DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

Academic Year 2012–2013

M. Danelutto

Version September 5, 2014

Teaching material – Laurea magistrale in Computer Science and Networking

CONTENTS

PREFACE

This material covers most of the arguments presented in the course of "Distributed systems: paradigms and models" given at the "Laurea Magistrale" in Computer Science and Networking (joint initiative by the University of Pisa and Scuola Superiore St. Anna) during the Academic year in 2012–2013. In particular, the material covers all arguments related to structured programming models and paradigms. The only arguments missing among those presented in the course are those relative to networks (wireless networks and peer-to-peer paradigm) which are covered by other, existing textbooks, as mentioned on the course web site at `http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spm/start`.

Being a kind of "working" version, you should frequently and carefully read the *errata corrige* available at the author's web site (`http://www.di.unipi.it/~marcod`) or on the course DidaWiki at `http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticar spm`). Please feel free to give any suggestion or to signal any kind of error directly to author writing to the address `marcod@di.unipi.it` with `Subject: SPM notes`.

M. Danelutto

February 2012, Pisa, Italy

ACKNOWLEDGMENTS

This book is the result of a long lasting activity in the field of structured parallel programming. I owe part of my experience to a number of colleagues that cooperated with me since 1990, including M. Aldinucci, F. André, B. Bacci, H. Bouziane, J. Buisson, D. Buono, S. Campa, A. Ceccolini, M. Coppola, P. D'Ambra, C. Dalmasso, P. Dazzi, D. Di Serafino, C. Dittamo, V. Getov, S. Gorlatch, L. Henrio, P. Kilpatrick, S. Lametti, M. Leyton, K. Matsuzaki, C. Migliore, S. Orlando, C. Pantaleo, F. Pasqualetti, S. Pelagatti, C. Perez, P. Pesciullesi, T. Priol, A. Rampini, D. Ratti, R. Ravazzolo, J. Serót, M. Stigliani, D. Talia, P. Teti, N. Tonellotto, M. Torquati, M. Vanneschi, A. Zavanella, G. Zoppi.

marcod

CHAPTER 1

PARALLEL HARDWARE (NEEDS PARALLEL SOFTWARE)

In recent years, a substantial improvement in computer and networking technology made available parallel and distributed architectures with an unprecedented power. The hardware revolution not only affected large parallel/distributed systems (e.g. those hosting thousands to millions of CPUs/cores in large "data center-like" installations) but also personal computing facilities (e.g. PCs, laptops and small clusters). This is mainly due to the shift from single core to multi core design of mainstream chipsets. The change from single core to multi core systems impacted also the programming model scenario. It introduced the need of (and the urgency for) efficient, expressive and reliable parallel/distributed programming paradigms and models.

In this chapter, we first outline the advances in computer architecture that took place in the last years and that are still taking place while we write these notes. Then we briefly point out the urgencies arising from these advances related to the programming model in the perspective of providing programmers with frameworks that simplify the task of developing efficient, reliable and maintainable applications for typical parallel and distributed architectures.

It is worth pointing out that within the whole book, we'll consider "parallel" and "distributed" computing as two facets of the same problem. Despite the fact parallel computing and distributed computing communities played different roles in the past and looked like two completely separated communities, we believe that distributed computing is a kind of "coarse grain" parallel computing or, if you prefer, that parallel computing if the "fine grain" aspect of distributed computing.

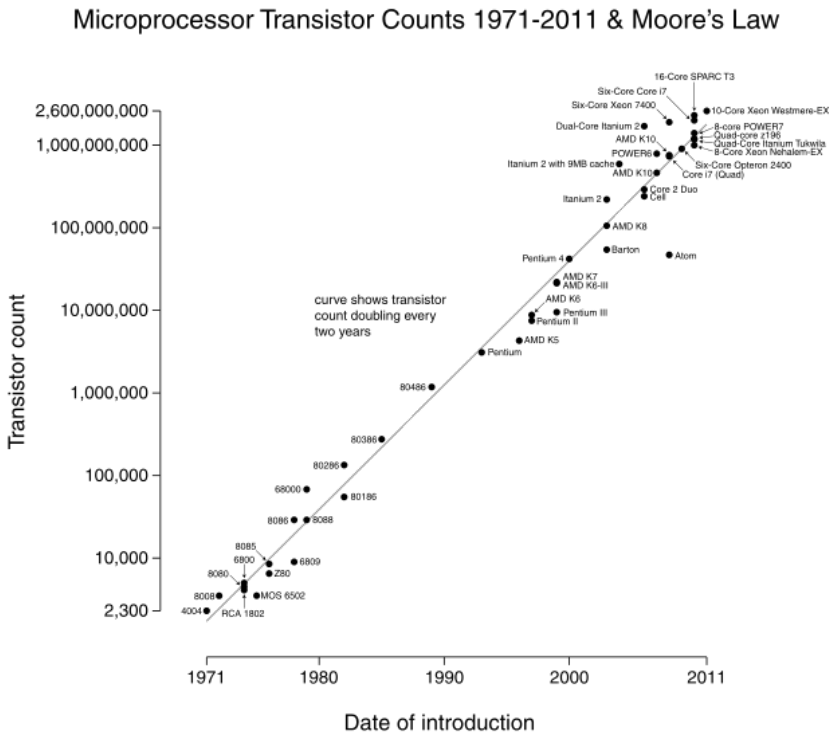


Figure 1.1 Moore's rule (Figure taken from Wikipedia)

1.1 HARDWARE FOR PARALLEL/DISTRIBUTED COMPUTING

We recognize several factors that contribute to the urgency of developing efficient programming models for parallel/distributed architectures. Among those factors, we would like to mention:

a) Evolution of CPU design and of the relative implementation techniques. When moving from single core to multi core chip design parallelism becomes the only way to exploit chip features.

b) Evolution of high performance computer architectures, such as those appearing in the Top500 list[1], that are moving more and more from dedicated architectures to tightly coupled assemblies of commodity processors through (almost) commodity interconnection network. These machines hosts hundreds of thousands of cores, and their exploitation requires correspondingly parallel/distributed applications.

c) The advent of cloud architecture and systems. These systems host a large number of processing elements/cores and provide the users the possibility to "buy" or "rent" a partition of the cloud to be used to run their own applications. Clouds require efficient programming models paired with efficient management of the resources used, which again is a typical problem related to parallel/distributed architecture targeting.

We discuss these three aspect in more detail in the following sections.

[1]http://www.top500.org

The Von Neumann architecture is the one with one (or more) processing unit and a single (data and code) memory unit. Program is stored in memory and a register in the processing unit denotes the address of the next instruction to execute. It is named after the computer scientist John von Neumann (1903–1957). The interconnection between processing unit and memory is called Von Neumann bottleneck.

Figure 1.2 Von Neumann Architecture

1.1.1 Evolution of CPUs

Since '60, Moore law dominated the CPU design and implementation scenario. Moore's original statement dates back to 1965. In his publication "Cramming more components onto integrated circuits", on the Electronics Magazine, he said that:

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

As a matter of fact, in the past 40 year, the amount of components successfully placed on a single chip doubled approximately every two year.

The larger and larger number of components available on the chip have been used up to early '00 to improve the power of a single, Von Neumann style, processor, basically in two different ways:

- by increasing the word length (4 bits in 1971 to 8, 16, 32 and up to 64 bits in 2008)
- by adding more and more features that contribute to the overall CPU performance (pipelined units, super scalar design, different cache levels, out of order execution, vector/SIMD instruction set, VLIW execution units, hyper threading, virtualization support, etc.)

Along with the larger and larger number of components integrated onto the same chip, clock frequencies have been raised. This allowed to implement CPUs with shorter and shorter clock cycles. The consequences of this 40 year trend have been twofold.

On the one hand, the classical (imperative or object oriented) programming model was supported by all these CPUs and therefore old programs run on all these new CPUs. In some cases, re compilation of the source code is the only action required to port the code. In case the new architecture is from the same manufacturer, binary portability of object code is in general guaranteed[2] and even compilation therefore is not necessary.

On the other hand, the existing programs may be run faster and faster on the new architectures, due to the higher clock frequency and to the new features introduced in processor micro architecture.

This long running process stopped in early '00, however. Power dissipation related to the higher number of components packaged on the chip and to the higher frequencies used come to a point where chips required complex, highly efficient heat dissipation systems, possibly liquid based. The amount of power dissipated by a chip was higher that several hundred of

[2]Old instruction set are provided on the new architectures. Pentium processors can still run code compiled for 8086 in the '80 at the beginning of '00.

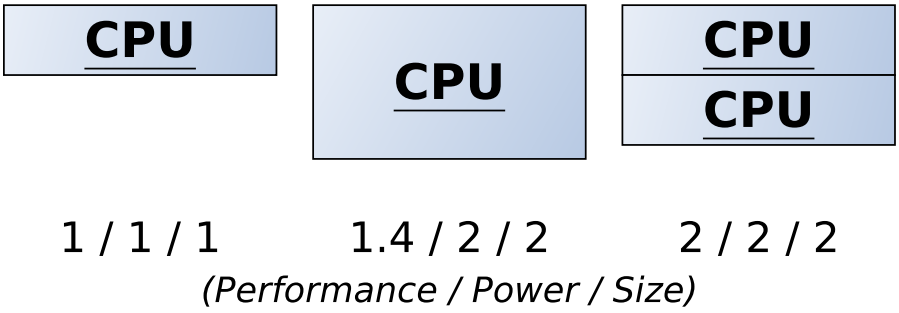


Figure 1.3 Pollack's rule

Watts. In the meanwhile, the new features added to CPUs required a substantial amount of the overall chip area but provided quite limited improvements in performance.

As observed by Fred Pollack, an Intel engineer:

> performance increases roughly proportional to the square root of the increase in complexity.

whereas is is known that power consumption is roughly proportional to complexity of the chip (and therefore to its area). This means that doubling the complexity of a chip only leads to a 1.4 increase in performance. However, if we use the double area to implement two smaller processors (half size of the 1.4 performance one), we get system that sports the double of performance of the original chip dissipating the double of power of the original chip.

This observation led to the development of an entirely new trend: engineers started designing more CPUs on the same chip, rather than a single, more complex CPU. The CPUs on the same chip are named *cores* and the design of the overall chip is called *multi core*.

As the chip implementation technology keeps improving with rates similar to those of the Moore's law era, a new version of the Moore law has been developed, stating that

> the number of cores on a chip will double approximately every two years.

Cores in multicore chips have been redesigned to get rid of those features that require a huge area to be implemented and only contribute a modest improvement in the overall CPU performance. As an example, cores in multicore chips usually do not use out-of-order instruction execution. This allows to design smaller chips, consuming less power and, mainly, being better packable in more and more dense multicore configurations.

Multicores already dominate the CPU scenario. Today it is almost impossible to buy a computer with a single core inside, but in case of "netbooks" or ultraportable devices[3]. Most of PCs already[4] sell with dual or quad cores inside, and top end machines currently use dual quad core or eight core processors.

However, multicore architectures need explicitly parallel programs to be used efficiently. Each one of the cores in the architecture must have one or more active thread/process assigned (in average) to keep the system busy and therefore to keep the system efficiency high. This radically changed the situation. Parallel programs are *needed*. There is no chance to take an old program and run it on a multicore getting a better performance, unless the original program has already been designed and implemented as a parallel program. Indeed, due to the simplification made to the cores, it could also be the case–and it is, usually–that

[3]although currently available tablets and smart phones already include chips such as the Tegra one: a dual ARM + nVidia GPU chip

[4]these notes are being written beginning of 2011

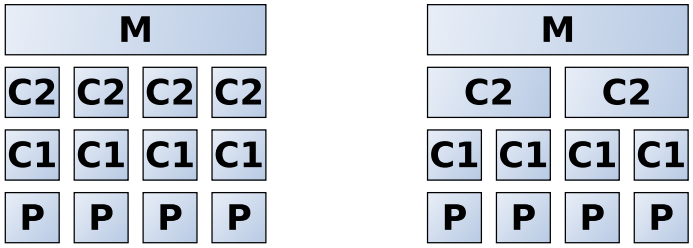


Figure 1.4 Different architectures for multicore SMP

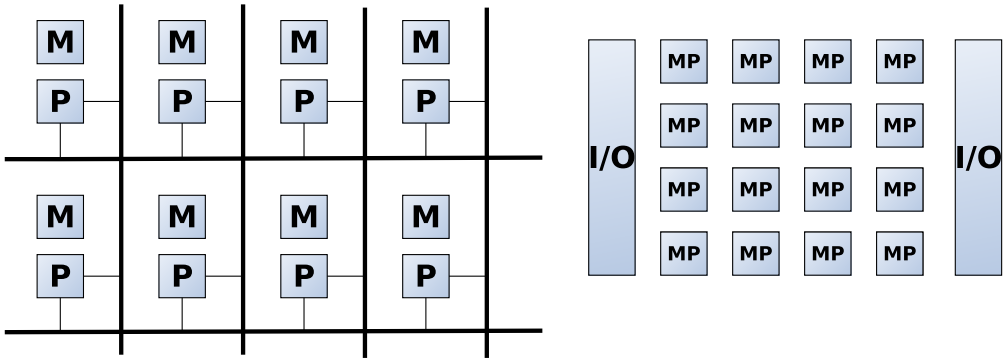


Figure 1.5 Typical architecture of many core chips

non parallel old code runs slower on new processors. Despite the fact multicore architectures need explicitly parallel code to be exploited, their architectural design is quite traditional: a number of CPU, with private caches accessing a *common shared memory space*. Therefore the architectural model exposed by the micro architecture is a plain UMA SMP (Uniform Memory Access, Symmetric MultiProcessor) model (see Fig. 1.4).

However, as technology evolves, more and more cores are available on chip and *many core* architectures are already available today, with 64 and more cores on the same chip. These architectures include Tilera chips (up to 100 cores[5]) and the Intel experimental 80 core chip[6]. These CPUs make a further step in architectural design in that they release the UMA shared memory concept in favour of a regular interconnection of cores (usually based on bi-dimensional mesh topologies) with own, private memory that can be accessed routing proper messages through the on chip core interconnection network. As a consequence, the memory model becomes a NUMA model (Non Uniform Memory Access, that is accesses to different memory locations/addresses may require substantially different times) and more advanced techniques have to be used to enforce data locality and therefore achieve complete exploitation of the architecture peculiar features.

Up to now we concentrated on two CPU families: "traditional multicore" CPUs (relatively small number of cores, UMA-SMP design) and "many core" CPUs (high number of cores, NUMA memory, regular interconnection structure). There is a third kind of architectures that are being developed and used currently, namely the GP-GPU ones (General Purpose Graphic Processing Units). These architectures actually provide a huge number of cores (in the range 10-100, currently) but these cores are arranged in such a way they can

[5]http://www.tilera.com/products/processors.php
[6]http://techresearch.intel.com/articles/Tera-Scale/1449.htm

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (Kw)
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12659.9
2	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT	186368	2566.0	4701.0	4040
3	DOE/SC/Oak Ridge National Laboratory United States	**Jaguar** - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.	224162	1759.0	2331.0	6950
4	National Supercomputing Centre in Shenzhen (NSCS) China	**Nebulae** - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 Dawning	120640	1271.0	2984.3	2580
5	GSIC Center, Tokyo Institute of Technology Japan	**TSUBAME 2.0** - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP	73278	1192.0	2287.6	1398.6
6	DOE/NNSA/LANL/SNL United States	**Cielo** - Cray XE6, Opteron 6136 8C 2.40GHz, Custom Cray Inc.	142272	1110.0	1365.8	3980
7	NASA/Ames Research Center/NAS United States	**Pleiades** - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband SGI	111104	1088.0	1315.3	4102
8	DOE/SC/LBNL/NERSC United States	**Hopper** - Cray XE6, Opteron 6172 12C 2.10GHz, Custom Cray Inc.	153408	1054.0	1288.6	2910
9	Commissariat a l'Energie Atomique (CEA) France	**Tera-100** - Bull bullx super-node S6010/S6030 Bull SA	138368	1050.0	1254.5	4590
10	DOE/NNSA/LANL United States	**Roadrunner** - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM	122400	1042.0	1375.8	2345

Figure 1.6 Top ten positions in the Top500 list, Feb 2012

only executed SIMD code (Single Instruction Multiple Data), in particular multi threaded code with threads operating on different data with the same code[7].

GP-GPUs are normally used as CPU co-processors. In case data parallel code has to be executed suitable to fit the GP-GPU execution model, the code and the relative data is offloaded from main (CPU) core to the GP-GPUs cores (and private, fast memory). Then some specialized GP-GPU code (usually called *kernel*) is used to compute the result exploiting GP-GPU data parallelism and eventually results are re-read back in the CPU main memory. Overall, GP-GPU exploitation requires a notable programming effort. Open programming models have been developed to operate on GP-GPUs, such as OpenCL[8] and GP-GPUs manufacturers provide they own programming frameworks to access GP-GPU functionalities (e.g. CUDA by nVidia[9]). However, these programming frameworks require a deep knowledge of target GPU architecture and programmers have to explicitly deal with all the CPU-GPU data transfers in order to use the GPUs. This makes parallel programming of GP-GPU very expensive, at the moment being, but the fact GP-GPUs already have on board a huge number of cores (e.g. 520 cores on nVidia Fermi GPU) allows to eventually reach impressive performances in plain data parallel computations.

According to chip manufacturers, current technology allows to increase by a plain 40% *per-year* the number of on chip cores. As a result, it is expected to have an increasing number of cores per chip in the near future. With such numbers, single chip programming *must* use efficient parallel/distributed programming models, tools and implementation techniques.

[7]At the moment being, GP-GPUs are often referred to using the term *many core* although in this book we will use the term many core to refer all those (non GPU) processors with a relatively high number of cores inside (in the range 100 to 1K)
[8]http://www.khronos.org/opencl/
[9]http://www.nvidia.com/object/cuda_learn_products.html

Processor Generation	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
Xeon 5600-series (Westmere-EP)	240	48	23578707.77	42434388.44	2847664
Xeon 5500-series (Nehalem-EP)	91	18.2	7777633.81	11877704.02	1046206
Opteron 6100-series "Magny-Cours"	34	6.8	6671696.8	9031446.6	980032
Xeon 5400-series "Harpertown"	26	5.2	3354399.4	4535729.23	395301
Opteron Quad Core	17	3.4	2100788.1	2794468	307530
POWER6	15	3	1128793.8	1480537.6	78752
POWER7	14	2.8	1678583	2233860.45	73808
Intel Xeon E5	10	2	2769524	3420620.8	162656
PowerPC 450	7	1.4	2176911	2618162.4	770048
Opteron 6200 Series "Interlagos"	7	1.4	2625043	3462200.2	363656
Power BQC	5	1	1427273	1782579.4	139264
Itanium 2 Montecito	4	0.8	227368	267980.4	42736
PowerPC 440	4	0.8	724683	906035.2	323584
Opteron Six Core	3	0.6	2731150	3586944	345602
Xeon 5200-series "Wolfdale-DP"	3	0.6	180579	230169.6	17280
Xeon 5500-series (Nehalem-EX)	3	0.6	1224940	1463569	161408
Xeon 5300-series "Clovertown"	3	0.6	334480	434227.2	38320
PowerXCell 8i	2	0.4	1168500	1537632	136800
Opteron Dual Core	2	0.4	265960	370024.4	48008
Xeon E7-x8xx-series (Westmere-EX)	2	0.4	129720	165888	17280
SPARC64 VIIIfx	1	0.2	10510000	11280384	705024
NEC	1	0.2	122400	131072	1280
SPARC64 VII	1	0.2	110600	121282	12032
PowerPC 970	1	0.2	63830	94208	10240
ShenWei	1	0.2	795900	1070160	137200
POWER5	1	0.2	75760	92781	12208
Core i5	1	0.2	61410	138930	9900
Xeon EM64T	1	0.2	53000	64972.8	9024

Number of Processors share for 11/2010

In addition to the table below, you can view the visual charts using the TOP500 charts page. A direct link to the charts is also available.

Number of Processors	Count	Share %	Rmax Sum (GF)	Rpeak Sum (GF)	Processor Sum
1025-2048	2	0.40 %	156020	199932	3328
2049-4096	60	12.00 %	2284042	2716312	221438
4k-8k	291	58.20 %	11685027	18923817	1769924
8k-16k	96	19.20 %	7607941	10538478	1037103
16k-32k	20	4.00 %	3375425	4351903	506760
32k-64k	17	3.40 %	5227787	8820874	783950
64k-128k	5	1.00 %	4655000	6694211	483678
128k-	9	1.80 %	8681851	12409784	1666146
Totals	**500**	**100%**	**43673092.54**	**64655310.70**	**6472327**

Figure 1.7 Sample summary figures relative to Top500 Nov. 2011 (top) and Nov. 2010 list (bottom)

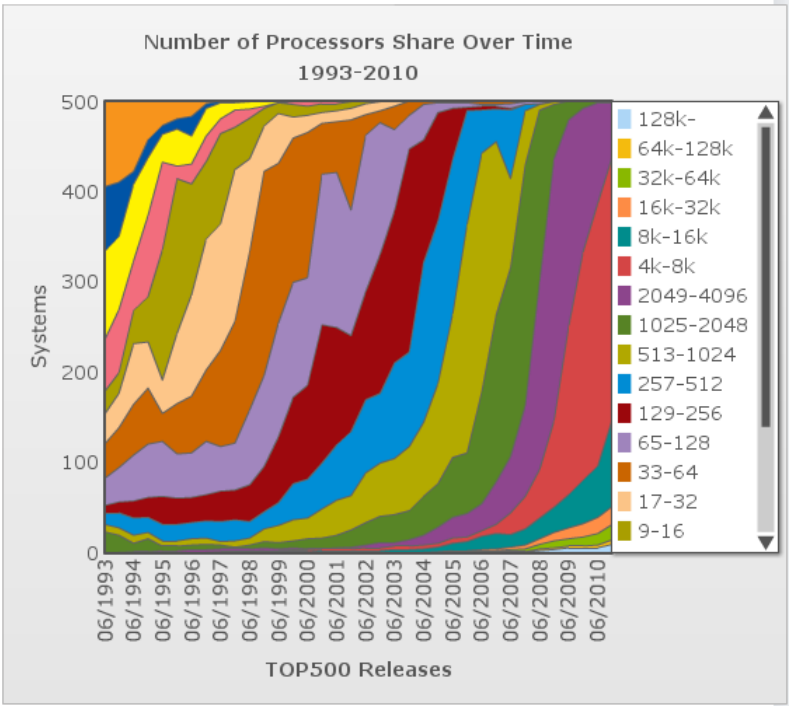


Figure 1.8 Historic/trend charts relative to Top500 (sample)

1.1.2 High performance computing architectures

Traditionally, we label as "high performance computing" all those activities involving a huge amount of computation/data. High performance computers are therefore those big "machines" usually owned by big institutions or departments, able to perform impressive amount of computations per unit of time and build out of an actually huge number of processing elements.

Since year, the `www.top500.org` web site maintains a classification of the world most powerful single computer installations. A list is published twice per year, in June and November, hosting the top 500 computer installations in the world. The installation are ranked by running a standard benchmark suite. Fig. 1.6 shows the top ten positions of the list published in Nov 2010. Data relative to any one of the lists published from 1993 on may be "summarized" also according to different parameters such as the number of processors/cores in the system, the kind of processors or interconnection network, the operating system used, etc. A sample of these summary tables is shown in Fig. 1.7.

But the more interesting possibility offered by the Top500 lists is the possibility to see the evolution of the features of the Top500 installations in time. Fig. 1.8 shows[10] one of the charts we can get summarizing the evolution in time of one of the features of the Top500 computers, namely the number of processors/cores included.

Looking at these time charts, several trends may be recognized in these high end systems:

- the number of processing elements is constantly increasing. Recently the increase in processor count has been partially substituted by the increase in core per processor count.

[10]the graph is in colors, there is no way to get it in black&white. Therefore it may be hardly read on printed version of the book. However, it can be easily regenerate "on screen" pointing a browser to the `http://www.top500.org/overtime/list/36/procclass` web site.

- most of the systems use scalar processors, that is "vector" processors–quite popular for a period in this kind of systems–constitute a less and less significant percentage of the totals.
- GP-GPUs are pervading the Top500 systems. More and more systems hosts nodes with GP-GPU coprocessor facilities. This makes the systems heterogeneous, of course.
- interconnection networks are evolving from proprietary, special purpose networks to optimized versions of a limited number of networks (Ethernet and Infinibad)
- Linux jeopardized all the other operating systems and currently runs more that 90% of the total of the systems
- these systems are used to run a variety of applications. The most popular one are financial applications (about 8.6% in Nov. 2010 list), research applications (about 16.4%) and "not specified" applications (about 34%) which unfortunately also include military applications.

From these trends we can therefore conclude that:

- we need programming environments able to efficiently manage more and more processing elements. The Top500 applications are mostly written using low level, classical programming environments such as MPI or OpenMP.
- we need programming environments able to efficiently manage heterogeneous architectures, such as those with GP-GPU equipped processing elements.
- we need programming environments supporting functional and performance portability across different parallel architectures, to be able to move applications across Top500 systems.

1.1.3 Cloud

The term *cloud computing* has been introduced in the '2000s and refers to the provisioning of computer resources to run some kind of application that may be sequential or parallel. The idea of *cloud* recalls the idea of a pool of resources with some kind of "undefined borders". The cloud resources are made available "on request" to the final user, that usually pays the access to the cloud per resource and per time used.

Clouds have been invented by industry/commercial companies, in fact. On the one side, several companies were using for their own purposes a huge number of workstations connected through some kind of private network. These collections were often dimensioned on peak necessities and therefore part of the resources were unused for a consisted percentage of time. Therefore these companies thought it was convenient to find ways of selling the extra power of this computing element agglomerate in such a way the investment made to set up the collections of computing elements could be more fruitful. On the other side, other companies found convenient to invest in computing resources and to sell the *usage* of these computing resources to persons/companies that could not afford the expense to buy, run and maintain the hardware resources they occasionally or permanently need but can anyway use some budget to use the resources from a provider.

As a matter of fact, in less than 10 years a number of companies are providing cloud services (including Amazon, Microsoft, Google) and the model is also currently being exported as a convenient way to organize companies internal services.

At the moment being, an assessed "cloud" concept can be summarized as follows:

> *cloud computing is a computing model that guarantees on-demand access to a shared pool of computing resources. The computing resources may include processing nodes,*

interconnection networks, data storage, applications, services, etc. and they can be configured to accomplish user needs with minimal management effort through a well defined interface.

where the key points are

Computing model This provides a computing model. The user must adopt this exact computing model to be able to use cloud computing.

On-demand Resources in the cloud are provided on-demand. There is no "reserved" resource in the cloud: cloud applications express their needs through suitable APIs and the cloud management software allocated the needed resources when the application is run. This, in turn, means that the user application may no rely on any physical property of the resources used. Different runs may get different physical resources. Therefore everything (hw) is accessed through proper virtualization services.

Shared The resources in the pool are shared among cloud users. Cloud management software will ensure separation among the logical resources accessed by different users.

Different resources The resources accessed through the cloud may be processing, storage or interconnection resources–on the hardware side–as well as application or service resources–on the software side.

Configurable Some configuration activity is requested to be able to access cloud resources. The configuration usually consists in writing some XML code specifying the "kind" of resources needed. In some other cases it may also consist in more specific programming activity interacting with the cloud APIs.

This definition of cloud is usually interpreted in three slightly different ways:

SaaS At the application level, clouds may be perceived as a set of services that can be accessed to compute results. We speak of *Software As A Service* (SaaS) in this case.

PaaS At the hardware level, clouds may be perceived as platforms of interconnected processing elements that execute user applications. In this case we speak of *Platform As A Service* (PaaS).

IaaS At an higher level with respect to PaaS we can perceive cloud as a virtual infrastructure, that is a hardware/software virtual environment used to run user applications. User applications in this case are rooted in the software tools provided by the infrastructure rather than directly on the virtualized hardware as it happened in the PaaS case. In this case we speak of *Infrastructure As A Service* (IaaS).

E.G.▷ As an example, if we develop a cloud application that uses the services provided by the cloud to reserve flight seats or hotel rooms to provide the final user the possibility to organize a trip to given place for a given amount of days, we are using a SaaS. If we prepare a cloud application were a number of virtual machines running our own customized version of Linux to run an MPI application on top of them, we are using a PaaS. Eventually, if we write a cloud MPI application we are using IaaS, as in this case the cloud will provide us the operating system + MPI virtual abstraction. ■

When using cloud for parallel computing care has to be taken to make an efficient usage of the resources claimed. The kind of actions taken also depends on the kind of usage (SaaS, PaaS or IaaS) we make of the cloud. Let us restrict to a PaaS usage. If we prepare our set of virtual machines to be eventually run on the cloud resources we have to be

aware that both different virtual machines may be allocated to different hardware resources and that different runs of our application may eventually get assigned different resources. Therefore, we must carefully access "system" resources through virtualization interfaces. Interconnection network should be accessed through is POSIX/TCP interface exposed by the PaaS, or the processing elements must be accessed to their generic specifications exposed (e.g. Intel IA32 architecture), for instance.

Due to this variance in the assigned resources some dynamic adaptation should be probably included in the application non functional code, to achieve the best efficiency in the usage of the "unknown" resources eventually assigned.

1.2 PARALLEL SOFTWARE URGENCIES

By looking at the three different aspects outlined in the previous Section, we can try to list a set of different urgencies (along with the connected *desiderata*) relative to the parallel programming environments. In this Section we briefly discuss the main items in this list.

Parallelism is everywhere. Parallelism is moving from HPC systems to all day systems. Personal computers, laptops and, recently, tablets and smart phones, have multi core processors inside. The amount of cores per processor is increasing constantly with time. We should expect to have a significant numbers of cores in any of these personal system within a few years. We therefore need to develop parallel computing theory, practice and methodologies rather than further developing sequential only their sequential counterparts. In particular, future programmers should be formed with parallelism in mind: we must abandon the idea that sequential algorithms come first and then they can be parallelized. These new programmers must start thinking to the problems directly in a parallel computing perspective.

Heterogeneous architectures are becoming important. With the advent of GP-GPUs most processing elements will have some kind of co-processor in the near future. GP-GPUs sport an impressive number of cores that unfortunately may be used only for particular kind of parallel computations, namely data parallel style computations. Parallel programming frameworks should be able to seamlessly support these heterogeneous architectures and should provide the application user with high level tools to access co-processor resources. This is even more important taking into account that heterogeneous architectures with FPGA based co-processors start to be available[11]. FPGAs are particular circuits that may be configure to perform different computations directly "in hardware". Being completely configurable they offer huge possibilities to improve critical portions of applications by directly programming "in hardware" these portions of code. However, this is a quite complex task definitely outside the possibilities of the application programmers. Programming frameworks should be able to provide high level access API also for this kind of accelerators. Outside the single processor features, even if we consider "distributed" architectures such as COW/NOW[12] architectures we have to take into account that rarely the systems hosted are identical. Some of them may be equipped with co-processors. More in general, upgrade due to normal system maintenance will introduce some degree of heterogeneity in the system.

[11] see Intel Atom E6x5C series `http://edc.intel.com/Platforms/Atom-E6x5C/` `http://newsroom.intel.com/community/intel_newsroom/blog/tags/system-on-chip`

[12] COW is the acronym of Cluster Of Workstations, whereas NOW is the acronym of Network Of Workstations

Parallel program portability is an issue. Parallel program portability is an issue, especially if non functional feature portability is considered in addition to functional portability[13]. Functional portability is usually ensured by the software platform targeted. An MPI program running on the first computer in the Top500 list may be quite easily ported on a different computer of the same list as well as on your own laptop. Of course the performances achieved and the amount of data processed will be affected. First of all, different amounts of memory may accommodate the execution of programs using differently sized input/output data. This is normal and it also happens in the sequential scenario. However, even in case the same data sizes may be processed different configuration in terms of the interconnection network and of the kind of processing elements used may lead to very different performance results for the very same parallel application. A decent parallel programming framework should ensure performance portability across different target architectures as much as possible, modulo the fact that nothing can be done in case of "insufficient" resources. Performance portability is not the only one non functional concern portability issue, however. Nowadays power management policies represent a significant non functional concern and again we wish to have parallel programming frameworks capable to support power management policy portability across different target architectures[14].

Code reuse is an issue. When developing a parallel application we will often be in the situation that a huge amount of sequential code already exists solving (part of) the different problems that have to solved in our application. These sequential codes are often the result of years of development, debugging and fine tuning. Rewriting them is therefore simply an unfeasible hypothesis. Any parallel programming framework must therefore support re-use of existing code as much as possible. In turn, this means that it must provide the possibility to link this code in those parts of the parallel application solving the particular problem solved by the sequential code and to orchestrate parallel execution of different instances of these sequential codes as well.

Dynamic adaptation must support non exclusive usage of parallel resources. When writing parallel code for cloud systems, as well as for architectures not in exclusive access, we must take into account of the varying features of the hardware used. Parallel software frameworks must provide support for the development of dynamic adaptation policies to get rid of the inefficiencies introduced by the varying features of the target architecture. The support must be efficient enough to support reconfiguration of the running application with a minimal overhead.

[13] see Sec. 2.3 for the definition of functional/non functional properties in a parallel computation

[14] e.g. we want programming environments capable to guarantee that a "power optimal" implementation of an application on target architecture A may be ported to architecture B preserving power optimality, either by simple recompilation or by some kind of automatic application restructuring

CHAPTER 2

PARALLEL PROGRAMMING: INTRODUCING THE CONCEPT

A generally assessed definition of parallel computing can be stated as follows:

> *Parallel computing is the use of two or more processing elements in combination to solve a single problem.*

At the very beginning of the parallel computing story "processing elements" where actually full computers, while nowadays they can be full computers as well as cores in general purpose (CPU) or special purpose (GPU) units. Also the term "two or more" was really referring to numbers in the range of ten, while nowadays it may range up to hundreds of thousands[15].

Despite the different types and number of processing elements used, however, parallel computing is a complex activity requiring different steps to produce a running parallel program:

- First, the set of possibly *concurrent activities* have to be identified. Concurrent activities are those activities that may be performed in concurrently/in parallel. Activities that may be performed concurrently exclusively depend on the features of the problem we want to solve. When defining the set of concurrent activities we defined both *which* and *how many* concurrent activities we have.
- Then, proper *coordination* of the concurrent activities has to be designed. Coordination includes synchronization as well as access to logically shared data from within different concurrent activities identified in the first step. Coordination features are derived from the features of the problem to solve and from the features of the set of concurrent activities devised in the first step.
- Last but not least, the concurrent activities and the related coordination activities *must be implemented*–in parallel–with the tools available and trying to maximize the

[15]The "supercomputer" currently in the top position of the `www.top500.org` list sports 186368 cores (Nov. 2010 list)

benefits of parallel execution. As an example, trying to minimize the wall clock time spent to solve the problem at hand. While implementing the concurrent activities with the available tools we can eventually realize that some of the logically concurrent activities are not worth begin parallelized as no benefits derived from their implementation "in parallel". Possibly, the tools available to support parallel implementation of concurrent activities will be such that the concurrent activities individuated in the first step may not be used to produce any kind of improvement–in terms of performance–of the application, and therefore we have to go back to step one and look for an alternative decomposition of our problem in terms of concurrent activities. In any case, the implementation of concurrent activities in parallel performed in this last step *depends on the tools and on the features of the parallel architecture we are targeting.*

E.G.▷ As an example, let us suppose we want to implement a parallel application transforming a color image in a black&white image. Conceptually, the image is an array of pixels. Each pixel is defined by a color. The set of concurrent activities we can individuate is such that we have one concurrent activity for each one of the pixels of the image. Each one of these activities will simply transform pixel representation according to a function $f : RGB \rightarrow B\&W$[16] mapping color pixels to black and white pixels. After having individuated the possibly concurrent activities needed to solve our problem, we investigate the kind of coordination needed among these activities. In our case, all the individuated concurrent activities may actually start in concurrently, provided concurrent accesses are supported to the different image pixels. Then we have to wait for the termination of all the concurrent activities in order to establish that our computation is completed, that is that our problem has been solved.

Let us consider we are using a shared memory multi core architecture with 16 cores to implement our application, and that the image to transform has been already allocated in memory. We can therefore implement each pixel transformation concurrent activity as a separate thread. We can start all the threads (one per pixel) in a loop and then wait for the termination of all those threads. The possibility to access single pixels of the image concurrently–for reading the color value and for writing the computed grey value–is guaranteed by the global shared memory abstraction exposed by threads.

We can easily recognize that the loop starting the threads and the one waiting for thread completion are huge, due to the huge numbers of pixels in the image, with respect to the number of cores in the target machine, which in turn determines the actual maximum parallelism degree of our application. Therefore we can "restrict" the number of parallel activities originally found in our problem by programming the application in such a way the image pixels are partitioned in a number of sub-images equal to the number of cores in our target architecture. The structure of the final parallel application remains the same, only the parallelism degree changes to match the parallelism degree of the architecture. This will obviously be a much better parallel implementation than the first one: no useless–i.e. not corresponding to the actual parallelism degree of the target architecture–parallelization has been introduced in the code.

What if the target architecture was a network of workstations instead? we should have used processes instead of threads and we should also have programmed some specific communication code to send the parts of the images to be processed to the different processes used. Due to longer time required both to start a process on a remote machine (w.r.t. the time needed to start a thread on a multicore) and to communicate over interconnection network (w.r.t. the time spent to implement an independent–e.g. not synchronized–access to the image pixels in main memory) this obviously requires a careful design of the implementation grouping large number of potentially concurrent activities into a much smaller number of processes to be run on different processing elements in the network.

■

[16] we denote with $f : \alpha \rightarrow \beta$ a function mapping values of type α to values of type β, as usual in functional style

Looking in more detail at the three steps detailed above, we can refine the different concepts just exposed.

Concurrent activities. When determining which are the possibly concurrent activities that can be used to solve our problem, we determine the candidates to be implemented in parallel. Two kind of approaches may be followed:

- we may look for *all* the concurrent activities we can figure out and assume that implementation will extract from the whole (possibly large) set those actually being worth to be implemented in parallel, that is the exact subset "maximizing" the performance of our application.
- Alternatively, we may directly look for *the best* decomposition of the problem in concurrent activities, that is the set of concurrent activities that, if implemented in parallel (all of them), lead to the better performance values in the computation of our problem.

The first approach clearly only depends on the features of the problem we want to solve, while the second one also depends on the features of the target architecture we want to use[17]. In general, different alternatives exists to devise the number and the kind of concurrent activities needed to solve a problem. Again, *all* possible alternatives may be devised in this phase or–more likely–one alternative is selected and other alternatives are possibly considered later on in case of negative feedbacks from the implementation phase.

Coordination. Coordinating concurrent activities is related to two different aspects: synchronization and data access. The synchronization aspects are related to problem of "ordering" the concurrent activities. Some activities may need results from other activities to start their computation and therefore some synchronization is needed. In other cases a set of concurrent activities must be started at a given time, for instance after the completion of another (set of) concurrent activity (activities). The data access aspects are related to the possibility for concurrent activities to access common data structures. Concurrent accesses have to be regulated to ensure correctness of the concurrent computations. Regulation of common (shared) data access depends on the kind of accesses performed. Concurrent read accesses to a read only shared data structure usually do not pose problems, while concurrent write accesses to a shared data structure must be properly scheduled and synchronized. Also, coordination may be achieved via accesses to shared data structures or via message passing. The usage of shared memory accesses and message passing can be combined to optimize coordination of concurrent activities in environments supporting both mechanisms.

Implementation. Implementation requires considering several problems, including parallel activities implementation, coordination implementation and resource usage. *Parallel activities* may be implemented with several different and partially equivalent mechanisms: processes, threads, kernels on a GPU, "hardware" processes, that is processes computed on specialized hardware such as programmable FPGA co-processors or specific hardware co-processors. *Coordination* implementation may exploit different mechanisms, such as those typical of shared memory environments or of message passing environments. Very often the coordination activities must take into account the possibilities offered by interconnection networks, internal or external to the processing elements. As an example, in clusters of workstations the interconnection network may support forms of broadcasting and multicasting that greatly enhance the possibilities to implement efficient collective[18] coordination

[17] target architecture in terms of both software (compilers and run times available) and hardware (number and kind of processing elements/resources available.
[18] i.e. involving a set of parallel activities/actors

operations. In a multicore, the UMA or NUMA[19] memory hierarchy may force particular allocation of shared data in memory in order to achieve efficient parallel activities coordination. *Resource usage* concerns the allocation of parallel activities to the resources (cores, processing elements, memory locations, etc.) actually available. In turn, this means that both *mapping*–assigning parallel activities to processing resources–and *scheduling*–starting and stopping parallel activities–policies and strategies must be devised. In general most of the problems related to mapping and scheduling are NP-complete problems, and therefore in most cases proper heuristics are used instead of exact algorithms.

2.1 CONCURRENT ACTIVITY GRAPH

The concurrent activities individuated in the first step performed to produce a parallel implementation of a given application is the starting point for all the implementation activities, the ones performed in the thirds step mentioned at the beginning of this Chapter. Actually, the starting point we have to consider includes both the concurrent activities and the actions needed to coordinate them in the parallel application.

We therefore introduce a formalism aimed at modeling the concurrent activities needed to implement a given parallel application: the *concurrent activity graph*. The concurrent activity graph is defined as a graph:

$$\mathcal{G} = (A, N)$$

where the nodes ($n \in N$) are concurrent activities and arcs represent two different types of relations among nodes, that is $A = C_d \cup D_d$, where :

- the *control dependencies* C_d, are those dependencies explicitly given by the programmer and to establish an ordering in the execution of concurrent activities. $\langle n_1, n_2 \rangle \in C_d$ represents the fact that the activity represented by node n_2 *must* be executed after the end of the activity represented by node n_1.
- *data dependencies* are those dependencies $\langle n_1, n_2 \rangle \in D_d$ modeling the fact that a data item produced by concurrent action modeled by node n_1 is needed to compute the concurrent activity modeled by node n_2.

Activities that do no have dependencies (neither control nor data dependencies) may be executed in parallel. Activities linked by a control or data dependency must be executed sequentially, although they can be executed in parallel with other concurrent activities.

Within the graph $\mathcal{G}$ we can identify *input activities* as those activities that are activated by the presence of application input data, and *output activities* as those activities that produce the data relative to final parallel application result.

Within the graph, we also identify the *critical path*, as the "longest" path of activities connected by control or data dependencies from an input node to an output node. "Longest" in this context has to be understood in terms of performance. As suggested by the name, critical path is the natural candidate when looking for optimizations in a parallel application. In fact, as a critical path starts with the availability of the input data and application results may only be delivered when all the output activities, including the one concluding the critical path, are terminated, the best time we can achieve in the execution of our parallel application is the time taken in the execution of the whole critical path[20].

[19] (Non)Uniform Memory Access. In a UMA architecture, all accesses to main memory costs the same time independently of the position of the core requiring the access. In a NUMA architecture, different cores may take different times to access the same shared memory location. The differences in time may be considerable.

[20] i.e. of the set of concurrent activities included in the critical path

We want to point out that the graph $\mathcal{G}$ is a logical graph, that is it corresponds to the logically concurrent activities we individuated among the overall activities to be performed in order to get the problem solutions.

Later on, we will speak of similar *implementation* graphs that refer to the parallel activities performed in the implementation of our concurrent activities graph. That graph could be sensibly different and is definitely dependent on the features of the target platform (hw and sw).

2.2 COORDINATION

Once the set of possibly concurrent parallel activities has been individuated, the next step consists in designing the proper coordination of these concurrent activities. Those activities that turn out to be *completely independent*–i.e. those activities that do not have dependencies on each other–may be coordinated in a simpler way than those activities that instead depend somehow one on the other.

A number of different coordination mechanisms may be used to purpose:

Execution coordination mechanisms These are those mechanisms that support coordinated creation (and possibly, termination) of a set of concurrent activities. Traditional coordination mechanisms comprehend *cobegin/coend* constructs, that is constructs that allow a set of concurrent activities to started at the same time and then to await the termination of these activities all together. Pthreads, the *de facto* standard in parallel programming on shared memory architectures, do not support particular execution coordination mechanisms, although these mechanisms may be easily implemented on top of the `phthread_create` and `pthread_join` primitive mechanisms.

Communication mechanisms Several different types of communications mechanisms are used, varying in the features of the communication implemented. Features of interest include:

- synchronous/asynchronous communication. In a synchronous communication, both partners should execute the send/receive at the same time, in order to have the communication performed. This means that some synchronization is required to ensure the property. In an asynchronous communication, the sends are asynchronous (unless the buffers used for the communication are full) while the receive is blocking (the receiver blocks in case no messages have been sent yet).
- symmetric/asymmetric communication. In a symmetric communication there is one sending partner and one receiver partners. Asymmetric communications include one-to-many and many-to-one communications. The terms refer to the number of senders (first word) and of receivers (last word). Particular cases of asymmetric communications are the broadcast, multicast, gather and reduce operations. These *collective* operations involve a set of partners. In the *broadcast*, the sent message reaches all the partners in the set. In the *multicast*, different parts of the message reach different subsets of the partner set. In both cases, the sender may or may not belong to the partner set (internal or external sender). Another particular cases of asymmetric collective communication are the gather and the reduce operations. We have again in this case a set of partners, but this time these are "sending" partners. There is a single "receiving" partner, that may or may not belong to the set. All the partners in the set send a specific message. In the *gather* communication, the receiving partner (internal or external) received

the data as a "collection", with one entry per sending partner. In the *reduce*, a binary, associative and commutative operator is applied onto all the items sent and the receiver eventually receives the resulting "sum" only (i.e. the result achieved by "summing" according to the binary operator).

Synchronization Synchronization mechanisms include different mechanisms differing in the kind and in the granularity of the synchronization, i.e. in the grain of the computations that have to be synchronized:

- *mutual exclusion* mechanisms, that is mechanisms that may be used to ensure that a given section of code is executed only by one of the concurrent activities in the program at a given time. Lock, mutexes, semaphores, critical sections and monitors are all mechanisms supporting mutual exclusion. Locks, are used for fine grain mutual exclusion, that is they usually are used to protect fine grain sections of code, and usually are implemented by means of active waiting (also referred to as *busy waiting*). Active waiting is tolerated due to the small amount of time needed to complete the protected code. Semaphores, critical sections and monitors are usually implemented without active waiting: concurrent activities that attempt to execute a section protected by one of these mechanisms are blocked and de-scheduled up to the moment the critical section is "free", that is no other concurrent activity is executing that section. Blocking and (de)scheduling activities are organized in such a way fairness is guaranteed.
- *barrier* mechanisms, implement "collective" synchronization, in that a barrier involved a number of concurrent activities and guarantee that all these activities reach a given point in their execution (the barrier) before proceeding. Typically, barriers are used to take care of all these situations where we must be sure that all (or part of) the concurrent activities are completed certain computations before proceeding to the next step.

The choice of different mechanisms to implement concurrent activities *orchestration*[21] may greatly impact the final "non functional"[22] features of our parallel application.

It is worth pointing out that the choice of an abstract coordination mechanism may be also affected by the availability of primitive/non primitive support for that mechanisms in the implementation framework.

E.G.▷ Let us suppose that in order to coordinate the execution of the parallel activities of our parallel application we decide to use a cobegin/coend execution coordination mechanism, while the target programming framework is only supporting phtreads. Pthread mechanisms only provide start and join methods. Therefore we have to implement some kind of abstraction, based on these mechanisms, to support cobegin/coend. If the overhead introduced by the abstraction is significant, the advantage of using this execution coordination mechanism may be completely impaired. Therefore, we should go back to the phase were we decided how to organize the concurrent activity coordination and consider some alternative possibility. ■

2.3 FUNCTIONAL AND NON FUNCTIONAL CODE

When considering the code needed to implement a parallel application–i.e. the code produced at the end of the implementation phase as described in the previous Section–we can easily understand it is composed of two distinct and very different kinds of code:

[21] the term *orchestration* is often used to refer to the coordination execution of a set of parallel/distributed activities concurring to the implementation of the same application
[22] see Sec. 2.3

Functional code The code needed to compute the actual results of the parallel computation. In case we want to compute weather forecast, the computation of the weather parameters (temperature, rain probability, etc.) out of the set of observed measures (current temperature, wind direction and intensity, humidity, etc.) is made by functional code.

Non functional code The code needed to orchestrate the computation of functional code in parallel. This includes code to set up parallel activities, to schedule these activities to the available resources and to coordinate their execution up to completion. In case of weather forecast, the split of functional code computation in such a way different sub-domains of the input data may be processed independently and in parallel and then a unique result may be build from the partial results computed on each sub-domain is all managed by non functional code.

In general, functional code is the code needed to compute *what* the application has actually to compute, whereas the non functional code is the code needed to determine *how* these functional results are actually computed in parallel. Functional code is *domain specific*[23] code and it has often been developed and tuned in years/decades. Non functional code is *platform specific*[24] instead.

The kind of expertise needed to write good code is therefore different in the two cases:

- to write good functional code a good expertise in the application domain is required, which is typically the expertise of specialized *application programmers*, whereas
- to write good non functional code a good expertise in the parallel programming is required, which is quite independent of the application area but very dependent on the target platform used.

2.4 PERFORMANCE

Performance is the main goal of parallel programming. The (quite difficult) task of implementing a parallel application solving in parallel a given problem is only afforded if we have a reasonable expectation that eventually the parallel application will provide "better" results, in terms of performance, with respect to the sequential application(s) solving the same problem.

Now the "performance" of an application may be defined in different ways. In Chap. 5 we will carefully introduce and discuss performance measures. What is in general expected from a parallel program in terms of performance is that

> *the time spent to execute the parallel program using n processing elements is about $\frac{1}{n}$ of the time spent executing any sequential program solving the same problem.*

As we will see, several factors impair the possibility to achieve such behaviour. In particular, the features of the problem to be solved have to be taken into account. First of all, not all the problems are suitable to be parallelized. Some problems have been recognized as inherently sequential and therefore will never benefit from parallelism/parallel implementation. Then, even those problems suitable to be somehow parallelized may contain a fraction of the work needed to solve the problem which is not parallelizable.

A well know law, the Amdhal law, states that the amount of non parallelizable work in an application determines the maximum speedup we may achieve by parallelizing that application.

[23] the term *domain specific* is used to denote code/features depending on the application domain at hand
[24] again, we refer to "platform" as the ensemble of software and hardware in the target architecture.

Assume you have an application where f percent of the total work is inherently sequential– it could not be parallelized. And assume that the whole application needs T_S units of time to be completed in sequential. Therefore fT_S units of time will be spent in the application *serial fraction* and $(1-f)T_S$ units in its non serial fraction. The best thing we can imagine is that using n processing elements the $(1-f)T_S$ time can scaled down to

$$\frac{(1-f)T_S}{n}$$

Therefore the speedup[25] we can achieve with our application, which is defined as

$$sp(n) = \frac{T_{seq}}{T(n)}$$

(the ratio among the time spent in sequential computation (1 processing element) and the time spent in the parallel computation with n processing elements) is

$$sp(n) = \frac{T_s}{fT_S + (1-f)\frac{T_S}{n}}$$

If we assume to have infinite processing resource available, the term $(1-f)\frac{T_S}{n}$ clearly goes to 0 and therefore the asymptotic speedup may be written as:

$$\lim_{n\to\infty} sp(n) = \frac{T_S}{fT_S} = \frac{1}{f} \tag{2.1}$$

Equation 2.1 is the Amdhal law. It has important consequences. In particular, it states that any application having $x\%$ of serial fraction could not achieve a speedup larger that 1 over x.

E.G.▷ If you have an application running in 10 hours with a 10% serial fraction, you will never succeed running this application in less than 1 hour ($\lim_{n\to\infty} sp(n) = \frac{1}{0.10} = 10$, $\frac{10hours}{10} = 1\ hour$). ■

Amdhal law establish a completely theoretic limit to the speedup we can achieve through the parallelization of the activities needed to solve a given problem. However, there are many factors that impair the possibility of achieving the limits established by Amdhal law. In particular, *overheads* incur in the implementation of parallel applications. Overhead incur in the execution of any non functional code of parallel applications. An overhead is to be intended as a time loss with respect to the computation of the parallel application (functional) results. As an example, setting up the parallel activities, either threads or local/remote processes, is an activity that introduces an overhead. If we consider the time needed to compute the solution of our problem, this does not comprehend the time needed to set up parallel activities. However, if we do not setup parallel activities we'll not be able to achieve any kind of speedup. Therefore overheads incur, they are somehow necessary to implement parallelism, but we have to manage to minimize them as much as possible. Another example of overhead is the one related to communications. If we set up a number of concurrent activities as processes on different processing elements and some data has to be distributed to the processes before they start computing, then the time needed to distributed the data through communication is plain "communication overhead".

Looking again at the Amdhal law, we can modify it in such a way overheads are taken into account. In particular we can image the time spent to compute in parallel the non serial fraction of the application is $\frac{T_s}{n} + T_{ov}$ instead of being simply $\frac{T_s}{n}$. Unfortunately, overheads

[25] the concept of speedup will be better introduced in Chap. 5.

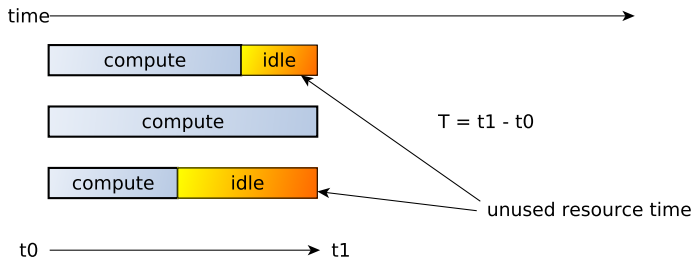


Figure 2.1 Unbalanced computation. Three concurrent activities (a_1, a_2, a_3) on three different processing elements, spend unbalanced amount of times to complete. The application completes when al the activities have been completed. Idle times on the processing elements computing a_1 and a_3 reduce application efficiency.

(T_{ov}) depend on the parallelism degree. Communication time depends on the number of entities participating in the communication or the time spent to start concurrent/parallel activities depends on the number of activities actually involved, as an example. Therefore Amdhal law should be rewritten as

$$\lim_{n\to\infty} sp(n) = \lim_{n\to\infty} \frac{T_s}{fT_S + (1-f)(\frac{T_S}{n} + T_{ov}(n))} = \frac{1}{f + (1-f)\frac{T_{ov}(n)}{T_S}} \tag{2.2}$$

In case of negligible overhead this clearly reduces to Eq. 2.1, but if the fraction of overhead with respect to the total sequential time is significant, the asymptote speedup is even smaller than one expected taking into account only the weight of the serial fraction. It is clear therefore that the less overhead we introduce in our parallel application with non functional code the better performance we can achieve.

2.5 OTHER NON FUNCTIONAL CONCERNS

When designing and implementing parallel applications, performance is the main non functional concern we are interested in, but other non functional concerns are also important. We outline in the following paragraphs the more important further non functional concerns in parallel programming.

2.5.1 Load balancing

When considering a set of parallel activities each computing a subset of what's needed to compute a global result, we must be careful not to organize these activities in such a way load unbalance is verified. Load unbalance is the situation where parallel activities in a set of activities get a different amount of work to be computed. As we are assuming that the results of all the activities are needed to produce the global result, in case of load unbalance we have to wait for the longest lasting activity to complete before producing the global results. While the longest lasting activity is completing, the other activities may be idle having already finished their computation. As a result, the overall computation will last more than in case of a balanced distribution of work (see Fig. 2.1).

We can therefore conclude that load balancing of parallel activities is fundamental to achieve good performance values. Several techniques may be used to achieve load balancing, including:

Static techniques These techniques may be used at compile time or at run time to establish a static subdivision of work among the concurrent activities actually run in parallel. In case the static assignment turns out to be not correct, either due to the

impossibility to exactly predict the amount of work assigned to concurrent activities or because of changes in the target architecture (e.g. additional load charged on processing elements of different applications sharing the same resources), the overall result will demonstrate poor performance.

Dynamic techniques These techniques are in general used at run time and achieve load balancing by dynamically assigning and re-assigning work to the concurrent activities. These techniques may achieve better load balancing in all those cases where the exact amount of work of activities could not precisely be defined before the activities actually take place at the expense of a larger overhead related to the management of dynamic work assignment/re-assignment.

E.G.▷ As an example, consider again the example made at the beginning of this Chapter, relative to transformation of color images into black&white images. If we statically divide the image pixels in partitions and we assign one partition per processing element, the partition overhead is negligible but in case one of the processing elements becomes overloaded, its partition could be computed sensibly later than the other ones delaying the delivery of the whole black&white image. On the other hand, if we dynamically assign to idle processing elements one of the pixels still to be computed, we will achieve a much better load balancing which is able to manage processing element overload situations such as the one mentioned above. However, the cost in terms of communications/accesses to shared memory and scheduling activities–which is overhead from our viewpoint–is (much) larger than in case of the static assignment. Therefore when considering static or dynamic assignment alternatives we have to measure this overhead, see which is the relation of the overhead introduced with respect to the overhead achieved due to load unbalance and eventually decide whether it is worth implementing the dynamic or the static assignment algorithm. ——————— ■

Load balancing is also impacted by the kind of processing resources used. In case of heterogeneous processing elements[26] load unbalance may also arise from the usage of different processing elements to compute comparable amounts of work.

2.5.2 Reliability

Reliability in parallel computing may be defined as *the ability to complete a computation in presence of (different kind of hardware/software) faults.*

Reliability, also referred to as "fault tolerance", is by itself a huge problem even in case sequential computations are taken into account. When parallel computation are considered is even more complex, due to the larger set of activities that have to be managed to get rid of the faults.

A number of techniques have been developed to implement fault tolerance. We mention just one of these techniques, without pretending to be complete but just to give the flavor of the problems that have to be solved.

Checkpointing. When executing an application we can insert non functional code taking "snapshots" of the application at given points during its execution. Each one of these snapshots, called usually *checkpoint* hosts the whole "state" of the computation, including values of variables in memory and the value of the program counted at the moment the checkpoint has been taken. The checkpoints are saved to the disk. Once checkpoint cp_{i+1} has been successfully saved to disk, checkpoint cp_i may be removed from disk. In case a hardware or software fault causes the fault of the incorrect (and usually immediate) termination of the

[26]e.g. processing elements in a cluster with different CPUs and/or different amounts of memory and different clock speed, or cores in a multicore with NUMA access to shared memory banks.

application, rather than restarting it from the beginning, the last checkpoint saved to the disk is used to restart the application. Memory representing variables is reinitialized with the variable values saved to disk and the control flow of the application is restarted from the program counter saved in the disk checkpoint.

Several problems must be taken into account, including: i) we need a mechanism to perceive faults and consequently restart the application from its last valid checkpoint, ii) we need to able to save the *application state* to disk, including side effects, iii) we need to insert into the application code proper non functional code "freezing" the application, dumping the application state to disk and eventually "unfreezing" the application[27].

Now consider the same mechanisms applied to a parallel application. We need to be able to take a "distributed" checkpoint, that is a checkpoint with parts related to the state of each one of the parallel activities making up the application but we also need to checkpoint the global state of the application. As an example we must take into account (and store to disk) *where the set of concurrent activities as a whole* was arrived in the execution of the parallel application. This means that if concurrent activity C_i was sending a message to C_j, we have to store this as C_i being sending the message, C_j being going to receive the message and with the actual contents of the message stored somewhere else in the "interconnection network" state, maybe. Also, consider the problem of synchronizing the checkpoints on the different processing elements used for our parallel computations. We need a two phase protocol where all the processing elements first agree on the event "it is time to take a checkpoint" and only when they all agree and the computations on all the processing elements have been "frozen" they may start taking the local snapshots.

Without entering in too much details it should be clear that organizing fault tolerance in a parallel application is quite a difficult task. However, it should also be clear that HPC *definitely requires* fault tolerance. If the MTBF (Medium Time Between Faults, the average time we can expect to elapse before another fault appears) in a normal computer ranges in month or years, when considering large supercomputers this time goes down to hours. This is mainly due to the very high number of components used in this large supercomputers, but the net effect is that in these installations the MTBF may be smaller than the time needed to run an application. This means that if no fault tolerance (non functional) code is properly inserted into the parallel application code, the application will never succeed to complete correctly.

2.5.3 Security

Security is related to the possibility to keep confidential data and code used to compute parallel applications as well as to guaranteed integrity of data and code transmitted across different processing elements connected by some interconnection network and participating to the parallel computation.

Several distinct issued related to security must be taken into account in parallel computing, especially taking into account that parallel machines are rarely assigned to a single user and that different kind of distributed architectures (COWs, NOWs and GRIDs) may be implemented on top of public (and therefore potentially unsecure) network segments. We just name a couple of them here and the we list additional issues specifically related to parallel computing.

Authentication. When using machines made up of a number of different computing elements, each with its own copy of an operating system, authentication is needed to be able to run a parallel program. In particular, all accesses to the machine resources (process-

[27] it is necessary to stop the application while taking the checkpoint to avoid inconsistencies, in general

ing elements) must go through proper authentication procedures/steps. NOW and GRIDs usually raise this problem, although both GRID middleware[28] and COW/NOW management software provide SSI [29] (Single System Image) facilities that solve this problem or anyway make it simpler. As an example, some GRID middleware use identity certificates and certification authorities to guarantee access to the grid resources to a given user. The user provides its certificate to the grid front end and the grid middleware takes care of propagating the certificate to the resources that will be actually reserved and used for the user application. However, in all those cases where no specific support for authentication is provided, parallel application programmers must insert proper authentication code (non functional code) into the application code.

Sensible data handling. When using data that should not be revealed outside, specific techniques must be used to cipher data flowing across untrusted network segments, such as those typical of public network infrastructure. Again, this is non functional code. The parallel application programmer must insert *in all the code relative to communication of either code or data* suitable routines to cipher transmitted in such a way in case they are snooped it is very difficult to evince their actual content. Message ciphering–at the sending side-and deciphering–at the receiving side–is anyway a costly activity. It is pure overhead from the functional viewpoint. Therefore its cost should be minimized as much as possible. This may require additional processing elements only dedicated to security handling to guarantee the better completion time of our application at the expense of some additional processing power dedicated to security overhead.

Sensible code handling. While sensible data handling may be managed inserting proper, non functional ciphering/deciphering code in the application code, the management of sensible code requires different techniques. "Code deployment", the distribution of code onto the different processing elements participating to a given parallel computation, is usually made with *ad hoc* tools: in case of distributed architectures, specific COW/NOW management tools or GRID middleware tools are used. In case of multi/many core architectures operating system or machine vendor tools are used instead. The problem of sensible code handling is more evident in network based architectures, for the very same motivations stated when dealing with sensible data handling. In this case the parallel application programmers has two possibilities to ensure secure code management during code deployment: either the deployment tools already support secure code management–and in this case it is only a matter of configuring these tools–or they do not support secure code management and therefore the parallel application programmer is forced to re-implement its own, secure deployment tools[30].

2.5.4 Power management

Power management is a (non functional) concern when dealing with parallel applications. The amount of cores used in current parallel applications may be huge and any saving in power will correspond to a significant decrease of cost. Each one of the concurrent activities identified when parallelizing an application may potentially run on a distinct processing element. Also, each sequential activity may be completed in a time roughly proportional to processing power of the processing element used for the computation. It is also known, however, the more powerful processing elements consume more than the less

[28] the set of libraries and run time tools used to run computational grids
[29] Single System Image abstraction provides the user with the illusion a set of distributed and independent processing elements are a unique system with parallel/distributed processing capabilities
[30] as an example ciphering code before the actual deployment through unsecure network segments.

powerful ones, and that GPUs may consume less power than CPUs when executing data parallel computations.

Therefore, when designing parallel applications care must be taken to avoid using unnecessarily "power hungry" computations. If a computation may be executed in time t on a (set of) processing element(s) P_a or in time $t' > t$ on a (set of) processing element(s) P_b consuming respectively w and w' Watts, in case both t and t' happen to be such that no deadline is missed[31], then the processing element(s) with the smaller power consumption must[32] be used. In our case, if $w < w'$ then we should use P_a otherwise we should use P_b.

Power management requires special non functional code in the application code to support *monitoring* of power consumption. Also, in case we want to support the possibility to move applications from more consuming processing elements to less consuming ones once we evince that the less consuming and less powerful processing elements guarantee anyway the completion of the activities "in time", we need further non functional code to be added to the parallel application, namely the one needed to stop an activity, move it to another processing element and–last but not least–to restart the moved activity after notifying all the activities in the parallel computation that location of the moved activity has changed[33].

2.6 "CLASSICAL" PARALLEL PROGRAMMING MODELS

We will use the term *classical parallel programming models* in this text to refer to all those programming models (or frameworks) that provide the programmer with all the tools needed to write parallel programs without actually providing any primitive *high level* programming paradigm. More explicitly, we will include in the category of "classical" parallel programming models all those models that basically provide suitable mechanisms to implement concurrent activities, communication and synchronization as well as concurrent activities mapping and scheduling, but do not provide higher level mechanisms such as those completely taking care of the implementation details related to a given parallelism exploitation pattern nor they provide suitable abstractions to separate functional (business logic) code from non functional one.

As an example, programming environments such as those provided by COW/NOWs with Linux processing elements interconnected through TCP sockets will be classified as "classic" parallel programming environments, as well as programming environments such as MPI (the Message Passing Interface [72]) or OpenMP (the current *de facto* standard for multi core programming) and Cuda/OpenCL (the current *de facto* standards for GP-GPU programming [59, 58]).

In this Section, we will shortly discuss POSIX/TCP environment. The interested reader my find a number of different documents describing the all those "classic" programming frameworks, but as POSIX/TCP is something which is well known and ubiquitously available, we will introduce it here in such a way later on we will/may use it to discuss different details related to the implementation of structured parallel programming frameworks.

2.6.1 POSIX/TCP

The POSIX/TCP framework relies on two pillars:

- The availability of the POSIX system calls, including those related to process management and to TCP/IP socket management, and

[31] in other words, they do not make the critical path execution longer
[32] could be conveniently
[33] This to ensure consistency in communications, as an example.

- The availability of some kind of shell command line interface supporting remote access features such as `ssh`, `scp`, `rsync`, `rdist`, etc.

The first pillar may be exploited to run parallel programs (i.e. parallel program components) on a variety of target architectures, without being concerned by operating system peculiarities. In particular, POSIX processes and threads constitute a kind of *lingua franca*[34] in the field of the mechanisms supporting the set up and running of concurrent activities. On the other side, TCP (and UDP) sockets represent a viable, quite general mechanisms[35] to implement communications between the parallel application concurrent activities.

The second pillar allows to build all those tools that are needed to support the deployment and the parallel/distributed run of the parallel program components.

We separately outline the main features, commands and system calls related to these two pillars in the next sections.

2.6.1.1 POSIX system calls In this section we outline the POSIX system calls related to the set up of concurrent activities, to communications and to synchronizations. We just outline the main system calls of interest. The reader may refer to the manual pages of any POSIX system (using obtained by running a `man syscall-name` command at the shell prompt) for further details on the exact syntax and usage of these system calls.

Concurrent activity set up. Concurrent activities may be set up in a POSIX framework as either *processes* or *threads*. The related system calls operate in the framework set up by the operating system. Therefore processes (threads) are processes (threads) of the same "machine" (either single processor or multicore). POSIX presents a peculiar interface for *process set up*, based on two distinct system calls. The `fork` system call *duplicates* the process invoking the syscall. The original process gets the process identifier of the new (copy of the original) process, while the new process get a "0" (process identifiers are integers in POSIX). The `exec` system call, instead, differentiates the code of the current process: the old code being executed before the exec syscall is abandoned and a new program is used as the process body taken from the executable file specified through the exec parameters[36]. Therefore the typical sequence of system calls used to create a process includes a `fork`, followed by a conditional statement checking the `fork` result: in case it is 0 (that is it is the *child* process running) an `exec` is called to differentiate the code. In case it is not 0 (that is it is the *parent* process running) the code continues with the rest of the process body. Processes execute in a separate address space. No sharing is supported unless the processes explicitly use SysV system calls to declare and use segments of shared memory (see communication paragraph below). However, the child process inherits a *copy* of all the variables defined and assigned by the parent process before the `fork` execution. The termination of a process may be awaited using a `wait` or a `waitpid` system call. The former waits for the termination of any one of the processes forked by the process invoking the wait. The second one waits the termination of a process with a specific process identifier. Both syscalls are blocking.

POSIX *threads* (*pthreads*) are set up with a `pthread_create` system call. Thread body must be specified by passing–via one of the parameters–a function receiving a `void *` argument and returning a `void *` result. A single parameter may be passed to the thread (the function parameter) via another `pthread_create` parameter. Threads share the global address space of the process invoking the `pthread_create` calls. This means that global variables are shared among threads. Threads may access them for reading and writing.

[34] POSIX API is implemented in Linux/Unix, Mac OS X and (with proper support tools) in Windows systems.

[35] even if not perfectly efficient, for certain types of communications

[36] There are several different version of the exec syscall depending on the kind of data type used to represent the parameters, actually: `execv`, `execl`, `execve`, `execle`

In case of multiple threads accessing variables, the accesses may/should be synchronized using locks and semaphores that are provided within the same `libpthread` providing the mechanisms for thread creation. The termination of a thread may be awaited using the blocking system call `pthread_join`.

Communication. POSIX provides different mechanisms supporting communications among concurrent entities: shared memory for threads, pipes, sockets, message queues and shared memory for processes. Shared memory is primitive when using pthreads. No special mechanisms are needed to declare or use shared memory, but the proper usage of synchronization mechanisms when performing concurrent accesses to shared variables. POSIX processes may use pipes or sockets to implement inter-process communications. Pipes are "in memory" files. They are declared using the `pipe` syscall. This returns a couple of file descriptors that may be used (one for reading and one for writing) to access the pipe. Writing to the write descriptor results in the possibility to read the same data on the read descriptor of the pipe. Usually pipes are created *before* forking a new process. Then either the parent process writes to the pipe and the child process reads from the pipe or vice versa. Of course, pipes created in the parent process may also be used to communicate in between two child processes. Operations on pipes require the usual `read` and `write` syscalls used to read or write plain disk files. Sockets may be also be used for interprocess communications. Sockets eventually provide descriptors such as those provided by a pipe, but there are a couple of significant differences: i) socket declaration is asymmetric. Two processes using a socket behave as a client/server system. The server declares a server socket and binds it to an address. Then it accepts connections by the client. The result of a connection is a socket providing the file descriptor used to access transmitted data. The client declares a client socket and connects it to the server socket. The connection returns the file descriptor used to access transmitted data. ii) the socket file descriptor supports *bi-directional* communication. Both client and server may write and read to the socket descriptor. Data written by client (server) may be read by server (client).
The syscalls used to operate on sockets include: `socket` to declare either client or server socket, `bind` to bind a server socket to an address, `connect` to connect a client socket to a server socket, `listen` to declare the amount of pending connections allowed on a server socket, `accept` to accept connections from clients on a server socket, `close` and `shutdown` to terminate a socket connection. POSIX supports `AF_INET` and `AF_UNIX` sockets. The former uses IP addresses (IP address + port number) and support communications between remote machines. The latter uses "filename" addresses and only supports communications between processes running on the same machine (instance of POSIX operating system). POSIX so called "SysV IPC" syscalls allows processes to share memory segments and to exchange messages via message queues. They also provide the semaphore mechanisms needed to synchronize accesses to the shared memory segments. The declaration of a shared memory segment is achieved using a `shmget` syscall. The shared memory segment may then be accesses via a normal pointer obtained from a `shmat` syscall. Different processes may access the same shared memory segment provided they agree on the shared memory segment identifier passed to the `shmget` syscall. With similar system calls (`msgget`, `msgsend`, `msgrcv`) shared message queues among processes and (`semget`, `semop`) shared semaphores may be used.

Synchronization. To synchronize activities of POSIX processes and thread different mechanisms may be used. Threads may use locks and semaphores, via `pthread_mutex_lock`/`pthread_mutex_unlock` and `sem_wait`/`sem_post` syscalls. Processes running on the same machine may use SysV semaphores (see above) to synchronize, as well as `AF_UNIX`, `AF_INET`, SysV message queues or pipes, used in such a way pure synchronization messages

are exchanged[37]. In order to synchronize the activities of processes running on separated machines, `AF_INET` sockets must be used.

2.6.2 Command line interface

In this section we shortly outline the main Unix commands used to deploy and run parallel/distributed programs on a network of POSIX workstations and to control the execution. As for the POSIX system calls, the interested user may get more details on these commands running a `man command-name` at the shell prompt.

Deployment. To deploy code and data to remote machines, the `scp`, `rsycn` and `rdist` commands can be used. `scp` copies files/directories to and from a remote machine. Sources and destinations may be specified as local files/directory (usual syntax) or remote ones (denoted with the machine name followed by ":" followed by the (relative or absolute) filename. Different flags may be used to recursively copy directories and to affect the ciphering algorithms used to secure the communication. `scp` usually requires the login/password of the remote machine, unless you adopt special procedures[38]. `rsync` synchronized local and remote directories (denoted with a syntax such as the one used for `scp`), that is copies the modified files of the source to the destination. It is very efficient as it only transfers the modified portions of file across network. Therefore code and data may be copied to the remote machine simply by `rsync`ing the proper directories. `rdist` performs a task similar to the one performed by `rsync`, but it is specialized to support file transfer towards a number of different remote machines. It uses a `distfile` configuration file that may include different "targets" (operations) each moving different local directories to (possibly different) sets of remote machines. In order to be able to use anyone of these commands, the user must have a valid account on the remote machines used.

Lifecycle control. To start processes on a remote machine the `ssh` command may be used. This command takes a (remotely valid[39]) command and executed it on a remote machine. The syntax used to specify the remote machine consists in a user name followed by the "@" and by the qualified name of the remote machine. In order to be able to use `ssh`, the user must have a valid account on the remote machines used. As for `scp` special care must be taken to avoid having the remote system asking the password before executing the requested command. To stop remote processes, the `ssh` command may be used to run proper `kill` commands. `ssh` may also be used to run monitoring commands on the remote systems (e.g. a `ps` listing the status of the user processes, or a `uptime` to monitor the machine load).

2.6.3 MPI

MPI (the Message Passing Interface) "is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users."[40]. The library basically provides the user with an API that includes library calls for different kind of communication and synchronization mechanisms. These calls are available in C, C++ and FORTRAN bindings, although a number of other programming frameworks and

[37] if a process has to wait an event signaled by another process, it may read from a message queue/pipe/socket just one byte, while the signaling process may write that byte at the right "synchronization" point

[38] to avoid the request of the remote login password, you should insert your public key (usually in the file `$HOME/.ssh/id_rsa.pub` in the `$HOME/.ssh/authorized_keys` file of the remote machine. The public rsa password must be generated with an empty passphrase.

[39] that is a valid command on the user account on the remote system

[40] see `http://www.mcs.anl.gov/research/projects/mpi/`

languages provide proper wrappings to call MPI services. In the following, we refer to the C binding of these library.

2.6.3.1 Concurrent activities set up The computational model of MPI is SPMD, Single Program Multiple Data. This means a unique program is compiled and copies of the program are run in parallel on the target architecture. Each copy of the program basically represents a "concurrent activity", according to our terminology.

MPI programs are plain C `main` functions. The programs are compiled through and MPI compiler (usually named `mpicc`) and executed through the MPI launcher (usually named `mpirun`). The launcher accepts parameters (`-np NPE`) that specify the number of "concurrent entities" to be used to run the program. These concurrent entities may be configured to be cores of the same machine, processing elements in a cluster or in the network of computers.

Within the C code of the MPI program, the MPI programmer may read the total amount of concurrent activities available (using a `MPI_comm_size` call). Then, the programmer may specify the activities of each one of the concurrent activities by using the value returned by the `MPI_Comm_rank` call, that represents the index of the current activity.

The typical structure of an MPI program is therefore the one in the following listing:

```
1  #include <mpich.h>
2
3  int main(int argc, char ** argv)
4  {
5    int size, rank;
6
7    MPI_Init(&argc,&argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   // I'm concurrent activity "rank"
11   // of a total "size" activities
12   switch(rank) {
13   case 0: { // concurrent activity 0 code
14     break;
15   }
16   case 1: { // concurrent activity 1 code
17     break;
18   }
19   default: { // other concurrent activities code
20   }
21   }
22   MPI_Finalize();
23   return;
24 }
```

Includes/mpi.c

Line 7 initializes the library. Then we get the total number of concurrent activities (`size`, with the call on line 8) and our identifier within the set of concurrent activities (`rank` with the call on line 9). The switch on lines 20–21 determines the program eventually run by the concurrent activity. The call on line 22 eventually terminates MPI activities right before the program end.

If we compile the program and we run it with the flag `-np 4`, we will have `size=4`, one concurrent activity executing the first `case` code, one executing the second `case` code and two further activities running the `default` code.

2.6.3.2 Communications MPI provides a plethora of communication mechanisms. We are not interested listing all of them here. We just introduce the classes of communications

available and the we show a sample communication code, to have a rough idea of the programming effort required. Each communication mechanism is implemented through one or more calls, to be executed by the concurrent activities participating to the communication. The communication mechanism provide by MPI include synchronous and asynchronous communications (with variants) and collective communications (one too many and many to one). All MPI communications take place within a "communicator" context, that is a collection of concurrent activities identified by their integer rank. Initially, all the concurrent activities are grouped in the `MPI_COMM_WORLD` communicator, then programmers may divide the communicator in sub-communicators, if the case.

Point to point communications are exemplified by a send and a receive call:

- `MPI_Send(buf,count,datatype,dest,tag,comm);`
  sends the contents of the buffer `buf` with `count` items of type `datatype` to the destination concurrent activity identified by the rank `tag` within the communicator `comm`. The message is tagged with the tag `tag`.

- `MPI_Recv(buf,count,datatype,src,tag,comm,status);`
  receives in the buffer `buf` up to `count` items of type `datatype` from the concurrent activity identified by the tag `src` within the communicator `comm`. A `tag` may be used to specify that only messages tagged in a given way should be considered. The `status` is a return parameter used to access the final status of the communication.

Different types of send exist in MPI, depending on the synchronization mechanisms used with the receiver: the standard one (the one implemented with the two primitives above) is asynchronous, with buffering defined at the implementation level, the "buffered" one is again asynchronous and allows buffering to be explicitly declared. Then there are two synchronous sends: the first one (the "synchronous" one in MPI terminology) may be started before the receive has actually been started, but completes with the receive. The second one (the "ready" mode one) may start executing only if the corresponding received is already started. If the receive has not yet been started and error code is returned.

Asymmetric receives may be used by specifying an `MPI_ANY_SOURCE` and/or an `MPI_ANY_TAG` parameters.

Collective operations include broadcast, gather, scatter, reduce, reduce+scatter, scan and alltoall communications. All of this primitives assume to have a "root" concurrent activity in the communicator. Then the operation is implemented between the root activity and the activities in the communicator. As an example, in a `MPI_Broadcast` primitive data from the root activity is broadcasted to all the activities in the communicator (including the root one).

With version 2 of the MPI standard, *one side communications* have been introduced. These are kind of remote memory access mechanisms. A concurrent activity may read or write variables of another concurrent activity, basically. The "one side" name cames from the fact the communication only requires calling an MPI routine on the "active" side (i.e. the concurrent activity reading or writing) with no need for a dual, "companion" call on the passive activity (the one being read or written).

2.6.3.3 Deployment Once MPI has been installed, deploying and running an MPI program on a set of cores/machines is a matter of issuing from the command line a command such as

```
mpirun -np XXX programname params ....
```

Depending on the implementation, different configuration files are needed to specify where the resources needed to run the program `XXX` concurrent activities. Usually, "names" of

resources[41] may be listed in a text or XML file, with the associated "parallelism degree" (e.g. the number of cores). The MPI launcher consults these files to set up the number of concurrent activities required through the command line parameter. If more concurrent activities than available processing elements (cores) are required, some round robin assignment is used.

2.7 IMPLEMENTING PARALLEL PROGRAMS THE "CLASSICAL" WAY

We discuss in this Section a trivial implementation of a simple parallel application on a POSIX/TCP system. The intent is to point out all the peculiarities and difficulties related to the usage of low level mechanisms to implement parallel applications.

2.7.1 The application

We want to implement a very simple application that squares a set of integers in parallel. Therefore, in our application:

- the input is constituted by a given number of integers
- each one of the integers has to be squared ("squaring" is our "business logic code")
- the output (on screen) is the set of squared integers

For the sake of simplicity, we do not require an "ordered" input, therefore if we ask the computation of squares of $x_1, \ldots, x_m$ then allow to have $x_i * x_i$ appearing on the screen before $x_{i-k} * x_{i-k}$.

2.7.2 The concurrent activity graph

Given the set of input data

$$x_1, \ldots, x_m$$

the concurrent activities set is given by

$$\{\forall i \ : x_i^2\}$$

as all the tasks (i.e. the tasks computing each x_i^2) are independent: no data dependencies are present.

2.7.3 Coordination

Being the concurrent activities independent we need no coordination among concurrent activities. Of course, we need to have the input data to start and we should stop only after computing all the concurrent activities.

2.7.4 Implementation

We assume to target a POSIX/TCP workstation network. Therefore we will use processes to implement concurrent activities and sockets to implement communications and synchronization. We also assume–in principle–to have concurrent activities partitioned across the available processing elements.

[41] e.g. IP addresses or fully qualified names

$$G=(N,A)$$

$$N = \{em, coll, w1, \ldots , wn\}$$
$$A = \{(em,w1), \ldots , (em,wn),(w1,coll), \ldots , (wn,coll)\}$$

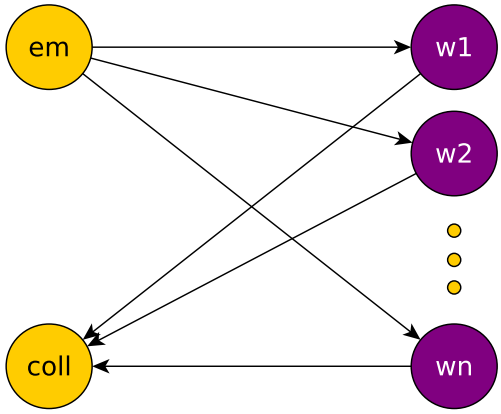


Figure 2.2 Implementation graph of the sample application

A possible implementation is the following one:

- a processing element (the *emitter*) hosts all the input data and delivers input data items (one at a time) to one of the processing elements belonging to a set of *worker* processes
- a set of identical *worker* processes will wait for a task to be computed from the *emitter* process, compute the task and deliver the result to a *collector* process
- a *collector* process will collect results from workers and print them to screen.

The logical parallel activities graph is the one depicted in Fig. 2.2.

2.7.5 Code relative to the implementation handling termination but not handling worker faults

We decide to implement communications as follows:

- the emitter process publishes a server socket. It accepts connections from workers. When a connection is accepted the emitter sends to the worker a task to be computed and closes the connection. The emitter process is single threaded, so one connection at a time is handled. This is reasonable due to the negligible amount of work needed to handle the connection.
- the collector process publishes a server socket. It accepts connections from the workers. When a connection is accepted the collector reads from the socket a result and prints it to screen. Then the connection is closed. Again, the collector process is single threaded.
- the worker process executes a loop. In the loop body:

1. it opens a socket and connects to the emitter server socket, sending its process identifier.
2. it reads a tasks from the socket connected to the emitter, closes the socket and computes the result relative to the task
3. it opens a socket and connects to the collector server socket, sends the result to the collector and closes the socket.

The usage of the TCP sockets in this "connectionless" mode is not typical nor encouraged. We adopted it here as it allows to clearly individuate the different phases in the parallel program: communication and computation phases, mainly.

The following listing details the code of the emitter process.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netinet/in.h>
8  #include <netdb.h>
9  #include <arpa/inet.h>
10
11 //  on demand scheduling of tasks to workers
12 //  receive request from worker i -> send task to compute to worker i
13 //  when no more tasks are available, send an EOS termination
14 //
15 //  usage is:
16 //     a.out portno collectorhost collectorport
17 //
18
19 #define MAXHOSTNAME 80
20 #define MAXWORKERS  16
21
22 #define TRUE (1==1)
23 #define FALSE (1==9)
24
25 int main(int argc, char * argv[]) {
26   int s, cs, si, retcode, k, pid, i;
27   unsigned int salen;
28   struct sockaddr_in sa,sai;
29         char hostname[MAXHOSTNAME];
30         struct sockaddr_in w_addr[MAXWORKERS]; // used to keep track
              of workers
31         int w_pid[MAXWORKERS];  // store address and pid of workers
32         int nw = 0;               // and count how many different
              workers
33
34   int task = 0;      // tasks are positive integeres, in this case
35   int tasklimit = 5;  // we'll send tasklimit tasks before stopping
36   int eos = -1;   // special task to denote End Of Stream
37
38         int loop = TRUE;         // to determine loop end
39
40   // code needed to set up the communication infrastructure
41         printf("Declaring socket\n");
42   si = socket(AF_INET,SOCK_STREAM,0);    // socket for inputs
43   if(si == -1) {perror("opening socket for input"); return -1;}
44   sai.sin_family = AF_INET;
```

```
45  sai.sin_port = htons(atoi(argv[1]));
46        gethostname(hostname,MAXHOSTNAME);
47  memcpy(&sai.sin_addr, (gethostbyname(hostname)->h_addr),
48    sizeof(sa.sin_addr));
49        printf("Binding to %s\n",inet_ntoa(sai.sin_addr));
50  retcode = bind(si,(struct sockaddr *) & sai, sizeof(sai));
51  if(retcode == -1) {
52        perror("while calling bind"); return -1; }
53        printf("Listening socket\n");
54  retcode = listen(si,1);
55  if(retcode == -1) {
56    perror("while calling listen"); return -1; }
57
58  while(loop) {
59    int found = FALSE;
60    salen = sizeof(sa);
61    printf("Accepting connections .... \n");
62    s = accept(si,(struct sockaddr *)&sa,&salen);  // accept a
        connection
63    if(s == 1) {
64      perror("while calling an accept"); return -1; }
65    printf("Accepted connection from %s porta %d (nw=%d i=%d)\n",
66      inet_ntoa(sa.sin_addr), sa.sin_port, nw, i);
67    retcode = read(s,&pid,sizeof(int));  // read request from worker
68          if(retcode == -1) {
69      perror("while reading task from worker"); return -1; }
70    // store worker data if not already present ...
71    for(k=0; k<nw; k++) {
72      if(memcmp((void*)&w_addr[k],(void *)&sa.sin_addr,sizeof(struct
          sockaddr_in))==0
73     && w_pid[k]==pid) {
74    printf("Old worker asking task\n");
75    found = TRUE;
76    break;
77      }
78          }
79            if(!found) {
80    memcpy((void *) &w_addr[nw], (void *) &sa.sin_addr, sizeof(struct
        sockaddr_in));;
81    w_pid[nw] = pid;
82    nw++;
83    printf("New worker asking task from %s pid %d port
        %d\n",inet_ntoa(sa.sin_addr), pid, sa.sin_port);
84      }
85
86    printf(" >> %d ",pid); fflush(stdout);
87    if(task < tasklimit) {  // send a task to the requesting worker
88      write(s,&task,sizeof(task));
89      printf("sent task %d to worker %s
          %d\n",task,inet_ntoa(sa.sin_addr), pid);
90      task++;   // next task to be sent
91    } else { // if no more tasks, then send an EOS
92      write(s,&eos,sizeof(task));
93      printf("sent EOS %d to worker %s
          %d\n",task,inet_ntoa(sa.sin_addr), pid);
94      // check worker list and decrease worker count
95      for(i=0; i<nw; i++) {
96        if(memcmp((void *) & w_addr[i], (void *) &sa.sin_addr,
            sizeof(struct sockaddr_in))==0
```

```
97              && w_pid[i] == pid) {
98          w_pid[i]=0; // nullify entry
99          nw--;
100         if(nw==0) // no more workers -> stop looping
101           loop = (1==0);
102           }
103         }
104             }
105       close(s);
106     }
107     // after sending EOS to all the workers, we can send an EOS also to
          the collector,
108     // we assume workers, before asking a task (and getting eos) already
          sent the last
109     // result computed to the collector
110           printf("Closing operations\nSending EOS to collector ... \n");
111           cs = socket(AF_INET,SOCK_STREAM,0);    // socket to access
                emitter
112           if(cs == -1) {
113                   perror("while creating the socket to emitter");
114                   return (-1); }
115           sa.sin_family = AF_INET;
116           sa.sin_port = htons(atoi(argv[3]));
117           memcpy(&sa.sin_addr, (gethostbyname(argv[2])->h_addr),
118                   sizeof(sa.sin_addr));
119           retcode = connect(cs,(struct sockaddr *)&sa,sizeof(sa));
120           if(retcode == -1) {
121                   perror("while connecting to emitter"); return(-1);}
122
123           write(cs,&eos,sizeof(int)); // sends a request to the emitter
124     close(cs);
125     printf("Terminating\n");
126     return 0;
127 }
```

Includes/emitter.c

Lines 34–36 define the tasks we will compute and the value we will use to terminate the processes when no more tasks must be computed. Lines 41–56 declare and bind a server socket using as the port number a parameter taken from the command line. Emitter loop is started at line 58 . Loop termination is controlled by the boolean variable `loop` initialized to true and set to false when the last "termination" task (`EOS`) has been sent to the last worker. Line 62 accepts (blocking call) a connection. The number of active workers is updated in lines 71–84. For each one of the incoming connections we check the IP address of the client (the worker) along with process id sent through the socket. This identification method allows to put more workers on the same node. When a previously unseen worker appears, it inserted in the list of already seen workers and the number of workers (`nw`) is updated. If there are more tasks to be executed (lines 87–90) a task is sent to the requesting worker using the bidirectional socket resulting from accept, the socket is closed (line 105) and the loop is restarted. If there are no more tasks, a special EOS tasks is sent (lines 93–103) the socket is closed and the loop restarted. Actually, at each EOS the number of active workers is decremented. When it reaches zero, the emitter loop is terminated. At this point, lines 107–126, the emitter sends a termination message also to the collector process, thus terminating the whole parallel computations, and terminates itself.

The collector process is defined through the following code:

```
1 #include <stdio.h>
2 #include <string.h>
```

```
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <netdb.h>
9
10 // receives results from the workers and displays them on the console
11 // usage:
12 //    a.out portno
13 //          number of the port for the result socket
14
15 #define MAXHOSTNAME 80
16
17 int main(int argc, char * argv[]) {
18   int s,si, retcode, i;
19   unsigned int salen;
20   struct sockaddr_in sa,sai;
21   char hostname[MAXHOSTNAME];
22
23   si = socket(AF_INET,SOCK_STREAM,0);    // socket to receive the
       results
24   if(si == -1) {
25     perror("while opening socket"); return -1;}
26   sai.sin_family = AF_INET;
27   sai.sin_port = htons(atoi(argv[1]));
28     gethostname(hostname,MAXHOSTNAME);
29   memcpy(&sai.sin_addr, (gethostbyname(hostname)->h_addr),
30     sizeof(sa.sin_addr));
31   retcode = bind(si,(struct sockaddr *) & sai, sizeof(sai));
32   if(retcode == -1) {
33     perror("while binding socket to addr"); return -1; }
34   retcode = listen(si,1);
35   if(retcode == -1) {
36     perror("while calling listen"); return -1; }
37
38   while(1==1) {
39     salen = sizeof(sa);
40     printf("Accepting connections\n");
41     s = accept(si,(struct sockaddr *)&sa,&salen); // accept a
         connection
42     if(s == 1) {
43       perror("while accepting a connection"); return -1;
44     }
45     retcode = read(s,&i,sizeof(int));// read a res from one of the
         Worker
46           if(retcode == -1) {
47     perror("while reading a result"); return -1; }
48           if(i==(-1)) {
49             printf("EOS -> terminating\n");
50     break;
51     }
52     printf("Read result: %d ",i); fflush(stdout); // and print it on
         console
53     close(s);
54         }
55         close(si);
56   return 0;
57 }
```

Includes/collector.c

It is very similar to the emitter code, so we will not comment it further. Eventually, each worker process is defined through the following code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>

/*
 *    function to be computed on the single task
 *    this is actually the only "business logic" code
 */

int f(int task) {
  sleep(1);  // just to simulate some work ...
  return (task*task);
}

/*
 *    receives integer tasks and delivers integer results, up to the
     reception
 *    of an EOS (a -1)
 *    this version opens/closes the sockets at each task, which is not
 *    definitely efficient ...
 *
 *    usage:
 *       a.out emitter_address emitter_port collector_address
     collector_port
 */

int main(int argc, char * argv[]) {
  int s, retcode;
  struct sockaddr_in sa;
  int pid = 0;


  pid = getpid();     // get my POSIX process id
  printf("This is worker %d\n",pid);
  while(1==1) {
    int task, result;

    s = socket(AF_INET,SOCK_STREAM,0);    // socket to access emitter
    if(s == -1) {
      perror("while creating the socket to emitter");
      return (-1); }
    sa.sin_family = AF_INET;
    sa.sin_port = htons(atoi(argv[2]));
    memcpy(&sa.sin_addr, (gethostbyname(argv[1])->h_addr),
      sizeof(sa.sin_addr));
    retcode = connect(s,(struct sockaddr *)&sa,sizeof(sa));
    if(retcode == -1) {
      perror("while connecting to emitter"); return(-1);}
```

```
52
53      write(s,&pid,sizeof(pid)); // sends a request to the emitter
54      retcode = read(s,&task,sizeof(task)); // get a task
55      if(retcode != sizeof(task)) {
56        perror("while receiving a task"); return -1; }
57
58      // check for EOS
59      if(task < 0)  {
60        printf("Received EOS\n");
61        break;  // if EOS terminate loop iteratons
62      } else {
63        printf("Received task %d\n",task);
64        // otherwise process the incoming task
65        result = f(task);
66        printf("Computed %d -> %d\n",task,result);
67
68        // send result to the collector
69        s = socket(AF_INET,SOCK_STREAM,0); // create socket to collector
70        sa.sin_family = AF_INET;
71        sa.sin_port = htons(atoi(argv[4]));
72        memcpy(&sa.sin_addr, (gethostbyname(argv[3])->h_addr),
73          sizeof(sa.sin_addr));
74        retcode = connect(s,(struct sockaddr *)&sa,sizeof(sa));
75        if(retcode == -1) {
76          perror("connecting to the collector");
77          return(-1);}
78
79        // send the result and close connection
80        write(s,&result,sizeof(result));
81        close(s);
82
83        // then cycle again
84      }
85    }
86    close(s);
87    return 0;
88  }
```

Includes/worker.c

An infinite loop is entered at line 38. The loop will terminate only when an EOS termination task will be received, via the break at line 61. In the loop body, first a socket is connected to the emitter server socket (lines 41–51), then the process id of the worker is sent as a "task request" to the emitter process and a task is read from the emitter server socket (lines 53–55). If the received task is an EOS, the loop is immediately terminated (line 61). Otherwise, a result is computed out of the received task (line 65) and sent to collector processor (line 80) after connecting a client socket to the collector server socket (lines 67–77).

This code may be compiled using standard tools (see the following makefile).

```
1  CC = gcc
2  CFLAGS =
3  LDFLAGS = -lpthread
4  TARGETS = emitter collector worker
5
6  all:  $(TARGETS)
7
8  clean:
9    rm -f $(TARGETS)
```

```
10
11 worker:    worker.c
12   $(CC) -o worker $(CFLAGS) worker.c $(LDFLAGS)
13
14 collector:  collector.c
15   $(CC) -o collector $(CFLAGS) collector.c $(LDFLAGS)
16
17 emitter:  emitter.c
18   $(CC) -o emitter $(CFLAGS) emitter.c $(LDFLAGS)
19
20 dist:    $(SOURCES)
21   rsync -avz ../FarmSocket ottavinareale:
22   rsync -avz ../FarmSocket backus:Downloads/
23   rsync -avz ../FarmSocket pianosau:
```

Includes/makefile

This is not the only code needed to run our program, actually. We need some scripting code to deploy the code on the target architecture nodes, to run it and to clean up possible processes left after an unexpected termination of the parallel computation.

In order to deploy the code on the remote nodes, we may use the bash script below.

```
1 #!/bin/bash
2 for m in $@
3 do
4   rsync -avz /home/marcod/Documents/Codice/FarmSocket  $m:
5 done
```

Includes/deploy.bash

In this case we perform an `rsync` on all those machines named in the script command line arguments.

We eventually need also some code to run the remote emitter, collector and worker processes. The following listing presents a Perl script compiling and running the processes on the remote machines. Emitter is run on the first machine passed in the argument list, with the port number taken from the argument list as well. Collector is run on the second machine passed in the argument list, with the port number taken again from the argument list. Eventually, all the other machine names are used to run worker processes. The run script also stores the names of the machines where a process has been started in the file `machines.used` in the current working directory.

```
1 #!/usr/bin/perl
2
3 ### launcher emitter_machine emitter_port collector_machine
      collector_port w_machine*
4
5 $emachine = shift @ARGV;
6 $eport    = shift @ARGV;
7 $cmachine = shift @ARGV;
8 $cport    = shift @ARGV;
9
10 open FD, ">machines.used" or die "Cannot open log file";
11 system("ssh $emachine \"\( cd FarmSocket\; make emitter\; ./emitter
      $eport \)\" & ");
12 print FD "$emachine\n";
13 sleep(5);
14 system("ssh $cmachine \"\( cd FarmSocket\; make collector\;
      ./collector $cport \)\" & ");
15 print FD "$cmachine\n";
16 sleep(5);
```

```
17
18 foreach $w (@ARGV) {
19   system("ssh $w \"\( cd FarmSocket\; make worker\; ./worker $emachine
         $eport $cmachine $cport \)\" & ");
20   print FD "$w\n";
21
22 }
23 close FD;
24 exit;
```

Includes/launcher.pl

The `machines.used` file is used in the "terminator" script killing all the processes possibly left running on the machines used to run our parallel application (see code below).

```
1 #!/usr/bin/perl
2 open FD, "<machines.used" or die "Cannot open log file for reading";
3 while(<FD>) {
4   $m = $_ ;
5   chop $m;
6   system("ssh $m kill -9 -1");
7 }
8 close FD;
9 system("rm -f machines.used");
10 exit;
```

Includes/terminator.pl

Functional and non functional code. In this implementation, the only functional code present in the C programs listed above is

- the code generating the tasks in the emitter code (lines 34, 35, 87 and 90)
- the code reading and writing tasks and results on sockets; in particular the `sizeofs` in read and writes depend on task and results data types.
- the code computing the result in the worker (lines 15–18 and 65).

All the rest of the code is non functional code, needed to set up and run the concurrent activities graph. As you can see, functional and non functional code are completely mixed up. As a consequence, debugging, fine tuning and maintenance of the application may be difficult.

2.7.6 Adding fault tolerance (partial)

In order to add a very simple for of fault tolerance, that is the possibility to detect a fault in a worker process and to reschedule the task last scheduled to the faulty worker to another worker, we have to modify the code presented and commented in the previous section quite a lot.

First of all, we need a way to `detect` worker failures. We can use standard mechanisms, such as heartbeats[42], but due to the fact that a TCP connections reveals the closing of one of the two connections ends to the other hand attempting to read/write to the connection socket, we may use a simpler mechanisms:

- we modify the worker process code in such a way the worker permanently connects to the emitter process,

[42] a technique such that active entities send periodical messages to the controller entity. In case of fault, the controller perceives a sudden stop in the messages and may detect the entity failure.

- then the emitter process gets a particular retcode when reading a request from the worker connected socket in case of worker termination (the read returns 0 as the number of bytes actually read in the buffer).
- once the emitter has detected the connection error, it assumes that the task previously scheduled to faulty worker may have been aborted before actually sending a result to the collector process. Therefore it has to be somehow rescheduled on a different worker. However, we must take care of the possibility that the worker died after sending the result to the collector. In this case, if the task is rescheduled for execution the collector will see two results for the same task.
- due to the necessity to keep a connection open for each one of the wokers, however, the emitter has to be implemented in a multi threaded way. A thread is started at each connection from one of the workers, gets the connected socket as a parameter and starts a loop reading requests and sending tasks. In turn, this means that more threads will access our dummy "task list" (the variables `task` and `tasklimit`), which must be protected with a `pthread_mutex` to ensure concurrent access correctness.

We need therefore to define more complex task, result and service message (request message) data types, as we will need to associate task ids to each one of the tasks, and copy the id to results, in such a way collector may signal the emitter whose tasks have been completed. This is needed to avoid duplication of computations relative to tasks sent to a faulty worker that indeed succeed sending a result for that task to the collector before dying. These data structures are defined as follows (file task.h):

```
1  //
2  // data structure taking care of the tasks to compute. We need to push
3  // back tasks that have not presumably been computed due to failures
4  //
5
6  #define TAG_EOS (-1)
7  #define TAG_TASK_REQUEST (-2)
8  #define TAG_COMPLETED (-3)
9
10 typedef struct _service {
11   int v;
12   int tag;
13 } SERVICE;
14
15 typedef struct _task {
16   int v;     // the "value" of the task
17   int tag;     // the tag (to recognize task/result and to reorder
18   struct _task * next;  // it's a linked list
19 } TASK;
20
21 typedef struct _result {
22   int v;       // the "value" of the result
23   int tag;       // the tag (to recognize task/result + reorder)
24   struct _result * next;  // it's a linked list
25 } RESULT;
```

Includes/task.h

The next listing shows a possible code of a worker process adapted to the new concurrent process schema.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
```

```
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <netdb.h>
8  #include <unistd.h>
9
10 #include "task.h"
11
12 /*
13  *   function to be computed on the single task
14  *   this is actually the only "business logic" code
15  */
16
17 int f(int task) {
18   sleep(1);  // just to simulate some work ...
19   return (task*task);
20 }
21
22 /*
23  *   receives integer tasks and delivers integer results, up to the
        reception
24  *   of an EOS (a -1)
25  *   this version opens/closes the sockets at each task, which is not
26  *   definitely efficient ...
27  *
28  *   usage:
29  *      a.out emitter_address emitter_port collector_address
        collector_port
30  */
31
32 int main(int argc, char * argv[]) {
33   int s, cs, retcode;     // task socket, result socket, error ret
34   struct sockaddr_in sa;
35   int pid = 0;
36
37
38   pid = getpid();     // get my POSIX process id
39   printf("This is worker %d\n",pid);
40     // connect permanently to the emitter process
41   s = socket(AF_INET,SOCK_STREAM,0);    // socket to access emitter
42   if(s == -1) {
43     perror("while creating the socket to emitter");
44     return (-1); }
45     printf("Opened E socket %d\n",s);
46   sa.sin_family = AF_INET;
47   sa.sin_port = htons(atoi(argv[2]));
48   memcpy(&sa.sin_addr, (gethostbyname(argv[1])->h_addr),
49     sizeof(sa.sin_addr));
50   retcode = connect(s,(struct sockaddr *)&sa,sizeof(sa));
51   if(retcode == -1) {
52     perror("while connecting to emitter"); return(-1);}
53
54   while(1==1) {
55     TASK task;
56     RESULT result;
57     SERVICE request;
58     // sends a request to the emitter
59     request.tag = TAG_TASK_REQUEST;
60                   request.v = pid;
```

```
61      retcode = write(s,&request,sizeof(SERVICE));
62                  if(retcode == -1) { perror("sending a request");
                        return -1; }
63      retcode = read(s,&task,sizeof(TASK)); // get a task
64      if(retcode != sizeof(TASK)) { // this could be corrected in case
65              // of large tasks (while(len!=sizeof) read)
66        perror("while receiving a task"); return -1; }
67
68      // check for EOS
69      if(task.tag < 0)  {
70        printf("Received EOS\n");
71        break;  // if EOS terminate loop iteratons
72      } else {
73        printf("Received task %d\n",task.tag);
74        // otherwise process the incoming task
75        result.v = f(task.v);
76        result.tag = task.tag;
77        result.next = NULL;
78        printf("Computed %d -> %d\n",task.v,result.v);
79
80        // send result to the collector
81        cs = socket(AF_INET,SOCK_STREAM,0); // create socket to collector
82        sa.sin_family = AF_INET;
83        sa.sin_port = htons(atoi(argv[4]));
84        memcpy(&sa.sin_addr, (gethostbyname(argv[3])->h_addr),
85          sizeof(sa.sin_addr));
86        retcode = connect(cs,(struct sockaddr *)&sa,sizeof(sa));
87        if(retcode == -1) {
88          perror("connecting to the collector");
89          return(-1);}
90
91        // send the result and close connection
92        write(cs,&result,sizeof(RESULT));
93        close(cs);
94        printf("Sent to C socket, E sock %d\n",s);
95
96        // then cycle again
97      }
98    }
99    close(s);
100   return 0;
101 }
```

Includes/workerFT.c

Eventually, the emitter process program may be defined as follows:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netinet/in.h>
8  #include <netdb.h>
9  #include <arpa/inet.h>
10 #include <pthread.h>
11
12 //  on demand scheduling of tasks to workers
13 //  receive request from worker i -> send task to compute to worker i
```

```
14 //  when no more tasks are available, send an EOS termination
15 //
16 //  usage is:
17 //     a.out portno
18 //           number of the port used to get worker task requests
19 //
20 //  this is the version checking worker failure (emitter side)
21 //  and just signaling a worker failure
22 //
23
24 // the structure hosting the tasks
25 #include "task.h"
26 // this hosts tasks initially scheduled for execution
27 TASK * task_list = NULL;
28 // this hosts tasks to be rescheduled (possibly)
29 TASK * task_limbo = NULL;
30
31 // to manage task list
32 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
33 pthread_mutex_t mutex_limbo = PTHREAD_MUTEX_INITIALIZER;
34 // moved outside main: global
35 int task = 0;      // tasks are positive integeres, in this case
36 int tasklimit = 10; // we'll send tasklimit tasks before stopping
37 TASK eos;    // special task to denote End Of Stream
38
39
40 void * worker_handler(void * vs) {
41   int s = *((int *) vs);
42   int retcode,i;
43   TASK * task_to_send;
44   int * taskNo = (int *) calloc(sizeof(int),1);    // host tasks
       computed
45   // start reading tasks
46   printf("worker_handler %ld started\n",pthread_self());
47   while(1==1) {
48     SERVICE request;
49     printf("Waiting for a request \n");
50     // read request: task request from woker or task completion msg
         from coll
51     retcode = read(s,&request,sizeof(SERVICE));
52     if(retcode != sizeof(SERVICE)) {  // should be != 0 (read can be
         split)
53       perror("while reading task from worker"); fflush(stderr); }
54     if(retcode == 0) { // connection broken
55       printf("NEED TO RUN A SUBSTITUTE WORKER AFTER THIS DIED !!! \n");
56       // you should know which is the last task sent, in such a way it
           can
57       // be rescheduled onto a different worker
58       printf("TASK PRESUMABLY NOT COMPLETED is TASK
             <%d>\n",task_to_send->tag);
59       pthread_mutex_lock(&mutex_limbo); // add task to limbo for
           reschedule
60   {
61     TASK * t = (TASK *) calloc(1,sizeof(TASK));
62           t->v = task_to_send->v;
63           t->tag = task_to_send->tag;
64           t->next = task_limbo;
65           task_limbo = t;
66     printf("Task <%d,%d> -> LIMBO\n",t->v, t->tag);
```

```
67          }
68        pthread_mutex_unlock(&mutex_limbo);
69        return (NULL);
70      }
71      if(request.tag == TAG_COMPLETED) { // this is from collector
72    // cancel request.v tag from limbo
73    TASK * p, *pp ;
74          p = pp = task_limbo;
75    pthread_mutex_lock(&mutex_limbo);
76          while(p!=NULL) {
77            if(p->tag == request.v) { // completed: to be removed
78        if(p == pp) { // first element to be removed
79                p = p->next;
80              } else {
81          pp->next = p->next; // TODO: free unused one !
82              }
83            }
84          }
85    printf("Computed task tag=%d removed from LIMBO\n",request.tag);
86    pthread_mutex_unlock(&mutex_limbo);
87    return(NULL); // eventually terminate the thread
88      } // else: emit a task to the worker and cycle (this is a worker
          manager)
89      printf("Got request from %d ",request.v); fflush(stdout);
90      pthread_mutex_lock(&mutex);
91      if(task_list != NULL) {  // send a task to the requesting worker
92      TASK * t = task_list;
93      task_list = t->next;
94            task_to_send = t;
95      taskNo++;
96          pthread_mutex_unlock(&mutex);
97      } else { // if no more tasks, then check limbo or send EOS
98          pthread_mutex_unlock(&mutex);
99          pthread_mutex_lock(&mutex_limbo);
100           if(task_limbo!= NULL) {
101       task_to_send = task_limbo ;
102             task_limbo = task_limbo->next;
103       printf("Task pool is empty but sending task from Limbo\n");
104     } else {
105             task_to_send = &eos;
106       }
107         pthread_mutex_lock(&mutex_limbo);
108     }
109     write(s,&task_to_send,sizeof(TASK));  // send either task or eos
110     if(task_to_send != &eos) {
111     printf("sent task %d to worker %d\n",task,i);
112     } else {
113     printf("Send EOS to worker %d\n",i);
114       close(s);
115     return((void *)taskNo);
116     }
117   }
118 }
119
120 #define MAXHOSTNAME 80
121 #define MAXTHREADS  16
122
123 int main(int argc, char * argv[]) {
124   int s,si, retcode, i;
```

```
125   unsigned int salen;
126   struct sockaddr_in sa,sai;
127         char hostname[MAXHOSTNAME];
128         pthread_t tids[MAXTHREADS];  // thread handlers
129         int tNo = 0;          // thread to allocate
130
131
132   // set up task structure
133   for(i=0; i<tasklimit; i++) {
134     TASK * t = (TASK *) calloc(1,sizeof(TASK));
135           t->next = task_list;
136     t->v = i;
137       t->tag = i;
138           task_list = t;
139   }
140   // set up eos
141   eos.v = -1;
142   eos.tag = -2;
143         eos.next = NULL;
144
145   // code needed to set up the communication infrastructure
146         printf("Declaring socket\n");
147   si = socket(AF_INET,SOCK_STREAM,0);   // socket for inputs
148   if(si == -1) {perror("opening socket for input"); return -1;}
149   sai.sin_family = AF_INET;
150   sai.sin_port = htons(atoi(argv[1]));
151         gethostname(hostname,MAXHOSTNAME);
152   memcpy(&sai.sin_addr, (gethostbyname(hostname)->h_addr),
153     sizeof(sa.sin_addr));
154         printf("Binding to %s\n",inet_ntoa(sai.sin_addr));
155   retcode = bind(si,(struct sockaddr *) & sai, sizeof(sai));
156   if(retcode == -1) {
157         perror("while calling bind"); return -1; }
158         printf("Listening socket\n");
159   retcode = listen(si,1);
160   if(retcode == -1) {
161     perror("while calling listen"); return -1; }
162
163   while(1==1) {
164           int * s = (int *) calloc(sizeof(int),1);
165
166     salen = sizeof(sa);
167     printf("Accepting connections .... \n");
168     *s = accept(si,(struct sockaddr *)&sa,&salen);  // accept a
          connection
169     if(*s == 1) {
170       perror("while calling an accept"); return -1; }
171     // now fork a thread to permanently handle the connection
172           pthread_create(&tids[tNo],NULL,worker_handler,(void *) s);
173     printf("Created worker_handle No %d\n",tNo);
174     tNo++;
175   }
176         printf("Closing operations\n");
177   return 0;
178 }
```

Includes/emitterFT.c

The `task_limbo` structure defined at line 29 is used to keep all those tasks that were sent to a worker that failed after receiving the task. When the emitter finishes scheduling "normal" tasks, he will attempt to reschedule the ones in the "limbo" whose tags have not been received from the collector as tags of "successfully computed" tasks.

Eventually, the code of the collector process is the following:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netinet/in.h>
8  #include <netdb.h>
9
10 #include "task.h"
11
12 // receives results from the workers and displays them on the console
13 // usage:
14 //    a.out portno emitterhost emitterport
15 //          number of the port for the result socket
16 //      then address of emitter (to send feedbacks on task completion)
17
18 #define MAXHOSTNAME 80
19
20 int main(int argc, char * argv[]) {
21   int s, ei, si, retcode, i;
22   RESULT result;
23   unsigned int salen;
24   struct sockaddr_in sa,sai,eai;
25   char hostname[MAXHOSTNAME];
26         char * emitterhost = argv[2];
27         int emitter_port = atoi(argv[3]);
28
29   si = socket(AF_INET,SOCK_STREAM,0);    // socket to receive the
       results
30   if(si == -1) {
31     perror("while opening socket"); return -1;}
32   sai.sin_family = AF_INET;
33   sai.sin_port = htons(atoi(argv[1]));
34     gethostname(hostname,MAXHOSTNAME);
35   memcpy(&sai.sin_addr, (gethostbyname(hostname)->h_addr),
36     sizeof(sai.sin_addr));
37   retcode = bind(si,(struct sockaddr *) & sai, sizeof(sai));
38   if(retcode == -1) {
39     perror("while binding socket to addr"); return -1; }
40   retcode = listen(si,1);
41   if(retcode == -1) {
42     perror("while calling listen"); return -1; }
43
44   while(1==1) {
45     SERVICE feedback;
46     salen = sizeof(sa);
47     printf("Accepting connections\n");
48     s = accept(si,(struct sockaddr *)&sa,&salen); // accept a
         connection
49     if(s == 1) {
50       perror("while accepting a connection"); return -1;
```

```
51      }
52      // read a res from one of the Worker
53      retcode = read(s,&result,sizeof(RESULT));
54            if(retcode != sizeof(RESULT)) {  // AGAIN, should be
                corrected
55      perror("while reading a result"); return -1; }
56            if(result.tag < 0) {
57              printf("EOS -> terminating\n");
58      break;
59      }
60      printf("Read result: <%d,%d> ",result.v,result.tag);
61      fflush(stdout); // and print it on console
62      close(s);
63            // now send a feedback to emitter process with the tag of the
64      // received task: in case it was selected for re-scheduling
65      // it will be removed, otherwise it will be rescheduled after
66      // completing "normal" tasks
67      ei = socket(AF_INET,SOCK_STREAM,0);    // socket to receive the
            results
68      if(ei == -1) {
69        perror("while opening socket"); return -1;}
70      eai.sin_family = AF_INET;
71      eai.sin_port = htons(emitter_port);
72      memcpy(&eai.sin_addr, (gethostbyname(emitterhost)->h_addr),
73        sizeof(eai.sin_addr));
74      retcode = connect(ei,(struct sockaddr *)&eai,sizeof(eai));
75        if(retcode == -1) {
76      perror("while connecting to emitter"); return(-1);}
77            // send feedback on computed task
78      feedback.tag = TAG_COMPLETED;
79            feedback.v   = result.tag;
80      retcode = write(ei,&feedback,sizeof(SERVICE));
81            if(retcode != sizeof(SERVICE)) {
82      perror("while writing feedback to emitter"); return (-1); }
83      close(ei); // one shot: TODO use permanent connection on dedicated
            ss
84          }
85          close(si);
86    return 0;
87  }
```

Includes/collectorFT.c

We leave the reader the possibility to evaluate the amount of code needed to introduce such a limited form of fault tolerance in our trivial parallel application. The reader may easily imagine how the code design, coding and code debug and tuning activities could be in case a real application is taken into account and complete fault tolerance is to be implemented.

CHAPTER 3

ALGORITHMIC SKELETONS

Researchers studying how to implement efficient parallel applications on parallel target architectures observed that the *patterns* used to exploit parallelism in *efficient* parallel applications are not a large number.

> *They observed that there are just a small number of common patterns that model common parallelism exploitation forms and that are used and re-used in different places, possibly nested to model the more complex parallelism exploitation strategies.*

These researchers also observed further and–possibly–more important facts:

- The correct and efficient programming–that is the design, coding, debugging and fine tuning–of these parallel patterns constitutes the major programming effort when implementing efficient parallel applications.
- The knowledge needed to implement efficient parallel patterns is completely different from the knowledge needed to program the functional logic of the application[43].
- Applications clearly separating the code used to implement parallel patterns from the application functional code were much more easy to develop, debug and tune, with respect to the applications were parallelism implementation code and functional code was completely intermingled.

As a consequence, these researchers started asking themselves if some benefit could be achieved from the design and development of parallel patterns *per se*, that is parallelism exploitation patterns studied without taking into account the final application functional code. In turn, this led–in different times and within different research communities–to the development of two distinct research branches:

[43] that part of the application code actually computing the result of the application

1. In late '80s, Cole introduced the *algorithmic skeleton* concept and a community of researchers started working and improving the algorithmic skeleton concept. This research community was mainly build by researchers coming from High Performance Computing community. Their major concern was the development of programming frameworks suitable to support efficient parallel programming, in particular with respect to the performance achieved in parallel programs. The community evolved and the concept of algorithmic skeleton evolved as well. Nowadays there are several research groups building on this concept around the world. Algorithmic skeletons are the main subject of this book and they will be thoroughly discussed from this Chapter on.

2. In the '00s, software engineering researchers evolved the design pattern concept into *parallel design pattern*s. The parallel design patterns has been and are currently investigated by a community which is fairly disjoint from the one investigating algorithmic skeletons. This notwithstanding the research on parallel design patterns produced (and it is currently producing) results that are similar to those produced in the algorithmic skeleton framework. Sometimes the results only differ in the "jargon" used to illustrate them. In more interesting cases they are similar but the different–more software engineering oriented–perspective used in investigating the results may provide useful hints to algorithmic skeleton designers as well. Despite the fact this book aims at providing an "algorithmic skeleton perspective" on structured parallel programming, parallel design patterns will be discussed in detail in Chap. 6 to point out similarities and synergies between these two worlds.

3.1 ALGORITHMIC SKELETONS: DEFINITION(S)

Algorithmic skeleton concept has been introduced by Murray Cole with his PhD thesis in 1988 [37]. Cole defined a skeleton programming environment as follows:

> The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

Algorithmic skeletons were defined as pre-defined patters encapsulating the *structure* of a parallel computation that are provided to user as building blocks to be used to write applications. Each skeleton corresponded to a single parallelism exploitation pattern. As a result, a skeleton based programming framework was defined as a programming framework providing programmers with algorithmic skeletons that may be used to model the parallel structure of the applications at hand. In a skeleton based programming framework the programmer has no other way (but instantiating skeletons) to structure his/her parallel computation. In the original definition of Cole, there was no mention to the fact that skeletons could be nested. Therefore any skeleton application was the result of the instantiation of a single algorithmic skeleton and thus, in turn, of a single parallelism exploitation pattern.

Algorithmic skeletons were introduced as Cole recognized that most of the parallel applications exploit well know parallelism exploitation patterns. Rather that requiring the application programmers to rewrite from scratch all the code needed to implement these parallelism exploitation patterns when a new application has to be implemented, it was

The term "'business logic" usually denotes the functional code of the application, distinct from the non functional code modelling, as an example, the implementation of parallelism in the application. In an application processing a set of images to apply to each one of the images a filter f, the code for the filter *is* business logic code, while the code needed to set up a number of parallel/concurrent/distributed agents that altogether contribute to the parallel computation of the filtered images *is not*.

Figure 3.1 Business logic

therefore more natural to provide the patterns as building blocks that the programmers could instantiate by providing the "business logic"[44] of their application.

E.G.▷ As a motivating example, consider parameter sweeping applications. A parameter sweeping application is an application where a single code is run multiple times to evaluate the effect of changes in input parameters. Each run of the code on a different input parameter set is completely independent of the other runs. Therefore a parameter sweeping application may be easily implemented adopting a master/worker (see Sec. 8.4.1 for the definition of the master/worker pattern) parallelism exploitation pattern. A master schedules different input parameter sets to a set of workers running the application code and gathers the result items produced by these workers in such a way the user may evaluate the effects of "parameter sweeping" and possibly iterate the process up to the point he/she finds the proper parameter set (the one providing the expected results). Now all the code needed to implement the master, to properly schedule input parameter sets to the workers and to gather the results is independent of the actual code used to implement the "function" computed by the application. The only dependency is on the types of the input parameter set and of the output results, actually. Therefore, an algorithmic skeleton can be provided to the user that accepts parameters such as the input and output types and the code needed to compute output out of the input data. The skeleton completely takes care of the implementation of the master/-worker parallelism exploitation pattern needed to implement the overall parameter sweeping application. Being the code of the skeleton designed and implemented by (presumably) expert system programmers, well aware of the features of the target architecture at hand, the application programmers will succeed developing an efficient parameter sweeping application without being required to have any kind of knowledge related to target architecture or to the techniques needed to efficiently implement a master worker schema. ——— ■

Later on, Cole published his "skeleton manifesto" [38] where skeletons are described in a fairly more abstract way:

> many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic patterns of computation and interaction. For example, many diverse applications share the underlying control and data flow of the pipeline paradigm. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognize these to enhance performance.

Eventually, the "Skeletal parallelism" home page maintained by Murray Cole[45] gives a more general definition of algorithmic skeletons:

[44] see business logic definition in Fig. fig:txt:business logic
[45] http://homepages.inf.ed.ac.uk/mic/Skeletons/

> There are many definitions, but the gist is that useful patterns of parallel computation and interaction can be packaged up as 'framework/second order/template' constructs (i.e. parameterized by other pieces of code), perhaps presented without reference to explicit parallelism, perhaps not. Implementations and analyses can be shared between instances. Such constructs are 'skeletons', in that they have structure but lack detail. That's all. The rest is up to us.

As we see, Cole's definition of algorithmic skeleton evolved in time. In the meanwhile, several research groups working on algorithmic skeletons contributed to the evolution of the algorithmic skeleton concept. Eventually, a new definition of algorithmic skeletons has been elaborated, which is nowadays commonly accepted. This new definition may be summarized as follows:

> An algorithmic skeleton is parametric, reusable and portable programming abstraction modelling a known, common and efficient parallelism exploitation pattern.

Such definition embeds several different, fundamental concepts:

- it models a *known, common* parallelism exploitation pattern. We are not interested in patterns that are not commonly found in working parallel applications. Possibly, uncommon patterns could be anyway used with those skeleton frameworks that provide some degree of extensibility (see Sec. 10.6).
- it is *portable*, that is an efficient implementation of the skeleton should exists on a range of different architectures.
- it models *efficient* parallelism exploitation patterns. Skeletons only model those patterns that have some known, efficient implementation on a variety of target architectures.
- it is *reusable*, that means it could be used in different applications/contexts without actually requiring modifications
- it is *parametric* in that it accepts parameters that can be used to specialize *what* is computed by the skeleton, rather than *how* the parallel computation is performed.

3.1.1 The skeleton advantage

What are the advantages provided by a programming framework based on algorithmic skeletons? Let us assume that a skeleton programming framework is available, providing the programmer with a convenient number of skeletons.

The implementation of parallel applications using such a framework requires the following steps:

1. the possible parallel structures of the application[46] are evaluated
2. the available skeletons are evaluated and a skeleton (nesting) is individuated as the one modelling the application parallel structure
3. the parallel application code is produced by instantiating the skeleton (nesting) chosen, providing the suitable functional (e.g. the business logic parameters) and non functional (e.g. parallelism degree of the skeleton(s) used) parameters
4. the application is compiled
5. the application is run on the target architecture

[46] that is the possible ways to exploit application internal parallelism

6. the application results are analyzed:
   - for functional correctness: functional correctness of skeleton implementation is guaranteed by the framework, but the correctness of the instantiation of the skeletons, as well as of the business logic code has to be checked anyway
   - for non functional correctness: performance has to measured, looking for potential bottlenecks or inefficiencies.
7. in case of inefficiencies, refinements of the current skeleton structure of the application as well as alternative skeletons (or skeleton nestings) are considered, and the process is restarted from phase 3.

This process is radically different from the process used to develop parallel applications with traditional programming frameworks, such as MPI for instance. In that case, the application programmer must go through the whole process of managing all the details relative to parallelism exploitation, while in this case *all the implementation details relative to parallelism exploitation patterns are managed by the skeleton framework.*

This has several consequences, that represent the fundamental advantages of programming frameworks based on algorithmic skeleton[47]:

- the abstraction level presented to the programmer by the programming framework is higher than the one presented by traditional parallel programming frameworks. This in turn has two consequences:
  - The whole process of *parallel* programming is simplified. Parallelism exploitation mainly consist in properly instantiating the skeleton abstractions provided by the framework rather than programming your own parallel patterns using the concurrency, communication and synchronization mechanisms available.
  - Application programmers may experiment *rapid prototyping* of parallel applications. Different application prototypes may be obtained by instantiating (that is by providing the proper functional/business logic code) different skeletons among those available or, in case this is allowed, different compositions of existing skeletons.
- The correctness and the efficiency of the parallelism exploitation is not a concern of the application programmers. Rather, the system programmers that designed and implemented the skeleton framework guarantee both correctness and efficiency.
- Portability on different target architectures is guaranteed by the framework. An algorithmic skeleton application can be moved to a different architecture just by recompiling[48]. Portability has to be intended in this case not only as "functional portability", but also as "performance portability" as the skeleton framework usually exploits best implementations of the skeletons on the different target architectures supported.
- Application debugging is simplified. Only sequential code (that is business logic code) has to be debugged by the application programmers.
- Possibilities for static and dynamic optimizations are much more consistent than in case of traditional, unstructured parallel programming frameworks as the skeleton structure completely exposes to the programming tools the parallel structure of the application.
- Application deployment and (parallel) run, that notably represent a complex activity, is completely in charge of the skeleton framework

[47] as clearly pointed out in Cole's manifesto[38]
[48] if the algorithmic skeleton framework supports the new target architecture, of course.

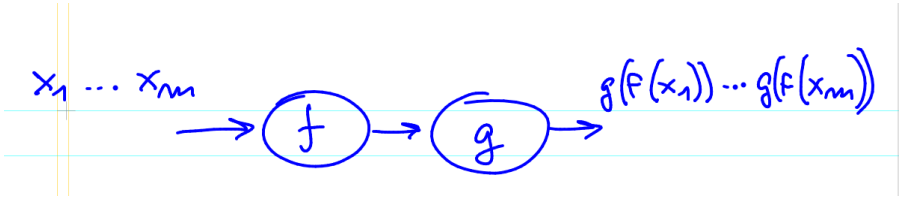


Figure 3.2 Two stage pipeline

3.1.2 The skeleton weakness

Of course, in addition to the advantages of algorithmic skeleton based programming frameworks with respect to traditional, unstructured parallel programming frameworks as listed in the previous section, there are also disadvantages than must be taken into account. The principal drawbacks are related to both "user" (i.e. application programmer) and "system" (i.e. system programmer) views:

- In case none of the skeletons provided to the programmer (or none of the possible skeleton nestings) suits to model the parallel structure of the application the application could not be run through the skeleton framework (see also Sec. 10.6).
- Functional and performance portability across different target architectures requires a huge effort by the skeleton framework designers and programmers. The effort required may be partially reduced by properly choosing the sw target architecture targeted by the skeleton framework, e.g. by considering some "higher level" target to produce the "object" code of the skeleton framework[49].

3.2 SKELETONS AS HIGHER ORDER FUNCTIONS WITH ASSOCIATED PARALLEL SEMANTICS

Following Cole's last and more general perspective, algorithmic skeletons can be defined as *higher order functions* encapsulating/modelling the parallelism exploitation patterns, whose functional parameters provide the "business logic" of the application. The parallel semantics of the skeletons defined in this way is usually given with a short statement detailing how parallelism is exploited in the computation of the higher order function.

As an example, having defined[50] the data type `stream` as

```
type 'a stream = EmptyStream | Stream of 'a * 'a stream;;
```

we can define a *pipeline* skeleton using the higher order function:

```
let rec pipeline f g =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> Stream((g(f x)),(pipeline f g y));;
```

The type of the pipeline skeleton is therefore

$$pipeline :: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha\ stream \rightarrow \gamma\ stream$$

[49] as an example, targeting MPI rather than UDP sockets + Pthreads. MPI provides a virtualization layer that takes care of handling differences in the data representation among different systems/processors, as an example. Therefore less non functional code is needed to run an MPI application than a Pthread/UDP application. In turn, this means the task of the skeleton compiler is easier.

[50] we'll use Ocaml notation through this notes, see [74] for the syntax

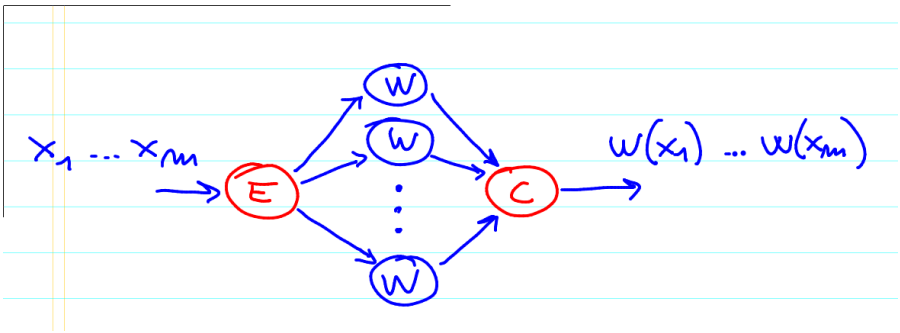


Figure 3.3 Task farm

The function definition perfectly captures the *functional semantics* of the skeleton: a pipeline computes the second stage (function g in this case) on the results of the application of the first stage (function f, in this case) to the input data. The *parallel* semantics of the skeleton is defined by some text, a kind of "metadata" associated to the function, such as:

> *the pipeline stages are computed in parallel onto different items of the input stream. If x_i and x_{i+1} happen to be consecutive items of the input stream, than $f(x_{i+1})$ and $g(f(x_i))$ will be computed in parallel, for any i*

In order to get a working application, whose parallelism is modelled as a pipeline, the user may instantiate the second order pipeline function by passing the parameters modelling first and second stage "functions" f and g.

E.G.▷ As an example, if users want to write an application to filter images and then recognize characters string appearing in the images, provided it has available two functions $filter :: image \rightarrow image$ and $recognize :: image \rightarrow string\ list$, he can simply write a program such as:

```
let main =
   pipeline filter recognize;;
```

and then apply the `main` to the input stream to get the stream of results. ——— ■

Despite the fact the parameters of a skeleton should only model the business logic of the applications (e.g. the pipeline stages), some non functional parameters are also required by skeletons in some well known skeleton frameworks. A notable example of non functional parameters is the *parallelism degree* to be used to implement the skeleton.

As an example, take into account the *farm* skeleton, modelling embarrassingly parallel stream parallelism exploitation pattern:

```
let rec farm f =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> Stream((f x),(farm f y));;
```

whose type is

$$farm :: (\alpha \rightarrow \beta) \rightarrow \alpha\ stream \rightarrow \beta\ stream$$

The parallel semantics associated to the higher order functions states that

> *the computation of any items appearing on the input stream is performed in parallel*

The parallelism degree that may be conveniently used was evident in pipeline. Having two stages, a parallelism degree equal to 2 may be used to exploit all the parallelism intrinsic to the skeletons. In the farm case, according to the parallel semantics a number of parallel agents computing function f onto input data items equal to the number of items appearing onto the input stream could be used. This is not realistic, however, for two different reasons:

1. items of the item stream do not exist all at the same time. A stream is not a vector. Items of the stream may appear a different times. Actually, when we talk of *consecutive* items x_i and x_{i+1} of the stream we refer to items appearing onto the stream at times t_i and t_{i+1} with $t_i < t_{i+1}$. As a consequence, it makes no sense to have a distinct parallel agent for all the items of the input stream, as at any given time only a fraction of the input stream will be available.

2. if we use an agent to compute item x_i, presumably the computation will end at some time t_k. If item x_j appears onto the input stream at a time $t_j > t_k$ this same agent can be used to compute item x_j rather than picking up a new agent.

As a consequence, the parallelism degree of a task farm is a critical parameter: a small parallelism degree do not exploit all the parallelism available (thus limiting the speedup), while a large parallelism degree may lead to inefficiencies as most of the parallel agents will be probably idle most of time.

In some skeleton frameworks, therefore, the parallelism degree is given as a parameter of the skeleton. The task farm skeleton is therefore defined as an high order function having (also) an integer as a parameter. This parameter represent their intended parallelism degree of the *implementation* of the skeleton. However it has no effect on the "function" computed by the skeleton. It has only effect on the "parallel semantics" metadata.

A task farm skeleton with a parallelism degree parameter may therefore be defined with the high order function

```
let rec farm f n:int =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> Stream((f x),(farm f y));;
```

whose type is now

$$farm :: (\alpha \rightarrow \beta) \rightarrow int \rightarrow \alpha\ stream \rightarrow \beta\ stream$$

with the parallel semantics defined as follows:

> the computation of consecutive items of the input stream is performed in parallel on n parallel agents.

These non functional parameters of skeletons generated quite a huge debate within the skeleton community. On the one side, as skeletons pretend to abstract all the aspects related to parallelism exploitation, non functional parameters such as the parallelism degree should not be left under application programmer responsibility. Rather, they should be properly handled by the skeleton implementation, possibly being computed using some performance model of the skeleton (see Chap. 5). On the other side, such parameters can be used by expert programmers to express the fact they want to use exactly that amount of resources. As an example, they may want to experiment different parallelism degrees according to the type of the execution constrains of the applications. If the computation of the results is not urgent, smaller parallelism degrees may be required, whereas, in case of very urgent computations, higher parallelism degrees may be asked through non functional parameters.

Consistently with the idea a skeleton is a programming abstraction hiding all the details related to parallelism exploitation pattern implementation, we agree on the idea that the

better choice is to leave to the skeleton implementation the management of all the non functional skeleton parameters. In case the programmer is allowed to provide such kind of parameters, they will be only taken as loose requirements on the execution of the skeleton: the implementation of the skeleton will try to accommodate user requests, but in case the implementation could figure out the requests are not reasonable or that better choices could be made, these better choices will be actually taken and the user will be consequently informed.

3.2.1 Stream modelling

An alternative way of defining stream parallel algorithmic skeletons assumes pipeline and farm just operate on a global input stream of data to produce a global stream of results. In this case, we may define pipelines and farms with much simpler higher order functions that do not operate on streams, actually, and then use these functions as parameters of another higher order function implementing stream parallelism. In this case, the pipeline skeleton will basically correspond to functional composition:

```
let pipeline f g =
  function x -> (g (f x));;
```

while task farm skeleton corresponds to identity:

```
let farm f =
  function x -> (f x);;
```

The higher order function taking care of exploiting parallelism can then be defined as follows:

```
let rec streamer f x =
  match x with
    EmptyStream -> EmptyStream
  | Stream (v,s) -> Stream ((f v), (streamer f s));;
```

such that, to define the filter/recognize program defined in Sec. 3.2 we can now write:

```
let main =
  streamer (pipeline filter recognize);;
```

Clearly, the parallel semantics associated to skeletons changes in this case, with respect to the parallel semantics we had when pipeline and farms actually processed streams. A parallel semantics is only associated to the *streamer* function as:

the processing of independent input stream data items is performed in parallel

whereas pipeline and farm have do not have associated any kind of parallel semantics.

This way of modelling stream parallelism corresponds to the idea of structuring *all stream parallel applications* as farms[51] with more coarse grain computations as functional parameter.

Again, consider the filter/recognize example. Let's suppose that the recognize phase is much more computationally expensive with respect to the filter phase. Using the explicit stream handling versions of pipeline and farm skeletons, we would write the program as

```
let main =
  pipeline (filter) (farm recognize);;
```

[51] recall the parallel semantics previously given for the farm and look at the analogies in between the *streamer* function and the former definition of the farm higher order function

Recalling the parallel semantics of these skeletons we can easily understand that given a stream with data items $x_1, \ldots, x_m$ (with x_i appearing on the stream after x_{i-k} for any k) the computations of

$$recognize(y_i) \quad recognize(y_{i+k})$$

(being $y_i = filter(x_i)$) happen in parallel, as well as the computations of

$$recognize(y_i) \quad filter(x_{i+k})$$

If we use the alternative modelling of stream parallelism described in this section, the program instead is written as

```
let main =
  streamer (pipeline filter recognize);;
```

and in this case the only computations happening in parallel are the

$$recognize(filter(x_i)) \quad recognize(filter(x_{i+k}))$$

3.3 A SIMPLE SKELETON FRAMEWORK

Although different programming frameworks based on skeletons have been developed that sport each different algorithmic skeletons, a subset of common skeletons may be found in most of these frameworks. These common subset contains skeletons from three different classes of skeletons: stream parallel skeletons, data parallel skeletons and control parallel skeletons.

3.3.1 Stream parallel skeletons

Stream parallel skeletons are those exploiting parallelism among computations relative to different, independent data items appearing on the program input stream. Each independent computation end with the delivery of one single item on the program output stream. Common stream parallel skeletons are the pipeline and task farm skeletons, whose definition in terms of higher order functions and parallel semantics has already been given in Sec. 3.2.

The **pipeline** skeleton is typically used to model computations expressed in stages. In the general case, a pipeline may have more than two stages. The restriction to two stages as given by the definition in Sec. 3.3.4 guarantees strong typing and the ability to define pipelines with more than two stages as pipelines of pipelines. Given a stream of input tasks

$$x_m, \ldots, x_1$$

the pipeline with stages

$$s_1, \ldots, s_p$$

computes the output stream

$$s_p(\ldots s_2(s_1(x_m))\ldots), \ldots, s_p(\ldots s_2(s_1(x_1))\ldots)$$

The parallel semantics of the pipeline skeleton ensures all the stages will execute in parallel. In Chap. 5 we will see that this guarantees that the total time required to compute a single task (pipeline latency) is closed to the sum of the times required to compute the different stages. Also, it guarantees that the time needed to output a new result is close to time spent to compute a task by the longer stage in the pipeline.

The **farm** skeleton is used to model embarrassingly parallel computations, instead. The only functional parameter of a farm is the function f needed to compute the single task. Given a stream of input tasks

$$x_m, \ldots, x_1$$

the farm with function f computes the output stream

$$f(x_m), \ldots, f_(x_1)$$

Its parallel semantics ensures it will process tasks such that the single task is processed in a time close to the time needed to compute f sequentially. The time between the delivery of two different task results, instead, can be made close to the time spent to compute f sequentially divided by the number of parallel agents used to execute the farm, i.e. its parallelism degree[52].

3.3.2 Data parallel skeletons

Data parallel skeletons are those exploiting parallelism in the computation of different sub-tasks derived from the same input task. In other words, the data parallel skeletons are used to speedup a single computation by splitting the computation into parallel sub-tasks. Data parallel computation do not concern, *per se*, stream parallelism. It may be the case that a data parallel computation is used to implement the computation relative to a single stream item in a stream parallel program, however. In this case, data parallelism is used to speedup the computation of all the items belonging to the input stream.

Typical data parallel skeletons considered in the skeleton based programming frameworks include *map*, *reduce* and *parallel prefix* skeletons. Those skeletons can be (or are) defined to work on different kinds of data *collections*. For the sake of simplicity, we consider their definition relatively to vectors.

Given a function f and a vector x the **map** skeleton computes a vector y whose elements are such that $\forall i\ y_i = f(x_i)$. The map skeleton is defined as follows[53]:

```
let map1 f x =
  let len = Array.length x in
  let res = Array.create len (f x.(0)) in
    for i=0 to len-1 do
      res.(i) <- (f x.(i))
    done;
  res;;
```

and its type is therefore

$$map :: (\alpha \rightarrow \beta) \rightarrow \alpha\ array \rightarrow \beta\ array$$

The parallel semantics of the map states that

each element of the resulting vector is computed in parallel.

It is worth to be pointed out that the same kind of reasoning relative to the farm parallelism degree also apply to the map parallelism degree. The only difference is in the kind of input data passed to the parallel agents: in the farm case, these come from the input stream; in the map case, these come from the single input data.

[52] under certain conditions. See again Chap. 5

[53] Ocaml provides a map function on Array data type, whose functional semantics is the same of our map skeleton. Using that function, we could have simply defined the map skeleton as `let map f = function x -> Array.map f x;;`

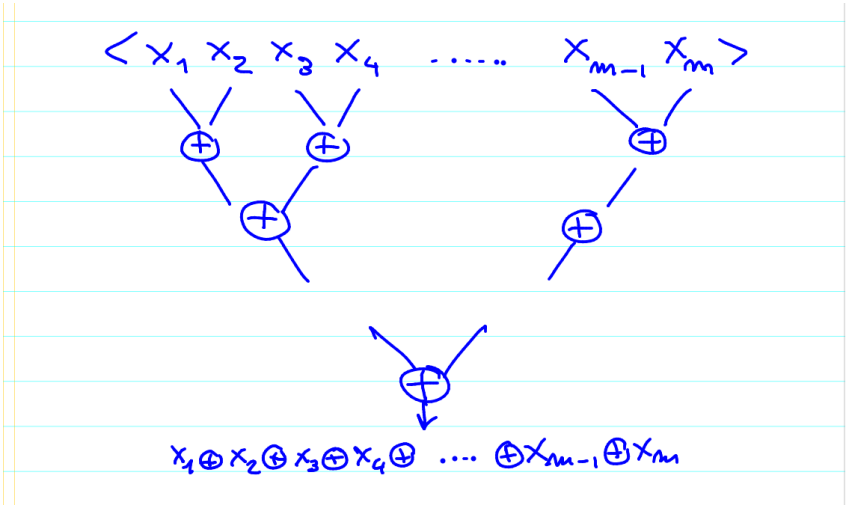


Figure 3.4 Computation performed by a reduce

The **reduce** skeleton computes the "sum" of all the elements of a vector with a function $\oplus$ (see Fig. 3.4). That is, given a vector x it computes

$$x_1 \oplus x_2 \oplus \ldots \oplus x_{len}$$

The reduce skeleton is defined as follows[54]:

```
let rec reduce f x =
  let len = Array.length x in
    let res = ref x.(0) in
      for i=1 to len-1 do
        res := (f !res x.(i))
      done;
      !res;;
```

and has type

$$(\alpha \to \alpha \to \alpha) \to \alpha\ array \to \alpha$$

In the most general form, the function f passed to the reduce skeleton is assumed to be both associative and commutative, and the parallel semantics of reduce states that:

> *the reduction is performed in parallel, using n agents organized in a tree. Each agent computes f over the results communicated by the son agents. Root agent delivers the reduce result. Leaf agents possibly compute locally a reduce over the assigned partition of the vector.*

The **parallel prefix** skeleton computes all the partial sums of a vector, using a function $\oplus$ (see Fig. 3.5). That is, given a vector x it computes the vector whose elements happen to be

$$x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_{len}$$

The paralell prefix skeleton is defined as follows

```
let parallel_prefix f x =
```

[54] Again, Ocaml provides fold_right and fold_left functions on Array, with the same functional semantics of reduce skeleton

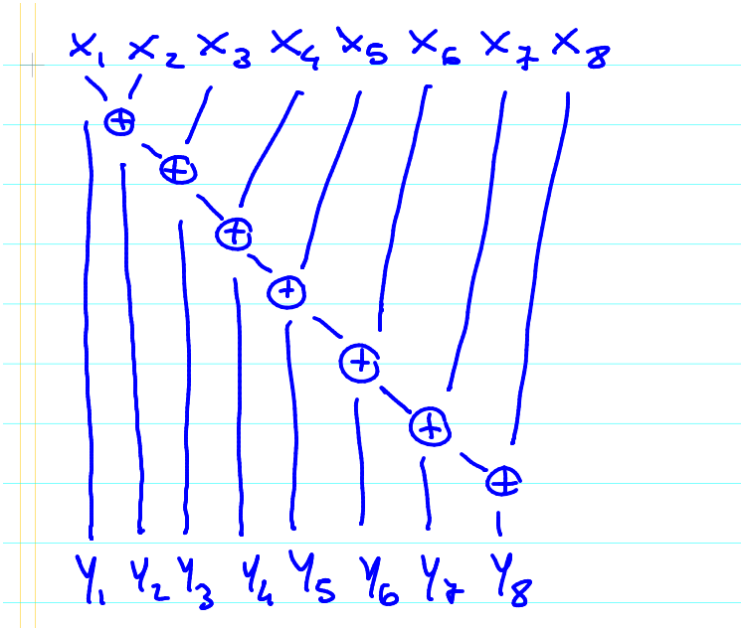


Figure 3.5 Computation performed by a parallel prefix

```
let len = Array.length x in
let res = Array.create len x.(0) in
  res.(0) <- x.(0);
  for i=1 to len-1 do
    res.(i) <- (f x.(i) res.(i-1))
  done;
res;;
```

and has type

$$(\alpha \to \alpha \to \alpha) \to \alpha\ array \to \alpha\ array$$

The parallel semantics states that

> *the parallel prefix is computed in parallel by n agents with a schema similar to the one followed to compute the reduce. Partial results are logically written to the proper locations in the resulting vector by intermediate nodes.*

There are many other data parallel skeletons. In particular, those computing the new item in a data structure as a function of the old item and of the neighboring items (these are called *stencil* data parallel computations).

As an example, let us consider the stencil skeleton on vectors. This skeleton may be defined with the following higher order function[55]:

```
let stencil f stencil_indexes a =
  let n = (Array.length a) in
  let item a i = a.((i+n) mod n) in
  let rec sten a i =
```

[55]in this case the "stencil" set is defined by defining the list of indexes that make up the stencil. The stencil_indexes list hosts increments to be added to the current item position to get the different elements belonging to the stencil. The array is considered circular, therefore the "successor" of the last array element is actually the first array item.

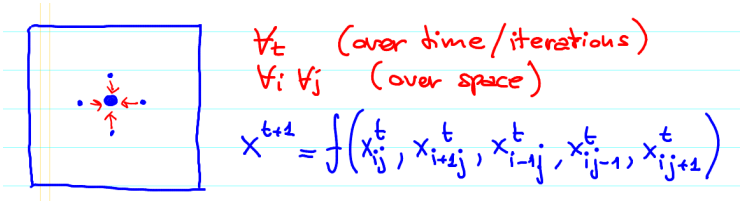


Figure 3.6 Computation performed by a simple stencil data parallel

```
    function
      [] -> []
    | j::rj ->  (item a (i+j))::(sten a i rj) in
  let res = Array.create n (f a.(0) (sten a 0 stencil_indexes)) in
  for i=0 to n-1 do
    res.(i) <- (f a.(i) (sten a i stencil_indexes))
  done;
  res;;
```

whose type is

$$(\alpha \rightarrow \alpha\ list \rightarrow \beta) \rightarrow int\ list \rightarrow \alpha\ array \rightarrow \beta\ array$$

The skeleton processes an input array with items

$$x_0, \ldots, x_n$$

computing an array

$$y_0, \ldots, y_n$$

whose elements are obtained as follows :

$$y_i = f(x_i, l)$$

where l is the list of the array elements x_k $\forall k \in stencil$.
The parallel semantics of this *stencil* can be stated as follows:

each one of the y_i items of the resulting vector is computed in parallel

A sample stencil computation is summarized in Fig. 3.6. Stencil skeletons are important as they can be exploited in a number of different "numerical" applications. In particular, a number of differential equation systems may be solved iterating a computation structured after a stencil pattern.

The last data parallel skeleton we consider is *divide&conquer*. This skeleton recursively divides a problem into sub problems. Once a given "base" case problem has been reached, it is solved directly. Eventually, all the solved sub-problems are recursively used to build the final result computed by the skeleton.

The high order function capturing the divide&conquer semantics may be written as follows:

```
let rec divconq cs dc bc cc x =
  if(cs x) then (bc x)
  else (cc (List.map (divconq cs dc bc cc) (dc x)));;
```

where `cs` is the function deciding whether input problem has to be split, `dc` is the function splitting the input problem into sub problems, `bc` is the function computing the base case and `cc` is the function "conquering" sub problem solutions, that is the function computing

the solution of a problem out of the solutions ot its sub problems. The type of the skeleton high order function is therefore

$$(\alpha \rightarrow \ bool) \rightarrow (\alpha \rightarrow \alpha\ list) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\beta\ list \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

The parallel semantics associated to the divide&conquer skeleton states that

computation of Sub problems deriving from the "divide" phase are computed in parallel.

This is a very popular algorithmic skeleton. It is classified as data parallel as in the general case parallelism comes from the decomposition of some particular data structure.

E.G.▷ Sorting is the classical example used to explain divide&conquer. In order to sort a list of integers, we may split the list in two sub-lists, order the two sub lists and then merge the ordered sub lists. If we take care of splitting the list into sub lists in such a way elements of the first sub list happen to be smaller or equal to the elements appearing in the second sub list, the merge phase is reduced to juxtaposition of the two ordered sub lists. Therefore we can define this sort as

```
divconq (singleton) (pivotSplit) (identity) (juxtappose)
```

where `singleton` returns true if the argument list is of length equal to 1, `pivotSplit` takes a list, extracts the first element (the pivot) and returns the two lists made of all the original list elements smaller that the pivot plus the pivot itself and of the original list elements larger than the pivot, the *identity* returns its parameter unchanged and eventually `juxtapose` takes two lists and return the list made of the elements of the first list followed by the elements of the second one.

■

3.3.3 Control parallel skeletons

Control parallel skeletons actually do not express parallel patterns. Rather, they express structured coordination patterns and model those parts of a parallel computation that are needed to support or coordinate parallel skeletons.

Control parallel skeletons typically include:

Sequential skeleton which is used to encapsulate sequential portions of code in such a way they can be used as parameters of other (parallel) skeletons.

Conditional skeleton which is used to model *if-then-else* computations. Such a skeleton can be expressed by the following higher order function:

```
let ifthenelse c t e x =
  match(c x) with
    true -> (t x)
|   false -> (e x);;
```

whose type is

$$ifthenelse : (\alpha \rightarrow bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

Iterative skeleton which is used to model definite or indefinite iterations. As an example, a *skwhile* iterative skeleton may be represented through the higher order function

```
let rec skwhile c b x =
  match(c x) with
    true -> (skwhile c b (b x))
```

```
  | false -> x;;
```

whose type is

$$skwhile : (\alpha \rightarrow bool) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

These skeletons do not model any parallelism exploitation patterns, although some of them may lead to parallel computations. As an example:

- A conditional skeleton *if-condition-then-else* may be implemented in such a way the computation of the *condition*, *then* and *else* parts are started concurrently. Once the *condition* is evaluated, either the *then* termination is awaited and the *else* computation is aborted or vice versa, depending on the result given by the computation of *condition* (this parallel semantics implements a kind of *speculative* parallelism).
- An iterative skeleton may be implemented in such a way a number (possibly all) the iterations are executed concurrently, provided that there are no *data dependencies* among the iterations (this is a kind of `forall` skeleton, actually, which is very similar to the map one. The only difference concerns the way the parallel computations are derived: in the map, they come from the decomposition of the compound input data structure; in the `forall` they come from different iterations of the same loop).

3.3.4 A skeleton framework

With the definitions given up to now, we can imagine a *skeleton programming framework* as a system providing at least:

1. a number of skeletons, described by a grammar such as:

```
Skel ::= StreamParallelSkel |
         DataParallelSkel | ControlParallelSkel
StreamParallelSkel ::= Farm | Pipe | ...
DataParallelSkel ::= Map | Reduce |
                     ParallelPrefix | ...
ControlParallelSkel ::= Seq | Conditional |
                        DefiniteIter |
                        IndefiniteIter | ...
Seq ::= seq(<sequential function code wrapping>)
Farm ::= farm(Skel)
Pipe ::= pipeline(Skel,Skel)
Map ::= map(Skel)
Reduce ::= reduce(Skel)
ParallelPrefix ::= parallel_prefix(Skel)
Conditional ::= ifthenelse(Skel,Skel,Skel)
DefiniteIter ::= ...
...
```

   The framework provides application programmers with the API to write `Skel` expressions denoting the skeletal program as well as some statement invoking the execution of the skeletal program. As an example, in Muskel [14] a `Skel` program can be declared as:

```
Skeleton filter = new Filter(...);
```

```
Skeleton recon = new Recognize(...);
Skeleton main = Pipeline(filter, recon);
```

2. a way to invoke the execution of the skeleton program. As an example, in Muskel this can be demanded to an `Application Manager` by passing it as parameters i) the program to execute, ii) the input stream and iii) the output stream manager[56]:

```
Manager mgr =
  new Manager(main,
    new FileInputStreamManager("datain.dat"),
    new ConsoleOutputStreamManager());
mgr.start();
```

Alternatively, the skeleton framework may provide library calls directly implementing the different skeletons. As an example, in frameworks such as SkeTo [84] the skeletons are declared *and* executed via suitable library calls: to use a map skeleton followed by a reduce skeleton the user may write a code such as

```
dist_matrix<double> dmat(&mat);
dist_matrix<double> *dmat2 =
   matrix_skeletons::map(Square<double>(), &dmat);
double *ss =
   matrix_skeletons::reduce(Add<double>(),
                            Add<double>(), dmat2);
```

In both cases, provided the framework is properly set up, the evaluation of the skeleton program leads to the proper exploitation of the parallel patterns modelled by the skeletons used. In the former case (Muskel, which is based on Java), the execution of the program through a plain Java interpreter will lead to the execution of the first and second pipeline stages on two different computing resources interconnected using TCP/IP sockets and Java RMI protocols. This, in turn, implements the pipeline parallel semantics as defined in Sec. 3.3.4. In the latter case (SkeTo, which is based on C++), the execution of the program compiled using a C++ compiler and linked with an MPI library leads to the execution in parallel, onto the available processing elements of both the `map` and the `reduce` skeletons, in sequence.

3.3.5 Skeleton programs

Skeleton programs written using a framework such as the one envisioned in the previous section have a well defined structure, that is usually represented by a *skeleton tree*.

The skeleton tree has nodes representing algorithmic skeletons and leaves representing sequential portions of code (the business logic code). Skeleton parameters are represented in node leaves (Fig. 3.7 shows an example of skeleton tree).

We will use in the following skeleton trees in different contexts: as an example, to represent parallel program classes, to expose the parallel structure of an application or to represent compiler data.

With a framework such as the one outlined in the previous sections, a skeleton tree may be defined according to the following grammar[57]:

[56] input and output stream managers are *de facto* iterators producing the items of the input stream and consuming the items of the output stream, respectively

[57] The grammar abstracts all non skeleton parameters, that are not significant when representing the parallel structure of the skeleton program.

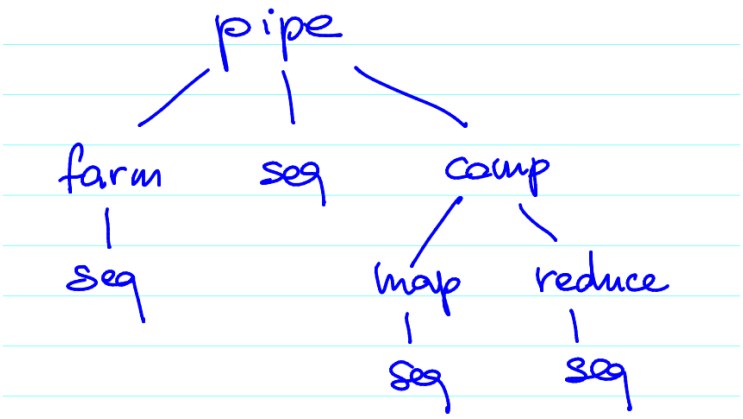


Figure 3.7 Sample skeleton tree

```
Sk ::= Seq(Code)
| Farm(Sk) | Pipe(Sk,Sk)
| Map(Sk) | Reduce(Sk) | Stencil(Sk)
| While(Sk) | For(Sk) | ...
```

As an example, the skeleton tree of a program filtering a stream of images by means of two filters, with one of the two filters being data parallel, may be represented by the term

```
Pipe(Map(Seq(dpFilter)),Seq(secondFilter)
```

or, in case the second filter is parallelized through a farm, by the term

```
Pipe(Map(Seq(dpFilter)),Farm(Seq(secondFilter))
```

These expressions only expose the parallel structure of the two programs. There is no parameter modelling the parallelism degree of the map and farm, as an example, nor there are parameters modelling the data types processes by `dpfilter` or `secondFilter`.

3.4 ALGORITHMIC SKELETON NESTING

Algorithmic skeletons, as proposed by Cole in his PhD thesis [37] were conceived as patterns encapsulating a single parallel pattern. In the thesis these patterns were used alone. Therefore, an algorithmic skeleton based application exploited just a single, possibly complex form of parallelism.

The algorithmic research community immediately started to investigate the possibilities offered by algorithmic skeleton composition. For a while, two perspectives have been investigated:

"CISC" skeletons Several researchers imagined to provide users with a large number of algorithmic skeletons [56], each modelling a particular parallelism exploitation pattern[58]. These skeletons were derived analyzing existing applications and looking for *any* pattern that demonstrated to be useful and efficient (e.g. scalable, performant,

[58] We introduce the terms "CISC" and "RISC skeletons" although this terminology has never been used before in the skeleton community. However, we think that CISC and RISC perfectly summarize the two approaches discussed here.

etc). Due to the large amount of different parallel applications, and due to the large number of significantly or slightly different parallelism exploitation patterns, eventually CISC skeleton set became huge. In turn, this implied that possible users (the programmers) were forced to dig into the skeleton set the one matching their needs. The effort needed to individuate the "best" algorithmic skeleton matching the parallelism to be exploited in the user application was sensible, due to large number of similar items in the skeleton set.

Furthermore, in case of CISC skeletons, nesting of skeletons was in general not allowed. This means that if you succeeded to model your application with one of the available skeletons and later on you realized the achieved performance was not satisfactory, you were not allowed to *refine* the skeleton structure of the program introducing further skeletons as arguments of the existing ones. Instead, you were forced to remove the (single) skeleton used and to pick up another skeleton in the set provide to the programmers. In turn, this could require changes in the parameters passed, that is in the business logic code, which is usually something we want to avoid as much as possible[59].

"RISC" skeletons Other researchers imagined to follow a RISC approach in the design of the skeleton set provided to the application programmers. That is, they investigated skeleton sets with a small number of items. These skeletons modelled very basic parallelism exploitation patterns. Each of the "code" parameters of the skeletons, however, could be recursively provided as skeletons. The envisioned scenario behind this design choice allowed complex parallelism exploitation patterns to be defined as composition of primitive parallelism exploitation patterns.

In this case, the application programmer were required to know only a small set of base skeletons plus another small number of "composition rules". It is clear that composition rules play a significant role in this case.

After a while, most of the researchers working on algorithmic skeletons opted for some kind of variant of the second approach and most of the algorithmic skeleton framework developed so far include the possibility to nest skeletons. However, different frameworks offer different composition/nesting possibilities. To understand the different cases, we start discussing how (possibly existing) sequential portions of business code may be included in a algorithmic skeleton program (Sec. 3.4.1). Then we will discuss the problems related to implementation of full algorithmic skeleton compositionality (Sec. 3.4.2) and eventually we will discuss the more common "non full" composition rules, namely those based on a *two tier* approach (Sec. 3.4.3).

3.4.1 Sequential code wrapping

In an algorithmic skeleton program sequential portions of code are needed to implement the application business code.

E.G.▷ As an example, in an application whose parallelism exploitation is modelled after a two stage pipeline skeleton, the functions computed by the two stages represent business logic code and may be supplied re-using existing code, already proven correct and fine tuned, properly wrapped in a form suitable to be used within the pipeline skeleton. ____ ■

Different algorithmic skeleton frameworks chose different approaches to support re-use of sequential portions of (business) code.

[59] Any change in the business logic code requires proving correctness by the application programmer. Correctness of the skeleton implementation, instead, does not need to be proved by the application programmers.

Some of typical design choices concerning sequential code re-use are related to the possibility to reuse the following kind of code:

- code written in a single sequential programming language vs. code written in a language of a given class of enabled sequential languages.
- side-effect free, function code vs. generic code, including static variables
- source code vs. object code
- source language level wrapping in functions/procedures vs. object code wrapping

Let us analyze pros and cons of these different possibilities:

Single vs. multiple languages. Restricting the possibility to use existing code written in a single sequential programming language allows to implement simpler and possibly faster skeleton programming frameworks, as there is no need to set up particular, multi language procedures. However, when different languages provide object code interoperability, that is when programming language C_i allows to seamlessly call procedures/functions written in language C_j, then considering the possibility to wrap both C_i and C_j code does not require additional efforts.

Functions vs. generic code. In most cases, code with side effects may hardly be safely integrated into algorithmic skeletons. Although nothing prevents in principle the possibility to use side effect code as a pipeline stage or as a map worker, the side effects determine a number of problems. As an example

- you cannot in general move (e.g. to improve the computation performance) a skeleton component (e.g. a map worker or a pipeline stage) from one machine to another one preserving the semantics of the application (side effects could have modified data which has not been comparably modified on the target machine);
- in case of fault of one of the concurrent activities relative to a skeleton, you are not able to restore the computation status, as you don't know which was the exact current status (side effect status is not known).
- you cannot guarantee in general the functional correctness of the implementation of the skeleton framework (you cannot guarantee that two runs with the same input data return the same output results, as different sub computations contributing to the global algorithmic skeleton computation may take place in different runs on different machines).

Algorithmic skeleton frameworks usually *require* that the application programmer guarantees that the sequential portions of code used as skeleton parameters are pure functions.

Source vs. object legacy code. In general the business code that programmers wish to re-use is constituted by some source code possibly calling some external libraries. In case only re-use of source code is allowed, this may exclude the possibility to use well know and effective *libraries*.

Therefore, algorithmic skeleton frameworks usually allow to use legacy libraries in the wrapped business code, provided that this dependency is somehow declared. The declaration of the dependency allows the framework to ensure the presence of the called library in all places where the wrapped code calling the library will be eventually executed, as an example, or, in case of fault, to restore a good "restart point" for the computation on a different processing node.

Source vs. object code wrapping. Wrapping source code is generally easier than wrapping object code. In particular, to wrap source code standard compiler tools can be exploited (e.g. pre-processing, annotations, source-to-source compilers). In case object code is considered for wrapping the "structure" of the code must be known or at least the way input and output parameters are managed must be known.

3.4.2 Full compositionality

Full compositionality in a skeleton framework means that a skeleton[60] may be used as actual parameter for *any* code parameter of *any* skeleton. This property is also known as *flat nesting*, and it has several important implications:

1. Full compositionality implicitly assumes that any skeleton composition actually makes sense, as an example that a map used as a pipeline stage represents a notable and useful parallelism exploitation pattern, as well as a pipeline used as a map worker. As we will discuss in Sec. 3.4.3 some skeleton compositions do not make sense, actually.
2. If we allow any skeleton to appear as a code parameter in another skeleton we assume that the type of the function of the skeleton is the same of the function used as parameter in the second skeleton. This, in turn, raises two different problems:
   - All function parameters must have the same generic type. Then we can use this type as type of *all* the skeletons in the framework. As an example, we can use the generic type $f : \alpha \rightarrow \beta$.
   - We must find a way to conciliate stream parallel skeletons with function parameters. A pipe will have type $pipe : (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha\ stream) \rightarrow (\gamma\ stream)$. Therefore in order to be able to use a pipe as a stage of another pipe we must conciliate the fact $\alpha \rightarrow \gamma$ is not the same type as $\alpha\ stream \rightarrow \beta\ stream$.
3. The support of full compositionality implies that, given two skeletons Δ_1 and Δ_2 modelling two different kind of parallelism exploitation patterns, the techniques used to implement the two patterns must be *compatible*. In other words we must ensure that the joint exploitation of the two parallelism exploitation patterns in $\Delta_1(\Delta_2)$ (or in $\Delta_2(\Delta_1)$) does not lead to any kind of conflict or inefficiency.

These problems represents an obstacle to the support of full compositionality. In particular, the third point above is the critical one. We don't know at the moment how to implement efficiently compositions of arbitrary parallelism exploitation patterns. As a consequence, limitations are considered related to algorithmic skeleton composition. The more common limitation is to allow only certain skeleton composition, as detailed in the next Section.

3.4.3 Two tier model

During the design of P3L [23] in early '90, the group of the Dept. of Computer Science in Pisa come out with a well defined composition rule for algorithmic skeletons, that was able to solve most of (possibly all) the problems related to algorithmic skeleton composition.

P3L provided a small set of skeletons, including both stream parallel and data parallel skeletons. Each one of the P3L skeletons could use another skeleton in a code parameter. Skeletons could be used to instantiate pipeline stages as well as map, reduce or farm workers. However, the resulting skeleton tree[61] were *valid if and only if stream parallel skeletons were*

[60] with all its parameters properly instantiated
[61] the tree with skeletons at nodes and parameters in the skeleton node leaves

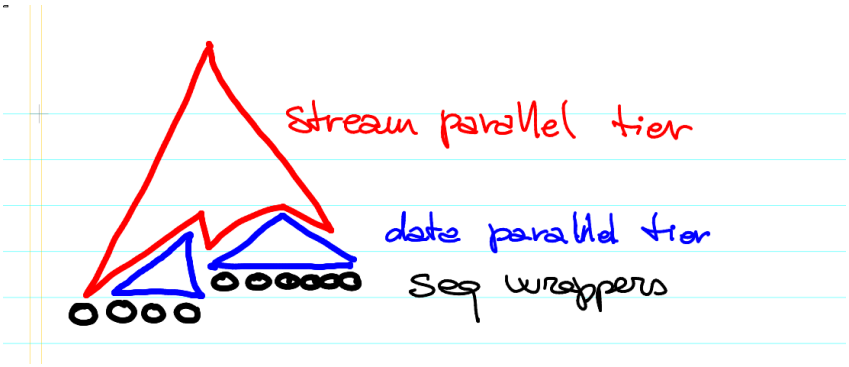


Figure 3.8 Two tier skeleton nesting

used in the upper part of the tree, data parallel ones in the lower part and sequential skeleton (i.e. wrapping of sequential, business logic code) in the leaves.

The upper part of the skeleton tree represents the first stream parallel tier and the lower part (excluding the leaves) the second, data parallel tier (see Fig. 3.8).

This choice was operated observing that:

- data parallelism with internal stream parallel patterns does not offer substantial performance improvements over the same kind of data parallelism with sequential parameters only
- data parallelism may be refined structuring its data parallel computations as further data parallel patterns
- both stream parallel and data parallel patterns eventually should be able to use sequential code parameters.

This model is still in use in several programming frameworks supporting skeleton composition, including Muesli and (in a more restricted perspective) SkeTo.

3.5 STATELESS VS. STATEFULL SKELETONS

Classical skeleton frameworks always considered, designed and provided the final user/programmer with *stateless* skeletons, that is skeletons whose functional behaviour may be completely modelled through higher order *functions*. This is mainly due to the class of parallel architectures available when the first skeleton frameworks were conceived and implemented. At that time, "parallel" architectures was a synonym of "distributed memory" architectures. Clusters, as well as *ad hoc* multiprocessors did not provide any kind of shared memory mechanisms and therefore separate processing elements had to be programmed with programs accessing disjoint rather than shared data structures. This situation has changed, as nowadays a number[62] of multi/many core architectures provide quite efficient shared memory mechanisms.

Moreover, it is known that a number of cases exists where a concept of *shared state* may be useful to model parallel computations. In some cases, it may be useful to maintain in a state variable some counters relative to the current parallel computation. In other cases it is worth allowing concurrent activities to (hopefully not frequently) access some centralized and shared data structure.

[62] the vast majority, actually

E.G.▷ A stencil data parallel computation looking for solutions of some differential equation system will iteratively proceed up to the moment the difference between the current and the the previously computed solution approximation is smaller than a given ϵ. A parallel application with this behaviour can be programmed as a map reduce, with the map computing the new approximated solution as well as the "local" difference among the current and the old approximate solution and the the reduce computing the maximum absolute "local" distance. The computation ends when this value happens to be below the threshold ϵ. A much more natural way of expressing the same computation will be one where the data parallel workers update a globally shared $maxDiff$ variable. This requires, of course, that each worker in the data parallel template is able to perform an atomic *update-with-local-value-if-larger-than-the-global-shared-one* operation. ■

E.G.▷ A parallel ray tracer application may be programmed in such a way all the "rays" may be computed in parallel, provided that the scene parameters (shapes, positions, surface feature of the objects, light position, etc.) may be accessed concurrently. A natural solution consists in programming a data parallel skeleton providing a copy of the scene data to all the workers in the data parallel template. A possibly more efficient solution–e.g. in case of large scene data–consists in programming the data parallel workers in such a way they can access–*read only*–the scene data. ■

Managing and scheduling complex access patterns to a shared data structure represents a complicate and error prone activity, requiring proper planning of shared data access scheduling and synchronization. It also represents one of the classical source of bottlenecks in existing parallel applications: when shared data structures must be accessed concurrently, the concurrent activities rarely may achieve ideal speedup. The accesses to the shared data structure, in fact, make up the serial fraction that limits the speedup achieved in the parallel application (see Amdhal law in Sec. 2.4). Therefore, shared data access has always been prohibited in structured parallel programming models based on the algorithmic skeleton concept (remember that an algorithmic skeleton has to be a well known, *efficient* parallelism exploitation pattern, according to the skeleton assessed definition).

With the growing and ubiquitous presence of shared memory multi/many core architectures, however, the problem of being able to manage some kind of shared global state is becoming again more and more interesting.

All the skeleton based programming frameworks listed in Chap. 13 that have been labeled as supporting multi/many core architectures, exploit multi cores to support a number of parallel activities, actually, but they do not–at the moment and with the exception of FastFlow–exploit the existence of a shared memory support to provide the user/programmer with some kind of shared state abstraction available to the existing skeletons.

mettere uncheck SM

However, the structure of the parallel computation exposed through the skeleton composition used to model parallel applications may be used to improve the efficiency in dealing with shared state within parallel computation.

We illustrate in the following sections some principles and techniques useful to provide such shared state support in skeleton based frameworks.

3.5.1 Shared state: structuring accesses

We start distinguishing the different kind of accesses performed on shared state data structures. The particular features of each kind of access will then be used to determine the better techniques supporting the implementation of the shared state in the skeleton framework. The kind of accesses we distinguish include:

Readonly This is not actually a shared state, as no modification is made to the shared structure. The only concurrent accesses are those aimed at reading (parts of) the shared structure.

Owner writes This is another particular and quite common kind of shared state access: a number of concurrent entities access the data structure for reading but only one of the concurrent entity is allowed to modify a single sub-partition of the shared state structure.

Accumulate In this case, the shared state data is accessed for writing by all (more than one of) the concurrent entities accessing the state, but the only kind of write operation is such that each write turns out to be a

$$x_{new} = x_{old} \oplus y$$

were $\oplus$ is a commutative and associative operator and y is a parameter provided by the concurrent entity requiring the update of the shared state variable.

Resource This is the kind of shared state access that mimics an exclusive access to a shared resource. It corresponds in obtaining a *lock* on the shared state variable, then modifying the variable and eventually releasing the lock.

These kind of accesses allow different strategies to be implemented to support shared state implementation. These strategies vary depending on the kind of architectural support available for shared memory accesses. In particular:

- *Readonly* state may be handled by copying. In case the data structure is large and shared memory is supported, accesses to the shared data structure do not need any kind of synchronization nor cache coherence needs to be ensured on the cache copies of the shared data structure (parts).
- *Owner writes* state variables may be handled in such a way the "owner" concurrent entity has allocated in its private memory[63] the "owned" state variables and other concurrent activities must access these values accessing the "owner" memory. This means implementing remote accesses in case of distributed architectures or implementing some kind of cache coherence algorithm in case of shared memory target architectures.
- *Accumulate* shared state may be implemented by encapsulating the accumulated state in a concurrent activity that receives asynchronous requests *accumulate(y)* by the computation concurrent activities and applies the accumulate operation on a local copy of the state. Read request relative to the accumulated value may be implemented in the same way, may be causing a *flush* of all the write requests pending before actually reading the value to be returned back to the requesting concurrent activity. In case of target architectures providing support for shared memory, this kind of state may be implemented in a more distributed way, profiting of cache coherence mechanisms. The value of the accumulate state will therefore migrate to the places where the accumulate write requests will be generated.
- *Resource* state variables should be handled with full synchronization in order to guarantee the exclusive access to the state values.

The consequences of the usage of state variables within structured parallel computations depend on the kind of shared state and, with a relatively smaller weight, on the kind of

[63] we refer to memory as *main memory or caches* in this case, depending on the kind of target architecture

parallel pattern used. Let us assume the kind of pattern used only impacts the number of concurrent activities accessing the shared state[64]. Then:

- Accesses to *readonly* shared stated does not impact parallelism exploitation. A little overhead may incur in the access to readonly state variables with respect to the overhead needed to access regular variables[65].
- Accesses to *owner writes* shared state will behave like accesses to *readonly* state. No parallel activities are serialized due to state access as only the owner concurrent activity may write (its part of) the state variable. A possible, small overhead may be measured when accessing–reading–shared variables if at that time the same (part of the) state variable is being written by its owner concurrent activity.
- Accesses to *accumulate* shared state will be implemented mostly asynchronously. Again, these accesses will not impair the potential parallelism: no logically concurrent activities will be serialized *unless* specific "flush/barrier" operations are provided to synchronize the accesses.
- Accesses to *resource* state variables may completely serialize potentially parallel/concurrent activities, due to the full synchronization ensured in the shared state accesses.

E.G.▷ Consider a farm, and assume we want to count the number of tasks computed by the string of workers in the farm. We may use an *accumulate* integer shared variable. The accumulator operator $\oplus$ is the actual addition operator, which is commutative and associative. Each farm worker will issue a *accumulate(1)* request after completing the current task computation. The shared state variable will only be read at the end of the computation, possibly by the farm collector or by some other concurrent activity taking place after farm termination. No serialization is imposed on worker activity due to access to this simple shared state.

■

E.G.▷ Consider a pipeline and assume each state uses a *resource* shared state to compute the stage function. We have to distinguish two cases:

1. the access to the shared state represents a small fraction of the stage computation, or
2. it represents a consistent fraction of the computation.

In the former case, each stage access to the shared state is serialized. As a consequence, only the remaining fraction of stage computation may be actually performed in parallel. Therefore, pipe parallelism is reduced according to the Amdhal law (see Sec. 2.4).

In the latter case, the computation of the stages of the pipeline are *de facto* serialized. The whole pipeline behaves therefore as a sequential stage with latency equal to the sum of the latency of the stages (see Chap. 5).

■

3.5.2 Shared state: parameter modeling

State modeling at the skeleton syntax level may be implemented using labelling. We therefore first *declare* the kind of shared state we want to use, then we use the name of the shared state variable as any other variable/parameter. The implementation will take care of planning the correct kind of accesses to the shared variables.

Consider our sample skeleton framework introduced in Sec. 3.3.4. That framework may be extended with some additional syntax to declare state parameters:

[64]this is a quite simplistic assumption, but relatively to the classification of shared states used in this Section, it makes sense
[65]non state ones

```
Id ::= <valid identifier>
Ids ::= Id | Id, Ids
StateParams ::= readonly Ids; | ownerwrites Ids; |
  accumulate Ids; |  resource Ids;
```

Then the `Ids` used to declare state parameters may be used in all places where normal parameters are used as skeleton parameters.

In our higher order function description of skeletons we can for instance assume to have, in addition to what already introduced in previous sections, the state data type defined as:

```
type 'a state = ReadOnly of 'a
  | OwnerWrites of 'a
  | Accumulate of 'a
  | Resource of 'a;;
```

Then we can define a stateful farm, accumulating the number of tasks computed by the worker as follows:

```
let a = Accumulate(ref 0);;  (* declare state *)

let accumulate a y =   (* define accumulate function *)
  match a with
     (Accumulate i) -> i:=(!i+y)
   |  _ -> failwith ("shared state kind not allowed");;

let rec farmWithAcc f a =
  function
    EmptyStream -> EmptyStream
  | Stream(x,y) -> (accumulate a 1);
                   Stream((f x),(farmWithAcc f a y));;
```

This obviously models functional semantics of the farm with state[66]. In fact, the function computed by the program is the one resulting from the sequential computation of all the tasks. As the accumulation on the shared variable happens via an associative and commutative operator, the final result is the same achieved in case of any other, non sequential scheduling of the farm workers.

However, other kind of "stateful" skeletons may be more hardly modeled following this approach. As an example, a farm skeleton using a *resource* kind of shared state could be easily written as

```
let rec farm f ssr =
  function
   EmptyStream -> EmptyStream
|  Stream(x,y) -> Stream((f ssr x),(farm f ssr y));;
```

Then we can define some auxiliary functions as follows:

```
let ssr = Resource (ref 0);;

let inc ssr =
  match ssr with
    Resource r -> r:=(!r +1)
```

[66] this particular kind of state

```
|   _ -> failwith("shared stated kind not allowed in inc");;

let dec ssr =
  match ssr with
    Resource r -> r:=(!r -1)
|   _ -> failwith("shared stated kind not allowed in inc");;

let f ssr x =
  if(x > 100) then ((inc ssr);x*2)
  else ((dec ssr); x/2);;
```

Eventually, a farm computing function f above can be defined as:

```
farm f;;
```

having type:

$$int\ ref\ state -> int\ stream -> int\ stream$$

However, this only computes the function computed by the sequential schedule of all the farm tasks. Therefore it cannot be used anymore as functional semantics of the statefull skeleton.

We can summarize the problems related to shared state managing in skeletons as follows:

- In order to model stateful skeletons, the shared variables must be exposed in the skeleton code. Then depending on the skeleton parallel pattern and on the accesses made to the shared variables suitable implementation policies may be devised.
- However, different access patterns lead to different implementation strategies. Therefore also the access patterns should be exposed.
- In the general case, shared state modelling requires much more complicated high level functions to capture concurrent access properties.

CHAPTER 4

IMPLEMENTATION OF ALGORITHMIC SKELETONS

Algorithmic skeleton frameworks can be implemented according to different models. A first, important alternative is among implementations introducing new programming languages and implementations actually introducing only libraries for well know, sequential programming languages. In both cases, skeleton programming frameworks may be implemented using different software technologies. Traditionally, algorithmic skeleton frameworks have been implemented exploiting *implementation templates*. Most of the early implementation of skeleton frameworks actually exploit this technology. In late '90s, a *macro data flow* based implementation model has been proposed. More recently, implementations of skeleton frameworks exploiting *AOP* (Aspect Oriented Programming), *annotations* and *software components* have been designed and implemented. In the following sections, we will discuss these different approaches.

4.1 LANGUAGES VS. LIBRARIES

Algorithmic programming frameworks may be provided to the application programmers in two fundamentally different flavors:

- As a brand **new programming language**, hosting skeletons as proper language constructs and compiled (or interpreted) in such a way the skeleton programs may be suitably executed on the target parallel architecture at hand, or
- as a **library** written in a particular *host language*.

4.1.1 New language

In case of skeleton frameworks implemented as new programming languages, the application programmer is exposed to a new language sporting skeletons as primitive constructs. No-

table examples of this approach are the P3L [23] and ASSIST [87] programming frameworks. In these cases, the programmer writes programs that include particular statements/declarations declaring skeletons as well as the `main` program to be executed. As an example, in P3L, a skeleton program looks like:

```
seq filter in(Image x) out(Image y)
$c++{
    // ... c++ code producing y out of x here
$c++}

seq recon in(Image x) out(String y[])
$c{
  /* C code producing the array of strings
     y out of the Image x here */
$c}

pipeline main in(Image a) out(String c[])
  filter in(a) out(Image b)
  recon in(b) out(c)
end pipeline
```

The program, once compiled through the `p3lcc` compiler may be run as a plain Unix filter processing inputs from the standard input and delivering outputs to the standard output. The sequential pieces of code wrapped into the `filter` and `recon` sequential skeletons are compiled using plain C++ or C compilers, as indicated among the $ and { characters introducing the sequential code. The program is defined by the topmost skeleton, i.e. the skeleton not parameter of any other skeleton, which turns out to be the pipeline `main` in our case. Proper variable scoping is defined in such a way the variables named across different skeleton calls (e.g. variable c used as output of `filter` and input of `recon`) model communication channels among skeletons[67].

When a new programming language is introduced to provide skeletons, a set of compiler/interpreter tools are provided such as those of Fig. 4.1. A *compiler* translates source code into intermediate/object code. This object code is then linked with proper libraries (e.g. communication libraries) and eventually run on the parallel computing resources at hand by the *linker/launcher* tool.

The compiler and the linker/loader tools may be implemented by orchestrating different compiler/linker tools, actually. In most cases, the new programming languages providing skeletons are not compiled directly to object code. Rather, they are compiled to some intermediate "source + library call" code in some *host* sequential programming language that is then compiled to object code using the host sequential language compiler. This happens to avoid re-writing sequential compilers when these are already available. The "library calls" here are usually calls to some communication library. As an example, in P3L the `p3lcc` compiler produced C + MPI "object code". This code was eventually compiled with the `mpicc` compiler. The resulting code was eventually run on the parallel computing elements with the `mpirun` command.

4.1.2 Library

When the algorithmic skeletons are provided as libraries, a completely different approach is followed.

[67] In case of P3L, these channels were implemented using MPI channels [32]

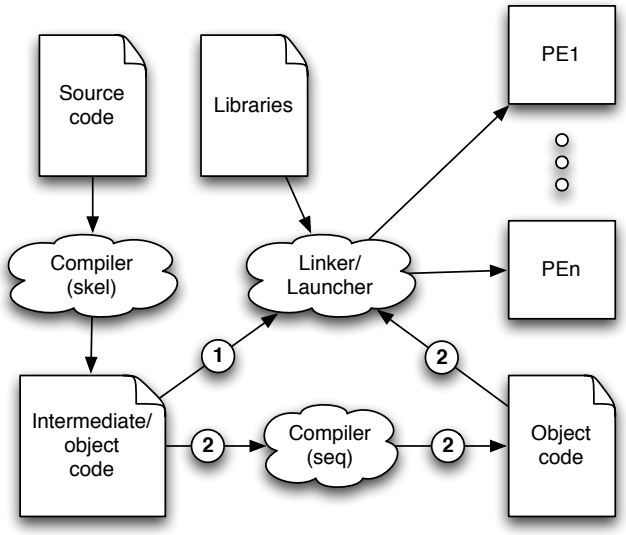


Figure 4.1 Tool chain for skeleton frameworks providing a new language (workflow ① and ② represent alternatives)

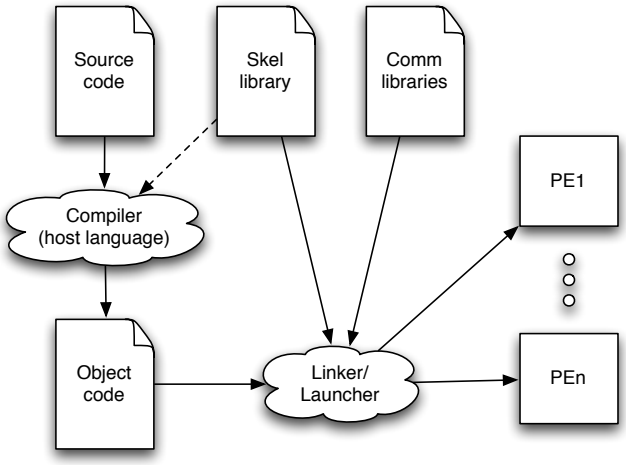


Figure 4.2 Tool chain for skeleton frameworks implemented as libraries

Skeleton programs are written in some "host sequential" programming language (host source code). Skeletons are either declared or simply called via host language library calls. The library implementation, in turn, implements the skeleton functional and parallel semantics. The skeleton library implementation may use different kinds of communications libraries (e.g. MPI, POSIX/TCP, JXTA) as it happens in the case of skeleton frameworks implemented as new programming languages. In this case the tool chain to be implemented looks like the one in Fig. 4.2.

The distinction among "declaration" and "call" of skeletons is worth to be explained in more detail. Several skeleton frameworks (e.g. Muesli[60] and Muskel[14]) *declare* the skeleton program and than a specific statement is inserted in the program to actually compute the skeleton program.

The Muskel code shown in Sec. 3.3.4 shows how this approach works. First a `main` skeleton is declared, using proper composition of objects of class `Skeleton`. Then an `ApplicationMananager` object is used to `start` the computation of the program.

Other skeleton frameworks, instead, use library calls to directly execute skeleton code. The SkeTo code in Sec. 3.3.4 shows an example of map and reduce skeletons which are called for execution. In fact, the philosophy of SkeTo is such that the programmer may intermingle skeleton calls to sequential code as if they were plain, sequential library calls[81, 68].

4.2 TEMPLATE BASED VS. MACRO DATA FLOW IMPLEMENTATION

Two quite general models have been used to implement parallel programming frameworks based on algorithmic skeletons:

- *implementation templates*, and
- *macro data flow*.

The former is the model used in the firsts algorithmic skeleton framework implementations. It has been used later on and it is currently used in the implementation of the vast majority of skeleton based parallel programming frameworks (e.g. in eSkel, Muesli, SkeTo, ASSIST). The latter has been introduced in late '90. It is radically different both in the techniques used to implement the algorithmic skeletons and in the possibilities opened in terms of optimizations, fault tolerance and portability. With several variants, it has been adopted in different frameworks, including Muskel (where it has been developed first), Skipper, Calcium and Skandium[68].

In the following subsections we outline the two implementation technique and we point out the main similarities and differences.

4.2.1 Template based implementations

In a template based implementation of a skeleton framework, each skeleton is eventually implemented by instantiating a "process" *template*[69]. Historically, we use the term *process template* as the typical concurrent entity run in parallel in the first skeletons frameworks was the *process*. Currently, we should better refer to the templates as *concurrent activity templates* to denote the fact the concurrent activities implementing a given skeleton actually may be a mix of processes, threads, GPU kernels etc.

> A process template is a parametric network of concurrent activities, specified by means of a graph $G = (N, A)$. Each node N_i in the graph represents a concurrent activity.

[68] actually both Calcium and its successor Skipper, use an implementation based on evaluation stacks. This notwithstanding the execution mechanism eventually looks like the one used in macro data flow implementations.

[69] In literature, the term "implementation template" has been used as well

An arc $\langle i, j \rangle$ from node N_i to node N_j denotes a communication moving data (or pure synchronization signals) from node N_i to node N_j. Each node has associated an informal "behaviour", that the program executed on that node. Parameters in the specification of a process template vary.

As an example, the parallelism degree of the template is normally a parameter, as well as the business code to be executed on the node or the type and number of the channels (links) insisting on the node.

Each process template is associated with some further metadata. The metadata provide further information needed to make a proper usage of the template. Examples of metadata associated to the template include:

Target architecture : a description of the target architecture(s) were the template may be conveniently and efficiently used.

Skeletons supported : a description of the skeletons that can be possibly implemented in terms of the template.

Performance model : a description of the performance achievable with the template, parametric with respect to both application dependent parameters (e.g. the execution times of the sequential portions of business logic code used in the template) and system dependent parameters (such as communication latency and bandwidth relative to the target architecture).

Let us suppose a process template is described by a tuple such as:

$$\langle name, \{Sk\}, \{Arch\}, PerfMod, G = (N, A) \rangle$$

and that at least one template exist for any skeleton provided by the skeleton framework. Then the compilation of a skeleton application can be obtained as follows:

Phase 1: skeleton tree The source code of the application is parsed and a skeleton tree, such as those described in Sec. 3.3.5, is derived

Phase 2: template assignment The skeleton tree is visited and a template is assigned to each one of the nodes in the tree. The template T_j assigned to a node of type[70] Sk_i is such that

- C1 the template T_j is a template suitable to implement skeleton Sk_i
- C2 the template T_j is suitable to be implemented on the target architecture considered
- C3 possibly, the performance model of the template T_j demonstrates better performance than those of any other template T_i satisfying the conditions C1 and C2 above.

Phase 3: optimization The annotated skeleton tree[71] is visited again and the parameters of the templates are filled up, using either heuristics or the performance models to devise correct actual parameter values.

Phase 4: code generation The "object" code for the parallel application is generated. The object code is usually built of a set of programs in a high level language with calls to suitable communication and synchronization libraries, such as C/C++ with MPI calls or OpenMP pragmas. During code generation usually makefiles and metadata needed for code deployment on the target architecture are produced.

[70] modeling a skeleton
[71] each node is annotated with a template

It is worth pointing out that the very same process may be implemented in a library, rather than in a set of compiling tools. In this case:

- Phase one derives from the fact skeletons are declared as library objects, and therefore there is no need to parse source code to get the skeleton tree
- Phases 2, 3 and 4 are executed *on-the-fly* when the execution of the skeleton application is started, via the proper library call. In general, this means the "execute" call sets up a number of concurrent activities that then specialize according to the different processes needed to implement the templates. As an example, Muesli skeletons are implemented as a library and Muesli uses a template based approach. The skeletons composing the program are *declared* as C++ objects. Then a execution is invoked. The execution starts an MPI run. MPI process identifiers are used to specialize each MPI process to perform as expected by the template used to implement the application skeleton (skeleton nesting).

Considering the whole process leading to the generation of the skeleton application object code we can make the following considerations.

4.2.1.1 Template selection In many cases, only one template–the "best" one–exists for each skeleton supported by the framework. In that case the selection phase is clearly void. When more than a single template exists for a given skeleton and a given target architecture, the selection may rely on information provided by the performance models associated to the template, as we detailed in the phases listed above. Two alternative approaches may be followed, however: i) heuristics may be used (such as "template T_j is preferable in case of nestings, while $T_{j'}$ has to be preferred for standalone skeletons), or ii) a random template may be selected from those available for the skeleton and architecture at hand, but then the optimization phase may substitute the template with another one, if the case. In a sense, this second approach moves duties from the first skeleton visit to the optimization visit.

4.2.1.2 Optimization phase The optimization phase may be relatively large with respect to the other phases. Optimization activities may include computation of the optimal parallelism degree, evaluation of alternative skeleton nestings providing the same semantics with different non functional features, fine tuning of the template compositions. We mention here just three examples of these optimization activities.

Parallelism degree computation. The skeleton tree may be annotated with the templates without actually specifying their parallelism degree. Once the number of processing elements available for the application execution is known, the annotated skeleton tree may be processed to assign part of the available resources to each one of the templates. P3L implemented this phase in two steps: during the first step, the skeleton tree was visited bottom up and each template was assigned an "optimal" number of resources. Optimality was defined with respect to the template performance models. Then a cyclic process was initiated aimed at taking away the minimum amount of resource from the assignment that leave a balanced assignment (i.e. a resource assignment such that there are no bottleneck skeletons). The process was iterated up to the point the number of assigned resources was smaller or equal to the number of available resources. An alternative approach could consist in generating all the feasible assignments of the available resources to the templates in the skeleton tree, then to evaluate all the assignments with the performance models and eventually keep the one(s) getting the best results. This process is much more complex, as *all the possible* assignments are considered. The other alternative (the P3L one mentioned above) only works on "good" assignments: it starts from an optimal assignment and reduces to the number of resources actually available moving through assignments that are not optimal, but keep balanced assignment of resources to the skeleton/template tree assignments.

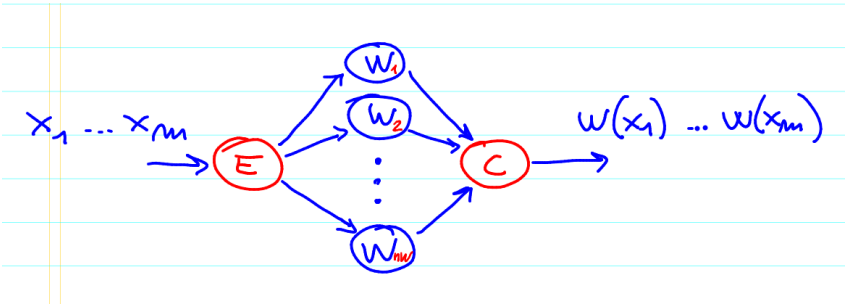


Figure 4.3 Farm template for complete interconnection architectures

Skeleton nesting rewriting. Well know functional equivalences exists between skeleton trees (see also Sec. 10.1). As an example a

```
Pipe(Farm(Seq(s1)),Farm(Seq(s2)))
```

is functionally equivalent to a

```
Farm(Pipe(Seq(s1),Seq(s2)))
```

Now it is clear that optimization phase my decide to rewrite a skeleton tree into a functionally equivalent skeleton tree, provided that some advantage is achieved in terms of non functional properties of the skeleton tree. It is also clear that, for any rule stating the functional equivalence of skeleton tree S_i and S_j, it could be the case that with respect to a non functional concern it is better to use the first form while with respect to another non functional concern it is better to use the second form.

Template composition tuning. As it will be clearer later on, template composition may turn out into having a template T_i delivering data to template T_j such that T_j processes this data with a function $f = f_n \circ \ldots \circ f_1$, T_i delivers data resulting from the application of $g = g_m \circ \ldots \circ g_1$ and $g_m \equiv f_1^{-1}$. A typical example is the case of two templates where the first one builds an array out of the single rows computed in parallel by the template worker processes and the second template unpacks any matrix received in rows to submit the rows to another set of worker process[72]. In this cases, the composition of the two templates may be optimized simply taking away the g_m computation from the first template and the f_1 computation from the second template.

4.2.1.3 Code generation Code generation depends a lot on the target environment considered. Algorithmic skeletons frameworks usually target environments with high level languages and communication libraries (C/C++ with MPI or OpenMP, Java with TCP/IP sockets). In this case, the concurrent activities used to implement the templates are processes or threads whose body may be produced using simple macros as well as instantiating objects and providing the properly wrapped business logic code parameters. In case lower level programming environments are targeted, the code generation may be sensibly larger. As an example, if assembly language + communication libraries were assumed as the target environment, this would require a considerable effort relative to the code generation and optimization.

4.2.1.4 Commonly used templates There are several process templates that are used in different skeleton frameworks. We describe a couple of templates to further clarify the process template concept.

[72] this case may arise when we compute a `Pipe(Map,Map)` application, as an example

Farm template for complete interconnection architectures. A farm skeleton models embarrassingly parallel computation over streams. One common template used to implement the farm skeleton uses an *emitter* process, a *collector* process and a string of *worker* processes (see Fig 4.3).

- The emitter takes care of receiving the tasks from the farm input stream and to schedule them–in a fair way–to the worker processes.
- Worker processes receive tasks, compute them and deliver results to the collector process.
- The collector process simply gathers results from the workers and delivers those results onto the farm output stream.

The performance model of the process template establishes that the farm service time is given by the maximum among the time spent by the emitter to schedule a task to one of the workers, the time spent by the collector to gather a result from one of the workers and to deliver it to the farm output stream and by the time spent by one of workers to process a task divided by the number of the workers in the farm.
This farm template has several parameters:

- the parallelism degree n_w, denoting how many workers are actually included in the process template.
- the scheduling function implemented by the emitter (e.g. round robin or on demand)
- whether or not the input task ordering has to be preserved onto the output stream results (this is possibly implemented by the collector)
- the type of input tasks
- the type of output tasks
- the computation performed by each one of the workers, expressed as another, possibly sequential, skeleton

This template may be used on any architecture supporting full interconnection among processing elements. A "companion" task farm template may be obviously defined using threads instead of processes. In that case, symmetric multiprocessors can be targeted.

Farm template for mesh architectures (version 1). When targeting mesh interconnected architectures, a farm template may be defined in terms of an emitter process, a string of emitter routers, a string of workers, a string of collector routers and a collector process. The interconnection among those elements is such that the mesh interconnection available is used:

- the emitter process is placed at position (i, j) in the mesh
- the n_w emitter routers are placed at positions $(i, j + k)$ with $k \in \{1, n_w\}$
- the n_w worker processes are placed at positions $(i + 1, j + k)$ with $k \in \{1, n_w\}$
- the n_w collector routers are placed at positions $(i + 2, j + k)$ with $k \in \{1, n_w\}$
- the collector process is placed at position $(i + 3, j + n_w + 1)$

as shown in Fig. 4.4. The behaviour of the different processes in the mesh is the following:

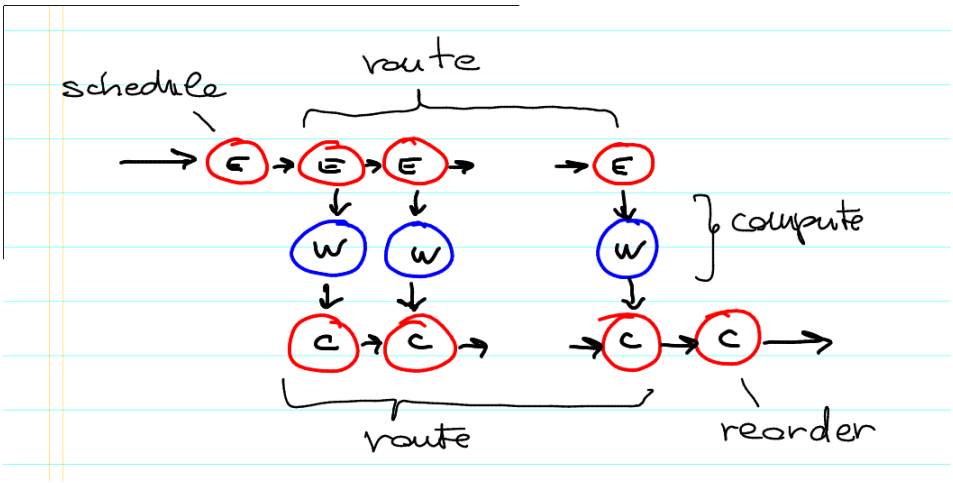


Figure 4.4 Farm template for mesh connected architectures

- the emitter process receives tasks from the task input stream and outputs passes these tasks to the first emitter router
- emitter router of index i receives $n_w - i$ tasks and passess them to the next emitter router. Then it receives another tasks and passes this tasks to the worker of index i. Eventually it starts again receiving and passing tasks.
- worker processes receive tasks to be computed from the emitter routers, compute them and deliver tasks to the corresponding collector routers
- collector routers receive tasks either from the upper worker or from the left collector router and deliver them to the right collector router (or to the collector, in case the router is the last one in the chain)
- the collector process receives results from the n_w collector router and delivers these results to the farm output stream.

The performance model of this template states that its service time is

$$T_s = max\{T_e, \frac{T_w}{n_w}, T_c\}$$

where T_e T_w and T_c are respectively the time spent to route a task, to compute a task or to route a result. The parameters of the template include the parallelism degree n_w as well as the parameters related to the computation performed: types of tasks and results, skeleton to be computed in the workers.

Farm template for mesh architectures (version 2). This second template is built of a number of processes arranged in a pipeline. Process i receives data from process $i-1$ and delivers data to process $i+1$. Each process behaves as follows:

- it receives $n_w - i$ tasks, and passes these tasks to next process, then
- it receives one task, computes it and passes the results to next process
- it receives $i-1$ results from previous process and delivers these results to next process

This template works in three phases: during the first phase (taking $n_w T_e$ units of time) all processes receive a task to compute; in the second phase, all processes compute a task

(spending T_w units of time); in the last phase all processes route results to the last process (in $n_w T_c$ units of time), the one in charge of delivering results onto the farm output stream. As a result the template processes n_w tasks in $n_w T_e + T_w + n_w T_c$ units of time thus delivering a service time of approximately

$$T_S = \frac{n_w T_e + T_w + n_w T_c}{n_w}$$

Taking into account that the time spent to route a task or a result (T_e or T_c) is small, the service time in this case can be further approximated with

$$T_S = \frac{n_w T_e}{n_w} + \frac{T_w}{n_w} + \frac{n_w T_C}{n_w} \cong \frac{T_w}{n_w}$$

4.2.2 Macro Data Flow based implementation

In late '90, an alternative implementation method for programming frameworks providing algorithmic skeletons has been introduced [43]. Rather than compiling skeletons by instantiating proper process templates, the skeleton application are first compiled to *macro data flow graphs* and then instances of this graphs (one instance per input data set) are evaluated by means of a distributed macro data flow interpreter.

Let's start defining macro data flow graphs.

> *A macro data flow grap (MDFG) is a graph $G = (N, A)$ whose nodes N are macro data flow instructions (MDFI). A macro data flow instruction, in turn, is a tuple*
>
> $\langle id, f, tokens, dests \rangle$
>
> *where*
>
> - *id is the instruction identifier,*
> - *f represents the function computed by the instruction on the input tokens,*
> - *tokens is a list of $\langle value, presencebit \rangle$ pairs, whose value field will eventually host some data coming from another MDFI and presencebit is a boolean stating whether or not the value item is significant (has already been received),*
> - *dests is a list of $\langle i, id, pos \rangle$ representing the position pos in the the token list of MDFI id to be filled by the i-th result produced from the computation of f on the $n = len(tokens)$ values of the input tokens $f(value_1, \ldots, value_n)$*
>
> *The arcs of the MDFG G, $a_i = \langle s_i, d_i \rangle \in A$, model the fact that a data item (output token) produced by MDFI with $id = s_i$ is directed to a MDFI whose $id = d_i$.*
> *A input token assignment of a MDGF G is a set of $\langle value, dest \rangle$ pairs, hosting the data values used to initialize the dest tokens in the graph.*

This is nothing different from usual data flow graphs used in sequential compilers to conduct data flow analysis. The term *macro* distinguish our graphs as having instructions whose function f is a coarse grain computation, while data flow analysis often use fine grain data flow instructions (e.g. instructions whose functions represent simple arithmetic operators). In case of macro data flow instructions used to support skeleton implementation, the functions used are typically those computed by the sequential business code wrapped into skeleton parameters.

The computation of a MDFG G with an input token assignment ITS proceeds as follows:

- the data items in the input token assignment are moved to the tokens identified by their *dest* fields.
- while *fireable* instructions exists in the MDFG, these instructions are executed and their results are directed to the proper token destinations. A fireable instruction is a MDFI whose input tokens have all been received, that is, have all *presencebit*s true.

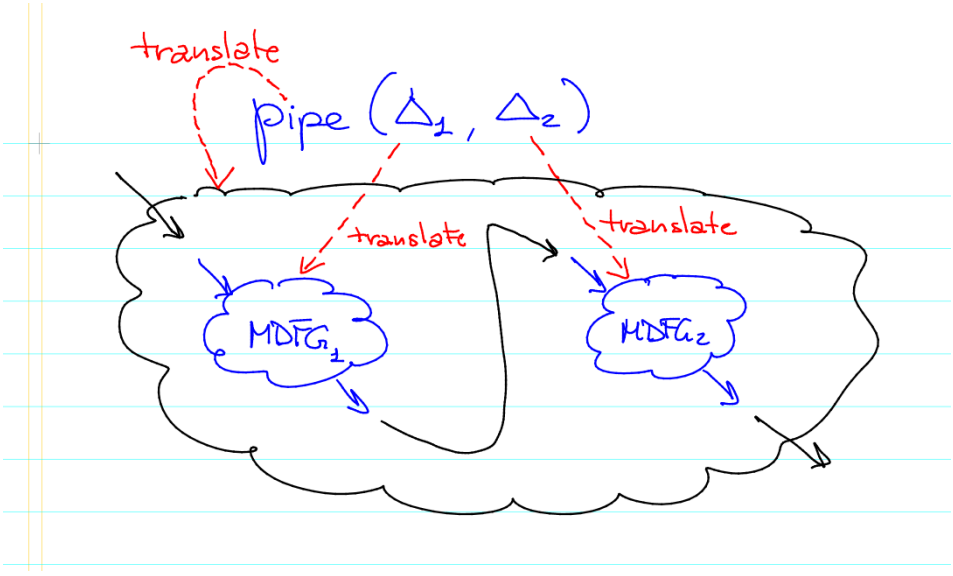


Figure 4.5 Recursive compilation of a pipeline skeleton to MDF graphs

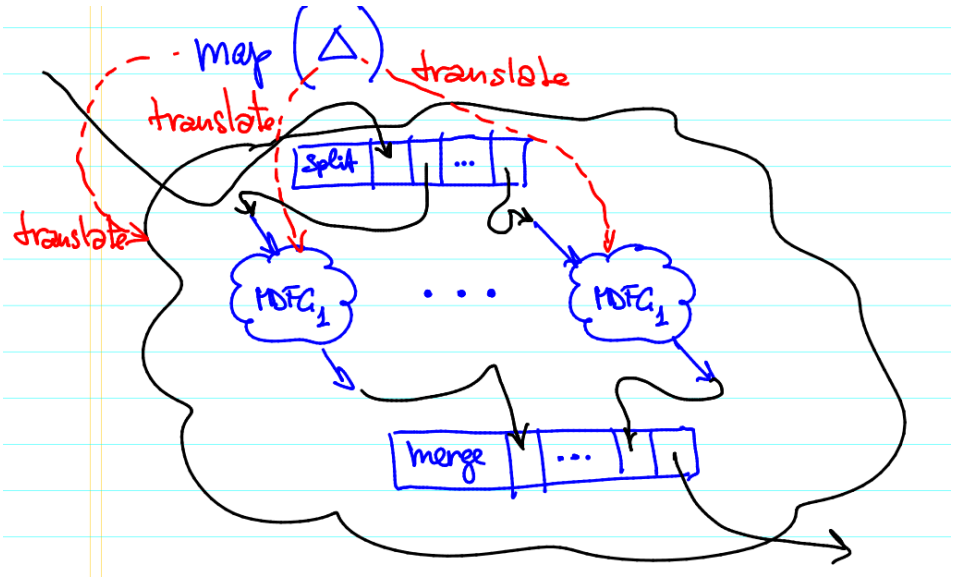


Figure 4.6 Recursive compilation of a map skeleton to MDF graphs

When a macro data flow implementation is used, each skeleton is associated with a MDFG. That graph implements the parallelism exploitation pattern embedded in the skeleton. As an example, a pipeline skeleton `pipe(S1,S2)` will be associated with a two instruction MDFG: the first instruction, with a single output token computes the function denoted by `S1` and delivers the result to the second instruction. This latter instruction has again a single input token and computes the function denoted by `S2`. In case `Si` is actually a sequential skeleton `seq(Ci)`, the wrapped sequential code `Ci` will be used to compute the function of the corresponding MDFI. In case `Si` is represented by another skeleton (composition), then recursively the instruction computing `Si` will be recursively substituted by the graph associated to the skeleton (composition) denoted by `Si`.

This process actually defines a formal recursive procedure translating a skeleton tree into a MDFG:

- **translate(Seq(Code))** returns a single MDFI computing the function defined in terms of **Code**. The instruction has as many input tokens as the input parameters of the function defined by **Code**, and has many destinations as the output parameters (results) defined by **Code**

- **translate(Skeleton(Param1,...,ParamN))** returns a graph G including as sub graphs (one or more instances of) the graphs generated by **translate(Param1)** to **translate(ParamN)** properly arranged according to the semantics of **Skeleton** (see Fig. 4.5 and Fig. 4.6)

Once the translation of a skeleton tree to a MDFG has been defined, we can define how a skeleton application may be executed:

- the application is compiled with the **translate** function defined above and a MDFG G is obtained, then *concurrently*:

  - *Input data management* For each input data item appearing on the input (stream) of the skeleton application an input token assignment is derived and a copy of the graph G is instantiated with that assignment. Then a "fresh"[73] graph identifier gid is computed and all the instructions in G are modified:

    * their identifier id is substituted with the pair gid, id
    * all destination $\langle id, pos \rangle$ in all instructions are substituted by $\langle gid, id, pos \rangle$

    The resulting graph G' is inserted into a *MDFG repository*.

  - *Computation* The MDFG repository is scanned and the fireable instructions in the repository are concurrently executed by means of a *concurrent macro data flow interpreter*. Results computed by the fireable MDFI are either placed in the proper tokens of the MDFG repository MDFIs or–in case of special "output" destinations stored in *dest* fields of a MDFI–passed to the output data management.

  - *Output data management* The final results computed by the concurrent MDF interpreter are delivered to application output–screen, disk file, etc.

This way of implementing skeleton frameworks has a clear advantage with respect to the template based one. The process of implementing skeleton programs is split in two parts: first, we devise efficient ways of compiling skeletons to MDFGs. Then we provide an efficient way of executing instantiated MDFGs on the target architecture at hand. As a consequence:

- The two steps may be separately optimized.

- Porting the skeleton framework to a different target architecture only requires porting the concurrent MDF interpreter.

- Introduction of new skeletons only require the definition of the proper translation of the skeleton to MDFGs and it can be easily integrated in an existing framework by properly extending[74] **translate** function.

[73] never user before
[74] i.e. adding a `case` clause in the **translate** implementation switch

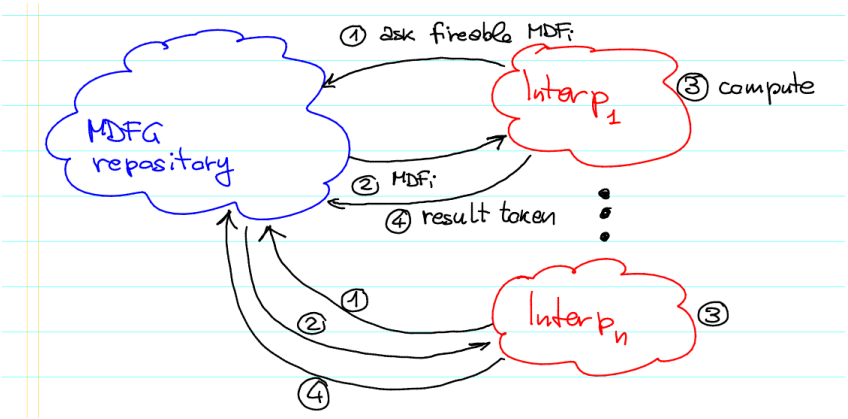


Figure 4.7 Concurrent MDF interpreter logical design

4.2.2.1 Concurrent MDF interpreter Clearly, the critical component of the skeleton framework implemented using MDF is the concurrent MDF interpreter. In order to be effective:

- The interpreter must include a suitable number of concurrent activities (*concurrent interpreter instances*), each being able to "interpret" (that is to compute) any of the MDFI appearing in the MDFG repository. This number represents the parallelism degree of the MDF interpreter and *de facto* establishes a bound for the maximum parallelism degree in the execution of the skeleton application.

- Each concurrent interpreter instance must be able to access the logically global, shared MDFG repository both to get fireable MDFI to compute and to store the computed output tokens.

- The logically global and shared MDFG repository must be implemented in such a way it does not constitute a bottleneck for the whole system. When a reasonably small number of concurrent interpreter instances is used, the implementation of the MDFG repository may be kept centralized. When larger numbers of concurrent interpreter instances are used, the MDFG repository itself must be implemented as a collection of concurrent and cooperating sub repositories.

The logical architecture of the concurrent MDF interpreter is outlined in Fig. 4.7.

4.2.2.2 Optimizations The efficient implementation of a distributed/concurrent (macro) data flow interpreter is as challenging as the efficient implementation of a template based skeleton framework. Several different optimizations may be introduced, aimed at improving the efficiency of the interpreter:

Scheduling optimizations Instructions scheduled on remote nodes include the input tokens to be processed. In case of large tokens, scheduling of MDFI to the concurrent interpreters may include special output tokens (dest) flags, telling the concurrent interpreter to maintain a copy of the result token and to send a unique *result identifier* as a result token instead of (or in addition to) sending the data back for proper delivery to the next MDFI input token. When an instruction is scheduled with this result in the list of its input tokens, it is preferably scheduled on the concurrent interpreter instance already having the input token data and therefore the communication overhead is reduced. In case of frameworks targeting shared memory architectures, such

as SMP multicores, the same technique may be used to direct computations to cores already having in their caches a valid copy of the input tokens[75].

MDFG linear fragment scheduling In case a linear chain of MDFI is present in a MDFG, once the first MDFI of the chain has been scheduled on concurrent interpreter i, all the subsequent instruction of the chain are also scheduled to the same concurrent interpreter. This allows copies of output tokens to the MDFG repository and subsequent copies of input tokens from the repository to the concurrent interpreter to be avoided.

4.2.3 Templates vs. macro data flow

Template and macro data flow implementation of skeleton frameworks both show pros and cons:

- Template based implementations achieve better performances when computing tasks with constant execution time, as the template based implementation has less run time overhead with respect to the macro data flow based implementations. On the other side, computations with tasks with highly variable computing times are more efficient on macro data flow based implementations as these implementations sport better[76] load balancing properties with respect to template based implementations.

- Porting of macro data flow based frameworks to new target architectures is easier than porting of template based implementations. In fact, MDF implementation porting only requires the development of a new MDF concurrent interpreter, whereas porting of template based implementations requires a complete rewriting of the skeleton template library, including performance models of the templates.

- Template based implementation are more suited to support static optimization techniques, such as those rewriting skeleton trees, as a substantial compile phase is anyway needed to implement skeletons via templates. The very same optimizations in MDF based implementations require some additional effort due to the fact the run time has actually no more knowledge of the skeleton structure of the program it is executing: it just sees MDFI from a MDFG.

4.3 COMPONENT BASED SKELETON FRAMEWORKS

Algorithmic skeletons may obviously be interpreted as building blocks for parallel applications. Therefore it is natural to try to migrate the algorithmic skeleton concept to the software component scenario.

Software component are usually intended as black boxes with a well defined interface defined in terms of provide ports–ports providing some kind of service–and use ports–ports declaring some kind of needed, external service. Each software component has associated–mostly informally–a semantics, defining the kind of services provided by provide ports in terms of the services used through the use ports. A framework supporting software components supports application development via composition of system-defined as well as user-defined components. The user picks-up the components modelling the blocks of his application and eventually links those components in a component "assembly" (a graph of components, in the software component jargon) by properly pairing provide and use ports.

[75] in both cases, we refer this technique as *affinity scheduling*
[76] an also achieved more easily

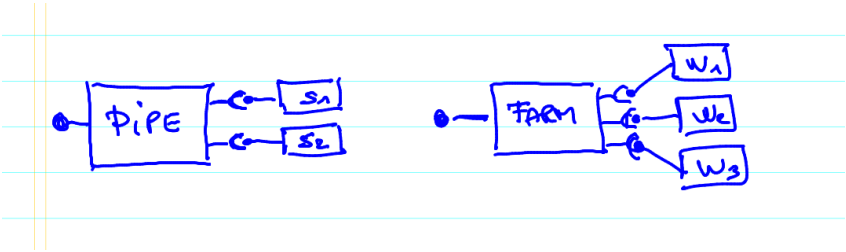


Figure 4.8 Sample skeleton components (not composite)

Any algorithmic skeleton can therefore be viewed as a *parallel component providing* as a service the higher order function modelled by the skeleton and *using* other the services provided by components modelling its skeleton parameters.

The simpler skeleton component is clearly the one wrapping some legacy code. It only has a provide port. Through this port, a computation of the wrapped code onto some kind of input parameters may be asked. The computation happens sequentially. Then other skeleton components may be defined. As an example, a pipeline component will have one provide port and two use ports, as depicted in Fig. 4.8 (left). Or a farm component may be defined with one provides port and a number of use ports, each interfacing the farm component with a worker component, as depicted in Fig. 4.8 (right).

These components modelling algorithmic skeletons may be used as other more classical components within the software component framework, provided the component framework supports compositionality. In this case, the composite component pipeline is perceived as a monolithic component from ay other component invoking the services provided by its provide ports. Only the system programmer implementing the skeleton component actually perceive the structured internals of the skeleton components.

Skeleton frameworks have rarely been provided as software component frameworks, with a notable exception: GMC Behavioural Skeletons (GCM/BS see Sec. 10.5). GCM stands for Grid Component Model and it is a component model specially designed to support grid computing within the CoreGRID project[77]

GCM/BS exploited all the positive features of a software component framework to provide the programmer of parallel component applications with skeleton composite components (the behavioural skeletons). These components not only implemented all the parallelism exploitation details related to the modelled skeleton in way that turns out to be completely transparent to the component application programmer, but also included advanced features (see Chap. 10) that allowed non functional concerns such as performance tuning and optimizations to completely managed by the skeleton component, without any kind of support/intervention require to the component application programmer.

[77]CoreGRID is an EU funded Network of Excellence (NoE) that run in the years from 2005 to 2008 (see `www.coregrid.net`). GCM has been developed within the Programming model Institute of CoreGRID. The reference implementation of GCM has been developed within the GridCOMP EU funded project (`gridcomp.ercim.org`) on top of the ProActive middleware.

CHAPTER 5

PERFORMANCE MODELS

When dealing with parallel computations we are in general interested in achieving "the best performance" possible. This expectation may be read in several different and complementary ways:

- First of all, we are interested in knowing which kind of performance we can achieve when using a given parallelism exploitation pattern–or skeleton composition–*before* actually starting writing application code. In case expected performance of our pattern doesn't match our requirements we can simply avoid to go on implementing a useless application.

- In case several different[78] implementations are possible, we are interesting in knowing which one will provide the better performances. Alternatively, we may be interested in knowing which ones will eventually provide performance figures matching our performance requirements, in such a way we can go on implementing the "simplest"–e.g. the less expensive in terms of coding effort–one.

- Last but not least, we may be interested in knowing how much overhead is added by our implementation, that is how much the measured performance will eventually differ from the theoretic one.

In all cases, we need to have some kind of *performance model* of our application available. The performance model measures the performance of an application on a given target machine in function of a full range of parameters depending either on the application features (parallelism exploitation patterns used, time spent in the execution of the sequential portions of code, dimensions of the data types involved, etc.) or on the target machine features

[78] different as far as parallelism exploitation is concerned, but functionally equivalent

(communication times, processing power of the different processing elements, topology of the interconnection network, features of the (shared) memory subsystem, etc.).

In other words, we need a function $\mathcal{P}$ mapping values of parameters $p_1, \ldots, p_n$ to values of some performance indicator. More precisely we are interested in a set of functions $\mathcal{P}_1, \ldots, \mathcal{P}_h$ each modeling a different performance indicator (e.g. service time, latency, completion time, etc.).

Once this functions have been defined, we can use these performance model functions to

- *predict* performance of an application,
- *compare* alternative implementations of an application, or
- *evaluate* the overheads of an implementation of an application.

Of course, the evaluation of the overheads in the implementation of an application also requires some methods to *measure* the performance values achieved on the target architecture (see Sec. 5.6).

5.1 MODELING PERFORMANCE

We start taking into account which kind of performance indicators are useful. When dealing with performance of parallel applications we are in general interested in two distinct kind of measures:

- those measuring the absolute (wall clock) time spent in the execution of a given (part of) parallel application, and
- those measuring the throughput of the application, that is the rate achieved in the delivering of the results.

The first kind of measures include measures such as

Latency (L) the time spent between the moment a given activity receives input data and the moment the same activity delivers the output data corresponding to the input.

Completion time (T_C) the overall latency of an application computed on a given input data set, that is the time spent from application start to application completion.

The second kind of measures include measures such as

Service time (T_S) the time intercurring between the delivery of two successive output items (or alternatively, the time between the acceptance of two consecutive input items), and

Bandwidth ($\mathcal{B}$) the inverse of the service time.

These are the basic performance measures of interest in parallel/distributed computing. Applications with a very small service time (a high bandwidth) have a good throughput but not necessarily small latencies. If we are interested in completing a computation within a deadline, as an example, we will be interested in the application completion time rather than in its service time.

However, when modeling performance of parallel applications we are also interested in some derived[79] performance measures such as:

[79] i.e. non primitive, expressed in terms of other base measures

Speedup ($s(n)$) the ratio between the *best known* sequential execution time (on the target architecture at hand) and the parallel execution time. Speedup is a function of n the parallelism degree of the parallel execution.

$$s(n) = \frac{T_{seq}}{T_{par}(n)}$$

(here we denote with T_{seq} the best sequential execution time and with $T_{par}(n)$ the parallel execution time with parallelism degree equal to n).

Scalability ($scalab(n)$) the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

$$scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

Both these measures may be computed w.r.t. service time or latency/completion time, that is the T_x in the formulas may be consistently instantiated by service times or by completion times, provided we do not mix the two.

Another "derived" measure useful to model performance of parallel computations is efficiency:

Efficiency the ratio between the ideal execution time and actual execution time. Efficiency is usually denoted with ϵ.

$$\epsilon(n) = \frac{T_{id}(n)}{T_{par}(n)}$$

As the ideal execution time may be expressed as the sequential execution time divided by parallelism degree[80]

$$T_{id}(n) = \frac{T_{seq}}{n}$$

then we can express alternatively efficiency as follows:

$$\epsilon(n) = \frac{T_{seq}}{nT_{par}(n)} = \frac{s(n)}{n}$$

5.1.1 "Semantics" associated with performance measures

Each performance measure has an associated "semantics".

L, T_C the latency and the completion time represent "wall clock time" spent in computing a given parallel activity. We are interested in minimizing latency when we want to complete a computation as soon as possible.

T_S, $\mathcal{B}$ service time represents (average) intervals of time incurring between the delivery of two consecutive results (acceptance of two consecutive tasks). We are interested in minimizing service time when we want to have results output as frequently as possible but we don't care of latency in the computation of a single result. As bandwidth is defined as the inverse of the service time, we are interested in bandwidth maximization in the very same cases.

[80] this is the best we can do with n parallel "executors", actually: evenly divide the overall work among the parallel executors, without adding any kind of additional overhead.

$s(n)$ Speedup gives a measure of how good is our parallelization with respect to the "best" sequential computation. Speedup asymptote is given by $f(n) = n$. This because, if $s(n) > n$ then the portions of the algorithm executed in parallel on the n processing elements will take a time $t_p < \frac{T_{seq}}{n}$ and therefore we could have performed sequentially the n computations taking t_p on a single processing element, taking an overall time $t = nt_p < n\frac{T_{seq}}{n} < T_{seq}$ which is in contradiction with the assumption T_{seq} was the "best" sequential time. Therefore we will never expect to have speedups larger than the parallelism degree. Actually, there are several reasons that make possible the achievement of *super linear speedups*[81]. In parallel computing we will be constantly looking for higher speedups, of course.

$scalab(n)$ Scalability looks slightly different than speedup. It measures how efficient is the parallel implementation in achieving better performances on larger parallelism degrees. As for the speedup, the asymptote is $f(n) = n$ for the very same reasons. However, scalability does not provide any comparison with the "best" sequential computation. In fact, we can have optimal scalability and still spend more time than the time spent computing sequentially the same program for any reasonable value of n.

$\epsilon(n)$ Efficiency measures the ability of the parallel application in making a good usage of the available resources. Being the speedup bound by $f(n) = n$ the efficiency is bound by $f(n) = 1$. It must be clear that if we have two parallel implementations I_1 and I_2 of the same application with efficiencies ϵ_1 and ϵ_2, provided that $\epsilon_1 > \epsilon_2$ we can't conclude that implementation I_1 is "better" than implementation I_2. This is for sure true if "better" refers to efficiency. But if we are interested in completion time, as an example, it may be the case that the more efficient implementation also gives a larger completion time. We should not forget that a sequential implementation has $\epsilon(n) = 1$. In fact in this case $n = 1$ and therefore

$$\epsilon(1) = \frac{T_{seq}}{nT_{seq}} = \frac{T_{seq}}{1T_{seq}} = 1$$

Of course, if ϵ_1 and ϵ_2 refer to the same parallelism degree n, then $\epsilon_1 > \epsilon_2$ actually means that I_1 is "better" than implementation I_2. In fact, in this case we have

$$\epsilon_1 > \epsilon_2$$

$$\frac{T_{seq}}{n \times T'_{par}(n)} > \frac{T_{seq}}{n \times T''_{par}(n)}$$

$$\frac{1}{T'_{par}(n)} > \frac{1}{T''_{par}(n)}$$

$$T'_{par}(n) < T''_{par}(n)$$

5.2 DIFFERENT KIND OF MODELS

When using performance models with skeleton based parallel programming frameworks, we are obviously interested in deriving *usable* performance models of the skeletons provided

[81] as an example, this may be due to the fact that input/processed data does not fit cache sizes in sequential execution while it fits in each one of the parallel activities in which the sequential program is eventually split. The gain due to the cache hits substituting the main memory accesses could be larger than the overhead necessary to set up the parallel activities and to distribute then the data partitions.

by the framework. These performance models will be hopefully *compositional* in such a way the performance of a given application expressed in terms of a skeleton nesting can be derived by properly composing the performance models of the skeletons appearing in the nesting.

The skeleton performance models, in general, are strictly dependent on the implementation of the skeletons on the target architecture. In turn, the implementation of a skeleton will probably have itself a performance model.

Therefore we can refine our concept of "skeleton performance model" by distinguishing between several different performance models:

Skeleton performance models These are *abstract* models of the theoretic peak performance we can get out of a skeleton. They are abstract in that they do not depend on the implementation of the skeleton but rather depend only on the parallel features of the parallelism exploitation pattern modeled by the skeletons.

E.G.▷ A performance model of the pipeline service time states that the service time of the pipeline is the maximum of the service times of the pipeline stages, as it is known that the slowest pipeline stage sooner or later[82] becomes the one behaving as the computation bottleneck.
If we consider a task farm, instead, the service time of the farm is given by the service time of the workers (T_w) in the farm divided by the number of workers (n_w), as hopefully n_w results will be output by the workers every T_w units of time. ——— ■

These performance models only take into account "abstract" measures, such as the service time of a stage or of a worker. They to not take into account overheads introduced by communications between pipeline stages, for instance. In other terms, they do take into account actual implementation of the skeletons. They model the theoretical performance provided by the skeleton.

Template performance models These are much more concrete models that express *the performance of a template implementing a skeleton on a given target architecture.* As we are considering implementation on a particular target architecture these performance models assume to know details (parameters, actually) modeling typical mechanisms of the target architecture, such as latency and bandwidth of communications, of accesses to memory, etc.

E.G.▷ The template performance model of a pipeline on a complete interconnection target architecture (e.g. a cluster of workstations with an Ethernet interconnection network) takes into account the communication overhead in addition to the other parameters taken into account by the pipeline skeleton performance model. Therefore, while stating that the service time of the template is the maximum of the service times of the pipeline template stages, the service time of each stage is given[83] by the latency of the stage *plus* the time spent to complete the communication needed to receive the input data *plus* the time spent to complete the communications needed to deliver results to next pipeline stage. Now let us assume that our pipeline has two sequential stages only with latencies L_1 and L_2 respectively. And let us assume that communication of input data to stage one takes t_{c_1}, of output from stage one to stage two t_{c_2} and of final results t_{c_3}. Then the *skeleton performance model* for service times will state that[84]

$$T^{sk}_{S_{pipe}}(2) = max\{T_{S_1}, T_{S_2}\} = max\{L_1, L_2\}$$

[82] depending on the amount of buffering among the different pipeline stages
[83] we assume to have sequential stages only
[84] service time of a sequential stage is equal to its latency

whereas the template performance model for the linear chain process template will state that

$$T^{templ}_{S_{pipe}}(2) = max\{(T_{S_1} + t_{c_1} + t_{c_2}), (T_{S_2} + t_{c2} + t_{c_3}) > T^{sk}_{S_{pipe}}(2)$$

and therefore efficiency of the template is smaller than the efficiency (theoretic) of the skeleton.

Now, in case the target architecture is such that communication and computation times may be overlapped[85] we will have

$$T^{templ}_{S_{pipe}}(2) = T^{sk}_{S_{pipe}}(2) = max\{L_1, L_2\}$$

as the communication costs are negligible[86], in this case.

Last but not least, consider what happens in case the template is used in a context where there is no chance to have two distinct processing elements available to support the execution of the two stages. In this case, the template concurrent activities may be "collapsed" on a single processing element and the consequent template performance model will state that:

$$T^{templ}_{S_{pipe}}(1) = T_{S_1} + T_{S_2}$$

or, in case communications are exposed due to impossibility to overlap communications with computation

$$T^{templ}_{S_{pipe}}(1) = T_{S_1} + T_{S_2} + t_{c_1} + t_{c_3}$$

■

In addition to the performance models related to parallelism exploitation–skeleton and template models–performance models are also needed to model target architectures. Therefore we introduce:

Architecture performance models These are the models exposing *the time spent in the different mechanisms implemented by the target architecture* to the template performance models. Typical architecture performance models are those modeling inter-processing element communication or relative power of processing elements.

E.G.▷ Communication costs may be modeled as a linear combination of setup time and message transfer time. With quite significant approximations, communication cost of a message having length l on an distributed architecture with no overlap among communication and computation, may be modeled as:

$$T_{comm}(l) = T_{setup} + \frac{l}{B}$$

being B the bandwidth of the communication links.

The same mechanisms, when considering many/multi cores with communications implemented in memory may be modeled as

$$T_{comm}(l) = T_{synchr}$$

if and only if the communication fits a single producer single consumer schema. In this case, one thread writes the message in main (shared) memory, and another thread reads the message. The producer and consumer threads must synchronize in such a way consumer does not consume "empty" messages and producer does not send messages in buffers that still hold the previously sent message. However, if the communication schema fits a different schema, the model changes. As an example, when a single produce multiple consumer schema is used, then additional synchronization cost has

[85] this happens when interconnection network is implemented through dedicated "communication processors", as an example. In that case, the communication time spent on the processor is negligible as the only cost paid is the offloading (to the communication processor) of the message descriptor.

[86] masked by the computation times

to be added to take into account to model contention among consumers to read the message. In general, these additional synchronization costs are proportional to both the length of the message and to the number of consumers involved. ——— ■

5.3 ALTERNATIVE APPROACHES

When developing any performance model we are faced to the problem of deciding the *level of accuracy* of the models. Depending on the number of the parameters taken into account, as well as on the final "form" of the performance models looked for, different results may be achieved. In particular, different approximation levels may be achieved, that is different accuracies may be achieved in the predicted results.

Most of the skeleton based programming frameworks developed so far considered performance models sooner or later. Some of them took into account performance models "ab initio", either to support template assignment or to compare achieved performances with the theoretic ones. Others only introduce performance models after an initial development that did not take them into account. Often, models are introduced and used to understand what's going wrong with templates that do not sport the expected behaviour.

In most cases, anyway, one of two distinct approaches is followed in the design of the performance models:

Approximate model approach In this approach, most of the peculiar features of the skeleton, template or architecture[87] are "summarized" in global parameters or neglected at all. The result consists in quite simple models (performance formulas) than may indeed capture all the details actually impacting the performance modeling process.

Detailed model approach This alternative approach, instead, tries to model any detail of the skeleton, template or architecture as accurately as possible. The result is usually a quite complex formula computing the performance value out of a large number of parameters.

Both approaches have been followed in the past. The former promises to provide simpler performance modeling and therefore to enhance the possibilities to exploit these models in different contexts. The latter promises to be much more accurate at the expense of models much more "difficult" to compute[88].

Unfortunately, very often the detailed model approach explodes both in terms of the parameters considered and in terms of the complexity of the final models. In addition, the detailed model approach requires much more detailed verification and validation activities than the approximate model approach. Eventually, the approximate model approach is the one currently being followed in the waste majority of cases when performance models are considered and used.

E.G.▷ Let us consider a classical farm template consisting of an *emitter/collector* process and of a string of n_w worker processes. The emitter/collector process gathers input tasks from the input stream and schedules these tasks for execution on one of the available workers. Workers, in turn, receive tasks to be computed, compute them and send back to the emitter/collector process the results. The emitter/collector process–concurrently to its task scheduling activity–collects results, possibly re-order results in such a way the input task order is respected in the output result sequence, and eventually delivers such re-ordered results onto the farm output stream. This template is often referred to as *master-worker pattern* although–in

[87] depending on the model considered
[88] although this "difficulty" could appear negligible with respect to the complexity of the "optimization" process of parallel programs

our perspective–this is actually a template suitable to implement both the task farm and map/data parallel skeletons.

In order to model service time of the master/worker template following the "approximate" model approach we ignore as much details as possible while observing that:

- the emitter/collector is actually the result of merging of task scheduling and result collection activities
- the string of worker processes dispatch an overall service time of $\frac{T_w}{n_w}$, being T_w the service time of the single worker.

The three activities–task scheduling, task computation and results gathering and dispatching–happen to be executed in pipeline. Therefore the service time of the master worker will be the maximum of the service times of the three stages. However, first and third stages are executed on the same processing element (the one hosting the emitter/collector concurrent activity). Therefore the service time eventually will be modeled (and approximated!) by the formula

$$T_S(n) = max\{\frac{T_w}{n_w}, (T_e + T_c)\}$$

possibly complicated by taking into account communication times.

If we follow a "detailed model" approach, instead, we should start considering that the emitter/collector spends a time composed by 2 task communication times (one to receive a task from the input stream and one to deliver it to one of the workers), plus a scheduling time (to decide which worker should be given the task), plus 2 result communication times (one to receive the result from a worker and the other one to deliver the result to the output stream). In turn, these times will be modeled by using a kind of "setup + transmission time" expression. Then we should consider that also service time of the worker includes a task and a result communication time. Eventually we should evaluate how these expressions relate to the overall functioning of the master worker template, that is to reason about matching to be guaranteed between task scheduling, computation and result gathering times in order to avoid bottlenecks in the master/worker computation. Possibly a much more complicated formula will be obtained with a structure similar to the one listed above relative to the "approximated" approach.

■

5.4 USING PERFORMANCE MODELS

Let us make the following assumptions:

- for each skeleton Sk_i provided by our programming framework, we have available a set of skeleton performance models

$$\mathcal{M}^{Sk_i} = \langle T_S^{Sk_i}(n), T_C^{Sk_i}(n), s^{Sk_i}(n), \epsilon^{Sk_i}(n), \ldots \rangle$$

- for each template Tpl_i suitable to implement skeleton Sk_i we have a similar set of template performance models

$$\mathcal{M}^{Tpl_i} = \langle T_S^{Tpl_i}(n), T_C^{Tpl_i}(n), s^{Tpl_i}(n), \epsilon^{Tpl_i}(n), \ldots \rangle$$

- for each measure of interest M_i and for each target architecture targeted A_j we have a model

$$M_i^j = f_i^j(p_1, \ldots, p_k)$$

where p_k are the base parameters modeling the target architecture (e.g. communication setup time, communication bandwidth, etc.).

We already stated in the introduction section of this chapter that these performance models will be eventually used to predict performance of applications, to compare different implementations of the same application or to evaluate how much an achieved performance is close to the ideal one.

In the the following sections, we discuss how these activities exploit performance models at compile time, at run time, or post-run time.

5.4.1 Compile time usage

At compile time different usages can be made of the performance models:

- in case of template based skeleton framework implementation performance models are used to *compare alternative* template-to-skeleton assignment, as explained in Sec. 4.2.1
- again, with template based implementations performance models may be used to *predict* the impact of accelerator usage (e.g. the impact of using a GPU to implement a map skeleton). This could be also considered a variant of the "compare alternatives" usage, as *de facto* this corresponds to evaluate whether the performance achieved when using the accelerator turns out to be better than the one achieved when the accelerator is not actually used.
- when using a MDF implementation, performance models may be used to decide which kind of tasks are worth to be delegated for concurrent execution to a concurrent MDF interpreter instance. As a matter of fact, tasks that require a computation time[89] smaller than the time required to communicate the task to the remote interpreter and to retrieve the result back from the remote interpreter are not worth to be scheduled for concurrent execution.
- again, when using a MDF based implementation, the performance models may be used to fine tune the template actually used to implement the distribute MDF interpreter.

However, the main and more important usage of performance models consists in automatically deriving, without any kind of application programmer support, unspecified non functional parameters relative to the execution of the skeleton/template at hand.

We stated in Sec. 3.2 that some non functional parameters are requested, in some skeleton based programming frameworks, that could be derived automatically[90]. The optimal parallel degree of a template is one of such parameters.

In case we are implementing a parallel program modeled by a single skeleton, the optimal parallelism degree may be obtained as follows:

s1 We start considering the number of processing elements available P. We identify this as the maximum parallelism degree of our application.

s2 We evaluate the performance model of interest $\mathcal{P}(p)$ for the all parallelism degrees $p \in [0, P]$.

s3 We assign the template the parallelism degree $p_i \mid \forall j \neq i \ \mathcal{P}(p_j) \leq \mathcal{P}(p_i)$

Although the process is linear in the amount of processing elements/resources available, the whole computation will not take a significant time, provided the formula expressing $\mathcal{P}$ is reasonably simple *and* P is not huge.

[89] on the local node, that is on the processing element in charge of scheduling the task to a remote interpreter
[90] we mentioned the parallelism degree, as an example

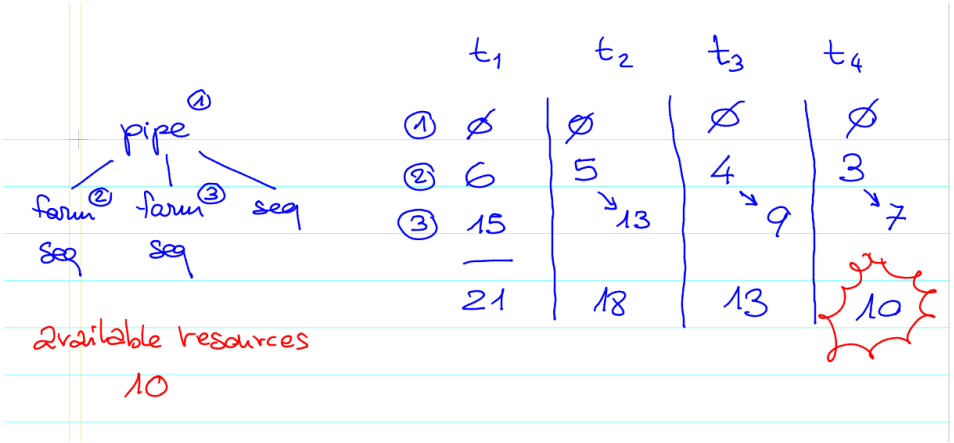


Figure 5.1 Computing resource assignment in a skeleton tree. Initially the two farms are given 5 and 15 processing elements. Then we visit the skeleton tree bottom up and we start taking away one processing element from the first farm. According to the pipeline performance model this allows to take away 2 resources from the second farm. The process is repeated until the number of required resources happens to be smaller than the number of the available resources.

Alternatively, we can "solve" the performance model in such a way the exact values of the parameters maximizing the performance with the given resource constrains are found.

In case our application is modeled using a skeleton nesting, a process such as the one described in Sec. 4.2.1 may be used instead. In this case, performance models are used

- in the first phase (the bottom up assignment of "optimal" parallelism degree) to compute the upper bound (if any) to the theoretic parallelism degree of the skeleton-s/templates used, and
- in the second phase (the top down iterative process taking away resources from the assignments in such a way balancing is preserved) to estimate the performance of the "reduced" skeleton/templates, that is, of the skeletons/templates from which resources have been removed in the tentative of coming to a parallelism degree matching the resources actually available.

This process is exemplified in Fig. 5.1.

5.4.2 Run time usage

At run time, performance models are mainly used to verify the current behaviour of the application with respect to the expected behaviour. This means that:

- we measure actual performance achieved by the application on the target machine. This means we insert probes[91] into the application code and through the probes we gather the measures needed to assess the current application performance[92].
- we measure the target architecture parameters needed to instantiate the (template) performance model of our application

[91] methods used to fetch performance values. This may be implemented completely in software (e.g. by invoking proper operating system calls) or with some degree of hardware support (e.g. fetching values of proper "hardware counters" available on most modern processors.

[92] it is known that probe insertion perturbs the computation, therefore we must choose probes in such a way a minimal intrusion is enforced and, therefore, a minimal perturbation of actual application performance is achieved.

- we evaluate the expected performance model of the application and we measure the distance with respect to the performance values actually measured.

Two possibilities are open, once the actual and the expected performance measures have been got.

- a former, simpler possibility is to report the application programmer of the differences in between the measured and predicted performance values. No other action is taken to correct possible mismatches between measured and predicted performance in this case. Possibly, the application programmer may use the reported information to improve[93] the application.

- a second, more complex possibility consists in analyzing the differences in between the measured and predicted performances and in operating–without programmer intervention, actually–to restructure the parallel application in such a way the measured performance is made closer to the predicted one. We will further discuss this possibility in Sec. 10.5. At the moment being it is sufficient to mention that application restructuring may involve rewriting the high level application code using different skeletons or different skeleton compositions, as well as re-computing the template assignment to skeletons, in such a way more efficient templates are picked up instead of the inefficient ones. It is worth pointing out that in this latter case, the choice of the "better" templates to be used in spite of the currently used ones could not rely on evaluation of the associated performance models. These models are the very same that led to the current template choice. The discrepancies in between observed and predicted performance should be due to inaccuracies in the performance model or in temporary conditions arising on the target architecture which are not considered in the model. Therefore some kind of "artificial intelligence" or heuristic reasoning should be applied to get rid of the faulty situation.

5.4.3 Post-run time usage

The last way of exploiting performance models is the newest one in the algorithmic skeleton scenario. It has been first introduced by Mario Leyton in his Calcium skeleton framework built on top of the Java ProActive middle-ware [29].

In this case, performance models are used as follows (Leyton referred to this process as *codeblame*):

- during application computation, current performance figures are collected on a *per skeleton* basis.

- after the end of the application computation, predicted performance figures are computed (again, *per skeleton*).

- the predicted values are compared to the measured ones and differences are individuated.

- in case of sensible differences, an analysis is performed on the sub-terms constituting the performance models in such a way a precise characterization of the causes of skeleton (or template) under performing are individuated.

- eventually, such causes are exposed to the application programmer as "performance summary" of the application execution, along with more general data such as overall completion and service time.

[93] to fine tune the application varying some of the parameters used to instantiate the skeletons used, or even, in case of large differences, to completely restructure–in terms of the skeletons or templates used–the whole application.

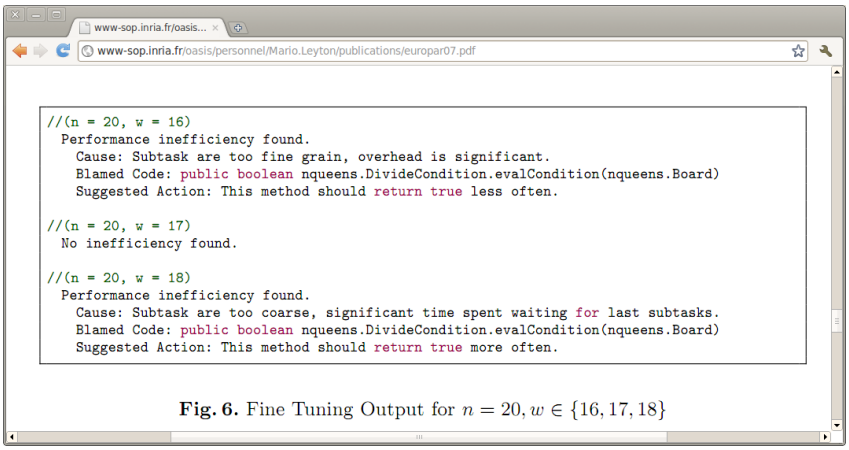


Fig. 6. Fine Tuning Output for $n = 20, w \in \{16, 17, 18\}$

Figure 5.2 CodeBlame figure from M. Leyton paper [29] The output is relative to three different runs (with parameters outlined in the comment lines) of a program solving the N-queens problem.

The programmer, looking at the causes of the inefficiencies found in the execution of the current version of the application, may modify the application in such a way these inefficiencies are eventually removed (Fig. 5.2 shows the kind of output presented to the user by the Calcium framework).

E.G.▷ Consider an application that can be implemented using data parallel skeletons, and suppose the proper skeleton to use is a map. Suppose also that the map may be applied at different items of a complex input data structure: as an example, a map may be applied either at the level of singletons belonging to a matrix or at the level of the rows of the same matrix. Now suppose the programmer chooses to exploit parallelism in the computation of the single element of the matrix. It may be the case that the grain of these computations is too fine and therefore the predicted performance happens to be far from the observed one. The skeleton framework may analyze the map model and perceive the inefficiency is given by the time needed to send data to remote processes (or to remote MDF interpreter instances) higher than the time spent computing the single matrix item. It can therefore report to the programmer a statement such as:

```
Map skeleton Ski:  fine grained worker.
   Please use coarser grain workers.
```

As a consequence, the application programmer has a clear hint on how to intervene to improve application efficiency. ■

5.5 SKELETON ADVANTAGE

When modeling performance of a parallel computation a number of different factors have to be taken into account. Such factors depend on the kind of concurrent graph activities involved, in general. In case we are considering the implementation of the parallel application rather that the application in abstract terms, the factors depend also on the target architecture features (both hardware and software). Therefore performance modeling turns out to be a quite difficult task, possibly more difficult than the development of the parallel application itself in the general case.

However, modeling algorithmic performance in skeleton programming frameworks is somehow simplified by different factors:

- First of all, the *exact parallel structure* of the parallel computation is completely exposed by the algorithmic skeletons used. No side, "hidden" interaction exist impacting

the performance modeling process. This is not usually the case in classical parallel programming frameworks were programmers work directly at the level of the base mechanisms in order to develop a parallel application.

- Second, when using hierarchical composition of skeletons, the performance modeling of the application may be incrementally build from the performance models of the base skeletons. Composition of performance modeling is achieved through proper development of modeling of a set of base measure–such as service time or latency– for each one of the skeletons provided to the final user/the application programmer. These base measures are the ones used in the definition of skeleton performance model as well and therefore the skeleton models recursively compose.
- Last but not least, when dealing with *implementations* rather than abstract parallelism exploitation patterns, the architecture modeling needed to support template performance modeling is somehow simplified as we can reason in terms of abstract components of the templates rather than of collections of very precise (base) mechanisms eventually used to implement the template on the target architecture at hand.

5.6 MONITORING APPLICATION BEHAVIOUR

Monitoring application behaviour requires two kind of activities:

- measuring times relative to the execution of the application as a whole. This is needed to be able to compute the actual completion time of the application as well as the service time achieved in delivering the results, just in case the application processes streams of tasks to produce streams of results.
- continuous measuring of the times relative to execution of different application parts. This is needed to be able to check the correct[94] execution of the application.

The former monitoring activity is simple and does not need actually too much modifications of the application code. Completion time and service time may both be measured using tools external to the application. In Linux/Unix the approximated wall computation time of an application may be taken by launching the application as a parameters of a `time` command rather than directly at the shell prompt. Once the completion time and the number of items computed is known, it is easy to derive the service time.

E.G.▷ In order to know the time spent in compiling the PDF of this book from its latex code, we can use the command shown below.

```
marcod@macuntu:$ time pdflatex main.tex > /dev/null

real 0m1.203s
user 0m1.156s
sys 0m0.044s
marcod@macuntu:$
```

The timed execution of `pdflatex` command with the output redirected to the null device is requested. The results of the computation are discarded, and at the end a summary of real, user and system time required to complete the command is show. “Real” time is the wall clock time spent to compute the program, as perceived by the shell process executing the command. “User” time is the one spent executing user code and eventually “sys” time is the one spent executing syscalls on the behalf of the “user” code. ———— ■

[94] with respect to performance non functional concerns

More precise time measures may be obtained reading time or clock cycles at the very beginning of the application code, then at the very end and eventually computing the differences among the measured values. Different system or library calls are available to the purpose in the different operating systems. In Linux `clock` and `gettimeofday` may be used as well as `omp_get_wtime` or `MPI_Wtime` in case OpenMP or MPI are used.

The latter activity is a little more complicated. It requires the usage of the same library calls possibly used to for the former monitoring activity. However, care has to be taken not to introduce significant impact in the original computation due to the monitoring mechanisms used and to the monitoring event frequency used.

This problem is known as the *intrusion problem*. The more probes you insert into the application code, the less accurate measures you take. Inserting a probe usually requires to perform some extra system call, which is quite expensive, in terms of time spent in the call, or library call, which is less expensive than a system call but may be anyway expensive. Moreover, the data read by the probes have to be stored somewhere, in order to be usable at some point during the execution of the application or after the current application execution. The memory footprint of the probe data is usually small, but it could be anyway a problem at least for those applications using huge amounts of virtual memory for their own "functional" data.

Care has to be taken also in reporting probe measures on the user terminal *while* the parallel application is running. If you insert a `printf` to show some probe measure, the resulting `write` system call is practically executed asynchronously with respect to the application code, and therefore it may look like it will not introduce any delay but the one relative to call to the `write`. However, the `printf` results will eventually be reported on the user terminal through some kind of network transport protocol. The resulting traffic will be supported by the same network supporting the parallel application communications and synchronizations. And this eventually *perturbs* the application behaviour.

When reasoning about parallel applications executed on multi cores, rather than on distributed architectures such as COW/NOWs, similar intrusion problems must be taken into account, related to thread–rather than process–scheduling and interaction.

In any case, parallel application probing is even more complicated than sequential application probing because the changes in relative timings of the concurrent activities due to probes may be such that some deadlocks or some bottlenecks completely disappear from the application run. When the probes are removed to run the application at its own maximum speed, they may come back again with definitely unpleasant effects[95].

Structured parallel programming helps in the implementation of application monitoring in distinct ways:

- The placement of probes is clearly suggested by the skeletons used in the application. In a pipeline, we can easily identify probe placement before and after the computation of a single task on each one of the stages involved. These probes may accumulate *average* service time for each one of the stages. Then pipeline termination procedure may be used to gather the stage service times all together in such a way the pipeline service time may be eventually computed, or the stage service times may be attached to the partially computed tasks flowing through the pipeline in case the pipeline overall service time has to be computed *during* the pipeline computation. In this case the computation of the pipeline overall service time may be made on the last pipeline stage.

[95] by the way Murphy law states this will happen when you'll be demonstrating the application to the final client paying for the application development :-)

- The implementation of the skeletons may be structured in such a way primitive measures are automatically gathered (may be depending on some skeleton configuration/-compilation flag) and returned upon the invocation of proper skeleton library calls. This is not a difficult task for the system programmer designing skeleton implementation, actually, as he/she has the complete knowledge related to the implementation and therefore he/she may figure out the best probing strategies to gather these base performance measures. This contributes to the separation of concerns between system and application programmers enforced by the structured parallel programming approach, actually.
- Due to the clear parallel structure of the skeletons, the perturbation effect of the probes may be estimated/modeled–at least in an approximate way–in such a way the measures returned by the probes actually estimate the performance of the application *without* the probes. Again, consider a pipeline. If we measure the service time by inserting a couple of probes reading time before a task computation is started and after it has been completed, we can add to the measured computation time an estimate of the communication overhead and return this value as the stage latency, which is not including–*de facto*–the overhead relative to the time measure probes.

5.7 PERFORMANCE MODEL DESIGN

Performance models represent a critical resource associated to skeletons and templates. Different approaches may be followed in order to design performance models that will be discussed in the following sections.

5.7.1 Analytical performance models

Traditionally, skeleton and template performance models have been developed following an "analytic" approach. This means that

- first performance modeling of the base mechanisms is designed such that we have analytical formulas expressing all the base performances
- then the performance model of the skeleton (template) is designed as a proper composition of instances of the base mechanisms performance models.

In this case, no general theory has been used to support performance model development but some quite usual mathematical background.

E.G.▷ The pipeline skeleton performance model may be developed, according to the analytical performance modeling approach, as follows:

- we first consider communications costs, and we conclude the cost of the communication of an item of type α and "sizeof" $sizeof(\alpha)$ among stages i and j is

$$T_{comm}(i,j) = T_{setup} + \frac{sizeof(\alpha)}{\mathcal{B}}$$

  being $\mathcal{B}$ the inter stage bandwidth.
- then we consider costs relative to the execution of the two stages (latencies) as L_1 and L_2.
- eventually we conclude the slowest stage will determine the pipeline service time:

$$T_s(pipe(S_1, S_2, \ldots, S_m)) =$$

$$max\{(L_0 + T_{comm}(i, i+1)),$$

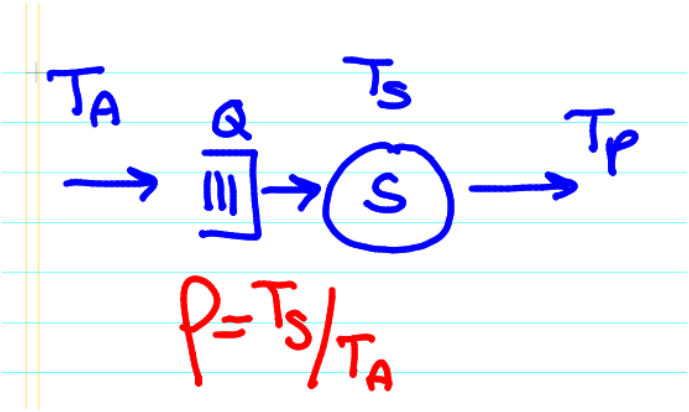


Figure 5.3 Server modeling

$$\forall i \in [1, m-1] : (L_i + T_{comm}(i-1, i) + T_{comm}(i, i+1)),$$

$$(L_m + T_{comm}(m-1, m))\}$$

Using the same approach, but ignoring communication overheads, we can design the performance model of a farm skeleton as follows:

- we first consider the n_w workers may compute n_w tasks in a time equal to T_w (the time spent computing a single task)
- then we consider the entity providing the tasks will have its own service time (t_e), as well as the entity gathering the task computation results (t_c).
- we look at the farm implementation as a three stage pipeline and therefore

$$T_s(farm(W)) = max\{t_e, t_c, \frac{T_w}{n_w}\}$$

In case we want to "solve" this model in such a way the optimal number of workers is computed, we can reason as follows:

- the service time is given by a maximum.
- the only term of the maximum depending on n_w is the $\frac{T_w}{n_w}$ term
- therefore we will have the minimum service time when this term will not be the maximum
- and therefore

$$n_w^{opt} = n \in \mathcal{N} : \frac{T_w}{n_w} = max\{t_e, t_c\}$$

that is

$$n_w^{opt} = \lceil \frac{T_w}{max\{t_e, t_c\}} \rceil$$

■

5.7.2 Queue theory

We can use some results from queue theory that basically model server systems in order to design skeleton (template) performance models. Queue theory is a quite complex field that has been extensively used in the past to model several distinct systems. Here we just

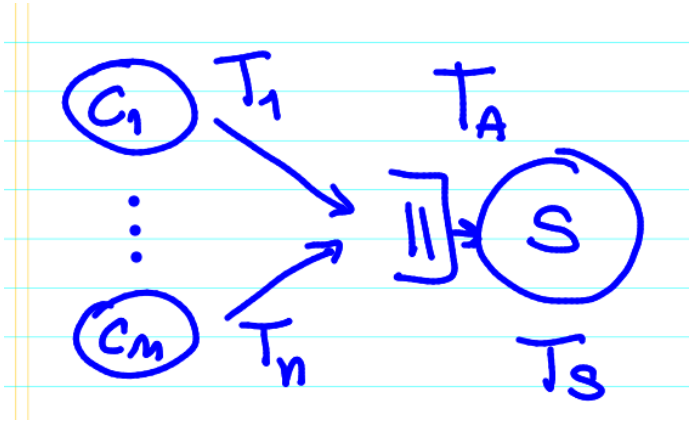


Figure 5.4 Multiple clients for the same server

mention a couple of results relative to M/M/1 queues[96] that may be easily re-used to model performance of skeletons and, more effectively, implementation templates. The interested reader may refer to existing publications and books. In particular, our students may refer to the notes for SPA course by M. Vanneschi available on line [86].

We introduce queue systems, such as the one represented in Fig. 5.3. A *server* S has a service time equal to T_S. It accepts server requests through an input queue. The inter arrival time of service requests is T_A. The *utilization factor* of the input queue Q is defined as

$$\rho = \frac{T_S}{T_A}$$

The inter departure time T_P of results relative to service requests from the server S is given by

$$T_P = \begin{cases} T_A & \textit{iff } \rho \leq 1 \\ T_S & \textit{iff } \rho > 1 \end{cases}$$

Two results are particularly useful, due to the kind of parallel patterns we are considering in this book.

1. Consider a system made of a server a several clients, such as the one depicted in Fig. 5.4. Suppose that each client C_i directs a service request to the server S each T_i. Then the inter arrival time to server S will be

   $$T_A = \frac{1}{\sum_1^n \frac{1}{T_i}}$$

   Intuitively, each client contributes sending a request every $\frac{1}{T_i}$ units of time. The frequencies of the different clients accumulate. Time is the inverse of the frequency.

2. Consider a system made of one client interacting with some servers, as depicted in Fig. 5.5. The client sends a service request towards server S_i with probability p_i. The

[96] queue systems differ depending on the features of servers and clients. Usually a queue system is named as a triple "A/B/C". A represents the kind of inter arrival time distribution, B represents the type of service time for the server(s) and C the number of servers (refer to Fig. 5.3). In our case, we are interested in results relative to queues M/M/1 and M/M/g where the M stands for "Markovian, that is stochastic, with no memory.

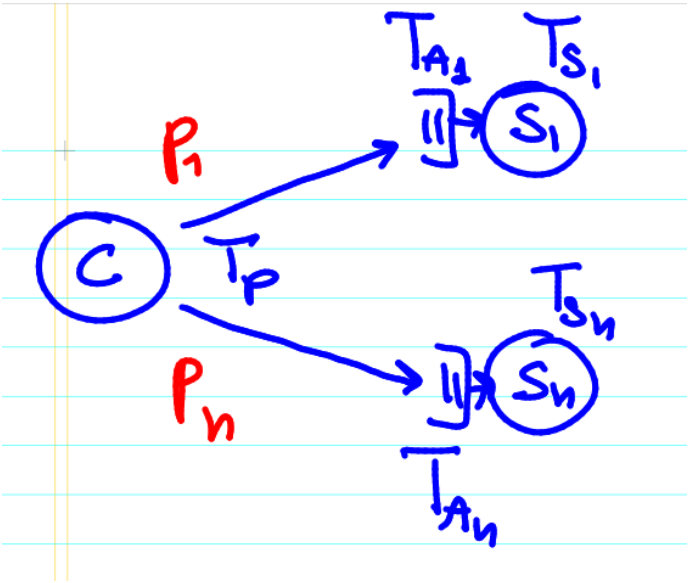


Figure 5.5 One client for the multiple servers

inter departure time of service requests from client C is T_C. Then the inter arrival time of requests to server S is

$$T_A = \frac{T_P}{p_i}$$

With these results, we can model several different aspects related to the performance of a skeleton.

E.G.▷ The emitter-collector-worker implementation of a task farm (or equivalently, of a data parallel map) may be modeled as a client (the emitter) directing task computation requests to n_w servers (the workers) and by a set of clients (the workers) directing "result" requests to a single server, the collector.

Consider the first part of the system. The emitter has an inter arrival time T_A^e. Being sequential, its inter departure time is $T_P^e = T_A^e$. Assuming a round robin scheduling of tasks to workers, the probability to direct a request to a given worker will be given by

$$p_i = \frac{1}{n_w}$$

Due to the second result listed above, the inter arrival time to generic worker will be therefore

$$T_A = \frac{T_A^e}{\frac{1}{n_w}} = n_w T_A^e$$

The utilization factor of the input queue of the generic worker will therefore be

$$\rho = \frac{T_w}{n_w T_A^e}$$

In order to have the worker delivering its service time T_w, the utilization factor should be smaller that 1. This means:

$$\rho = \frac{T_w}{n_w T_A^e} < 1$$

and therefore:

$$T_w < n_w T_A^e$$

$$n_w > \frac{T_w}{T_A^e}$$

which corresponds to the usual result we already mentioned several times relative to the optimal number of workers in a task farm. ■

CHAPTER 6

PARALLEL DESIGN PATTERNS

Design patterns have been introduced in the '90 to "describe simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence, they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succint and easily applied form" [48].

Parallel design patterns [69] have been introduced in early '00, with the intent to model solutions to recurring *parallel* problems. They represent a software engineering perspective on the parallel computing problems. The way they are presented–grouped in layers, with each layer including patterns of interest when looking at the parallel programming problems from a particular abstraction level–represents a kind of methodology suitable to support the parallel application developer in the design of efficient applications.

From our structured parallel programming course perspective, parallel design patterns represent a kind of "engineered way" of considering the problems related to parallelism exploitation. This is the reason why we introduce and shortly discuss parallel design patterns here.

6.1 DESIGN PATTERNS

A design pattern is a representation of a common programming problem along with a tested, efficient solution for that problem [48]. Design patterns are usually presented in an object oriented programming framework, although the idea is completely general and it can be applied to different programming models.

A design pattern includes different elements. The more important are: a *name*, denoting the pattern, a *description of the problem* that pattern aims to solve, and a *description of the solution*. Other items such as the *typical usage* of the pattern, the *consequences* achieved by

Element	Description
Name & classification	The name of the pattern and the class the pattern belongs to
Intent	The problem solved by the pattern
Also known as	The other names/terms used to describe the pattern
Motivation (forces)	The reason(s) why the pattern is introduced
Applicability	The typical situations where the pattern may be exploited
Structure	The structure of the pattern in terms of the component classes/class diagrams
Participants	The classes involved
Collaboration	The interactions among the involved classes
Consequences	The kind of consequences deriving from pattern application
Implementation	Known implementation
Sample code	Sample code or pseudo code relative to the implementation and usage of the patterns
Known uses	Examples of applications/algorithms/kernels using the pattern
Related patterns	Other patterns that may be of interest for or used with the current pattern

Table 6.1 Elements of a design pattern

using the pattern, and the *motivations* for the pattern are also included. Table 6.1 details the elements included in the description of a design pattern in [48].

Therefore a design pattern, *per se*, is not a programming construct, as it happens in case of the algorithmic skeletons. Rather, design patterns can be viewed as "recipes" that can be used to achieve efficient solutions to common programming problems.

Typical design patterns include patterns commonly used in wide range of object-oriented applications, such as

- the *Proxy* pattern, providing local access to remote services (Fig. 6.1 shortly outlines the proxy pattern as described in [48]),
- the *Iterator* pattern, providing (sequential) access to the items of a collection,

as well as more sophisticated patterns, such as

- *Mediator* pattern, encapsulating the interactions happening between a set of *colleague* objects,
- the *Strategy* pattern, providing encapsulation of a set of similar services along with methods to transparently pick up the best service provider,
- the *Observer* (or *Publish/Subscribe*) pattern, used to structure asynchronous, event based applications,
- the *Facade* pattern, providing a simplified interface to some kind of large code/class ensemble.

Overall, the programmer using design patterns to structure the sequential skeleton of his/her application will produce more robust and maintainable code than the programmer using a more traditional–non design pattern based–object oriented approach.

In the original work discussed in Gamma's book [48], a relatively small number of patterns is discussed. The debate is open on the effectiveness of adding more and more specialized patterns to the design pattern set [1]. This situation closely resembles the one found in the skeleton community some years ago. Skeleton designers recognized that a small set of skeletons may be used to model a wide range of parallel applications [22, 42] but some

Intent	provide a surrogate placeholder for another object to control access to it
Also known as	surrogate
Motivation	defer cost of creation, initialization or access
Applicability	remote proxy, virtual proxy, protection proxy, smart reference
Structure	class Proxy calls the class RealSubject when invoked from an instance of Subject–the RealSubject–issuing a Request()
Participants	Proxy (maintains a reference to the object, provides an interface to access the object, controls the accesses to the object), Subject (defines common interdace for RealSubject and Proxy), RealSubject (defines the real object represented by the Proxy)
Collaboration	propagate requests to RealSubject
Consequences	introduces indirection in accesses to the object
Implementation	e.g. by overloading the member access operation in C++ (->)
Sample code	sample code in C++ or Java presented here ...
Known uses	accesses to remote objects (RMI/RPC), implementation of distributed objects
Related patterns	Adapter (providing a different interface to an object), Decorator (adding responsibilities to an object).

Figure 6.1 The Proxy pattern

authors claimed that specialized skeletons should be used to model application dependent parallelism exploitation patterns [56].

Recently, some questions concerning design patterns have been posed, that still have no definite answer: should design patterns become programming languages? Should they lead to the implementation of programming tools that provide the programmer with more efficient and "fast" OO programming environments?[31, 65]

Again, this is close to what happened in the skeleton community. After Cole's work it was not clear whether skeleton should have been made available as some kind of language constructs or they should have been used only to enforce a well founded parallel programming methodology.

6.2 PARALLEL DESIGN PATTERNS

Parallel design patterns have been introduced [69] with the same aims of (sequential) design patterns, namely to describe solutions to recurrent problems in the context of interest. The context of interest in this case if obviously parallel software design rather than simple object-oriented software design.

As it happened in the case of classical design patterns, a parallel design pattern is described by means of a precise set of elements, namely:

- *Problem* – the problem to be solved.
- *Context* – the context where the pattern may be most suitably applied.
- *Forces* – the different features influencing the parallel pattern design.
- *Solution* – a description of one or more possible solutions to the problem solved by the pattern.

Sample parallel design patterns, as an example, solve typical parallelism exploitation problems, such as *divide&conquer*, *pipeline* and *master/worker* solving respectively the problem of efficient implementation of recursive, divide and conquer computations, of staged computations and of computations split in a number of independent tasks.

Parallel design patterns have been used to implement structured parallel programming frameworks very similar to the algorithmic skeleton ones [66, 50, 70] and a notable research activity has been generated that eventually led to the recognition of the fact that parallel design patterns may be a key point in the solution of the problems related to the development of efficient parallel software. The Berkeley report [21] published in late '00s clearly states that

> Industry needs help from the research community to succeed in its recent dramatic shift to parallel computing.

and lists among the other principles to be followed to efficiently address the problem of designing and developing efficient software for parallel machines the following principle:

> Architecting parallel software with design patterns, not just parallel programming languages.

In particular, the authors recognize that

> Computer scientists are trained to think in well-defined formalisms. Pattern languages encourage a less formal, more associative way of thinking about a problem. A pattern language does not impose a rigid methodology; rather, it fosters creative problem solving by providing a common vocabulary to capture the problems encountered during design and identify potential solutions from among families of proven designs.

A comprehensive list of parallel design pattern papers and projects may be found at the web page `http://www.cs.uiuc.edu/homes/snir/PPP/`. The rest of this Section summarizes the parallel design patterns as presented in [69].

6.2.1 The parallel design pattern design spaces

One of the most interesting aspects related to parallel design patterns as described in [69] is the partitioning of the design space of a parallel application in four separate but related *design spaces*:

- The **Finding concurrency** design space, including the parallel design patterns modelling the different kinds of parallel/concurrent activities structures that may be suitably used in a parallel application.
- The **Algorithm structure** design space, including the parallel design patterns modelling recurring parallel algorithm patterns.
- The **Supporting structure** design space, including the parallel design pattern that can be used to implement the algorithmic patterns in the previous design space.
- The **Implementation mechanisms** design space, eventually including the patterns modelling the base mechanisms used to implement a parallel application.

From our algorithmic skeleton perspective, the first two design spaces correspond to what we called "concurrent activity graph" definition/design, while the last two correspond to the "implementation" of the concurrent activity graph. The importance of this design space structuring relies in the "separation of concerns" implicitly assumed in between the programmers using the parallel patterns: application programmers–the ones supposedly using the patterns in the finding concurrency and algorithm structure design spaces–will only use their (application) domain specific knowledge to find out the most proper combination of high level patterns modelling parallelism exploitation in the application at hand. And this process will not require any knowledge relative to the underlying target architecture. Then the system programmers–the ones that are experts in the target architecture features–may exploit the design patterns in the supporting structure and implementation mechanisms

spaces to properly implement the patterns used at the higher level by the application programmers. This reflects the situation in the algorithmic skeleton world where application users basically use pre-defined skeletons that the system programmers are in charge to implement efficiently onto the target architecture.

In the following sections, we will discuss in detail the different patterns in the different design spaces, pointing out, if the case, similarities and differences with similar concepts/abstractions in the algorithmic skeleton universe. Fig. 6.2 shows the global design space, outlining the different patterns and pattern groups in the four design spaces.

6.2.2 Finding concurrency design space

The patterns in the "finding concurrency design space" may be used to start designing a parallel application, right after having considered the problem to be solved with the application and identified the most computationally intensive parts to be parallelized.

There are three different groups of patterns in this space. Two of them are actually related to the ways used to implement the parallel application (the decomposition and dependency analysis patterns) and one related to the fact the "finding concurrency" space is such that a solution if often found iterating with different steps through the decomposition and dependency analysis patterns (the design evaluation pattern).

In particular, the **Decomposition patterns** are used to decompose the problem into pieces that can execute concurrently, while the **Dependency analysis patterns** help grouping the tasks to be executed and anaylizing the dependencies among these tasks. The **Design evaluation pattern** is only used to "guide the algorithm designer through an analysis of what has been done so far before moving to the patterns in the algorithm structure design space" [69].

The decomposition patterns include just two patterns:

- the **Task decomposition pattern**, modelling the division of a complex application in a set of tasks that can be executed concurrently, and
- the **Data decomposition pattern**, modelling the division of the input data into subsets that can be (relatively) independently computed to obtain the final result.

The main forces influencing the design of these patterns are *flexibility*, *efficiency* and *simplicity*. Flexibility is needed to adapt the application design to different implementation requirements. Efficiency is needed as we are targeting efficient parallel application development. Last but not least, simplicity is required to help the application programmers to use to pattern in their parallel applications: too complicated patterns (or libraries, objects or APIs) have no sufficient appeal to convince the application programmers they are worth being used and exploited.

The dependency analysis patterns include three different patterns, instead:

- the **Group tasks** pattern aimed at modelling the more convenient grouping of tasks such that the management of dependencies is simplified,
- the **Order tasks** pattern aimed at figuring out how tasks (or groups of tasks) may be ordered to satisfy the application constraints related to task execution, and
- the **Data sharing** pattern aims at modelling the accesses to a shared data structure.

E.G.▷ Let's have a look in more detail to one of these patterns, which are not as obvious as the ones in the decomposition patterns set: the order task pattern. The order task pattern considers groups of tasks, as output by the group tasks pattern and tries to find the dependencies among those groups of tasks in order to determine an order of execution of the groups of

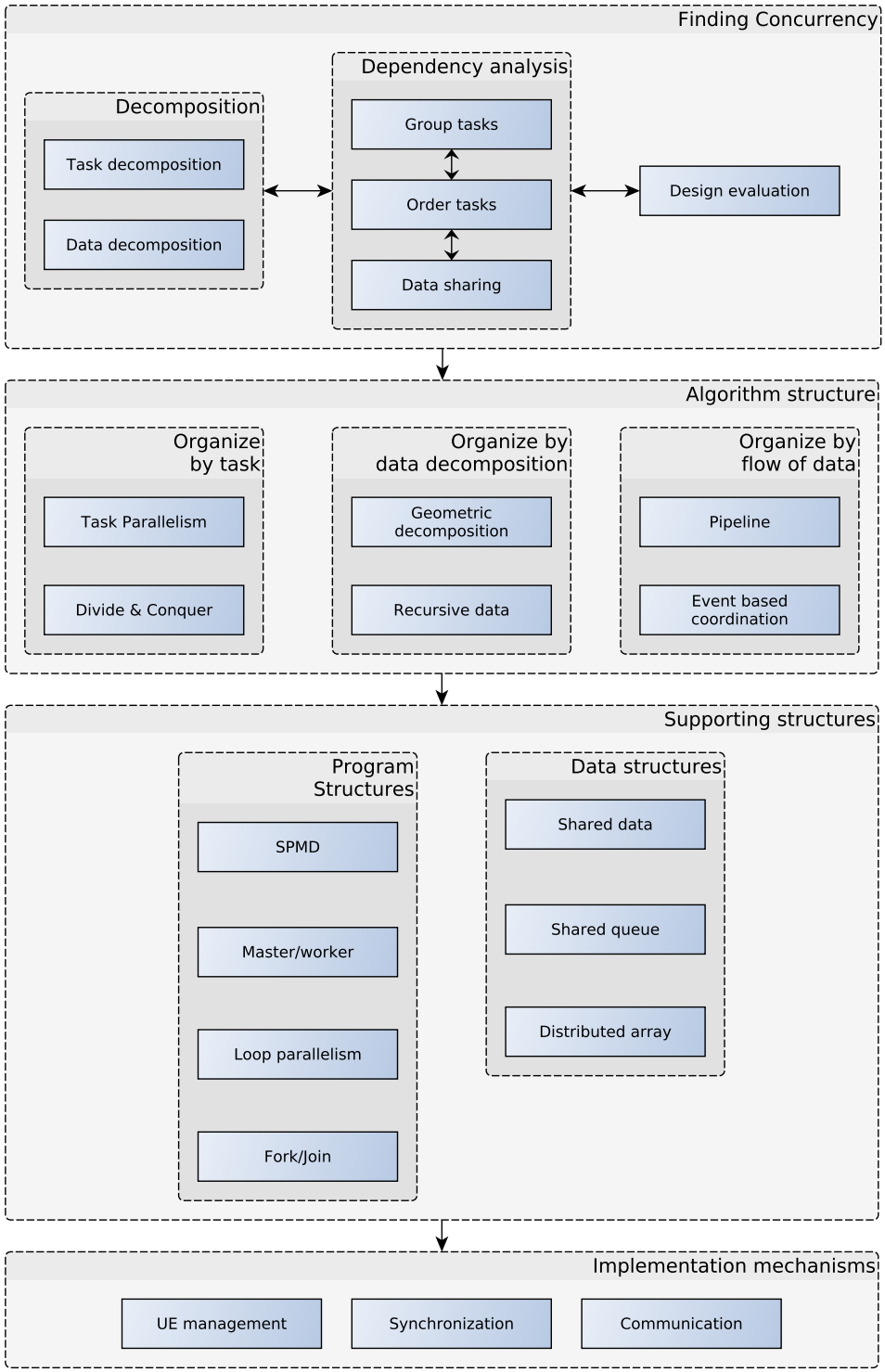


Figure 6.2 Design spaces & Parallel design patterns

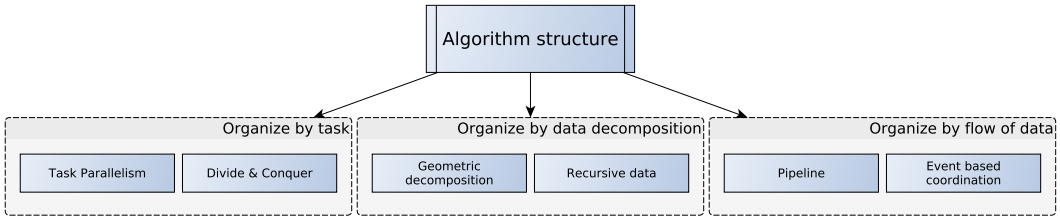


Figure 6.3 Algorithm structure design space: alternative pattern groups

tasks. In particular, temporal and data dependencies are considered. If neither temporal nor data dependencies are present, total independence is assumed, which is definitely a good feature when parallel execution of the tasks is taken into account. The order design pattern looks for a sufficient and necessary ordering, that is for an ordering that ensures that all the constraints are actually taken into account and satisfied and no unnecessary constrains are enforced. The solution proposed in [69] is to first consider data dependencies and then to consider the influence of external services (e.g. accesses to disk) that may impose further constraints with the possibility of unleashing the concurrent execution of all groups in case no data dependencies exist and no constraints come from external services. ——— ■

The overall output resulting from the analysis of the "finding concurrency" design space is a decomposition of the problem into different design elements, namely i) a task decomposition identifying the tasks that can be executed concurrently, ii) a data decomposition that identifies the data local to each one of the tasks, iii) a way of grouping tasks and ordering the tasks groups in such a way temporal and data dependencies are satisfied and iv) an analysis of the dependencies among the tasks.

6.2.3 Algorithm structure design space

The output from the "finding concurrency" design space is used in the "algorithm structure design space" to refine the design of our parallel application and to figure out a parallel program structure closer to an actual parallel program suitable to be run on a parallel target architecture.

There are three major ways of organizing a parallel algorithm:

- **by task**, that is considering the tasks as the base thing to be computed in parallel within the parallel algorithm,
- **by data decomposition**, that is considering the decomposition of data into (possibly disjoint) subsets as the base items to be processed in parallel within the parallel algorithm, or
- **by flow of data**, that is considering the algorithm defined flow of data as the rule dictating what are the steps to computed concurrently/in parallel within the parallel algorithm.

The three ways of organizing the parallel algorithm are somehow alternatives. While in the "finding concurrency" design space the application programmer is supposed to go through all the groups of patterns, here the programmer is required to choose one of the three alternatives and exploit on of the parallel design patterns in the group (see Fig. 6.3).

The **Organize by tasks** pattern group includes two patterns:

- the **Task parallelism** pattern, managing the efficient execution of collections of tasks. The proposed solution to implement the pattern works out three different points: how

tasks are defined, the dependencies among tasks and the scheduling of the tasks for concurrent execution.

- the **Divide&conquer** pattern, implementing the well know divide and conquer recursive solution schema.

The **Organize by data decomposition** pattern group includes two patterns:

- the **Geometric decomposition** pattern modelling all those computations where a given "geometric shape" of the concurrent executors may be recognized (linear vector, bi-dimensional array), and the parallel algorithm may be defined by defining the kind of computation(s) being performed at the shape items (vector items, array rows, array columns, array items, ...).
- the **Recursive data** pattern, modelling those parallel computations structured after the structure of some recursively defined data structure, e.g. a list–defined as a null list or an item followed by a list–or a tree–defined as a node or a node with a (given) number of "son" trees.

Eventually, the **Organize by flow of data** pattern group also hosts two patterns:

- the **Pipeline** pattern, where the flow of data is traversing a linear chain of stages, each representing a function computed on the input data–coming from the previous stage–whose result is delivered to the next stage.
- the **Event-based coordination** where a number of semi-independent concurrent activities interact in an irregular way and interactions are determined by the flow of data among the concurrent activities.

E.G.▷ In order to fix the concepts, let's analyze more in detail the "recursive data" pattern. The **problem** solved by the pattern is the implementation of (efficient and parallel) operations over recursively defined data structures. The **context** includes all those situations where the data contained in a recursively defined data structure (e.g. tree or list) are processed as a whole or in pieces. Tree structured data, as an example, may be processed to obtain a new data tree with all the nodes/leaves substituted by the result of a function f applied to the old nodes/leaves. Or we may process the tree structured data to search for a particular node, traversing the tree with a pre-fix algorithm. A number of problems have to be taken into account when describing this pattern. The most notable include data representation–usually pointers are used, which are neither too much portable nor practical to use for parallel processing of all the items in the data structure, as following pointers is usually a fairly sequential process, as an example–and dimensioning of the concurrent activities used to implement recursive data processing–e.g. the dept of the tree may be unknown and not computable up to the end of the computation. The **forces** driving the pattern design are different. First, we have to take into account the data representation problem. We need to define a proper representation, suitable to be used for parallel processing, may be radically different from the representation used for the recursive data structure in sequential programming. As an example a list may be represented with a vector rather than with the usual linked list data structure, to ensure the possibility to perform concurrent item accesses, even if insertion and removal of elements may require more complex implementations. Second, we must ensure the data representation is not only functional to the parallel processing needs, but it is should be also easy to understand and maintain. Last but not least, a tradeoff should be looked for in between the amount of concurrency exploited in the recursive data structure processing and the amount of overhead–concurrent activity setup, communication, synchronization–introduced. The **solutions** adopted within the pattern include structuring the pattern computation in phases, namely:

  – *data decomposition*: the data is restructured and decomposed as most suitable for being processed by the pattern

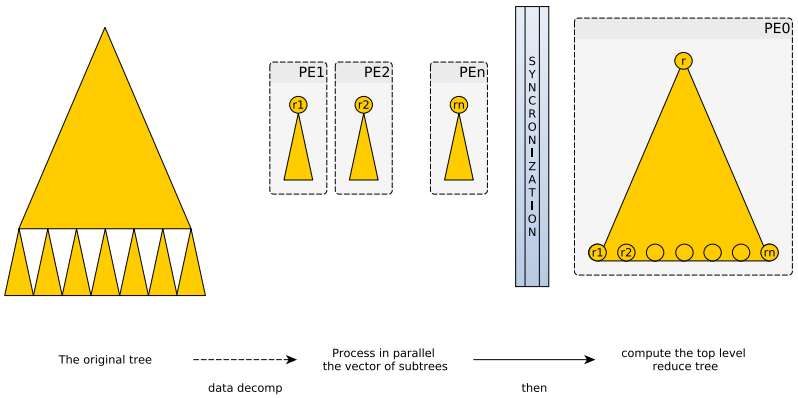


Figure 6.4 Recursive data pattern: sample usage (reduce of a tree)

- *structure*: figure out some kind of "collective"–and parallel–operation to be applied on the data decomposition results, possiibly in a loop, to compute the expected results
- *synchronization*: figure out the kind of synchronization needed to implement the (loop of) collective operations leading to the final result.

As an example, consider the problem of computing a "reduce" with an operator $\oplus$ over a binary tree. We can decompose the tree into the sub trees rooted at level i, then apply sequentially the "reduce"$\oplus$ operation onto all the sub trees in parallel and eventually sequentially compute the "reduce"$\oplus$ operation on the partial results computed at the different sub trees. In this case, the data "data decomposition" phase consist in obtaining the sub tree vector out of the original tree, the "structure" phase consist in applying–in parallel–the reduce over the sub trees and the "synchronization" phase involves waiting the termination of the computation of the sub tree reduces before actually starting computing the root reduce (see Fig. 6.4). ■

6.2.4 Supporting structure design space

After having explored different possibilities to find concurrency and to express parallel algorithms in the "finding concurrency" and "algorithm structure" design spaces, implementation is taken into account with two more design spaces. The first one is called **Supporting structures** design space. This space starts investigating those structures–patterns–suitable to support the implementation of the algorithms planned when exploring the "algorithm structure" design space. Two groups of patterns are included in the design space. The first one is related to program structuring approaches–this is the Program structures pattern group–while the second one is related to the commonly used shared data structures–this is the **Data structures** pattern group. The **Program structures** group includes four patterns:

- the **SPMD** pattern, modelling the computation where a single program code is run on different input data,
- the **Master/Worker** pattern, modelling concurrent execution of a *bag of tasks* on a collection of identical workers,
- the **Loop parallelism** pattern, modelling the concurrent execution of distinct loop iterations, and

	Task Parallelism	*Divide& Conquer*	*Geometric Decomp.*	*Recursive Data*	*Pipeline*	*Event-based Coordin.*
SPMD	⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆ ⋆	⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆
Loop Par.	⋆ ⋆ ⋆ ⋆	⋆ ⋆	⋆ ⋆ ⋆			
Master/worker	⋆ ⋆ ⋆ ⋆	⋆ ⋆	⋆	⋆	⋆	⋆
Fork/join	⋆ ⋆	⋆ ⋆ ⋆ ⋆	⋆ ⋆		⋆ ⋆ ⋆ ⋆	⋆ ⋆ ⋆ ⋆

Figure 6.5 Relationships between the *Supporting Structures* patterns and *Algorithm Structure* patterns (from [69]).

- the **Fork/Join** pattern, modelling the concurrent execution of different portions of the overall computation that proceed unrelated up to the (possibly coordinated) collective termination.

These patterns are well known in the parallel computing community[97]. The SPMD pattern is the computational model used by MPI and one of the most popular patterns used to structure parallel computations with the master/worker. Loop parallelism has been exploited in vector architectures, and it is currently one of the main sources of parallelism in both OpenMP and GPUs. Last but not least, the fork/join pattern perfectly models the `pthread_create/pthread_join` model of POSIX threads.

The **Data structures** group includes three patterns:

- the **Shared data** pattern models all those aspects related to the management of data shared among a number of different concurrent activities,
- the **Shared queue** pattern models queue data types implemented in such a way the queues may be accessed concurrently, and
- the **Distributed array** pattern, models all the aspects related to the management of arrays partitioned and distributed among different concurrent activities.

Also these patterns are well known in the parallel computing community. Shared data is managed in a number of different parallel applications and the correct and efficient management of the shared data is usually the more time and effort consuming activity in the whole parallel application development/design process. Shared queues are used to support interaction of concurrent activities in different contexts, from threads to processes and concurrent activities running on CPU co-processors. Eventually, distributed arrays are often used to implement data structures that are logically shared among concurrent activities but may be somehow partitioned in such a way single one of the concurrent activities "owns and manages" a single portion of the distributed array.

The authors of [69] classify the different patterns in this design space–in fact called "supporting structures" design space–with respect to their suitability to support the implementation of the different patterns in the "algorithm structure" design space.

As an example, Task parallelism is well supported by the four patterns in the Program structures group, whereas the Recursive data pattern is only (partially) supported by the SPMD and Master/Worker pattern (see Tab. 6.5, taken from the parallel patterns book).

In fact, by considering the "supporting structures" design space as the implementation layer of the "algorithm structure" design space, we *de facto* make a strong analogy with what happened in the algorithmic skeletons frameworks, where skeletons represent the "algorithm structure" patterns and the implementation templates represent the "supporting

[97] the community of researchers working in parallel computing but not necessarily experts in software engineering

Design space	#pag
Finding concurrency	21
Algorithm structure	57
Supporting structures	104
Implementation mechanisms	35

Figure 6.6 Space dedicated to the description of the four design spaces in [69]

```
int main(int argc, char * argv[]) {
  ...
  MPI_Init(&argc,&argv);

  my_id = MPI_Comm_rank(MPI_COMM_WORLD);

  switch (my_id) {
    case 0: { ... }
    case 1: { ... }
    ...
    default: { ... }
  }

  MPI_Finalize();
}
```

Figure 6.7 Sketch of sample MPI code

structures" patterns. It is worth pointing that the clear distinction between the "task parallelism" pattern (corresponding to the task farm skeleton) and the "master/worker" pattern (corresponding to one of the implementation templates used to implement task farms, maps and other kind of skeletons) has no correspondence in the algorithm skeleton framework where often the "supporting structures" master/worker pattern is claimed to be a skeleton, that is an "algorithm structure" pattern.

We want to point out here another interesting fact. Table 6.6 reports the pages used to describe the patterns in the four design spaces in [69]. The high level patterns and the low level ones are described using more or less the same space used to describe the patterns in the "supporting structure" design space. This is a clear sign that "there is more to say" on the supporting structures, as in parallel programming this is the only design space explored in depth along with the "implementation mechanisms" space. However, this latter space is a kind of "RISC" layer only providing very basic patterns used to build the full range of "supporting structures" patterns and therefore it may be described much more concisely.

E.G.▷ Consider the "SPMD" pattern. The **problem** solved with the pattern is the correct and efficient implementation of parallel computations where programmer provide a single program for all the concurrent activities. The single program may differentiate the instructions to execute depending on the concurrent activity index. The **context** where this pattern is used includes the large majority of the massively parallel applications, in particular all those developed using MPI. These applications are usually implemented with a single program such as the one outlined in Fig. 6.7. The program is executed on all the target machine processing elements. Each instance of the program receives a different integer id and the code executed at the different PEs is therefore differentiation with the switch statement. The **forces** driving the pattern design are the usual ones of most of the "supporting structures"

patterns, namely: portability, to ensure pattern reusability, scalability, to support different parallelism degrees according to the target architecture features, and efficiency, to ensure performance. A viable **solution** to SPMD pattern design consists in structuring the pattern with the phases outlined in the MPI code sketch above:

- *Initialize*: setting up all the necessary infrastructure for the pattern execution
- *Unique identifier*: getting the id necessary to differenciate the code within the unique program
- *Run the program on all the different processing elements*: run the program in such a way the all the concurrent program activities are completed
- *Finalize*: clean up everything and carefully terminate all the activities.

■

6.2.5 Implementation mechanisms design space

The second (and lower level) design space related to implementation of parallel applications is called **Implementation mechanisms** design space. This design space includes the patterns modelling the base mechanisms needed to support the parallel computing abstractions typical of parallel programming, that is:

- *concurrent activities*
- *synchronizations*
- *communications.*

In fact, the design pattern hosts only three distinct patterns:

- the **UE Management** pattern, related to the management of the **U**nits of **E**xecution (processes, threads)
- the **Synchronization** pattern, handling all those aspects related to ordering of events/-computations in the UE used to execute the parallel application
- the **Communication** pattern, handling all the aspects related to the communications happening in between the different UE implementing the parallel application.

The "UE management" pattern deals with all the aspects related to the management of the concurrent activities in a parallel application. Therefore creation, run and termination of concurrent activities are taken into account. Although in [69] only threads and processes are taken into account, the "UE management" pattern may be obviously adapted to handle the concurrent activities placed on CPU co-processors, e.g. the GPU kernels. The "synchronization" pattern deals with all aspects related to synchronization of concurrent activities and therefore covers aspects such as lock/fence[98] mechanisms, higher level mutual exclusion constructs (e.g. monitors) and collective synchronizations (e.g. barriers). Last but not least, the "communication" pattern deals with the aspects related to data exchange among concurrent activities and therefore covers aspects related to different kinds of point-to-point (e.g. send, receive, synchronous and asynchronous) and collective communications (e.g. broadcast, scatter, gather, reduce).

[98] mechanisms ensuring that at a given point the *view* of the shared memory items is consistent across all EU involved in a parallel application

6.3 SAMPLE PARALLEL DESIGN PATTERN USAGE: IMAGE PROCESSING APPLICATION DESIGN

We illustrate the kind of design methodology related to parallel design patterns summarizing different steps in the design of a parallel applications. We chose an application whose goal is to process–in parallel–a stream of images in such a way i) a "truecolor" filter is applied to each image–the colors are normalized–and ii) the image is sharpened–an edge enhance filter is applied.

6.3.1 Application analysis

The first step is not related to the design patterns, but rather is meant to identify the actual needs of the application in terms of parallel computing. As we want to process a (possibly long) stream of images and we know the images are high resolution images, we can simply evince that the computational load will be huge due to two distinct facts: i) the large number of images being processed, and ii) the large number of pixels relative to each one of the images. Therefore our aim will be to parallelize as much as possible both the processing of different images in the stream (which are independent of each other) and of different parts of the same image (if possible).

6.3.2 Exploring the "finding concurrency" design space

We first consider the patterns in "decomposition" pattern group. As different images appearing onto the input stream are independent, we can use the "task decomposition" pattern to model parallel execution of our application over the different images of the stream.

When moving considering the problem of parallel computation of the single image, we may consider to use the "data decomposition" pattern to implement both filters, the color normalization and the edge sharpening ones.

We then consider the "dependency analysis" pattern group. Here we may use the "order task" pattern to ensure that the processed images are output in the same order the corresponding unprocessed images appeared onto the input stream. We can also consider using the "data sharing" pattern to ensure access to shared data or portions of the images when we implement the two filters. As an example, we should consider the possibility to compute some general–i.e. referred to the whole image–color parameters and then to use this parameters to normalize independently different sub-portions of the image in the color normalization filter.

The "design evaluation" pattern may be used to evaluate the concurrency found in this step, before actually moving to explore the "algorithm structure" design space. Within the pattern, it is suggested to consider three different perspectives (forces) in order to come to a good "concurrent decomposition" of our application:

Suitability for target platform Although we still move at a very high level of abstraction, some general features of the target architecture should already be taken into account: i) number of processing elements (CPU/core, co-processor cores), ii) mechanisms supporting data sharing (shared memory, distributed shared memory, message passing) and iii) overhead related to the set up of concurrent activities. In particular, we must ensure

- we do not plan to use a number of concurrent activities larger than the number of available processing elements. In our case this means we should not plan to use one processing element per input image, as an example. In fact, by recognizing (task order pattern) that the edge filter is to be applied after the color one, we can plan to use a couple of processing elements to implement the processing of

a single image, and then we may assume to use more of this processing element pairs to implement in parallel the processing relative to different images of the input stream, up to the point we use all the available processing elements or we succeed processing all the available images before new ones are available onto the input stream.

- that the planned shared data structures may be actually reasonably implemented in terms of the mechanisms available to support sharing. In our case this means that we look at the target architecture features and in case it does not support primitive sharing mechanisms (e.g. shared memory or distributed shared memory) we rely only on communications to implement our shared data structures, e.g. assuming they are encapsulated in a "sharing server" answering all the read/write requests to the shared data structures.
- and that the overhead necessary to implement all the planned concurrent activities does not exceed the benefits coming from the parallel execution of our application. In our case this means we should be careful not setting up a concurrent activity when the time spent in computing the results of the concurrent activity sequentially is comparable to the time used to set up the concurrent activity itself.

Design quality In this case we need to ensure our concurrent activity plan ensures *flexibility*, *efficiency* and *simplicity*. In our case, flexibility is ensured by the possibility to vary different parameters, such as the parallelism degree in the "task decomposition" and "data decomposition" pattern, as well as in the possibility to collapse either level, that is to apply only the "task decomposition" pattern (therefore to exploit only stream parallelism) or to apply only the "data decomposition" pattern (that is to apply only data parallelism). Simplicity comes from the clean design ensured by the proper usage of the two patterns mentioned above. Efficiency comes from the considerations relative to the "suitability for target platform" and may only be stated in principle: it is well known that a good design may be completely impaired when the wrong implementation mechanisms are chosen to implement it, and therefore the actual efficiency will be clear only after having explored the whole set of design spaces.

Preparation for the next phase We must ensure that our concurrent activity plan is suitable to be worked out by the patterns in the "algorithm structure" design space. In our case, the regular decomposition "task+data" figured out should ensure the "algorithm structure" space may be efficiently explored.

Fig. 6.8 illustrates the results of the exploration of the "finding concurrency" design space.

6.3.3 Exploring the "algorithm structure" design space

Moving to the algorithm structure design space, we decide the abstract algorithms to be used to implement the concurrent activities identified in the previous step. As an example:

- we decide to use a "task parallelism" pattern from the "organize by task" group to implement the processing of differnet images of the input stream
- we decide to use a "geometric decomposition" pattern from the "organize by data decomposition" group to implement the processing of the single filter on the single image
- we decide to use a "pipeline" pattern from the "organize by flow of data" group to implement the sequence of the two filters applied onto one input image.

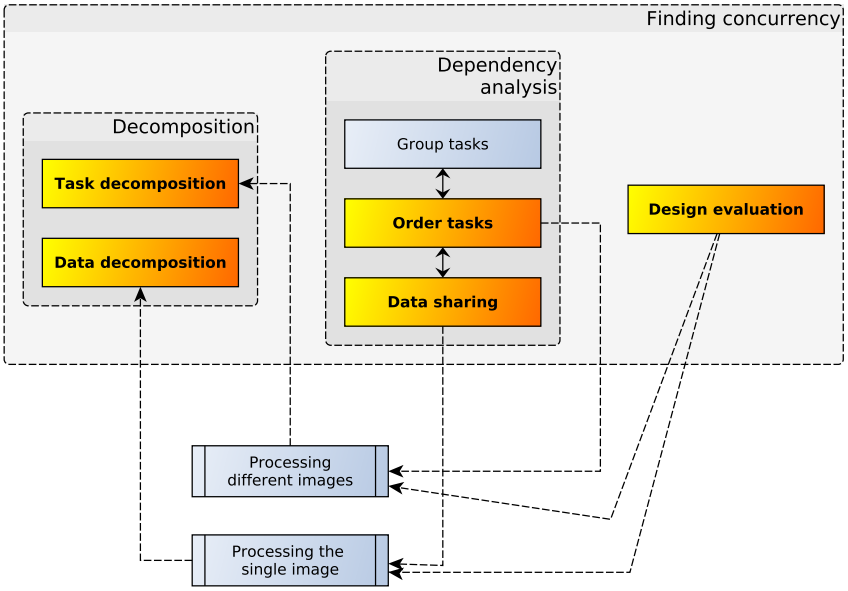


Figure 6.8 Finding concurrency design space explored: the patterns used are outlined in red/bold, the relation with the concurrent activities individuated are show as dotted lines.

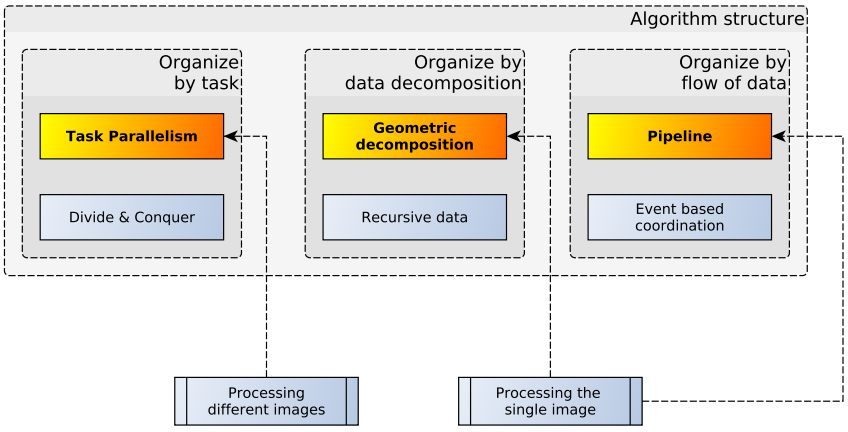


Figure 6.9 Algorithm structure design space explored: the patterns used are outlined in red/bold, the relation with the concurrent activities individuated are show as dotted lines.

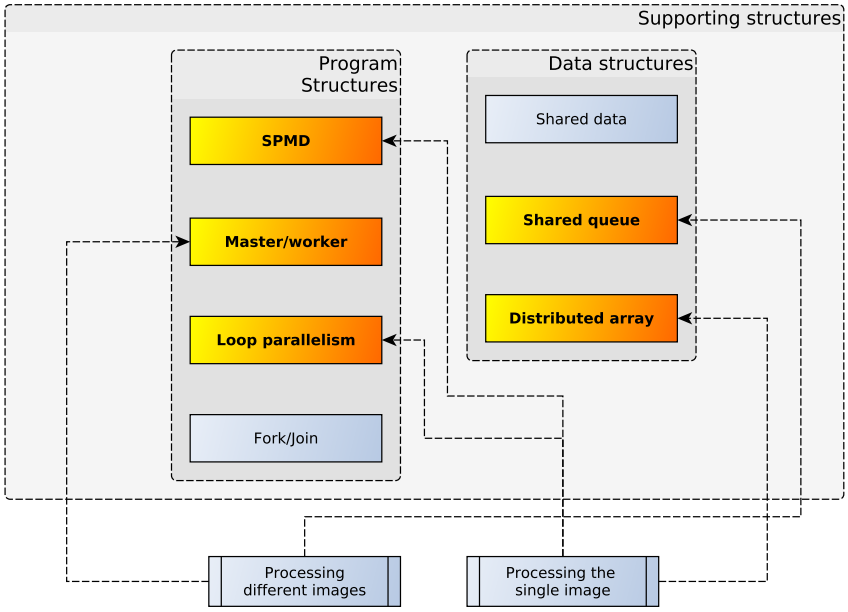


Figure 6.10 Supporting structures design space explored: the patterns used are outlined in red/bold, the relation with the concurrent activities individuated are show as dotted lines.

By picking up these patterns (see Fig. 6.9) we obviously must consider all the pattern related aspects. Remember a design pattern is a kind of "recipe" solving a particular problem. You–the application programmer–are charged of the task of instantiating the recipe to the case at hand.

6.3.4 Exploring the "supporting structures" design space

Eventually, we start considering more concrete implementation aspects by exploring the "supporting structures" design space.

As far as the "program structures" pattern group is concerned, we should look for patterns modelling the different concurrent activities individuated so far. The task parallel computation of different pictures of the input stream will be most suitably implemented using a "master/worker" pattern. Each worker will be in charge of processing a single image. The number of workers used–the parallelism degree–will determine the performance of our application. When considering the implementation of the different filter stages according to the "data decomposition" pattern, we have two choices that are worth being explored. We can use the classic "SPMD" pattern to model data parallel processing of the single filter. But we may also choose to use the "loop parallelism" pattern for the very same purpose. In case we use modern multi core architectures, the "loop parallelism" pattern may be a good choice, due to the facilities included in different programming environments (e.g. SSEx/AVX exploitation within g++ or the `#pragma parallel for` directive in OpenMP) to efficiently support this kind of parallelism. In case we use COW/NOW architectures, instead, the choice will be definitely to use the "SPMD" paradigm which is the one supported in MPI.

While considering the "data structures" pattern group, instead, we will probably eventually pick up the "shared queue" pattern to communicate (partially computed) images across different stages/workers, and the "distributed array" pattern to support data parallel implementation of the two distinct image filters.

6.3.5 Exploring the "implementation mechanisms" design space

Eventually, we have to explore the design space related to the implementation mechanisms. Let's assume we are targeting a cluster of multi core processing elements running a POSIX compliant operating system. A possible choice of patterns in this design space implementing the patterns chosen in the "supporting structures" design space will be the following one:

- both threads and processes will be used in the "UE mechanisms" pattern. In particular, we may choose to use processes to implement the master/worker pattern and threads to implement the data parallel computation of the two filter stages.
- locks and mutexes from the "synchronization" patterns will be used to implement the "shared queue" and the "distributed array" patterns at the single (multi core) processing element level.
- asynchronous point-to-point communications from the "communication" pattern will be used to implement the steps needed to feed workers in the master/worker pattern processing different images and stages in the pipeline processing different filters on the same image.

6.4 COMPARING PARALLEL DESIGN PATTERNS AND ALGORITHMIC SKELETONS

After having discussed how parallel design patterns are structured and how they can be used to design and implement parallel applications, let's try to set up a kind of "summary" of different features relative to both design patterns and to algorithmic skeletons, in the perspective of better understanding the pros and cons of both approaches. Fig. 6.11 summarizes how different features are supported in parallel design pattern and algorithmic skeleton (typical) frameworks.

The overall statement we can make out of this comparison is the following one:

> Algorithmic skeletons and design patterns tackle the same problem(s) from different perspectives: high performance computing and software engineering, respectively.
>
> Algorithmic skeletons are aimed at directly providing pre-defined, efficient building blocks for parallel applications to the application programmer, while design patterns are aimed at providing "recipes" to program those building blocks, at different levels of abstraction.
>
> A large number of algorithmic skeleton frameworks are available, while design pattern related methodologies are used in different–apparently non structured–parallel programming frameworks.

From this summary two important considerations may be made:

- on the one side, the pattern methodology may be used to improve the design of parallel applications with algorithmic skeleton based frameworks
- on the other side, some formalization of the design pattern concepts such that the "solution" element of the pattern may be provided as one or more library entries/object classes could support the definition of a pattern/skeleton programming environment/framework merging the more evident advantages of both worlds.

It is worth pointing out that the design patterns listed in Sec. 6.2 may be extended considerably and with profit. A number of different parallel design patterns have been actually proposed and used, which are not included in [69][99]. Just to make an example,

[99] e.g. see the ones described in the works collected at the `http://www.cs.uiuc.edu/homes/snir/PPP/` web site

	Algorithmic skeletons	**Parallel design patterns**
Communities	High performance computing	Software engineering
Active since	Early '90s	Early '00s
Style	Declarative, library calls	OO, method calls
Abstractions	Common, reusable, complete, nestable parallelism exploitation patterns	Abstract parallelism/concurrency patterns, Implementation patterns
Level of abstraction	Very high	Very high to quite low, depending on the design spaces
Nesting	Allowed, encouraged	Allowed, encouraged
Code reuse	Allowed, encouraged	Allowed, encouraged
Expandability	Poor/none, some frameworks support skeleton set extension through direct access to the implementation layer	Supported: new "recipes" may be added to the design spaces
Programming methodology	Instantiate and combine skeletons from a kind of "skeleton palette"	Figure out which patterns are worth being used, eventually implement the resulting structure on your own
Portability	Ensured by different implementations of the skeleton framework (or different implementation template libraries)	Ensured "by definition": design patterns are target architecture independent, although some can be used only on certain architectures
Layering	Usage layering: two tier model with task parallel skeletons at top, and data parallel skeletons at the bottom level	Design layers
Maintained programming frameworks	Multiple, C/C++, Java, Ocaml based, targeting COW/NOW, multi cores, multi cores with GPUs	Several recent parallel programming frameworks adopted design patterns concepts (e.g. TBB [54] or TPL [51])
Notable frameworks	P3L, ASSIST, Muesli, SkeTo, Skandium, FastFlow, SkePu, Mallba, OSL	COPPS
Automatic performance tuning	Possible, demonstrated, both static (performance models and template assignment) and dynamic (behavioural skeletons)	Possible, in principle
Performance	Assessed. Comparable with the ones achieved using classical, non structured programming environments.	In principle, same as for skeletons.
Time to deploy	Greatly reduced w.r.t. classical parallel programming frameworks, both design and implementation/tuning/debugging time	Quite limited design time. Implementation time comparable to the times needed when using classical parallel programming frameworks. Tuning and debugging time reduced.
Debugging	Debugging of sequential code portions (only) is needed	Parallel pattern and sequential code debugging nedeed

Figure 6.11 Summary of features in parallel design pattetns and algorithmic skeletons

consider the "pipeline" pattern included in "organize by flow of data" pattern group in the "algorithm structure" design space. The "supporting structures" design space does not provide a specific pattern suitable to be used to implement a pipeline algorithm. Although both the "SPMD" and the "master/worker" patterns may be adapted to execute "pipeline" stage tasks, a "pipeline" "supporting structure" pattern would be greatly useful and not too much difficult to introduce.

Actually, different authors–including the author of these notes–recognize that only three fundamental "supporting structures" are needed to implement any parallel application, provided a convenient number of "connector patterns" are also provided. These fundamental "supporting structure" patterns are the "string of workers"–a number of concurrent activities processing relatively independent tasks in parallel–the "chain of workers"–a number of concurrent activities each processing results from the previous one and delivering results to the next one in the chain–and feedback loops–communications channels driving back some results to previously executed concurrent activities. The "connector patterns" needed to set up proper networks of concurrent activities out of these supporting structure patterns-i.e. to manage communication and synchronization among the concurrent activities–include point-to-point and collective communications, lock/mutex/semaphores and barriers and logically shared data structures.

CHAPTER 7

SKELETON DESIGN

In this Chapter we discuss the problems related to skeleton (set) design. In particular, we want to answer two questions:

- How can we identify suitable skeletons/parallel design patterns?
- How can we design a good skeleton/parallel design pattern?

In order to answer these question we start considering again the definition of algorithmic skeleton we gave in Chap. 3:

> *An algorithmic skeleton is parametric, reusable and portable programming abstraction modeling a known, common and efficient parallelism exploitation pattern.*

From this definition we can figure out the principles that should inspire the process of identifying and designing new skeletons.

→ We should look for *parallelism exploitation patterns*. This means that parallelism is the main subject, the kind of parallelism modeled has to be clearly identified and described and, obviously, the pattern should be different from the patterns already modeled by other already defined skeletons or by a composition of already defined skeletons.

→ The skeleton should be *parametric*. It should be possible to specialize the skeleton by passing a proper set of functional and/or non functional parameters. In particular all the computations differing by features not relevant to distinguish the parallelism exploitation pattern used must be expressible with the same skeleton by simply instantiating it with different parameters.

→ The skeleton should be *reusable*: a number of different situations, possibly in different application domains should exist suitable to be modeled by the skeleton. In case

we have only one or two[100] situations where the skeleton may be used to model parallelism exploitation, the skeleton is not worth to be included in the skeleton set, unless it results to be the only way to model that particular situation and the few cases where the skeleton may be used cover very common application cases.

→ A skeleton is a *programming abstraction*. Therefore the new skeleton should clearly abstract all the details of the modeled parallelism exploitation pattern. Moreover, it must be provided to the final user[101] in a way which is compliant to the general rules used to define the other skeletons in the current skeleton set and respecting the syntax rules inherited from the skeleton "host language"[102].

→ The new skeleton should be *portable* and *efficient*: implementation should exist for the main class of parallel architectures and these implementations should be efficient, although efficiency may vary across different architectures. Take into account that we are defining a new skeleton–which is a kind of abstract concept–while the portability and efficiency are properties related to implementation, actually. The point here is that when we design a new skeleton we should be able to imagine how it can be implemented efficiently on a range of target architectures. This to avoid the situation where we have a very nice "instruction set"–the algorithmic skeleton set–but this instruction set could not be implemented efficiently and therefore no user will be prone to adopt it and to use it.

→ Last but not least, the new skeleton should abstract a *common* parallelism exploitation pattern. Here we have to make some distinction. Common parallel patterns are the things we are actually looking for, when designing a new skeleton. However, the adjective "common" may be interpreted in different ways. We can intend "common" as "appearing in wide range of different applications *from different application domains*", or we can release the "different application domains" requirement while keeping the rest of the sentence. In the former case, we will end up with true algorithmic skeletons. In the latter, we will end up with more *domain specific* skeleton sets. In principle, both true and domain specific skeleton sets are worth to be investigated and designed. We must point out that several of the skeleton frameworks designed in the past (or currently being maintained) were somehow "domain specific", at least in their author intentions: Skipper [78, 40] has been developed to support image processing and MALLBA [2] is aimed to support optimization algorithm design, as an example.

7.1 COLE MANIFESTO PRINCIPLES

In his algorithmic skeleton "manifesto" [38] Cole lists four principles which "should guide the design and development of skeletal programming systems". These systems can be read as principles dictating requisites on the skeletons that have to be eventually included in the skeleton set provide to the application programmer. Here we recall the Cole principles and we comment their impact on the skeleton design process.

1. Propagate the concept with minimal conceptual disruption. The principle, as illustrated in the manifesto, states that "simplicity should be a strength". If we also take into account that following a RISC approach in the design of the skeleton set[103] is definitely a good

[100] more in general, just a very few
[101] the application programmer
[102] the language used to express sequential portions of code and data types
[103] that is trying to define a small set of simple and efficient skeletons that may eventually be combined to achieve more complex parallelism exploitation patterns

practice, then it is evident that the development of *simple*, *general purpose* and *composable* skeletons is the final goal we must try to achieve. In addition, these skeletons must be provided to the application programmers in *a way as compliant as possible* with the rules and conventions of the sequential programming framework used to embed/host the skeleton framework.

2. Integrate ad-hoc parallelism. The principle was stated by Cole to point out that skeletons should support the integration of parallel forms not originally provided by the skeleton set designer. These parallel forms should be supported in such a way that they eventually integrate in the rest of the skeleton set. As an example, the user must be enabled to program its own parallel pattern and then the skeleton framework should allow the user to use the pattern as a skeleton in all the places where one of the original skeletons could be used. Cole's eSkel skeleton framework [26], as an example, provided the user with the possibility to autonomously manage a set of MPI processes allocated by the skeleton system and linked with proper communication channels to the rest of the skeletons used in the application. With respect to the different new skeleton design problems discussed in this Chapter, this principle has no particular effect. However, when considering the design of the overall skeleton set we must be careful to introduce some kind of *escape* skeleton suitable to support user[104] defined parallelism exploitation patterns and perfectly integrated with the rest of the skeleton set. In Sec. 10.6 we discuss how skeleton set expandability may be achieved in a macro data flow based implementation of a skeleton framework. In this case, the "escape" skeleton is the one where the user may directly define the macro data flow graph that constitutes the result of the compilation of the skeleton.

3. Accommodate diversity. The principle states that when designing skeletons "we must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility", that is we have to consider that simple skeleton are worth provided the simplicity of the skeleton does not impair the possibility to use the skeleton to model slightly different parallelism exploitation patterns. Here we can assume that this principle has two main consequences on the design process. On the one hand, the new skeletons should be properly defined using a significant set of parameters. Possibly, some of the parameters may be optional. The parameters should be used to differentiate the general skeleton behaviour and clear semantics should be given (functional and parallel) that states the effect of the combined usage of the parameters when instantiating the skeleton in a real application. On the other hand the new skeleton should comply the rules defined when looking for the possibility to "integrate ad-hoc parallelism", in such a way the user may extend parts of the skeleton to implement his/her slightly different pattern. As an example, when defining a farm, the user may wish to implement some particular scheduling policy, different from the system defined one, to support other kind of computations with respect to those supported by the original farm.

4. Show the payback. This principle is more related to the design of the overall skeleton set, actually. However, in order to build a skeleton framework able to "show the payback" to the final user, we should include in the skeleton set only new skeletons that i) may be used in a variety of well known situations, in different application fields, ii) should be high level, that is they should not include (too much) implementation dependent details, and iii) they should be easy to use, that is they should be perfectly integrated with the overall programming framework.

[104] application programmer

7.1.1 Summarizing ...

If we try to summarize what we discussed above, we can obtain the following set of requisites for a new skeleton to be included in the skeleton set provided to the application programmer:

a) it must model a common, efficient parallel pattern

b) it must be easy to use

c) it must be defined in terms of a comprehensive set of parameters

d) it should tolerate/support user defined variations

e) it must be efficiently implementable on a range of parallel target architectures.

7.2 LOOKING FOR (NEW) SKELETONS

In this Section we consider a methodology for discovering and designing new skeletons. Actually, we will distinguish two cases: one based on the analysis of existing parallel applications, and the other one based on the specific needs of a user implementing a particular application. In reality the second one constitutes and adaptation of the first one with just a different input/starting point.

7.2.1 Analysis

In this case, the idea is to look for new skeletons in existing parallel applications. Therefore the procedure may be outlined as follows:

1. We decide whether to analyze applications from different domains or from a single application domain. In the first case, the procedure will most likely terminate with the individuation of new general purpose skeletons, whereas in the second case it will most likely terminate with the individuation of domain specific skeletons. In both case, we select a number of applications *whose code is accessible* and/or *whose parallel algorithm is clear*.

2. We "distill" from the application individuated in the previous phase the parallel patterns used to express and model parallelism. This activity may require some kind of "reverse engineering" of the source code. In some cases some support by the application designers could be necessary to avoid digging huge amounts of source code. It has to be considered that in normal parallel applications the parallel structure of the application is not exposed "per se" and therefore we need to look at the mechanisms used to implement parallelism exploitation in order to be able to re-construct (reverse engineer) the high level parallel structure of the application.

3. We select the candidate new skeletons among those distilled in the previous phase that have no correspondent[105] skeleton in the currently available skeleton set.

4. We perform a comprehensive analysis of the candidate new skeletons. In particular, we try to understand whether the each new skeleton candidate is in case equivalent to some composition of other, new and simpler skeletons. In case, we try to understand whether these simpler skeletons are worth to be included in the skeleton set, that is if they actually model significant parallel patterns (or, possibly, significant parallel

[105]that is, whose parallel exploitation pattern could not be modeled using the skeletons currently available either alone or in proper nesting

pattern building blocks). If the new skeletons individuated cannot be further decomposed, we investigate if the new skeleton satisfies all the requirements listed at the end of the previous section, derived from either the algorithmic skeleton definition or from Cole's manifesto.

5. We proceed with new skeleton refinement. We design the skeleton name, syntax, parameter set and semantics. In the design of the skeleton parameter set we take into account both functional (e.g. worker functions, task and result data types) and non functional (e.g. parallelism degree, scheduling policies (if any)) parameters, and we also take into account any optional parameter that may be used to further differentiate the skeleton parallel pattern. When defining semantics, we define both functional and parallel semantics, as usual.

7.2.2 Synthesis

In this case the idea is to proceed in the definition of new skeletons starting from the particular needs of an application programmer trying to implement a specific parallel application. The process starts in case we verify that a pattern, significant and useful to implement (part of) the parallel application we are considering, cannot be expressed by any of the existing skeletons or by a composition of the existing skeletons. We then proceed as follows:

1. We perform an analysis phase such as the one listed in point 4 of the procedure outlined in Sec. 7.2.1.

2. Then we look for other cases where such a skeleton may be used. We therefore follow a procedure similar to the one listed in points 1 and 2 of the procedure outlined in Sec. 7.2.1 to individuate the application parallel patterns and we look for equal or "similar"[106] patterns. If a number of situations are individuated where the new pattern may be used (as is or slightly modified) we go on with the new pattern design process, otherwise we stop here.

3. We refine the skeleton definition, also taking into account the needs evidenced by the other situations where the new pattern could be eventually used, and we eventually define the new skeleton name, syntax, parameters and semantics.

7.3 SKELETONS VS TEMPLATES

An algorithmic skeleton is fundamentally a programming abstraction. A template, as introduced in this book, is an implementation of a skeleton on a given target architecture, including a process template (or a thread template) and some kind of performance model.

When looking for new skeletons it is easy to be convinced that a *recurring implementation* of some parallel pattern is a skeleton, especially during a process such as the one described in 7.2.1. We should take care of not confusing the parallel programming abstraction with the implementation, however.

E.G.▷ As an example, take into account a task farm skeleton. The programming abstraction presented to the user is such that the user can "submit" a set of independent tasks–possibly available at different times–and get them computed in parallel, with a speedup hopefully proportional to the number of parallel processing engines (processes on different processing elements, threads on different cores) used.

The emitter and collector concepts we introduced and we used several times in the book *are not* logically belonging to the task farm skeleton. Rather, they are related to a possible,

[106] with respect to our "new skeleton" candidate

relatively efficient implementation of the task farm. In fact, we pointed out how a similar implementation may also be used to implement a map skeleton, provided the emitter splits the incoming "collection" data into items to be delivered to the workers for independent computation and the collector gathers all the results from the workers and eventually "builds" the map result. ■

The point here is particularly important if we refer back to the discussion relative to CISC vs. RISC skeleton sets we introduced in Sec. 3.4. When looking for RISC skeleton sets, we may easily end up with the idea that the "elementary" building blocks are the simplest, possibly recurring, patterns used to implement the skeletons rather that the skeletons themselves. We may consider the emitter, a string of workers, a collector, a chain of workers, a tree of workers as the base building blocks of our parallel computations. This may be reasonable in perspective, but we have to take into account that:

- the "assembly" of these building blocks requires a deeper knowledge than the one required to nest "true" skeletons. As an example, to build a map out of the emitter, string of workers and collector building blocks, we need to be aware of the decomposition level used to feed workers. In a map skeleton, this is a parameter usually left in charge of the map implementation, and therefore under the responsibility of the system programmer rather than the of the application programmer.

- the level of abstraction of these basic building blocks could not avoid taking into account the target architecture, as skeletons in principle may do. An emitter building block for a complete interconnection architecture such as an workstation cluster or a symmetric multiprocessor (multicore) will be different from the same building block for another architecture not supporting complete interconnection, such as a mesh interconnected network of workstations/cores or a NUMA multi core. In addition to the already mentioned shift of responsibilities from system to application programmers this has also huge consequences on portability of parallel programs on different architectures.

CHAPTER 8

TEMPLATE DESIGN

An implementation template is an implementation of a specific skeleton on a specific target architectures. The designer of an implementation template is the "system programmer". The end user of the template will be the application programmer, although he/she will never directly use the template. Rather, he/she will use an algorithmic skeleton and the compiler tools (and/or the runtime) of the algorithmic skeleton framework will eventually implement the skeleton through an instantiation of the template.

When designing a template, a system programmer goes through a number of different phases, each requiring different kinds of "system" knowledge.

→ First of all, *possible alternative templates should be individuated.* All these alternatives must be suitable to efficiently implement the target skeleton, and therefore the system programmer has to keep in mind the precise semantics of the parallel pattern modeled by the skeleton. As these alternatives will eventually use the mechanisms and possibilities offered by the target architecture, the system programmer should be also an expert programmer of the target architecture.

E.G.▷ We consider the design of an implementation for a reduce skeleton on a complete interconnection architecture. Two alternatives may be considered (see Fig. 8.1):

1. a tree structured process/thread network where each node computes the reduction of the results provided by the children nodes, and
2. a string of workers locally computing the reduction of a partition of the original data and delivering the local result to a single node eventually computing the global result of the reduce.

———————————— ■

→ Then, *before* moving to the design of the template implementation, a *performance model has to be designed* capturing the performance related aspects of the template.

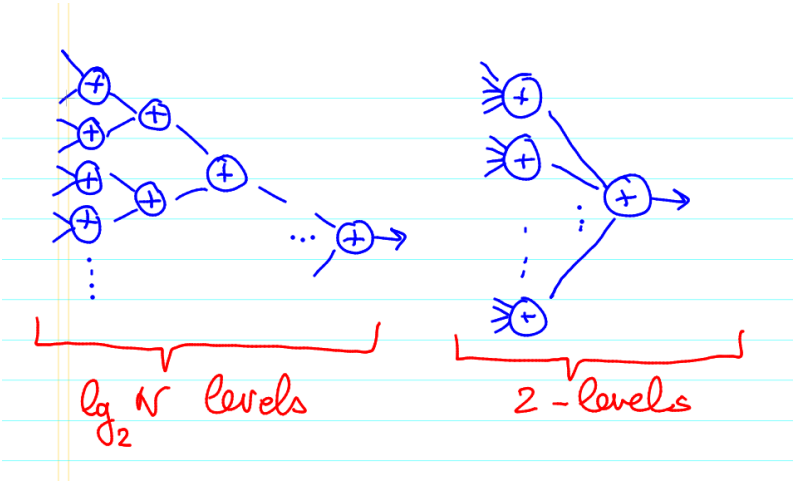


Figure 8.1 Alternative implementation templates for a reduce skeleton on a complete interconnection architecture

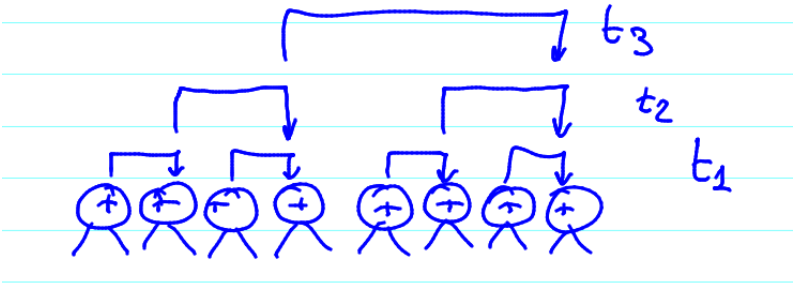


Figure 8.2 A third alternative implementation template for a reduce skeleton on a complete interconnection architecture

Not having yet started the implementation of the template, the performance model considered here is an abstract performance model. However, it has to take into account the "kind" of mechanisms used on the target architecture along with their general costs.

E.G.▷ We consider the two templates of Fig. 8.1. The first one has $log_2(N)$ communications $+ log_2(N)$ computations of the reduce operator $\oplus$ and needs $N-1$ processing elements (N being the number of the items in the data structure to be reduced). The second implementation uses $M+1$ nodes and takes $\frac{N}{M}T_\oplus + M(T_{comm} + T_\oplus)$. An alternative could be considered where the $\frac{N}{2}$ elements at the base of the tree are reused to perform the inner node computations (see Fig.8.2). The implementation is anyway taking $log_2(N)(T_\oplus + T_{comm})$. ■

→ Looking at the performance modeling developed in the previous phase, an *evaluation of the alternatives* has to be performed. The result of the evaluation process will be such that either a single "better" implementation alternative exists–and therefore it will be the only one considered for actual implementation–or there will be multiple alternatives with close performance figures–and in this case, they could all be considered for implementation.

E.G.▷ Fig. 8.3 plots the curves relative to the abstract completion time for the two reduce templates with varying parameters. From these performance models we may conclude that the "tree" template is better suited in case of computationally intensive $\oplus$ computations, whereas the other one is more suitable in case of "light" computation and more consistent–comparatively–communication times. Considering these results, we should keep both alternatives as depending on the size of the input data and on the time spent in communications and in the computation of the reduce operator $\oplus$ the first alternative may better than the second one or vice versa. ◼

→ Before moving to actual implementation of the template–that is, to the coding phase– the *template interface* has to be designed. This is the interface that will be eventually exposed to the compiling tools. In a sense, the template interface contributes to the definition of the "assembler" target by the skeleton framework compiler tools. The interface should therefore be rich enough to allow a certain degree of freedom in the usage of the template by the compiler tools and, in the meantime, simple enough to avoid complicated compiler algorithms. Taking into account that an implementation template is a *parametric process (thread) network* most of this phase is related to the proper design of the template instantiation parameters.

E.G.▷ In case of the reduce templates we may simply include parameters to give the number of parallel processing elements to be used, in both cases. The computation to be performed (the $\oplus$ code) and the data types processes (the type of the $\oplus$ operator *alpha*, being $\oplus : \alpha \times \alpha \to \alpha$) should also be included in the parameters. ◼

→ The last phase of the template design is the *actual implementation of the template*. An implementation of the template is produced exploiting at best the tools and mechanisms available on the target architecture.

8.1 TEMPLATE BUILDING BLOCKS

An implementation template for a $\langle skeleton, target\ architecture \rangle$ pair is never designed as a monolithic block. Usually, we design an implementation template composing some template building blocks. These templates building blocks include:

- Process networks with given topologies
- Thread sets with given behaviour (e.g. producer/consumer configurations)
- Communication infrastructures
- Shared state configurations
- Synchronization infrastructures.

Is is clear that the availability of a set of efficient template building blocks for a given target architecture covering all the aspects related to concurrent activity set up, as well as synchronization and communication handling greatly enhances the possibilities to implement efficient, correct and usable templates.

In general, when designing a template, the existing knowledge concerning implementation of non-structured parallel applications is used as much as possible, in such a way we capitalize on already existing results and we avoid duplication of efforts to come to the same results.

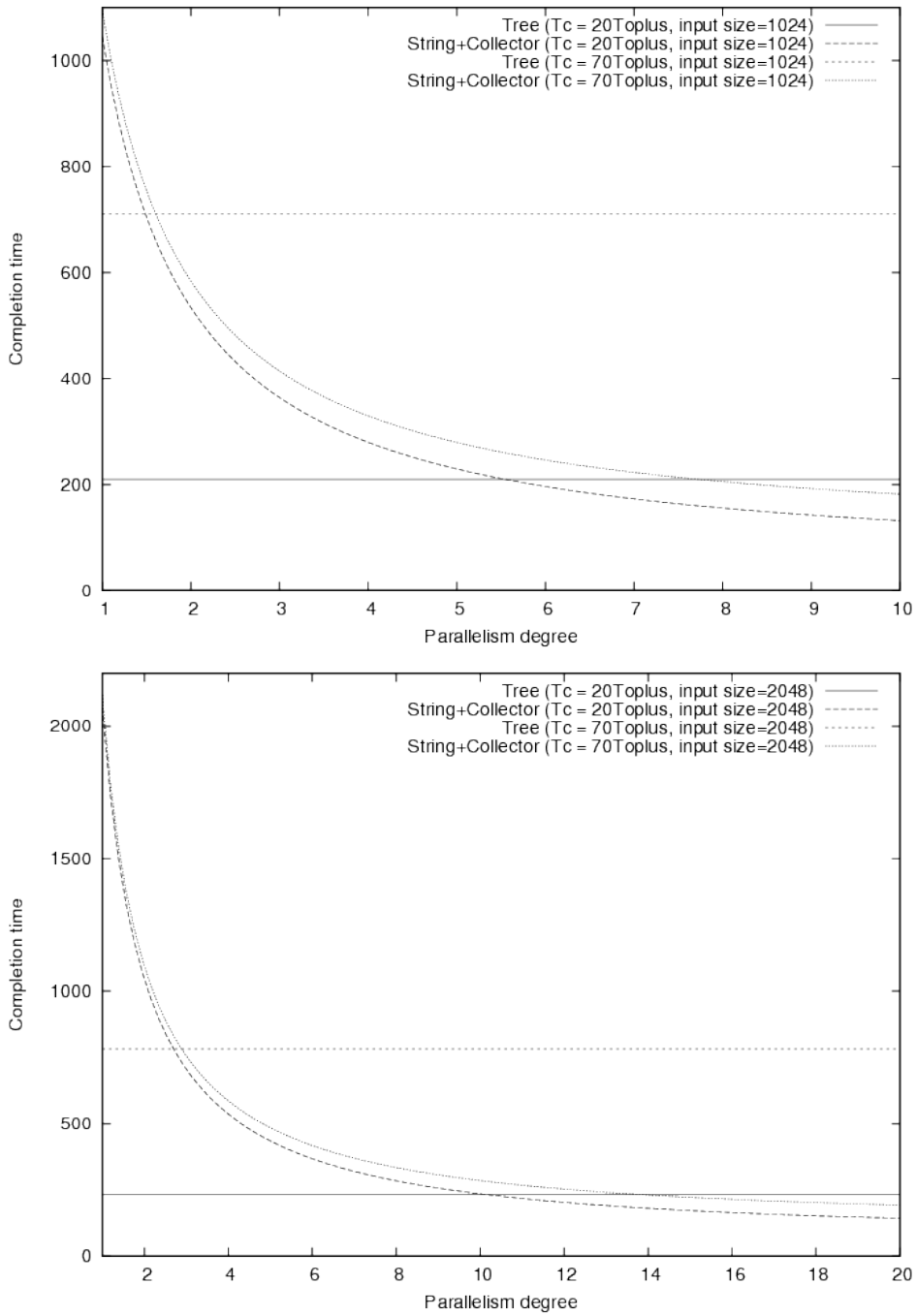


Figure 8.3 Performance comparison of the two reduce templates mentioned in the text. Left: input data structure of 1K items, right: input data structure of 2K items. $T_{\oplus}$ has been considered the unit time. T_{comm} is defined in terms of $T_{\oplus}$. Sample values plotted to show differences. The tree template has been considered with a fixed parallelism degree (the ideal one). The string+collector template has been considered with a varying number of workers in the string.

Here we want to discuss three different template building blocks, to give an example of how existing results may be exploited in the definition of an implementation template for skeletons. These three techniques may be composed with other techniques to implement different kind of skeletons.

8.1.1 Client-server paradigm

In a client server paradigm, a server provides some kind of services to a set of clients invoking those services through properly formulated service requests. The client server paradigm is pervasively present in computer science and networking. The DNS is structured after a client server paradigm as well as NFS or Email and the whole WEB systems, just to name a few examples.

The number of clients, the kind of the service (stateless, state-full), the kind of the server (single or multiple threaded) as well as the communication protocols used to deliver service requests and service results determine the performance of the client server paradigm.

In general, the performance of a client server system may be modeled using the queue theory as sketched in Sec. 5.7.2.

In our perspective, the client-server paradigm is useful to model the interactions within process template components. As we already pointed out, the interactions among a string of worker processes and some collector process gathering the worker results may be interpreted as a client-server paradigm, where the workers play the role of clients and the collector process plays the role of server. In a dual way, the interaction between a task scheduler and a string of workers computing the scheduled tasks can be interpreted as a client-server paradigm where the scheduler plays the role of client with respect to a set of workers that represent the servers.

More modern push/pull variants of the client-server systems may also be considered in this scenario. The classical client-server paradigm is the pull one. In this case the client sends a message with a service request to the server and the server answers back with the service result. Therefore the client initiates the server activities.

When a *push* client-server paradigm is adopted, instead, the server sends service results on its own initiative[107]. When designing templates, the push client-server paradigm can be used to model worker interactions with the scheduler in case auto scheduling is implemented. In this case, the scheduler process (the client, as we mentioned before) receives "push" messages from servers (the workers in the worker string) signaling the server may proceed computing a new task. As a consequence, the client sends back to the server a new service request[108].

All the existing technology used to implement servers for purposes different from structured parallel programming may be reused in templates. As an example, server multi threading techniques may be used to implement workers partially able to hide communication latencies even in case of communications not completely overlapping with computations (see Sec. 8.3.1).

8.1.2 Peer-to-peer resource discovery

In many cases, we face the problem of recruiting resources to be used in the implementation of a given template. Processing resources may be looked up in proper resource data bases or resource tables available from the system. More often we wish to be able to *discover*

[107] after an initial client "registration", of course

[108] actually, this situation may also be interpreted as a reverse server-client setting: the workers behave as clients with respect to the scheduled behaving as a server. Indeed, if the results have to be directed to a different concurrent entity–such as a collector process/thread–the reversal of client and server roles only moves the "push" paradigm to the collector side

the resources currently available to be recruited to our parallel computation. Some kind of parallel/distributed discovery mechanism is therefore needed.

A generally adopted discovery mechanism is based on (kind of) peer-to-peer techniques. When looking for a resource, we start a process similar to the peer discovery process in peer-to-peer computations:

- Either we flood the network with a proper discovery message–with its own Time To Live field–and then we wait for answers coming from the (remote) processing elements running the skeleton run time and available to support new computations, or
- Or we lookup a well known "service server" and we ask the server a list of the available peers/resources.

In both cases we may rely on all algorithms developed for the classical peer-to-peer paradigm. Therefore flooding may be organized in such a way i) flood messages do not survive a number of hops larger that the initial Time To Live, ii) flood messages are not forwarded back to the sending node, and iii) the reached peers directly contact the originator of the flood request rather than simply sending back a presence message to the host delivering them the flood discovery request.

In case we use this discovery approach, we need to insert in the template run time proper mechanisms to react to discovery flood messages, of course.

A variant of the pure peer-to-peer discovery may be used in networks supporting broadcast and multicast, such as all Ethernet variants. In this case, the host trying to discover resources to be recruited to the parallel computation may simply broadcast–or better multicast–a discovery message on a given port. The resources running the template run time will answer the discovery message. The order of arrival of the answers to the discovery message may also be interpreted as a measure of the "health" of the answering machine: answers coming late will be a symptom of poor network interconnection or of a very busy resource, in most cases.

Again, all existing peer-to-peer results here may be re-used to design a proper "resource discovery" template building block.

8.1.3 Termination

Termination of parallel programs is quite a non trivial tasks. If a number of parallel/concurrent/distributed activities are contributing to the computation of the very same final results it must be the case they both interact–through messages or through some kind of shared memory–and synchronize–again, either through messages or through proper synchronization primitives.

When designing templates, we must be careful to support termination of *all* the activities started, even in presence of errors–abnormal terminations.

A technique suitable for terminating all the activities in presence of normal termination (no errors) will be such that a tree is build at some point, whose nodes cover all the parallel activities eventually set up in the program and a node has as leaves all those activities that have to terminate after the termination of the node itself. This tree may be used to propagate messages from the root activity to the leave activities without incurring into the unpleasant situation of terminating activities *before* they actually finished interacting with other concurrent activities, and therefore leading to errors.

E.G.▷ In our stream parallel applications we can use the "end-of-stream" mark flowing through the stream channels to implement termination. The end-of-stream mark is a conventional message flowing through stream channels denoting the fact no more tasks/data items will flow along that particular channel. We have to be careful to "count" the number of incoming

marks and to produce the correct number of out-coming marks. As an example a farm emitter should emit n_w EOS marks after receiving one input EOS. Workers in the farm should simply propagate the received EOS. The collector in the farm should count n_w marks and forward only the last one. ■

It has to be clear that the channels supporting the termination messages may in general be different from the channels moving data (input tasks and results) across the template activities.

E.G.▷ Consider again the termination of farm template just mentioned. A relatively more efficient termination procedure will the one were the emitter sends the termination signal to the worker processes after receiving from the workers requests of tasks to be computed that are no more available. The workers simply terminate when they get the termination message. The emitter also sends a termination message to the collector after having sent the termination message to the last worker still requesting tasks to be computed. Eventually, the collector terminates when it receives the termination message from the emitter. ■

8.2 CROSS-SKELETON TEMPLATES

Implementation templates are designed to support a particular skeleton on a particular target architecture. However some templates are particularly significant and they may be used to support different skeletons. A typical case is given by the template implementing the task farm skeleton. The very same template may be efficiently used to support also data parallel skeletons such as the map one.

A different case is the one where a skeleton is implemented in terms of another (composition of) skeleton(s). Again the case arises when we consider data parallel skeletons. In a data parallel skeleton, an input data structure is split in partitions–possibly in single items–and then some kind of computation is performed on all these partitions. The data parallel skeleton result is build gathering the results of the partition computations.

This parallel pattern may be easily transformed in a stream parallel pattern as follows:

i) A three stage pipeline is used

ii) The first stage decomposes the input data structure into partitions. As soon as a partition has been created, it is delivered onto the stage output stream

iii) The second stage is a task farm, processing a stream of partitions to produce a stream of partition results

iv) The third stage is again a sequential stage, "collapsing" its input stream of results into the final result of the data parallel skeleton.

This corresponds to a skeleton-to-skeleton transformation, actually, even if the sequential parameters of the data parallel skeletons must be completed with some ad hoc code to produce a stream and to consume a stream in the pipeline of farm version.

If we implement a data parallel construct this way we are *de facto* using the stream parallel skeletons as if they were implementation templates. This is the reason why we mention this case in the template design chapter and we name it *cross-skeleton template*.

In many cases, especially when the target architecture has mechanisms providing good support to stream parallelism, this implementation template technique may be very effective. FastFlow [82] only provides stream parallel skeletons including task farms and pipelines. However, it may be used to implement data parallel skeletons as well as the FastFlow task farm template is parametric and the scheduling of the emitter may be user defined, as well as the gathering strategy of the collector process/thread. As a consequence,

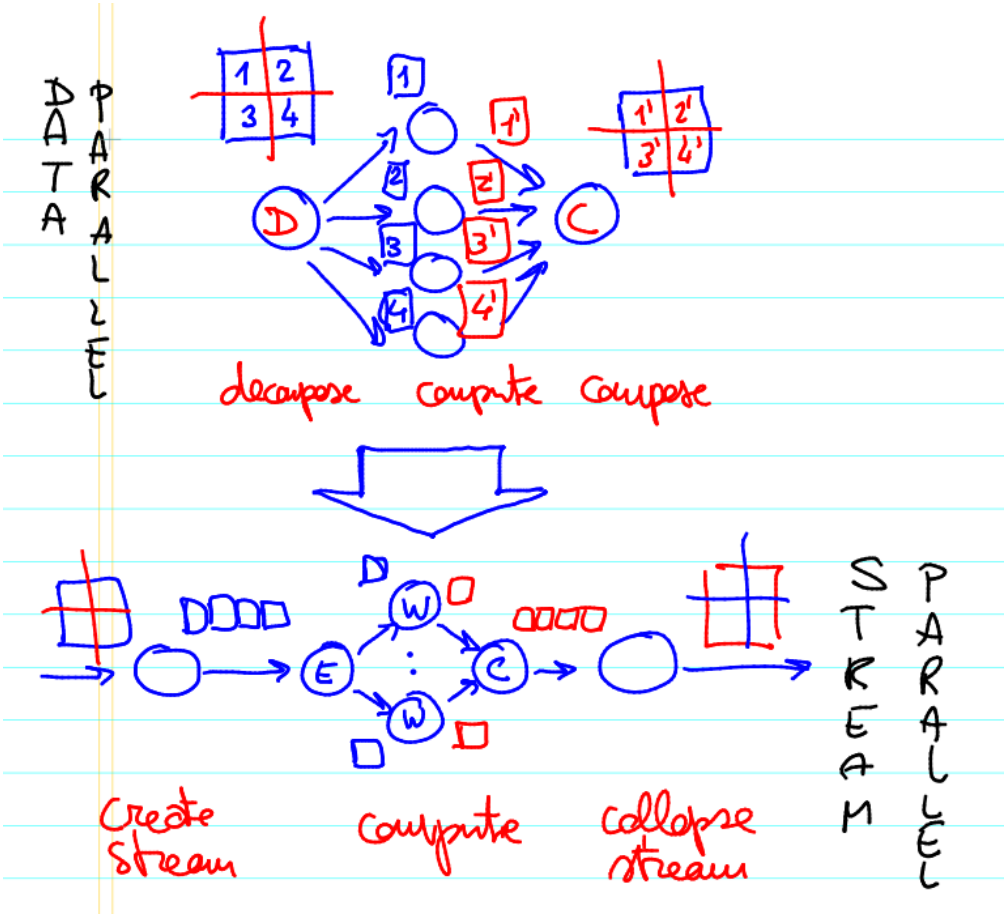


Figure 8.4 Implementing data parallel constructs in terms of stream parallelism

the stream parallel skeleton task farm may be used as cross-skeleton implementation template of the map data parallel skeleton (see Fig. 8.4).

We insist here the usage of the farm *template* is used as an implementation of the map *skeleton* rather stating that a map skeleton is provided because the details the user–i.e. the application programmer–has to express to use the task farm as a map implementation force him/her to think to the implementation rather than to reason only in terms of abstract parallel patterns.

8.3 SAMPLE TEMPLATE MECHANISMS

We introduce here some techniques that could be useful to implement templates on both shared memory and distributed memory architectures, such as multi cores and COW/NOWs. These techniques may be used as mechanisms in the implementation of templates. They are therefore template building blocks as the three listed in Sec. 8.1, but we discuss them separately as they are building blocks abstracting lower level features with respect to the other ones.

8.3.1 Double/triple buffering

When implementing communicating processes (or threads), a useful techniques consists in implementing double or triple buffering, provided hardware supports at least partial

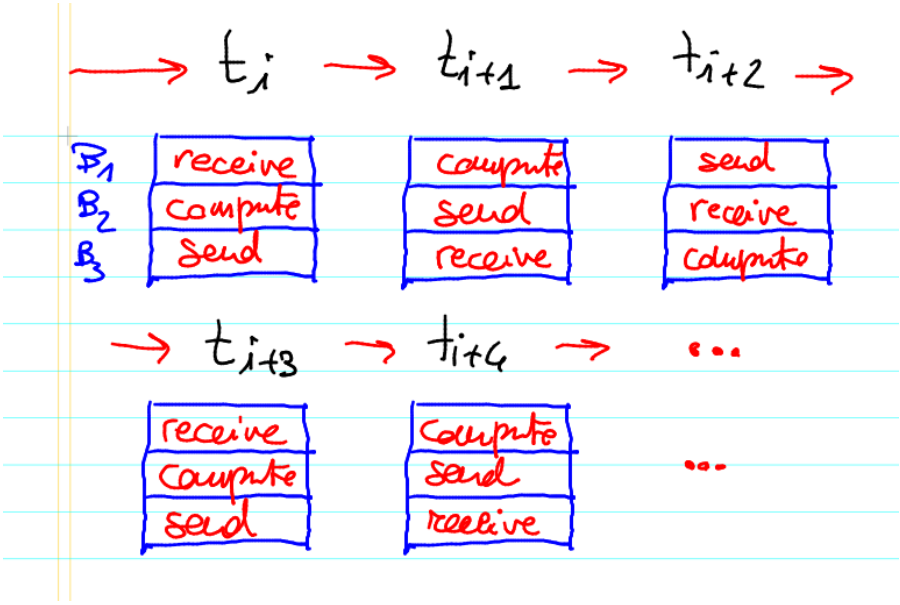


Figure 8.5 Triple buffering

overlap of communications and computations. Triple buffering can be used to implement any process receiving input data from and sending results to other processes. It works as follows (see Fig. 8.5):

- at a given time, one concurrent activity is in charge of receiving the "next" data to be used as input task, another concurrent activity is in charge of delivering the last result computed and a third activity is in charge of computing the last input task received. The three activities work on three different buffers: B_1 to receive data, B_2 to process data and B_3 to send results.
- after having concurrently completed the three steps–a logical *barrier* synchronization[109] is needed here–the three concurrent activities exchange their role. Activity i starts behaving as activity $(i+1)\%3$ in the previous step. Therefore the activity having received an input tasks starts computing it, the one that just computed a result starts sending it, and the one that just sent a result starts receiving a new task. Of course, each activity uses its own buffer for its new duty.

If there is complete overlap in between communication and synchronization, the triple buffering works as an "in place pipeline" and the time spent to compute one step is the maximum of the latencies of the three activities: computation, receiving a task, sending a result. The idea of re-using the buffers this ways also avoids unnecessary–and source of overhead–copying/sharing among the three concurrent activities.

Double buffering just uses two activities, one in charge of the computations and one in charge of the communications–both sends and receives.

8.3.2 Time server

Very often in a parallel computation you need to know wall clock time of some events. This may be needed for debugging as well as for fine tuning. However, events happening on

[109] in a barrier synchronization n activities wait each other. As soon as all the involved activities agree on having reached the "barrier", all the n activities may leave the barrier and continue their normal activity.

different processing elements may have access to separate clocks and therefore any message such as "the event e_i happened at time t_i" only makes sense with respect to over events generated on the same processing element–that is reading the same global clock.

When the relative ordering of the events is important this is obviously not enough. Several algorithms have been developed in the past to keep clocks of networked workstation synchronized or to support partial ordering of events happening on different workstations. One of the most popular algorithms suitable to establish a partial ordering among events happening in a workstation network is due to Lamport [61] and dates back to 1978. The algorithm is based on local logical clocks and computes a partial ordering among events happening on communicating workstations. NTP, the Network Time Protocol based on Marzullo's algorithm [64], is used to keep clocks of a network of workstations synchronized in modern operating systems (e.g. Linux).

In a number of cases, however, we may be interested in knowing an *approximated* estimate of the time of an event. In these cases, the usage of a centralized *time server* may be useful. Processes or threads wishing to signal/register an event, send messages to the centralized time server through a TCP server socket published on a well know port. The messages contain the *time stamp* of the event (i.e. the time when the event happened) registered using the local clock. The centralized time server, receives the messages from all the distributed participants to the parallel computation and stores the messages adding its own local time stamp relative to the reception of the message. Events are eventually reported back to the user/programmer sorted by their receiving time. As both local and time server time stamps are shown, the user/programmer may eventually sort out an approximate relative ordering of the distributed events as well as the time taken for a given activity to complete by computing the difference between the starting and the ending event time. If the activity has been performed locally to a single workstation, the difference will be computed on local time stamps. Otherwise, it will be computed on the time server "receive" time stamps.

This approach works pretty fine in case i) the latency of the communications to the time server is uniform (the same) for all the parallel computation partners and ii) the communications introduced to gather time stamps do not sensibly impact/disturb the implementation of the communications needed to carry on the parallel application. As usual, the usage of a *centralized* time server impacts scalability. In case a huge number of distributed actors are involved, the parallelization of the time server has to be taken into account. As an example, *local* time servers may be used that eventually[110] direct the gathered data to a centralized server.

8.3.3 Channel name server

When setting up a concurrent activity network on a distributed architecture we are faced with the problems of setting up proper communication channels. In fact, in order to map our concurrent activities we i) first need to assign each concurrent activity to one of the computing resources available and then ii) we need to set up communications channels interconnecting the interacting concurrent activities in the concurrent activity graph.

This second activity is referred as the "interconnection network setup" and takes place at the very beginning of the parallel computation, actually *before* the parallel computation starts. Therefore we can implement it using a centralized server as the bottleneck obviously introduced only affects set up time of the parallel computation, not the time spent in actually performing the computation itself.

A common technique consists in the following steps:

[110] at the end of the application, or during the application but implementing some kind of data compression to reduce the network traffic

i) a centralized *channel server* is created and run, providing two basic services: i) registration of a communication channel and ii) lookup of channel names. The registration of a channel registers the *address* of the channel (e.g. an TCP socket address (IP + port number), a SysV message queue (the queue id), etc.) associated with an application dependent and global *channel name*. The lookup of a channel name consists in sending back the associated address info or a lookup failure message in case no channel has been registered with that name up to the moment the lookup request is processed.

ii) Any concurrent activity started on one of the distributed processing resources initializes the input communication channels and register these channels to the channel server. This initialization/registering phase happens right before the actual "body" of the concurrent activity starts executing.

iii) Any concurrent activity that will eventually direct messages to another concurrent activity looks up the proper channel address by sending a lookup request to the channel server. In case the lookup fails, the lookup request is re-send after a short timeout and the process is iterated until the address of the channel is eventually received or a number or retries is reached such that the concurrent activity simply fails[111].

E.G.▷ As an example, suppose we are implementing a concurrent activity graph such as the one of Fig. 4.3 In this case, a number of concurrent activities will be deployed and started on a number of available resources. Assume E activity is started on resource R_E, C on R_C and W_i on R_{W_i}. Assume also we are targeting a cluster of workstations with communications implemented using TCP sockets. All the activities have an input channel, actually. The "names" of these channels are established in the concurrent activity graph [112]. Therefore, assuming the template name is "farm3":

- E will initialize a server socket with name "farm3-tasks-E-in" and will register its IP address and port to the channel server.
- C will initialize a server socket with name "farm3-results-C-in" and will register its IP address and port to the channel server.
- each W_i will initialize a server socket with name "farm3-task-Wi-in" and will register its IP address and port to the channel server.
- E will lookup for "farm3-task-W1-in", "farm3-task-W2-in", ..., "farm3-task-Wnw-in" on the channel server.
- each W_i will lookup "farm3-results-C-in" on the channel server.
- C will lookup for the address of the channel to use to deliver results (e.g. "pipe4-task-in").

and at this point all the concurrent activities will have all the necessary channel address.

■

8.3.4 Cache pre-fetching

When targeting shared memory architectures, such as the multi/many core architectures discussed in Chap. 1, efficiency in memory management is the major concern to be taken into account in order to be able to get performant implementations of parallel applications.

In particular, the highly parallel shared memory architectures[113] often present a NUMA (non uniform memory access) abstraction of the memory subsystem, and therefore it is

[111] this number has to be specified carefully, to avoid too early fails while other, complex or remote concurrent activities are still initializing their input channels

[112] therefore eventually in the template

[113] those with a number of cores in the 10-100 (or more) range

crucial to be able to map data into memory in such a way accesses are mostly performed as local accesses.

Cache management, that is the main mechanism supporting local memory accesses, is mostly in charge of the hardware/firmware architecture. A number of mechanisms and techniques have been developed in the past to make caches as efficient as possible in maintaining the application *working set*[114] in cache during the application computation. These techniques rely on properties of spatial and temporal locality of the memory accesses that may be observed in most of the (well written) applications, either parallel or sequential. Hardware/firmware cache management relies on two *pillars*: i) on demand data loading from upper levels of the memory hierarchy into caches and ii) (approximated) least recently used algorithms to select the "victim" cache lines to be substituted when new, fresh data is to be loaded in caches.

Recently, different mechanisms has been exposed to the programmers to improve cache management. One of this mechanisms is *cache prefetching*, that is the possibility to instruct caches to pre-load data that have not yet been accessed but that will be accessed in the near future. Different architectures offer different assembly instructions for data prefetching. Intel IA-32 and IA-64 micro architectures, as an example have different assembly instructions (IA-32 has those instructions in the SSE set, actually) suitable to pre-fetch a cache line containing a given address in all caches, or in part of the cache hierarchy (e.g. only from level 2 on).

Compiler tool chains expose these instructions as library calls that may be invoked by the programmers. As an example, the GCC compiler provides a "builtin" function `__builtin_prefetch(const void *addr, ...)` taking an address as a parameter and such that[115]:

> This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to __builtin_prefetch into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.
>
> The value of addr is the address of the memory to prefetch. There are two optional arguments, rw and locality. The value of rw is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value locality must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

It is clear that the correct (and performant) usage of this builtin requires both a deep understanding of the algorithm at hand and some analogous deep understanding of how the cache subsystem works and performs on the target architecture. If we try to pre-fetch too much/early data we can impair the mechanisms used to keep the application *current* working set instance in the cache. If we wait too much to issue the pre-fetch calls, the effect will be null. Therefore the correct usage of this kind of primitive is difficult and errors in its usage may lead to programs running worse than the original one.

When the structure of the parallel application is exposed, as in the algorithmic skeleton frameworks, something better may be done. By analyzing the structure of the parallel computation, we find out the data actually needed for the "immediately next" steps and we can therefore ask for their pre-fetch, possibly in the higher levels of the cache hierarchy,

[114] the set of data currently needed to carry on the computation
[115] http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

in such a way the current working set instance (usually in L1 cache) is left undisturbed but the forthcoming accesses to the new data items may be solved in L2 cache rather than in (NUMA) memory.

E.G.▷ As an example, consider a *map* skeleton. Assume the map workers are mapped to threads "pinned"[116] to different cores. The system programmer implementing the map template knows both the scheduling used to dispatch sub-computations to the map workers and the types and addresses of the input and output data relative to each one of the sub computation. Assume we have an input vector X of N floating point data items and the program computes a `map(X,f)`, that is it applies f to all the items of X. Assume also that for some reason worker W_j will be given the task to compute $f(x_i), f(x_k)$ and $f(x_y)$[117]. Then we may implement the map template in such a way that a task $\langle f, a, b\rangle$ is scheduled to the workers with an associated semantics such as

compute f(a) while prefetching b

to allow the worker to avoid waiting for b being fetched from NUMA memory into local cache(s), and eventually schedule to the worker a sequence of tasks such as

$$\langle f, x_i, x_k\rangle, \langle f, x_k, x_y\rangle, \langle x_y, \ldots\rangle$$

■

8.3.5 Synchronization avoidance

We consider another typical problem related to the correct implementation of parallel applications on shared memory architectures: concurrent accesses to shared data structures.

When accessing shared data structures proper synchronization primitives must be used to guarantee that the data structure accesses are correct independently of the scheduling of the concurrent activities that perform those concurrent accesses.

As an example, if a vector is accessed by a number of independent threads, usually the vector is protected with semaphores (in case of "long" accesses) or locks (in case of "short" accesses). Both semaphores and locks introduce overhead in the parallel computation. Locks usually implement busy waiting, and therefore processing power is used just to test that some concurrent access is finished. Semaphores may cause a context switch which is even more expensive in terms of time spent in the de-scheduling of the process/thread and of its subsequent re-scheduling once the conflicting concurrent access is actually finished.

Also in this case a deep knowledge of the algorithm and of the kind of concurrent accesses performed on the shared data structure may help the programmer to implement efficient synchronization mechanisms. As an example, such knowledge can be used to reduce the amount of synchronizations required as well as the to reduce the impact of these synchronization by picking up the more appropriate and efficient synchronization mechanisms.

Leaving the application programmer in charge of the complete responsibility for synchronization management may lead to a number of different problems and errors, however, ranging from inefficiencies in the parallel application to errors leading to deadlocks. In the past, programming language designers introduced a number of high level concurrent constructs capturing and encapsulating synchronization in concurrent accesses to data structures. We mention here *monitors*, as an example, introduced since '70s [52]. More recently, a kind of monitor has been introduced with `synchronized` methods in Java[118] and similar programming abstractions may be found in many modern concurrent programming languages. A monitor basically encapsulates a concurrently accessed data structure along with the

[116] i.e. bound, allocated on
[117] non adjacent items of the vector, otherwise the normal cache mechanism will be sufficient
[118] http://download.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

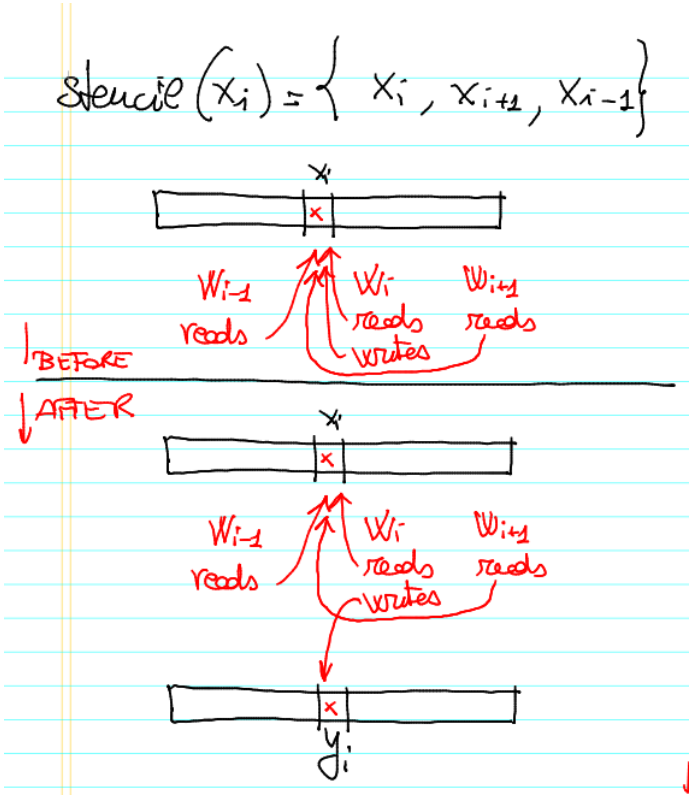


Figure 8.6 Removing synchronization in a stencil computation

methods used to access it. These methods are automatically synchronized in such a way no incorrect access to the data structure is allowed.

As algorithmic skeletons expose the structure of parallel computations and–as a consequence–also the structure of the accesses performed on the shared data structures, synchronization management may be easily left in charge of the system programmers implementing the skeleton framework, rather than in charge of the application programmer. As a consequence, the system designers–experts in both parallel programming techniques and in the mechanisms supported by the target architecture–may easily overcome the problems related to synchronization management.

E.G.▷ To illustrate the "skeleton advantage" in this case, we consider again the pure `map` skeleton. The collection processed by the map skeleton is logically shared among all the threads actually implementing the map workers. Therefore we could be reasonably convinced that accesses to the data collection *must* be synchronized. However, the map skeleton on collection $x_1, \ldots, x_n$ [119] is such that only the thread computing item i is actually reading and writing x_i. Therefore no synchronization is required and we can avoid protecting the x_i accesses with locks or semaphores.

A slightly different case happens when a `stencil map` skeleton is being computed. In this case, the worker computing element $f(x_i, stencil(x_i))$ reads x_i and the whole related stencil, actually. Those reads are concurrent with the reads performed by the worker computing $f(x_{i-1}, stencil(x_{i-1}))$ probably. However only worker computing element $f(x_i, stencil(x_i))$ will eventually write y_i, the $i-th$ position of the output collection. Therefore we can easily figure out that rather than protecting all the x_j reads and writes with locks/semaphores

[119] recall that $\texttt{map}(f, \langle x_i, \ldots, x_n \rangle) = \langle f(x_1), \ldots, f(x_n) \rangle$

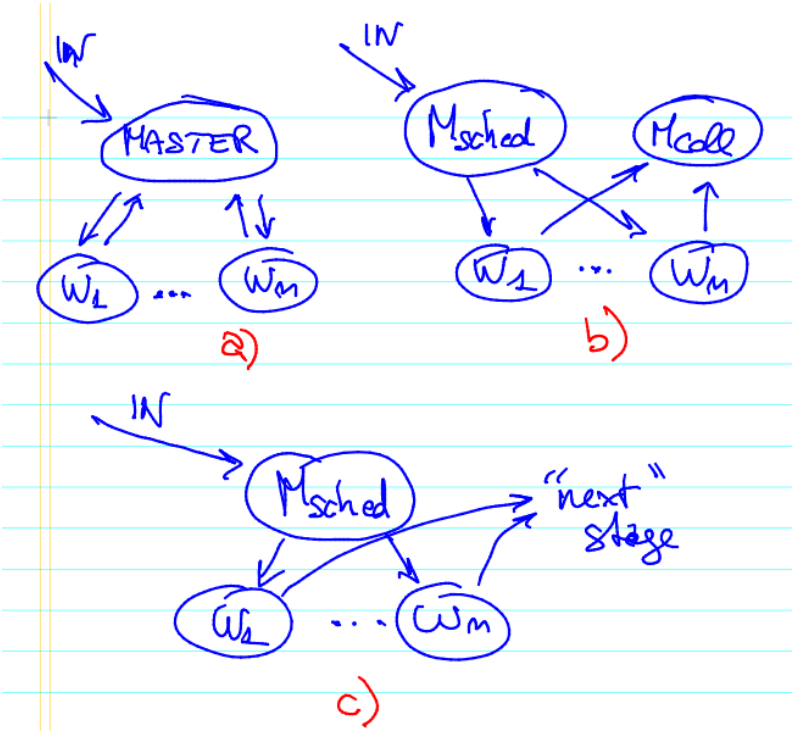


Figure 8.7 Master worker template alternatives

we can leave all accesses unprotected provided that the results are written to a different data structure, rather than "in place". In other words we can substitute the sub tree computing `stencil(f,stencil,x)`[120] with the subtree computing `stencil(f,stencil,x)` $\rightarrow$ `y`[121] provided that all subsequent accesses to x are properly renamed to access y instead. (Fig. 8.6 explains the concept assuming vector map with stencil build of the current item plus its two left and right neighbors.) ■

8.4 SAMPLE TEMPLATE DESIGN

In this Section we present two samples of the template design methodology outlined at the beginnig of this chapter, relative to a Master/worker template, suitable to implement different kinds of skeletons, and of a "farm with feedback" template, suitable to support divide and conquer like skeletons.

8.4.1 Master worker template

A master worker template is conceptually build of a *master* scheduling computations to a set of identical *workers* and collecting from the workers the computed results. This template may be used to implement, with small "variations" (possibly through proper parameters), a number of different skeletons, including task farm, map and stencil data parallel skeletons.

Here we consider the master/worker template as an implementation of a task farm skeleton, and therefore we "formalize" the procedure leading to the template already mentioned in several places in the previous Chapters.

[120] i.e. computing `new`$x_i = f(x_i, stencil(x_i))$
[121] i.e. computing $y_i = f(x_i, stencil(x_i))$

Herbert Kuchen in [75] presents a detailed discussion of the implementation of a task farm skeleton through different kinds of "master/worker-like" templates. In particular it discusses templates with a unique master (scheduling tasks and gathering results), with two masters (one for scheduling tasks and one for gathering results) and with no masters at all (to model farms embedded in other skeletons, such as a pipeline. In this case the scheduling is actually performed by the last process in the previous stage and the results are immediately directed to the next stage, rather than being collected by the master.

We took inspiration from this work (although master/slave implementation of task farms dates back to the beginning of the algorithmic skeleton history in early '90s) to start the first step in template design: *identification of the alternatives*.

Fig. 8.7 shows the alternatives we consider for the master/worker template after the Kuchen paper. The Figure actually shows alternative *implementation activity graphs* rather than complete templates. In fact, the process/thread network in Fig. 8.7 a) may be implemented using different communications mechanisms and different concurrent entities (processes, threads, lightweight GPU threads, etc.). We assume here to target a shared memory architecture, such as a NUMA multi core. We therefore restrict to the range of the mechanisms that can be conveniently used to implement the template.

As the second step, we *design simple performance models* for the alternatives chosen. In our case, we can conclude that:

- In case of template a) of Fig. 8.7 the service time of the farm template will be given by

  $$T_S(n) = max\{(T_{schedule} + T_{gather}), \frac{T_{worker}}{n}\}$$

  as the master should gather a result from a worker and deliver another task to be computed to the same worker in a single event.

- In case of the template b) of Fig. 8.7 the service time is the one we already mentioned, that is

  $$T_S(n) = max\{T_{schedule}, T_{gather}, \frac{T_{worker}}{n}\}$$

- In case of the template c) of Fig. 8.7 the service time will be the same one of template b) with the $T_{gather} = 0$, that is

  $$T_S(n) = max\{T_{schedule}, \frac{T_{worker}}{n}\}$$

Moving to the *evaluation of the alternative* phase, we must consider that in case of small schedule and gather times (as it happens in most cases), the three models will give quite close performance figures. Therefore we should conclude that all the three alternatives should be considered. Of course, the second and third alternatives give much better performance figures in case the cost of collecting worker results is not negligible.

Then we move to the *template interface design* phase. This activity basically consists in determining which parameters are suitable to be included in the template interface. We will consider the obvious ones:

i) the *input types*,

ii) the *output types*,

iii) the *the function computed* at the workers, and

iv) the *the parallelism degree*.

Of course, we should consider even more parameters, to make the template more and more *malleable* and therefore suitable to efficiently implement a larger number of skeleton instances, namely:

a) the *scheduling* policy,

b) the *reordering* policy (i.e. whether or not the relative ordering of the input tasks has to be considered to keep ordered in the same way the output results)[122], and

c) *any shared state* variable accessed by the workers, with the relative *shared access pattern*.

The last phase of the design is the actual implementation. We do not discuss too much details in this case. Taking into account we were targeting shared memory multi cores:

- We will use independent threads to implement master and worker activities.
- We will use some efficient implementation of a concurrent *queue* data structure to support scheduling and result collection activities.
- We will limit synchronizations among workers and master process to the concurrent access to these queues. In case of a single manager, we may also avoid to synchronize the access to the task queue if we assume (and consequently implement it) that:
  - the master is the only concurrent activity accessing the queue,
  - when a worker needs a new task to be computed the master selects a task from the queue and sends a pointer to the task descriptor to the requesting worker, while marking the task as already scheduled in the queue.

8.4.2 Farm with feedback

We consider a template to implement divide and conquer like skeletons. The main feature to be taken into account in this case is that the *depth* of the tree logically generated by the divide and conquer skeleton may be unknown looking at the input data and therefore some dynamic behaviour must be implemented in the template capable of running arbitrary dept divide and conquer trees.

Here we do not go through discussing all the phases of the design template process, actually, but we just point out the main decisions and achievements as follows:

- Among all possible *alternatives* we consider a tree structured template (e.g. one modelling one-to-one the actual abstract computation performed in the divide and conquer skeleton) and a slightly modified master worker template similar to the one discussed in Sec. 8.4.1. In the tree template each node evaluates the termination condition and, in case the base case has not been reached, recruits a number of resources equal to the number of outputs of the divide phase and delivers them the sub-blocks of the input data as returned by the divide to be computed. Then it waits for the sub results to come back from the "forked" resources and eventually it conquers the results and delivers the result to its "parent" node.

  In the master/worker like template, instead, we arrange things in such a way the master sends tasks to be computed to the workers and the workers may return either

[122] this seems obvious, but there are a number of situations were the asynchrony coming from unordered result delivery to next computation stages may greatly improve the application overall performance

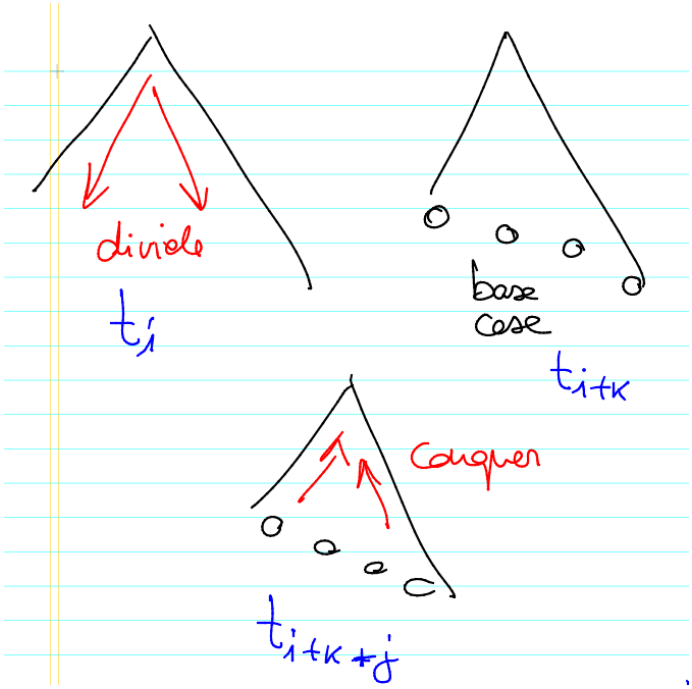


Figure 8.8 Divide and conquer phases: the depth of the three as well as the relative balance is unknown in general at the moment the input data is received

a result (when a base case has been reached) or a new (tuple of) task(s) to be computed resulting from a divide phase. We call this template *farm (or master/worker) with feedback* to evidence the fact tasks are either got from outside by the master or feedback to the master by the workers, as depicted in Fig. 8.9.

- *Evaluating the performance models* of the two templates, we conclude the time spent in recruiting new resources to support tree expansion in the tree template is too high. We therefore keep the farm with feedback template only.
- While *designing the interface* of the template, we decide to keep as parameters: i) the base case condition and the base case evaluation code, ii) the divide and the conquer code, iii) the types of the input, output and intermediate results and iv) the number of the workers in the master/worker. All parameters but the last one are functional parameters.

Using this template, implemented on a shared memory multi core, we are able to process a quick sort divide and conquer as follows:

1. the master sends the input list pointer to the first worker
2. the first worker evaluates the base case condition (false) and divides the list in two parts. These two parts are sent back to the master, as new tasks to be computed, along the feedback channels.
3. the master sends the two lists to first and second worker, that re apply the procedure described for worker one above.
4. more and more workers get busy dividing or computing the base case sort
5. eventually, the masters starts receiving ordered sublists that he arranges in the proper places of the final result.

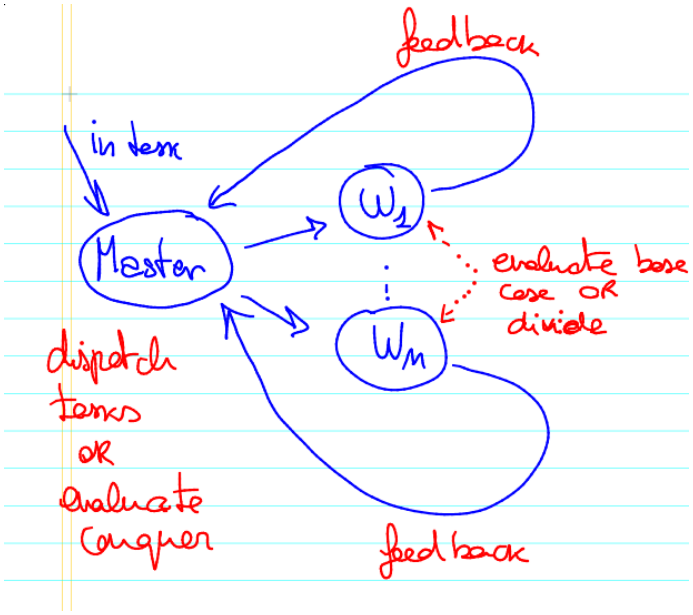


Figure 8.9 Farm with feedback template to implement divide and conquer skeleton.

It is worth pointing out that in case the "conquer" phase is not trivial, the master may delegate also the conquer tasks to the workers. Also, it is worth pointing out that without a proper memory support the feedback communications may represent a sensible overhead in the template computation.

8.5 SEQUENTIAL CODE OPTIMIZATION

As the very last section in this Chapter dedicated to template design, we want to focus on *sequential code optimization*. Properly wrapped sequential code constitutes the basic building blocks used to implement any kind of template. The usage of poor quality sequential code usually leads to poor quality parallel code, independently of the techniques and tools used to implement the different parallel templates used.

The focus in this course is not on the techniques needed to write good sequential code. However, there are some quite effective strategies that can improve a lot the quality of our parallel applications, namely those related to "vectorization".

Code vectorization is a technique aimed at executing in a very fast and efficient way operations on vectors, in all those cases where the operations applied to different vector elements happen to be independent. Vectorization techniques have been developed and applied in compilers targeting machines that support the parallel execution of multiple (floating point) operations on vector data. Back in the last century, only high end computer systems provided such possibility. Nowadays, most micro architectures provide the possibility to execute vector operations. As an example, Intel micro architectures provide different kind of vector operations (also called SIMD extensions) named SSE (with different series number: SSE2, SSE3, ...) or AVX while the PowerPC micro architecture notably provides the Altivec vector instruction set.

The base concept may be illustrated as follows:

- hardware provides a *vector register set* whose components (the vector registers) may be considered as a whole or as a collection of smaller registers. As an example, in a

vector register set of 16 registers of 128 bits each, each register may be referred as a 128 bit register, as 2 64 bit registers, as 4 32 bit registers, etc.

- hardware also provides a number of ALUs (Arithmetic Logic Units) such that they may operate computing a function of the correspondent parts of two vector registers and writing the result in the corresponding part of another vector register. As an example, hardware may provide 4 32 bit ALUs, such that we can compute the sum/-subtraction/multiplication/... of the 4 parts of two 128 bit vector registers in a single shot and write the four results in another 128 bit vector register.
- the instruction set of the micro architecture includes vector assembler instructions modelling the parallel operations supported by hardware replicated ALUs and by the vector register set (e.g. ADDVEC32 FPRVi, FPRVj, FPRVk computing the sum of the 32 parts of floating point registers FPRVi and FPRVj and storing the results in the floating point register FPRVk), as well as instructions to LOAD and STORE vector registers from and to the memory subsystem in parallel (e.g. LOADVEC32 FPRvi, Rbase, Rindex, loading the amount of 32 bit items necessary to fill up the floating point register FPRVi from memory addresses starting with Rbase+Rindex).
- the compiler investigates whether the iterations of a loop are independent and tries to compile the loop body using vector assembler instructions rather than the usual scalar instructions.

While providing parallel ALUs and vector registers in hardware is quite easy, the implementation of compiler policies and strategies to vectorize code is not a trivial task. Different compilers implement different amounts and kind of vectorizing techniques, however two principles are always respected:

- vectorization is not implemented by default, the programmer must use proper compiler flags to activate the vectorization options of the compiler, and
- vectorization only impacts the performance of the program, that is the vectorization techniques preserve the functional semantics of the programs.

Simple vectorization techniques may be applied on loops whose iterations are *independent*, that is where iteration i does not use data produced at iteration $i-k, k \in \mathcal{N}$[123] As an example, consider the loop:

```
for(int i=0; i<N; i++) {
  x[i] = a[i] * b[i];
}
```

In this case the results computed at iteration i (`x[i]`) do not depend on those computed in the previous iterations. The loop:

```
for(int i=1; i<N; i++) {
  x[i] = a[i-1] * x[i-1];
}
```

instead has dependent iterations, in that the i iteration uses the `x[i-1]` value produced at the previous iteration.

When trying to parallelise the first loop, the compiler will try to use the vector assembler instructions. In case the assembler provides a vector instruction computing the multiplication of 4 32bit floating point numbers stored in 128 bit vector registers, then the loop will be computed as if it was written as a:

[123]this is a quite simplistic definition of independent iterations, just to illustrate the concept, actually.

```
for(int i=0; i<N; i+=4) {
  asm_vector_register_load(FPRV1, a[i]);
  asm_vector_register_load(FPRV2, b[i]);
  asm_vector_mul(FPRV1, FRPV2, FPRV3);
  asm_vector_register_store(FPRV1, x[i]);
}
```

where the iteration computes the equivalent of four iteration of the original loop.

More complex vectorization techniques may be applied to different kind of loops, possibly after refactoring the loop code in such a way the dependencies present in the original code are removed. These techiques are quite complex and definitely outside the scope of this course.

How can we invoke vectorization on our sequential portions of code? Usually the compiler provides some kind of general purpose "optimization flag" that enables different kind of optimizations, including vectorization. In case of GNU compilers, the "optimization flag" is `-On` with n$\in (1, 2, 3, 4)$, with `-O3` activating the full range of vectorization optimizations. The effect of the vectrorization may be seen using the `-ftree-vectorizer-verbose=n` flag. With $n = 2$ the compiler report tells you which cycles have been vectorized. As an example, consider the code:

```
1  #include <iostream>
2  #include <time.h>
3
4  using namespace std;
5
6  struct timespec diff(struct timespec t0, struct timespec t1);
7
8  int main(int argc, char * argv[]) {
9
10 #define N 1024*1024
11
12   float x[N];
13
14   for(int i=0; i<N; i++)
15     x[i] = ((float) i);
16
17   for(int i=0; i<N; i++)
18     x[i] = x[i] * x[i];
19
20   for(int i=0; i<N; i++)
21     x[0] += x[i];
22   cout << x[0] << endl;
23 }
```

If we compile the code asking to optimize (and vectorize) it and to report vectorization results using GNU g++ with the command:

```
g++ -o veto3 vet.cpp -lrt -ftree-vectorizer-verbose=2 -O3
```

we will eventually get the following output:

```
Analyzing loop at vet.cpp:20

20: not vectorized: unsupported use in stmt.
Analyzing loop at vet.cpp:17
```

```
Vectorizing loop at vet.cpp:17

17: LOOP VECTORIZED.
Analyzing loop at vet.cpp:14


Vectorizing loop at vet.cpp:14

14: LOOP VECTORIZED.
vet.cpp:8: note: vectorized 2 loops in function.
```

stating that the first two loops have been correctly vectorized while the third one has been not.

Before discussing while the third loop has not been vectorized, let's have a look at the performances of this program with and without the vectorization enabled. We modify the program by adding some *ad hoc* profiling code as follows:

```
#include <iostream>
#include <time.h>

using namespace std;

struct timespec diff(struct timespec t0, struct timespec t1);

int main(int argc, char * argv[]) {

#define N 1024*1024
  struct timespec res;
  struct timespec t0, t1;

  float x[N];

  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t0);
  for(int i=0; i<N; i++)
    x[i] = ((float) i);
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t1);
  diff(t0,t1);
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t0);
  for(int i=0; i<N; i++)
    x[i] = x[i] * x[i];
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t1);
  diff(t0,t1);
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t0);

  for(int i=0; i<N; i++)
    x[0] += x[i];
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,&t1);
  diff(t0,t1);

  cout << x[0] << endl;
}

struct timespec diff(timespec start, timespec end)
{
  timespec temp;
  if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
```

```
41      temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
42    } else {
43      temp.tv_sec = end.tv_sec-start.tv_sec;
44      temp.tv_nsec = end.tv_nsec-start.tv_nsec;
45    }
46          cout << "Time elapsed: " <<  temp.tv_sec << "secs and "<<
                temp.tv_nsec/1000 << " msecs " << endl;
47    return temp;
48  }
```

Then we compile the code with and without optimizations (that include, but are not limited to, vectorization). When executing the non optimized code we get:

```
Time elapsed: 0secs and 13970 msecs
Time elapsed: 0secs and 7788 msecs
Time elapsed: 0secs and 4706 msecs
3.84357e+17
```

while after compiling the very same code with the `-O3` we get

```
Time elapsed: 0secs and 4633 msecs
Time elapsed: 0secs and 1132 msecs
Time elapsed: 0secs and 4186 msecs
3.84357e+17
```

The difference in the execution of the 3 loops is sensible. The first and the second loop experiment a 3 fold and 6 fold speedup. The last loop clearly only benefits from other optimization techniques (loop unrolling[124] in this case) and therefore the speedup is much smaller.

Now let's go back to the third loop not vectorized. The reason provided by the compiler is a `not vectorized: unsupported use in stmt` meaning some non legal action happens in the loop body statement. Here the "illegal" action is the reduction computed over the vector values. At each iteration, `x[0]` is read (value of the previous iteration) and updated, thus generating a "loop carried dependency".

Actually, in order to be vectorizable, loops should satisfy a full set of requirements, including:

1. only inner loops may be usually vectorized. Non inner loops may be vectorized in case the internal loop may be fully unrolled.
2. loop body should not contain call to external functions (not inlined)
3. loop body must contain straight-line code, with no branches inside
4. only countable (determinate) loops may be vectorized
5. there should not be dependences among loop iterations (iteration i reads results produces at iteration $i - k$)

The interested reader may find more details on vectorization in the reference texts and manuals [49, 55]. For the purpose of the course, we should state the following "rule":

[124] loop unrolling transforms loop body in such a way it is unrolled k times. After each unrolled iteration a test is made to check loop end condition. Then the loop iterations are divided by k. The whole techniques avoids the branch penalty incurring at each iteration of the original loop $k - 1$ times each k iterations, in this case. In case the original loop was a `for(i=0; i<N; i++) body` the unrolled version will be something such as `for(int i; i<N; i+=k) { body; if(!(i<N)) break; body; if(!(i<N)) break; ... }`

> any code wrapped to be used as building block in a template should be vectorized, if possible, either relying on the automatic vectorization capabilities of the compiler at hand, or possibly entering a vectorization aimed refactoring of the original code such that the resulting code could be eventually vectorized.

Take into account that this may have consequences on application parallelization. Take into account what happens if you are implementing a master/worker template to parallelize a map. The workers of the template will execute a chunk of the original iterations. If the map was computing $\forall\ i \in (0, 1, 2, 3, 4, 5 \ldots, n)$, the workers will be computing each part of these iterations. Let's assume that the workers (with no vectorization applied) compute their iteration chunks in 10 msecs and that the distribution of the chunks to the workers with the related gathering of the distinct workers results accounts for another 1 msecs. Then this application will probably scale pretty well on an eight core machine. However, if we apply vectorization on the worker code, assuming to achieve a speedup of 6 (as in the example discussed before) we will end up having workers computing in a time which is close to the overhead introduced to parallelise the overall computation thus leading to poor scalability of the application on the eight cores available.

CHAPTER 9

PORTABILITY

Portability has different meanings at different abstraction levels. *Application portability* refers to the possibility to port an existing application from a machine to another, possibly different, machine. Portability is fully achieved in this case, when the application may be run unchanged on the second architecture once it has been designed, developed and debugged for the first architecture. Application portability may be achieved different ways, including:

- *recompiling* of the source code, or
- simply *re-running* the original compiled code.

In the latter case, the two machines targeted will obviously be *binary compatible*, that is they will accept the same kind of object code files.

When considering programming frameworks, instead, *portability of the programming framework* across different architectures is fully achieved when the programming framework running on machine A may be moved and operated on machine B. This in turn, implies that application portability will be achieved, for all those applications running on top of our programming framework.

As for application portability, programming framework portability may be achieved in different ways:

- The easiest one is portability through *recompiling*. We take the source code of our programming framework, written in a language L_x and we recompile the whole source code of the programming environment on the second machine, provided a compiler for L_x is available on the second architecture.
- A more complex way of porting a programming environment requires both recompilation and rewriting of some parts of the programming framework implementation.

In particular, all those parts that were specifically implemented for solving problems peculiar of architecture A and whose solutions do not apply to architecture B.

- In the worst case, portability could simply not be achieved, as the features of the target architecture B are radically different from the features of target architecture A. In this case, most of the programming environment parts must be re-written almost from scratch. Probably, only the higher level tools, such as the parsers, may be preserved unmodified.

Portability of programs is an issue typical of parallel programming. In the sequential programming world it has a minor impact. The problem is mostly determined by the different features sported by different parallel and distributed architectures. These differences may concern the interconnection network between parallel/distributed processing elements (full interconnection, regular interconnection topology networks, etc.), the presence or the absence of some level of sharing in the memory hierarchies of the processing elements, the kind of accelerators available (e.g. GP-GPUs or FPGAs). This is not the case, or at least, the differences between different architectures are not so evident, when sequential architectures are considered. Most of the sequential architectures around[125] support very similar assembly code sets and as a result, the only significant differences between different architectures consist in the presence of particular functional units (e.g. SIMD units) speeding up data parallel computations. As a consequence, porting an application from architecture A to architecture B in most cases only requires a re-compilation of the source code of the application and, in a smaller but consistent number of cases, the object code of the application compiled on machine A may be simply moved and run to machine B without any kind of (or with minor, performance related) issues.

We are ready now to introduce a distinction between two different kinds of portability:

Functional portability Functional portability concerns the portability of the functionality of the application or programming framework. Functional portability of an image filtering application means that we can move/recompile the application to the new machine preserving the possibility to filter images obtaining the same set of filtered images we previously get from the application running on the first architecture.

Performance portability In addition to functional portability, we may be interested in portability of non functional properties of our applications/programming environments. Non functional features are those features that do not concur directly in the result computed by our applications (the function of the input data) but rather concur to *the way* these results are computed. As an example, performance, security, power management, fault tolerance are all non functional features of applications/programming environments. Performance portability guarantees that a program with a given performance on machine A may be ported to machine B preserving a comparable performance. The term "comparable", in this case, may be considered modulo the architectural differences between the two architectures. In other words, performance portability must guarantee that the best performance possible for the algorithms used relatively to the features of the target architecture are eventually achieved.

Now when considering performance portability in addition to functional portability new issues must be taken into account. Simple recompilation of an application on a new target architecture can give you functional portability but may be it does not guarantee performance portability.

[125] it is worth pointing out that these architectures actually do not exists anymore as single chips, but we refer here to usage of a single core as a sequential processing element

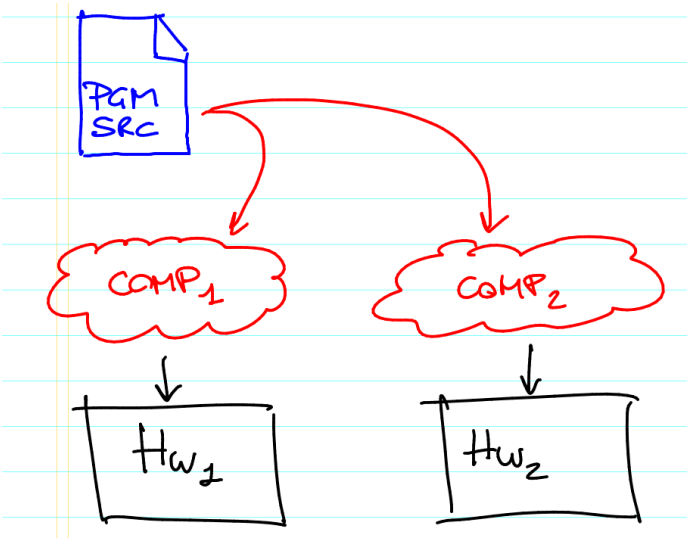


Figure 9.1 Portability through recompiling

In the sequential programming world, we are used to get performance portability for free. If you write a program for a PowerPC and then you recompile the very some code on an Intel processor, you'll get performance portability in addition to functional portability.

But in the parallel programming world things are a little bit different. If you write an MPI program using barriers and you run it on a machine with hardware support for broadcast[126], in case you move that program to a machine without the hardware support for broadcast, you will get definitely worst performances.

In the following sections we will discuss issues and solutions relative to both functional and performance portability of skeleton based programming frameworks. We distinguish two main approaches to portability, namely i) portability achieved through re-compiling and ii) portability supported through virtual machines.

9.1 PORTABILITY THROUGH RE-COMPILING

Portability through re-compiling relies on the existence of a set of compiling tools that may be used to re-compile the whole programming framework implementation on the new target machine.

9.1.1 Functional portability

In order to be able to functionally port a skeleton framework from a target architecture to another target architecture, what has to be guaranteed is that the programs running on the first architecture also run on the second one and compute the same results.

Let us consider what happens of functional portability when using template or macro data flow based implementations of skeleton programming frameworks.

Template based frameworks. In a template based skeleton framework, implementation of skeletons is achieved by instantiating process templates. Provided that the new target architecture:

[126] as an example, on a machine whose cores/nodes are interconnected using a mesh network plus a global bus for synchronizations

- supports the language used to implement the framework,
- supports the same kind of concurrent activity abstractions (e.g. processes, threads or kernels) used to implement the templates on the first architecture, and
- supports the communication and synchronization mechanisms used in the original process templates

then functional portability through re-compiling may be achieved.

The first requisite above is obvious. If you can't compile just a single component of the programming framework because no compiler exists on the new target architecture for the language used to write that component, the whole porting fails. Such a situation requires re-writing the component in another language for which a compiler exists, instead.

The second and third requisite could be handled in a different way. If some base mechanisms essential for framework implementation is not present, some new component must be designed, developed and integrated into the ported framework capable of emulating the missing mechanisms. Alternatively, the templates using the missing mechanisms must be completely rewritten in terms of the mechanisms presents on the new target architecture.

Macro data flow based frameworks. In a macro data flow based skeleton programming framework, functional portability may be achieved if

- a compiler for the language used to implement the framework is available,
- the mechanisms used to implement the macro data flow interpreter are supported on the new target architecture.

Therefore, as far as functional portability is concerned, we can conclude functional portability may be guaranteed if template implementation or concurrent MDF interpreter implementation may be conveniently ported onto the new architecture in the two cases. As a macro data flow concurrent interpreter will be in general modeled by a single parallelism exploitation pattern, while the templates in a template library model more parallelism exploitation patterns, we can also conclude that functional portability is slightly easier in MDF based skeleton frameworks with respect to template based frameworks

9.1.2 Performance portability

When dealing with performance portability, things are a little bit more complex.

Template based frameworks. In a template based skeleton framework performance portability is mainly achieved through proper restructuring of the templates used to implement the skeletons and the skeleton compositions. The general structure of the programming framework tools could be mostly left unmodified provided that new templates, or modified/adapted versions of the existing ones, are adopted for the new architecture targeted. As an example, let us consider the "farm template for complete interconnection architectures" of Sec. 4.2.1.4. In case we move to target an architecture with mesh interconnection of internal cores such as the Tilera multi-core, we must take into account the farm templates such as those discussed later on in the same Section ("farm templates for mesh architectures"). This was not necessary in case functional portability is only considered. In that case, we could have set up further communication library emulating the full interconnection required by the first template on a mesh architecture. This introduces more overhead in the implementation, but as we are only interested in functional portability, it is sufficient to guarantee portability.

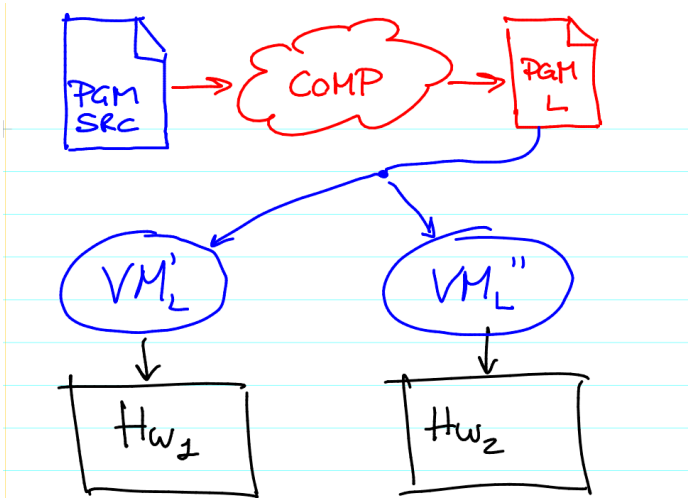


Figure 9.2 Portability through virtual machines

It has to be taken into account that depending on the new architecture features the design of a new template or the re-adaptation of an existing template for any one of the skeletons provided by the framework may require a huge design and implementation effort.

MDF based frameworks. In this case performance portability is mainly guaranteed by an efficient porting of the concurrent MDF interpreter. As again stated in Sec. 4.2.2, the efficient implementation of a concurrent MDF interpreter does require a substantial effort. However, once this effort has been spent, the whole framework may be ported seamlessly to the new architecture, as the features of the MDF graphs generated when compiling the skeletons will not impact performances on the new machine if the concurrent MDF interpreter has been successfully "performance" ported.

Therefore, in case of performance portability, we can conclude that it is slightly easier to achieve it in case of template based frameworks, as the general task is split into smaller tasks (the implementation of the single templates) rather than being kept unique (the implementation of an efficient concurrent MDF interpreter).

9.2 PORTABILITY THROUGH VIRTUAL MACHINES

Up to now, we have only considered portability assuming our programming frameworks are recompiled when moving to a new target architecture. There is an alternative way of implementing portability, however, which exploits virtual machines.

Let us assume that a skeleton programming framework exists targeting the Java Virtual Machine. The framework itself may be a library based or a new language eventually compiled down to Java, template based or macro data flow based. It does not matter. The important thing is that the skeleton applications eventually run as JVM byte-code. Alternatively, let us assume that a skeleton programming framework exists targeting C+MPI. Again it does not matter how the framework is implemented nor the programming model (library or new language) it exposes to the skeleton application programmer. The important thing is that the skeletons applications are eventually run as MPI applications.

In both cases, the functional portability of the programming frameworks may be simply guaranteed by the portability of JVM or MPI. What you need to port your programming

environment on a new target architecture is an implementation of the JVM or an implementation of MPI on the new target architecture.

What about performance portability, then? In general performance portability could not be guaranteed for the very same reasons listed when discussing compiler based framework portability. Virtual machine only guarantees portability and efficiency of the virtual machine mechanisms. The efficiency achieved on a single architecture is the "best" available for those mechanisms. However, this does not guarantees that the final framework efficiency is convenient. Again, let us consider porting of a programming framework from a machine supporting hardware broadcast and multi-cast mechanisms to a machine that does not support in hardware such mechanisms. Any template[127] relying on the efficiency of broadcast/multi-cast mechanisms will suffer from severe performance penalties on the second machine. As a consequence, true performance portability will be achieved only by rewriting the templates[128] in such a way they properly exploit the mechanisms available on the new target architecture.

9.3 HETEROGENEOUS ARCHITECTURE TARGETING

Targeting heterogeneous architectures poses problems similar to those arising when porting programming environments across different architectures. An heterogeneous architecture is such that at least two different kind of processing elements exist, that can be used to execute parallel/distributed activities. Sample heterogeneous architectures include multi-core CPUs with attached GPUs as well as interconnections[129] of different multi-core processing elements, e.g. a mix of Intel Core and IBM PowerPC processors interconnected via Ethernet.

Heterogeneity poses several problems, that partially overlap to those analyzed when dealing with portability issues:

- *Specialized compilers* must be used to target different nodes in the heterogeneous architecture. This means that the very same process template must be compiled with different tools depending on the choices made in the mapping phase, that is when "processes" are bound to processing elements.

- *Specialized templates* the templates used to compile/implement skeletons may differ for different processing elements. As an example, a map template used on plain multi-cores may radically differ from the one used on CPUs with an attached GPU. In this latter case, data parallel kernel executions may be delegated to the GPU, achieving a sensible speedup.

- *Specialized concurrent MDF interpreter instances* must be available to be run on the different processing elements used to implement the overall MDF interpreter.

In addition to requiring specialized compiler and templates, however, heterogeneity also raises some problems in the high level part of the compiling tools relative to template based programming frameworks. In particular, when templates are chosen to be associated to the skeletons appearing onto the application skeleton tree (see Sec. 4.2.1) decisions have to be made relatively to the possibility to choose templates that only work on a given subset of the target architecture. As template assignment has eventually direct consequences on the final performance of the application, this is a critical point.

[127] but this is true also taking into account the implementation of the concurrent MDF interpreter
[128] or the concurrent MDF interpreter
[129] by means of standard interconnections networks

E.G.▷ Consider the case of architectures build of the interconnection of multi-core nodes, where only a part of the multi-core nodes are equipped with GPUs. Let us assume the application to implement has a skeleton tree such as the following:

```
Pipe(Map(Seq(C1)),Map(Seq(C2)))
```

Let us suppose we have available map templates both for multicore only and for multi-core+GPU processing elements interconnections. Whether or not the GPUs are more efficient of CPUs on a given computation could not be decided without actually looking at the input data of the map skeletons. Regular data[130] will probably better perform on GPUs than multi-cores. Furthermore, in case of unbalance of the pipeline stages, it may be the case that one of the stages does not need an extreme parallelization, as the other stage is actually the pipeline slowest stage and therefore it is the other stage that eventually determines the performance of the whole pipeline.

Now when compiling this program on the heterogeneous architecture, we have basically two possibilities related to the template choice:

- We proceed as usual, looking at the performance models of the templates in order to establish which template composition will best implement the skeleton tree at hand. In other words, we completely implement the template assignment during the first phases of the process leading to the generation of the skeleton application object code. Or alternatively
- We perform a preliminary template assignment and later on, at *run time*, we possibly change the choice made in such a way we succeed to exploit the behaviour data gathered while executing the program[131]. This implements a kind of *self*-management of the template assignment process. Further details on this will be given in Sec. 10.5.

■

A different set of problems arise when targeting heterogeneous architectures in a macro data flow based framework. In this case, the compilation of the skeleton application to MDF graphs should not be changed. What changes is the implementation of the concurrent macro data flow interpreter. In particular:

- Different interpreter instances have to be developed, such that each instance is optimized for one of the heterogeneous processing elements. It is possible that different (heterogeneous) processing elements may run the same kind of interpreter instance (e.g. processing elements with slightly different processors). In the general case, different instances of the interpreter may support the execution of different subsets of MDF instructions: as an example, GPU powered interpreter instances may support the execution of data parallel macro data flow instructions, while more classical, non GPU powered architectures only support such instructions through execution of explicitly parallel MDF graphs (that is, not in the execution of a single MDF instruction).
- The scheduling of fireable MDF instructions to the concurrent interpreter instances has to be consequently modified. Being functionally different instances of the interpreter running on different heterogeneous nodes, specialized scheduling functions must be implemented as the set of remote interpreters is not anymore “anonymous”.
- In turn, the specialization of the scheduling policies to accomplish the diversity among available concurrent interpreters may require a more complex management of the logically centralized MDF graph repository. In case of distributed implementation of

[130] that is data that lead to identical executions in the subcomputations of the map
[131] such data may make evident that the GPU template was the best one or vice versa, as an example

the repository with load balancing guaranteed through fireable instruction stealing, also the stealing strategies must be adapted in such a way the existence of different interpreter instances is taken into account.

9.4 DISTRIBUTED VS MULTI-CORE ARCHITECTURE TARGETING

Historically, skeleton based programming frameworks have been developed to target distributed architectures[132]. The skeletons frameworks Muesli, Muskel, ASSIST, SkeTo (just to name those still maintained) have all been designed and developed to target clusters and network of workstations. Later on, some of these frameworks have been redesigned and improved in such a way multi core architectures could also be exploited. At the moment being, Muesli [34], Muskel [17] and Skandium [63] all support multicore machine targeting.

Multi core targeting is basically supported in two distinct ways:

Architecture modeling changes We assume that each multi core node in the target COW/NOW architecture is itself a network of processing elements. As a consequence, a number of processes–from the template composition implementing the skeleton program–or of concurrent interpreters–in case a MDF implementation is used–equal to the number of supported cores is placed on the single multi core.

This implies slight modifications in the skeleton framework implementation. Basically only requires to expose different cardinalities for the processing elements hosted in the target architecture, in such a way multiple cores are counted as multiple processing elements.

Of course, in case the templates used may optimize communications depending on the features of the links interconnecting nodes, the links between "virtual processing elements" modeling cores of the same processor should be modeled as "fast" links[133].

Template/MDF interpreted specialization Alternatively, and depending on the kind of implementation adopted in the framework, either the implementation templates of the skeletons, or the structure of the concurrent MDF interpreter instance should be changed to take into account the existence of multi core nodes in the target architecture.

When considering a *single* multi core as the target architecture of our skeleton framework, similar choices may be taken, but in general, better optimization can be achieved using the "specialization" strategy rather than the "architecture modeling" one.

In any case, the design of the templates or the design of the MDF interpreter must tackle problems that are quite different from the ones significant when COW/NOW architectures are targeted, as reported in Table 9.1.

In the following sections we briefly discuss the issues related to COW/NOW and multi-/many core targeting.

[132]We refer in this section as "distributed architectures" those resulting from the interconnection through some kind of interconnection network of complete processing elements, such as the cluster of workstations, the network of workstations and the computational grids. We refer with the term "multi/many core" those architectures build out of multiple cores, possibly on the same chip, sharing some level of the memory hierarchy, such as Intel Core processor or symmetric multi processors (SMP) such as the shared memory machines hosting different multi core chips on several distinct boards.

[133]that is links using specialized inter-thread communication mechanisms rather than slower inter-process mechanisms

	Distributed	Multi/many cores
Templates or MDF interpreter	Process based	Thread based
Communication & synchronization	Inter PE (TCP/IP)	Inter Thread (Shared mem + locks & semaphores)
Exploiting locality	Process mapping	Thread pinning Memory access
Optimization opportunities	Communications Process mapping Communication grouping	Shared memory accesses Thread pinning Memory layout (padding)

Table 9.1 Problems related to template/MDF interpreter implementation for distributed/shared memory architectures

9.4.1 Template/MDF interpreter implementation

On distributed architectures, the main mechanism exploited to implement concurrent activities is the *process* abstraction. Processes are used to implement concurrent activities in templates as well as those related to MDF interpreter implementation. As a consequence, *no memory is shared* among the different concurrent entities used to implement the skeleton framework. All the interactions among the concurrent activities must be implemented via explicit communications.

When targeting multi cores, instead, the main mechanisms exploited to implement concurrent activities is the *thread* abstraction. As a consequence, interactions among different concurrent activities may be implemented through shared memory *and/or* via explicit communications as in the case of concurrent activities implemented through processes.

Using processes may seem a little bit more complicate than using threads, due to the fact no shared memory abstraction is present and all the interactions must be implemented via explicit communications. However, while threads support "more natural" interactions via shared memory, they also support default sharing of *all* the memory space. Therefore it is easy to introduce errors and inefficiencies[134].

E.G.▷ A simple example will better illustrate the case. When implementing a map template using threads the naif implementation of accesses to the partitions of the original, collective data may introduce false sharing that could impair the actual speedup achieved in the execution of the map template. In particular, a map template with threads operating onto sub items of a vector whose items are of size smaller that the cache line size implies the arising of a false sharing situation. Items i and $i+1$ of the vector will be processed by different threads but they will actually be allocated to the same cache block. As a consequence, two accesses updating positions i and $i+1$ at different cores will imply the start of a cache coherency protocol which is actually useless, due to the fact that thread processing position i will never need accessing position $i+1$ and vice versa[135]. ■

[134] in this sense, inefficiencies can be seen as particular kind of "non functional errors

[135] after the termination of the map, of course, this may happen, but not within the execution of the map. Within the map thread processing item i may at most require the access to current (not the new one) value of position $i+1$, which does not require cache coherency.

9.4.2 Communication & synchronization

Communications and synchronizations represent a crucial matter in parallel and distributed programming. Therefore they play a fundamental role in both template and MDF based skeleton framework implementation.

When targeting distributed architectures, the main communication mechanisms are those provided by the interconnection network used to host the target architecture processing elements. In common COW/NOW architectures, either Ethernet or some kind of more advanced interconnections such as Myrinet [73] or Infinibad [53] are used. When using standard Ethernet, the TCP/IP protocol stack is used for communications, either at the level of transport protocols (TCP, UDP) or at the network level (IP). Streams and datagrams are used to implement both communications and synchronizations. Synchronizations are usually implemented via small messages[136]. When using more advanced interconnection networks, proprietary communication and synchronization mechanisms are used instead. TCP/IP protocol stack is also usually provided on top of these basic, proprietary mechanism, often resulting in quite large overhead when using it to implement communications or synchronizations[137]. It is worth pointing out that both Ethernet and the more modern and performant interconnection networks support some kind of primitive broadcast or multi cast communications. These mechanisms are fundamental for the design of both efficient templates and optimized concurrent MDF interpreters. As an example, in the former case, they can be used to support data distribution in data parallel templates, while in the latter case, they can be used to support load balancing mechanisms in distributed MDF graph pool implementation as well as different coordination mechanisms supporting the distributed/parallel implementation of the interpreter, such as fireable instruction stealing or fault tolerance discovery mechanisms.

When targeting multi/many core architectures, instead, accesses to shared memory are normally used to achieve an efficient implementation of both communications and synchronizations. Communication is implemented writing and reading to properly synchronized buffers in shared memory. Synchronization, including the one needed to implement the communication buffers is implemented using locks, semaphores or any other synchronization primitive available. In this case, it is fundamental that very efficient synchronization mechanisms are used, due to huge amount of cases where this mechanisms are used and to the relatively *fine grain*[138] computations involved in many cases in the synchronization operations. Some skeleton framework tackle this issue since the very beginning of the framework design. Fastflow is entirely build on a very fast and efficient single producer single consumer queue which has been demonstrated to be correct while being lock free and fence free on common cache coherent multi cores such as the ones currently provided by Intel and AMD. The implementation through shared memory of communication and synchronization primitives naturally supports the implementation of the collective broadcast/multi cast operations.

9.4.3 Exploiting locality

Locality is the property of algorithms of performing "near" data accesses during program execution. In other words, very often algorithms are such that if a data structure or a portion of code is accessed at time t_i the probability of another access to the data/code in the near future (time $t_{i+\Delta}$) is very high (temporal locality) as well as the probability

[136] messages hosting a very small payload, may be just one byte, which is not significant as payload as the send and receive of the message have associated the synchronization semantics.
[137] Windows support for Infiniband provides a TCP/IP stack implementation on top of Infiniband. The stack introduces a sensible overhead with respect to the one introduced by primitive Infiniband mechanisms.
[138] that is small amount of computation in between successive synchronization operations

of accessing locations close[139] to the ones just accessed (spatial locality). Locality is the property that makes current architectures work. Without locality, caches will be useless. Without locality most of the memory access[140] would be in main memory, whose access times are order of magnitude larger that the access times of caches. Ensuring locality exploitation is therefore fundamental for efficient implementation of algorithms and, therefore, of sequential programs.

Locality is even more important in parallel programming. Considering the parallel/distributed/concurrent activities usually are performed on processing elements that are relatively far from each other–in terms of the time spent to exchange data–it is clear that a program not exploiting locality will never be efficient.

When considering process based implementations of skeleton frameworks this is particularly evident. If we design a template were data is continuously moved to and from different processing elements because of poor locality optimization, this program will spend a lot of time moving data rather than actually computing results.

E.G.▷ As an example, suppose we implement a pipeline template where each stage gets tasks to be computed from an emitter/collector process and delivers results to the same process. In this case to compute a k stage pipeline on a stream of m tasks we need $2km$ communications as each stage must first receive a task to compute and then send the result back to the emitter/collector process. If we use the classical "linear chain" of k processes to implement the stages, the communications needed to compute the same stream are just one half mk. In the former template, we did not exploited the locality in the access of partially computed result $f_h(f_{h-1}(\ldots(x_i)\ldots))$ which is only accessed by stages S_h (the producer) and S_{h+1} (the consumer). ■

However, also in case of thread based implementations, locality plays a fundamental role.

E.G.▷ When working with threads and shared memory, we can consider a different example. Let suppose we want to implement a map template working on vectors. Therefore we distribute over a thread pool the computation of

$$\forall i : y_i = f(x_i)$$

If we distribute map iterations over threads such that iteration i is placed on thread $t = i\%n_{thread}$ we actually will achieve a much worse performance than in case we distribute iterations on threads such that iteration i is scheduled on thread $t = i/n_{thread}$, as this latter distribution allows to achieve a much better cache management due to the higher locality in the memory accesses[141].

■

Therefore, when targeting both distributed COW/NOW and multi/many core architectures, preserving and exploiting the locality proper of the parallel skeletons used is a fundamental issue. Many of the "optimizations" listed in the next section are *de facto* aimed at exploiting locality in parallel computations.

9.4.4 Optimizations

Several optimizations may be taken into account when targeting (porting to) distributed/-multi core architectures. Actually, being the parallel structure of our program completely

[139] whose address is not too different
[140] instruction fetches *and* data accesses
[141] as an exercise, try to imagine the trace of data addresses generated towards the caches from two different threads (running on two different cores).

exposed through the used skeletons, any known optimization technique relative to the particular patterns modeled by the skeletons may be used. As already discussed this is one of the fundamental advantages of the algorithmic skeleton approach to parallel programming.

Locality preserving/exploiting optimizations are probably the most important ones. In this context, we should mention different techniques:

- Allocation of the concurrent activities compiled from the application skeleton composition in such a way locality is preserved. This includes mapping communicating processes on "near"–from the viewpoint of communication overheads–processing elements as well as pinning[142] thread to cores in such a way threads accessing the same data structures share the largest portion of memory subsystem (e.g. second level caches).
- Grouping of concurrent activities on the same processing element, in case these activities happen to be *tightly coupled*, that is they share a lot of data structures or they send each other a large amount of data. This kind of grouping decreases the parallelism degree of the application, in general, but may lead to advantages in terms of a smaller number of communications or smaller overhead in shared data access that possibly overcome the losses due to the smaller parallelism degree.

Other common optimization techniques impact the way concurrent activities interact. Communication optimizations belong to this class of optimizations, as well as shared memory access optimizations.

Communication optimizations. Communication optimizations relate to the way communication among different concurrent activities are performed. In presence of a large number of communication between two processing elements of the target architecture, for instance, *communication grouping* may be used. With this technique, a number of communications happening in the same time interval and moving data from processing element PE_i to processing element PE_j are implemented as a single communication event with a larger message body. As communications between separate processing elements usually require some setup time plus a time directly proportional to the length of the message, this techniques allows $k-1$ setup times to be spared when k communications are performed. Taking into account that communication setup may be quite large (tens of microseconds when using Ethernet) this represents a sensible advantage. Of course, this means communications must be directed to a kind of "communication manager" that analyzes the traffic in pre-defined time chunks and possibly decides to group different communications into a single one. This also implies that the latency perceived by the single process may be larger than the one achieved when not using grouping, although the performance of the overall communication system is better.

Shared memory access optimizations. Shared memory access optimizations consist in a quite different set of techniques. To avoid problems while accessing shared variables[143] variable layout in memory may be optimized as well as the usage of names to denote shared variables. Variable layout in memory may be optimized to avoid that items of the same data structure accessed by concurrent activities of the same template share the same cache line. This avoids the penalties of the cache coherence mechanisms while preserving program fairness at the expense of a (slightly) larger memory footprint. The technique mainly consists in inserting *padding data* between consecutive items (or blocks of items) of the shared data structure in such a way accesses formerly referring to the same cache line become accesses referring to different cache lines.

[142] pinning is the operation aimed at fixing thread execution on a given core
[143] we assume shared memory architectures support cache coherence

E.G.▷ Consider a vector of integers. Assume cache lines may host 8 integers (32 bytes). Assume also the vector is processed by a `map` computing $x_i = f(x_i)$ and that each one of the workers in the map get chunks of 10 vector items to be processed. Worker 1 will process two cache line of data: the first one–assuming the vector is cache aligned in memory–will be entirely read and written. In the second line, worker 1 will just access the first and second elements, while the rest of the vector items in the line will be accessed by worker 2. This situation is a false sharing situation. When writing the new value for x_8 (the first element of the second cache line) worker 1 will probably cause the start of a cache coherency protocol aimed at propagating the new value to the cache line hosted on the processor where worker 2 has been scheduled, that in the meanwhile is processing $x_{10}, x_{11}, \ldots, x_{19}$ ——————— ■

Renaming of variables is used instead to allow better optimizations to be implemented in the sequential language compiler when accessing potentially shared data structures. By properly renaming variables that will eventually be accessed–for writing–by different concurrent activities due to the template/skeleton semantics we avoid the renamed variables be marked as "shared" and therefore accessed through time consuming access synchronization mechanisms.

CHAPTER 10

ADVANCED FEATURES

In this Chapter, we discuss some "advanced" features algorithmic skeletons. The term "advanced" has two basic meanings, in this context: i) the features discussed here deal with problems that are not commonly handled in parallel programming frameworks, or, at least, they are not handled with the same efficiency and success typically achieved in skeleton frameworks, and ii) these features were not included in the first algorithmic skeleton frameworks, but represent an evolution in the implementation of these programming frameworks towards more and more efficient implementation of the algorithmic skeleton concept.

The "advanced" features concern both program design (e.g. the possibility to transform programs by rewriting during the design phase) and skeleton framework implementation (e.g. adaptivity or fault tolerance). Indeed, most of these techniques do not represent *new* concepts. Rather, the techniques apply existing concepts to the algorithmic skeleton scenario that boosts the possibility to exploit the parallel computation structure knowledge typical in algorithmic skeletons.

10.1 REWRITING TECHNIQUES

Rewriting techniques have been exploited in a variety of contexts. Functional programming languages traditionally provide the necessary concepts to support program rewriting. Let us take into account our second order modeling of skeletons introduced in Chap. 3. We defined a `map` skeleton. Looking at the definition of this skeleton we can easily get convinced that (being $\circ : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$ the functional composition and " $:'': (\alpha \to \beta) \to \alpha \to \beta$ the operator applying a function to some input data[144])

$$(\texttt{map}\ f) \circ (\texttt{map}\ g) \equiv \texttt{map}(f \circ g) \tag{10.1}$$

[144]the first ":" is the apply operator, the second one (without quotes) is the usual "type" operator

In fact, when the left function is applied to a data structure such as $\langle x_0, \ldots, x_n \rangle$ (let us suppose the x_i here are items in a vector, as an example), we get

$$(\texttt{map}\ f) \circ (\texttt{map}\ g) : \langle x_0, \ldots, x_n \rangle = (\texttt{map}\ f) : \langle g(x_0), \ldots, g(x_n) \rangle =$$

$$= \langle f(g(x_0)), \ldots, f(g(x_n)) \rangle$$

whereas when the right function is applied to the same data structure, we get

$$\texttt{map}(f \circ g) : \langle x_0, \ldots, x_n \rangle \ = \langle (f \circ g) : x_0, \ldots, (f \circ g) : x_n \rangle =$$

$$= \langle f(g(x_0)), \ldots, f(g(x_n)) \rangle$$

which is the same result.

More rewrite rules may be proved correct when considering skeletons as introduced in this book. As an example, assuming we denote with Δ skeletons or skeleton compositions, we may simply verify that, for any skeleton Δ_i

$$\texttt{farm}(\Delta_i) \equiv \Delta_i \tag{10.2}$$

The `farm` skeleton, in fact, does only impact the parallelism degree used to compute a given program, not the functional semantics of the program computed through the farm. For the very same reason, we can state that

$$\texttt{pipe}(\Delta_1, \Delta_2) \equiv \texttt{pipe}(\texttt{farm}(\Delta_1), \texttt{farm}(\Delta_2)) \tag{10.3}$$

By further rewriting the last expression, using the 10.2 rule, we get another interesting example of skeleton tree rewriting, namely:

$$\texttt{farm}(\texttt{pipe}(\Delta_1, \Delta_2)) \equiv \texttt{pipe}(\texttt{farm}(\Delta_1), \texttt{farm}(\Delta_2)) \tag{10.4}$$

It is worth to point out that rewriting rules 10.1 and 10.2 are *primitives* while the rewriting rule 10.3 and 10.4 are clearly derived applying rewriting rule 10.2.

The importance of rewriting rules such as 10.1 and 10.2 is mainly related to non functional behaviour of your skeleton program.

E.G.▷ Let us assume a programmer has written a program, whose parallel structure is modeled by the skeleton tree

$$\texttt{pipe}(\texttt{Seq}(C_1), \texttt{Seq}(C_2))$$

and let us suppose the monitored performance values state that $T_{S_{C_1}} = t_1 < t_2 = T_{S_{C_2}}$ Then an easy way of improving the program "non functional" behaviour–the performance–is to apply rule 10.2 and transform the program as

$$\texttt{pipe}(\texttt{Seq}(C_1), \texttt{farm}(\texttt{Seq}(C_2)))$$

Mostly likely, this could require some minor change in the source code of the skeleton program but has a notable impact on program performance eventually achieved on the target architecture[145] ■

The rewriting rules just discussed should be read also in a more structured way, depending on the target architecture considered.

[145]We are assuming here that the inter arrival time of tasks to the pipeline is small enough in such a way it does not constitute a bottleneck and that the farm implementation is feasible, that is, the time spent to schedule tasks to the farm workers and to gather from these workers results is not larger than the time spent to compute the single result. If either of these assumptions does not hold, than the rewriting is useless, of course.

E.G.▷ Consider rule 10.1 and assume we want to implement a program operating on a stream of input tasks by applying to each one of the tasks a data parallel skeleton, as an example through the skeleton program

$$\texttt{pipe}(\texttt{map}(\Delta_1), \texttt{map}(\Delta_2))$$

If we assume:

1. that a "sequential composition" skeleton exist, $\texttt{comp}(\Delta_1, \Delta_2)$ performing in sequence, on the same processing elements, the computations of its parameter skeletons Δ_1 and Δ_2, and
2. that a single stage `pipe` may be used, representing the fact the computation modeled by the single parameter skeleton of the pipe is to performed on a stream of input data,

then we can state that

$$\texttt{pipe}(\texttt{map}(\Delta_1), \texttt{map}(\Delta_2)) = \texttt{pipe}(\texttt{map}(\texttt{comp}(\Delta_1), (\Delta_2))) \tag{10.5}$$

Now, either left hand side or right hand side expressions may be used to implement the same computation. What are consequences on the non functional features of our skeleton program?

- If left hand side expression is used:
  - Computation of $\texttt{map}(\Delta_1)$ and $\texttt{map}(\Delta_2)$ happen concurrently on different items of the input stream. As a consequence, we expect a better service time, being it expressed as the maximum of the service times of the two maps rather than being computed as the sum of the two
  - More communication are involved. We have to distribute the input tasks to the workers of the first map, than to distributed the results of the first map to the workers of the second map and eventually to collect the final result from the workers of this second map.
- If the right hand side expression is used:
  - The computation of each input data item happens serially in a single map, therefore the latency (and the service time) is higher, but
  - Less communication overhead is needed to implement the parallel computation (we omit the communications in between workers of the first map and second map, that in this case happen to be the same).

All in all we can conclude that the left hand side program is *finer grain* than the right hand side. Depending on the target architecture at hand *and* on the features of the surrounding skeletons (if any), one of these two equivalent programs may result "better" than the other one. ■

10.2 SKELETON REWRITING RULES

We summarize in this Section the more common rewriting rules that can be used with stateless skeletons such as those defined in Chap. 3. The rewriting rules are *named*, that is we give each of them a conventional name. Moreover, part of them are given *directionally*, that is instead of giving an equivalence such as $\Delta_1 \equiv \Delta_2$ they define a transformation from one term to the other one, such as $\Delta_1 \rightarrow \Delta_2$. The existence of both $\Delta_1 \rightarrow \Delta_2$ and $\Delta_2 \rightarrow \Delta_1$ rules gives the full equivalence between the two skeletons ($\Delta_1 \equiv \Delta_2$), of course.

Farm introduction $\Delta \rightarrow \texttt{farm}(\Delta)$

A farm can always be used to encapsulate any skeleton tree Δ

Farm elimination $\texttt{farm}(\Delta) \rightarrow \Delta$
A farm may always be removed.

Pipeline introduction $\texttt{comp}(\Delta_1, \Delta_2) \rightarrow \texttt{pipe}(\Delta_1, \Delta_2)$
A sequential composition may always be substituted by a pipeline.

Pipeline elimination $\texttt{pipe}(\Delta_1, \Delta_2) \rightarrow \texttt{comp}(\Delta_1, \Delta_2)$
A pipeline may always be transformed into the sequential composition or its stages.

Map/comp promotion $\texttt{map}(\texttt{comp}(\Delta_1, \Delta_2)) \rightarrow \texttt{comp}(\texttt{map}(f), \texttt{map}(g))$
The map of a composite may always be rewritten as a composite of maps.

Map/pipe promotion $\texttt{map}(\texttt{pipe}(\Delta_1, \Delta_2)) \rightarrow \texttt{pipe}(\texttt{map}(f), \texttt{map}(g))$
The map of a pipeline may always be rewritten as a pipeline of maps.

Map/comp regression $\texttt{comp}(\texttt{map}(f), \texttt{map}(g)) \rightarrow \texttt{map}(\texttt{comp}(\Delta_1, \Delta_2))$
The composite of maps may always be rewritten as a map of a composite.

Map/pipe regression $\texttt{pipe}(\texttt{map}(f), \texttt{map}(g)) \rightarrow \texttt{map}(\texttt{pipe}(\Delta_1, \Delta_2))$
The pipeline of maps may always be rewritten as a map of a pipeline.

Pipe associativity $\texttt{pipe}(\Delta_1, \texttt{pipe}(\Delta_2, \Delta_3)) \equiv \texttt{pipe}(\texttt{pipe}(\Delta_1, \Delta_2), \Delta_3)$
Pipeline is associative.

Comp associativity $\texttt{comp}(\Delta_1, \texttt{comp}(\Delta_2, \Delta_3)) \equiv \texttt{comp}(\texttt{comp}(\Delta_1, \Delta_2), \Delta_3)$
Sequential composition skeleton is associative.

10.3 SKELETON NORMAL FORM

An important consequence of the rewriting rules discussed in the previous section is the "normal form" concept for skeleton programs developed in late '90.

When considering a skeleton rewriting system, such as the one defined by rewriting rules listed in the previous section, we may be interested in defining the "better" rewriting–with respect to some non functional feature of our program–possible of any given program Δ.

This "better" rewriting is called "normal form" to represent the fact that this is the alternative[146] program representation that would normally, that is in absence of special situations, considered for the implementation.

A normal form for a small subset of stream parallel skeletons has been defined in [12]. The skeleton framework considered only includes pipeline, farm, sequential composition and sequential wrapping skeletons, denoted, respectively, by `pipe`, `farm`, `comp` and `seq`, as usual.

The rewriting rules considered basically only include farm elimination/introduction, which is the rewriting rule 10.2 above.

Authors also consider a simple performance modeling for the skeletons included in the framework. Their performance modeling states that

$$\begin{array}{lcl} T_S(\texttt{pipe}(\Delta_1, \Delta_2)) & = & max\{T_S(\Delta_1), T_S(\Delta_2)\} \\ T_S(\texttt{farm}(\Delta)) & = & max\{T_{emitter}, T_{collector}, \frac{T_{worker}}{n_w}\} \\ T_S(\texttt{comp}(\Delta_1, \Delta_2)) & = & L(\Delta_1) + L(\Delta_2) \\ T_S(\texttt{seq}(C)) & = & L(C) \end{array}$$

with $L(x)$ being the latency of x and $T_{emitter}, T_{collector}$ and T_{worker} being the service times of farm emitter, collector and workers processes.

[146] with respect to the rewriting system considered

Under these assumptions, the "normal form" of any skeleton program Δ is given as follows:

- the set of the "leaves" of the skeleton tree Δ is computed, as they appear left to right in the skeleton tree. This set, $C_1, \ldots, C_n$ is the set of the sequential wrapping skeletons appearing in the skeleton program. This set is called *fringe* and it can be recursively defined on the structure of the skeleton program as follows:
  - $fringe(\texttt{seq}(C_i)) = C_i$
  - $fringe(\texttt{comp}(C_1, \ldots, C_n)) = \{C_1, \ldots, C_n\}$
  - $fringe(\texttt{pipe}(\Delta_1, \Delta_2)) = append(fringe(\Delta_1), fringe(\Delta_2))$
  - $fringe(\texttt{farm}(\Delta)) = fringe(\Delta)$

  with *append* defined as usual, appending at the end of the first parameter the items relative to the second parameter.
- the "normal form" of the original program is given by

  $$\mathcal{NF}(\Delta) = \texttt{farm}(\texttt{comp}(fringe(\Delta)))$$

E.G.▷ Consider the program

$$\texttt{pipe}(\texttt{seq}(C_1), \texttt{farm}(\texttt{seq}(C_2)))$$

The left to right set of leaves is C_1, C_2 and therefore the normal form is

$$\texttt{farm}(\texttt{comp}(C_1, C_2))$$

■

Under the assumptions made by the authors in [12], the normal form guarantees the "best" service time for a stream parallel skeleton composition. This means that the following inequality holds, for any legal stream parallel skeleton composition Δ:

$$T_S(\mathcal{NF}(\Delta)) \leq T_S(\Delta)$$

The rationale behind the concept of normal form in this case[147] is the reduction of the overheads related to communications, taking into account that the coarser grain the workers are in a farm, the best speedup and efficiency you can get out of it. The overall result of applying the normal form can therefore be stated as follows:

> *whenever you have to parallelize a stream parallel program (skeleton composition) the best thing you can do is to "glue up" the largest grain worker by merging all the sequential portions of code needed to produce the final result, and then you can "farm out" the coarse grain worker.*

It is clear that the normal form as defined above also minimizes *latency*. If we consider the two equivalent skeleton programs Δ and $\mathcal{NF}(\Delta)$ the absolute wall clock time spent to compute $f(x_i)$, being f the function computed by Δ and x_i the generic input stream item will be smaller when computing f as $\mathcal{NF}(\Delta)$ than as Δ, due to the smaller communication overheads incurring.

E.G.▷ It is worth pointing out that the usage of the term "normal form" in this context does not comply with the "normal form" definition in rewriting systems. In rewriting systems the normal form is the term to which a given term may be reduced that could not be anymore re-written. In our skeleton context, normal form represents the "more performant" rewriting of a given term, which is the term that cannot be re-written any more without incurring in some kind of performance penalty. ■

[147] stream parallel skeletons only, stateless, with this particular performance model of the implementation of the skeletons provided to the programmer

10.3.1 Model driven rewriting

Most of the rewriting rules given in Sec. 10.2 are bi-directional, that is, they can be used to rewrite the left hand side skeleton expression into the right hand side one or vice versa.

If we consider a skeleton program Δ and:

- the target architecture,
- the implementation of the skeletons appearing in Δ
- the performance models associated to the implementation of the skeletons appearing in Δ

the problem of finding the "best" rewriting of Δ (with respect to some performance measure $\mathcal{M}$ is quite difficult.

If a set of rewriting rules $\mathcal{R} = \{\Delta_1 \rightarrow \Delta_2, \Delta_3 \rightarrow \Delta_4, \ldots\}$ exist, a program Δ may be used to generate the *rewriting tree* $RwTr(\Delta, \mathcal{R})$ such as:

- Δ is the root of the tree $RwTr(\Delta, \mathcal{R})$
- if a rule $\Delta_i \rightarrow \Delta_j$ applies to node $N_k \in RwTr(\Delta, \mathcal{R})$ (that is N_k is a skeleton expression Δ_{N_k} with a sub term matching Δ_i), then N_k is linked to a son node where the occurrence(s) of Δ_i in Δ_{N_k} have been replaced by Δ_j.

The $RwTr(\Delta, \mathcal{R})$ may be infinite. We therefore use the performance models relative to $\mathcal{M}$ to restrict the number of nodes in the tree as follows[148]:

- we start with $RwTr(\Delta, \mathcal{R})$ being made of the only node Δ with no son nodes.
- we expand the leaves in the tree by one step, i.e. we consider, for each leaf node N_k as many new son nodes as the number of rules in $\mathcal{R}$ matching the skeleton tree of N_k and obtained rewriting the skeleton tree of N_k with these rules.
- we evaluate the performance measure of interest $\mathcal{M}$ on all the new leaves and we flag all the new nodes with the corresponding $\mathcal{M}$ values. Then, for any new node N_i added (with its associated performance measure p_i)
  - if $p_i \geq min\{p_j \mid N_j \in \{set\ of\ leaves\ in\ the\ previous\ iteration\ tree\}\}$ we stop expanding N_i
  - otherwise, we insert the new node N_i in the skeleton tree
- if the tree has at least one new node, we cycle expanding the tree leaves again.

The tree generated this way is finite as we assume to have available a finite number of processing resources to implement the skeleton tree and any performance measure chosen will be eventually bound by the amount of resources available. As the process of building tree nodes is monotonic, we will eventually reach the performance limit and therefore terminating the generation of the rewriting tree.

In the final tree, the leaf node N_j with the better associated performance value p_j will eventually represent our "best rewriting" of Δ with respect to the set of rules $\mathcal{R}$.

[148] this is a *branch-and-bound* like approach

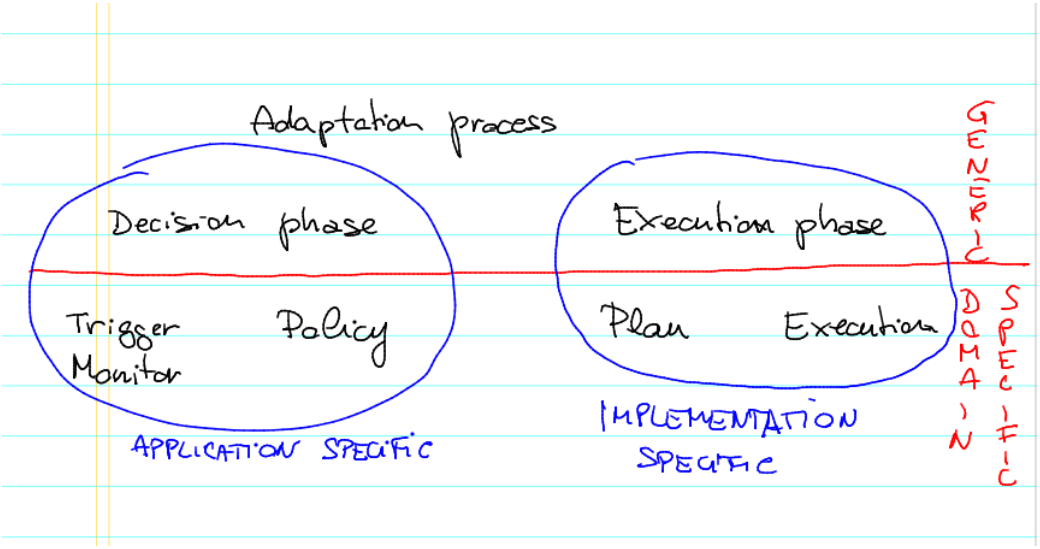


Figure 10.1 General adaptation process

10.4 ADAPTIVITY

Most of times, parallel programs execution needs to take into account with variations of the execution environment. These variations may be:

Endogenous that is depending on the parallel application features. As an example, a parallel application may be such that computations phases have different "weights" and therefore, in case they are executed on a constant number of processing resources, in some phases the available resources are underutilized and in other phases they are over utilized. This obviously leads to poor application performance, in general.

Exogenous that is depending on the target architecture features and/or behaviour, which represent causes somehow "external" to the parallel application. This is the case of non exclusive target architecture access, for instance. If another (parallel) application is run on the very same resources our application is currently using, the load share may lead to a poor performance of our application. Or, in case there is a kind of "hot spot" in Operating System activities, the same kind of impact on the running parallel application may be observed, even in case of exclusive target architecture access[149].

Therefore, dynamicity in the conditions related to parallel application execution generates two distinct kind of problems:

- *resource under utilization*, which is bad as unused resources may be dedicated to more useful tasks instead, and
- *resource over utilization*, which is bad as more resources could greatly improve the overall application performance, if available.

In turn, these two kind of problems have different effects on the execution of parallel applications:

- load unbalancing may be verified,

[149] this is the case of operating system activities scheduled at fixed times, such as backup activities, for instance.

- performance is unsatisfactory,
- efficiency is poor.

Therefore it is natural to think to the implementation of some kind of *adaptivity process* that could dynamically adapt the application execution to the varying features of the execution environment or to the varying needs of the application itself.

E.G.▷ Adaptation has been included in some applications, not using skeletons, actually. As an example *multi-grid* applications, i.e. those applications that analyze phenomena that happen to have different effects in different space portions usually adapt the computation to space portions features. Let suppose you want to compute the fluid dynamics of airflow on an airplane wing. Simulation must be more accurate in the space portions where more complex dynamics take place, such as on the front wing borders. Less accurate simulation is enough on flat wing surface space portions, instead. Therefore, if we want to tackle the problem with a kind of data parallel approach and we divide the wing space in equal portions, the portions simulating airflow on the wing border will over stress the computing resources they have got assigned while (comparatively) the portions simulating airflow on wing flat surface will under utilize the (same dimension) resources assigned. This require some kind of adaptation. In multi grid applications such as the one just mentioned this may consist in

Static adaptation The partition of the overall simulation space takes into account the features of the computation and splits more deeply those areas subject to more intensive computations, in such a way eventually all the partitions generated achieve a good load balancing.

Dynamic adaptation The initial partitioning is uniform (a number of partitions with similar dimensions) but then, at run time, in case a partition turns out to be "too computationally intensive", it may be dynamically split in such a way the resulting partitions use both the resources previously assigned to the original partition and further resources dynamically recruited.

Now, when unstructured parallel programming models are used, the adaptation is actually programmed by intermingling adaptation code with the application code (functional and related to parallelism exploitation). This requires notable extra-knowledge to the application programmer and also contributes to poor maintainability of the resulting code. ——— ■

The structure of general purpose adaptation process suitable to manage dynamicity in parallel applications have been studied by different authors. A general adaptation schema has been proposed in [8] which is represented in Fig. 10.1. In this case:

- The adaptation schema is clearly organized in two steps:

  DS Decision step: the behaviour of the execution of the application is monitored and analyzed. Possibly, decisions are taken concerning the adaptation of the application, in case it does not perform as expected.

  ES Execution step: a plan is devised to implement the decisions taken in the decide step and this plan is eventually executed in such a way the decision could be implemented.

  In turn, the decision step may be logically divided into *monitoring* and *analyze* steps and the execution step may be logically divide into *plan* and *execute* steps, with obvious "semantics" associated to each one of these phases.

- The adaptation schema is also logically divided in two layers:

  GL Generic layer. This encapsulates all the *generic*–i.e. non specific of the particular application at hand–decisions and actions.

DSL Domain specific layer. This encapsulates all the details more specifically related to the particular application domain.

- Eventually, the whole adaptation process has some parts that are application specific and some that are implementation specific. In particular:
  - The monitor, plan and execute phases are implementation specific. They rely on the mechanisms provided to perceive application behaviour (called *sensors*) and on those provided to actuate decisions (called *actuators*).
  - The analyze phase is completely application dependent and *de facto* it embodies all the details specific of the application at hand.

Let us analyze more in detail the two steps of the adaptation process.

10.4.0.1 Decision step In the first part of the decision step the current behaviour of the application execution is monitored. The necessary monitoring activity is usually planned at compile time by inserting proper monitor "probes" into the application parallel code. This activity is usually referred to by the term *code instrumenting*. Code instrumenting can be organized such that it is automatically handled by the compiling tools or performed under user (the programmer, in this case) responsibility. In the former case, the compiling tools must be exposed with the parallel structure of the application to be able to place the probes in the more appropriate places within the application source code. In general, the monitoring phase of the decision step is greatly impacted by the number and kind of probes available on the target architecture.

E.G.▷ Most of modern architectures (CPUs) provide *hardware counters* that can be used to measure clock cycles spent in a computation, as well as cache hits/misses and other measures of interest to determine the performance of an application. In case these hardware counters are not available, then more approximate measures can be obtained through operating system probes. The quality of the monitoring results is consequently more approximated. —— ■

In the second part of the decision step the current behaviour of the application–as exposed by the monitoring phase–is analyses and decisions are possibly taken concerning adaptation actions to be performed. This phase must be supported by some kind of abstract performance modeling of the application at hand. The performance figures computed using the abstract models should be compared to the performance figures actually gathered through monitoring. In case the two measures do not match, adaptation is required.

Once established that adaptation is required, the analysis phases must be able to figure out which *abstract adaptation actions* may be taken to handle the mismatch in between observed and predicted performances. Possible abstract actions include increase/decrease the parallelism degree and load re balancing, as an example, but also more complex actions such as complete restructuring of the parallel application, that is changing the parallelism exploitation pattern–or the pattern composition–used to implement the application.

The analysis phase may be implemented with one of two possible "general behaviours" in mind:

- It can be implemented as a *reactive* process. The analysis phase reacts to the monitored events not matching the expected application behaviour and decides to take *corrective* actions impacting future application behaviour.
- It can be implemented as a *proactive* process. The analyze phase proactively takes decisions that try to anticipate possible misfunctioning of the application. It digs monitored events and performances to look for symptoms of future misfunctioning and reacts to the symptoms–rather than to actual misfunctioning–in such a way the appearance of misfunctioning is prevented.

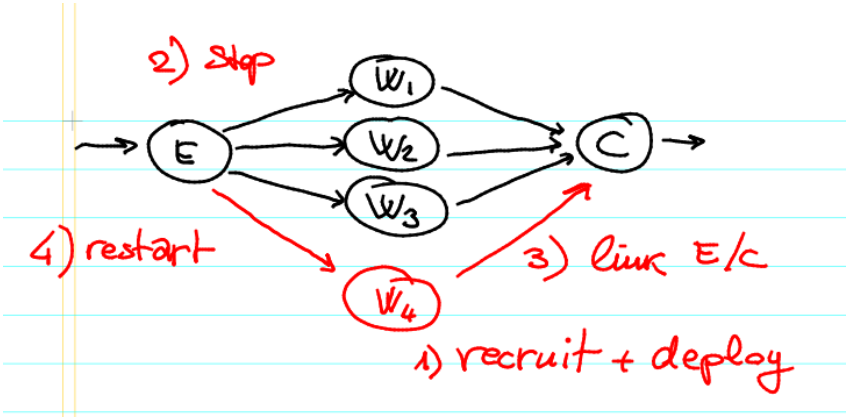


Figure 10.2 Adding worker plan

10.4.0.2 Execution step The execution step is actually built out of two distinct activities. The first activity is aimed at devising a suitable implementation plan for the decisions taken in the decision step. The second one is aimed at actually implementing the decisions taken on the running application according to the plan just devised.

During the plan devising phase, the ongoing computation may proceed without any kind of interference. As a matter of fact, apart from the monitoring phase of the decision step and the commit phase of the execution step, the current computation is not impacted by the adaptation process. Also, during the devising phase the parallel structure of the application–exposed through the skeleton tree actually used in the application–is exploited to devise the precise points of intervention on the running application to be used to implement the adaptation decision taken. In particular, solutions requiring a minimum downtime of the running application will be taken into account.

E.G.▷ When implementing the decision to increase the parallelism degree of a farm, we both the following implementation plans are feasible[150]:

1. stop farm template execution, then recruit a new processing resource, then deploy and start worker code onto the recruited resource, then link the worker to the emitter and collector processes, restart farm execution.
2. recruit a new processing resource, then deploy and start worker code onto the recruited resource, then stop template execution, then link the worker to the emitter and collector processes, restart farm execution (see Fig. 10.2).

Clearly, the second plan requires much smaller downtime than the first one as during all the process of resource recruiting and of remote code deployment, the farm template may continue its normal operations. ——————————— ■

At the end the plan devising phase, we may assume an ordered list of actions $\langle a_1; \ldots; a_k \rangle$ is devised that will implement the decisions taken in the decision phase. Each action a_i is defined in terms of the *actuators* provided by the implementation of the skeletons affected by the adaptation decision. This means that in case of template based skeleton frameworks the templates implementing skeletons should provide actuators to implement any reasonable adaptation action. In case of a macro data flow based framework, the implications are more relative to the possibility of change *on-the-fly* the template used to implement the distributed macro data flow interpreter, instead.

[150] we assume the farm is implemented through an emitter-workers-collector template such as the one discussed in Sec. 4.2.1

If we take a "template first" viewpoint, instead, the actuators provided by the templates *determine* the kind of dynamic[151] adaptation decision the system can actually implement. It is clear, in fact, that if the system does provide, as an example, mechanisms to stop the execution of our template and to modify the mechanisms or the configuration used during template execution, it will be impossible to actuate any kind of decision taken during the adaptation process.

The second phase of the execution step simply scheduled and applies actions $\langle a_1; \ldots; a_k \rangle$ onto the current running application. At the moment being we are only considering consecutive actions, that is we assume that the sequence $a_i; a_{i+1}$ has to be executed as

first execute action a_i *and then execute action* a_{i+1}.

Nothing prevents to adopt a more structured action plan language. As an example, we can assume that actions are defined by a grammar such as:

```
Action ::=
    <simple action>
|   Action then Action
|   Action par Action
```

where we interpret the second clause as "do first action then do the second one" and the third one as "do the first and the second action in parallel".

Also, we should consider the possibility that each action corresponds to a finite sequence of "actuator mechanisms calls", as the actuator mechanisms provided by the system may range from low level ones–simple, RISC style mechanisms–to high level ones–corresponding to full actions or event to entire parts of actions plans.

E.G.▷ To illustrate how the adaptation process discussed in this section works, we consider a simple data parallel computation and we discuss all the aspects related to adaptation in detail.

The application we consider is such that it can be expressed using a single data parallel skeleton. Computation at the workers is sequential. The expected performance will be roughly proportional to the number of workers used in the implementation of the data parallel skeleton[152].

During the monitoring phase we must gather information concerning the *progress* of our application. If the data parallel skeleton actually worked on a stream of data parallel tasks, it would be probably enough to gather data relative to the service time of the data parallel skeleton. This needs only one kind of sensor, namely the one reporting the inter-departure time of results from data parallel template "collector" process, that is from the process in charge of delivering the results onto the output stream. However, if no stream is present[153], we must be able to monitor the progress in the computation of a single data parallel task, the only one submitted for computation. In this case, all worker processes must be instrumented in such a way they report the percentage completed of the task(s) they received. This is a little bit more complex to be implemented, as a sensor. First of all, each worker must know a way to devise the percentage of work already performed. This is clearly an application dependent issue. Therefore only the programmer may probably be able to write this sensor. In some simple cases, the sensor may be determined from the skeleton and from the template structure. As an example, in embarrassingly parallel computations on arrays, each sub-item y_{ij} of the result is computed applying some kind of f to the corresponding sub item x_{ij} of the input data structure. The implementation template will probably assign a range of indexes $\langle [i_{min}, i_{max}], [j_{min}, j_{max}] \rangle$ to each one of the workers. Therefore each one of the workers will execute a

```
for(i=i_min; i<i_max; i++){for(j=j_min; j<j_max; j++)
```

[151] and run time
[152] we assume to use a template based skeleton framework
[153] as in a pure data parallel computation

and as a consequence at each iteration it will be able to give the percentage of the iterations already performed.

During the analyze phase we reason about the progress of our data parallel computation. We may observe different situations that may possibly be corrected with a corrective action:

- workers computed sensibly different amounts of the assigned work. This denotes *load imbalance* than in turn will eventually produce poor performance. Therefore the computations assigned to the workers that have currently computed much less than the other workers may be split and partially re-distributed to new workers, recruited to the purpose. Alternatively, the computation assigned to these workers may be split and the "excess" portions may be delegated to the workers currently performing better than expected, if any.

  It is worth pointing out that

  * load imbalance may be due to extra load charged on the processing resources used by the under performing workers, but also
  * load imbalance may be due to intrinsically heavier work loads assigned to the workers. Equal size partitioning of sub-items to workers is not the optimal choice in case of sub-items requiring sensibly different times to be computed, as an example.
- workers computed fairly equal amounts of the assigned work, but this amount does not correspond to the expected amount. In this case, we can assume that the target architecture eventually is "slower" than expected and therefore we can assume to re-distribute work among a larger number of workers, with extra workers recruited on-the-fly to the computation.
- workers computed fairly equal amounts of the assigned work, but this amount sensibly exceeds the expected amount. In this case, we can assume that the target architecture eventually is "faster" than expected and therefore we can assume to re-distribute work among a smaller number of workers, dismissing some of the workers currently used.

During the planning phase, several alternative decisions can be made. As an example:

- when working on an embarrassingly parallel data parallel skeleton, the plan should include the actions:
  * stop worker with excess load, split load still to be computed in two parts, then *in parallel*
    · re-assign the worker the first part, restart the worker *and*
    · stop a worker with low load left to compute, re-assign the second partition to the worker and restart it.
- when working on a stencil data parallel skeletons, the plan should include the actions:
  * wait for a barrier to be executed[154]
  * then proceed as in the case of the embarrassingly parallel skeleton, that is stop the worker with excessive load, split load ... etc.

■

10.5 BEHAVIOURAL SKELETONS

Behavioural skeletons have been introduced in early 2000s with the aim of supporting autonomic management of non functional features related to skeleton implementation. In particular, the first non functional property taken into account was performance.

> *A behavioural skeleton is the result of the co-design of a parallelism exploitation pattern and of an autonomic manager, taking care of some non functional feature related to the parallelism exploitation pattern implementation.*

[154]this is the mechanisms usually employed to synchronize workers before starting a new iteration, in such a way all the workers start with a consisted "neighborhood to collect stencils

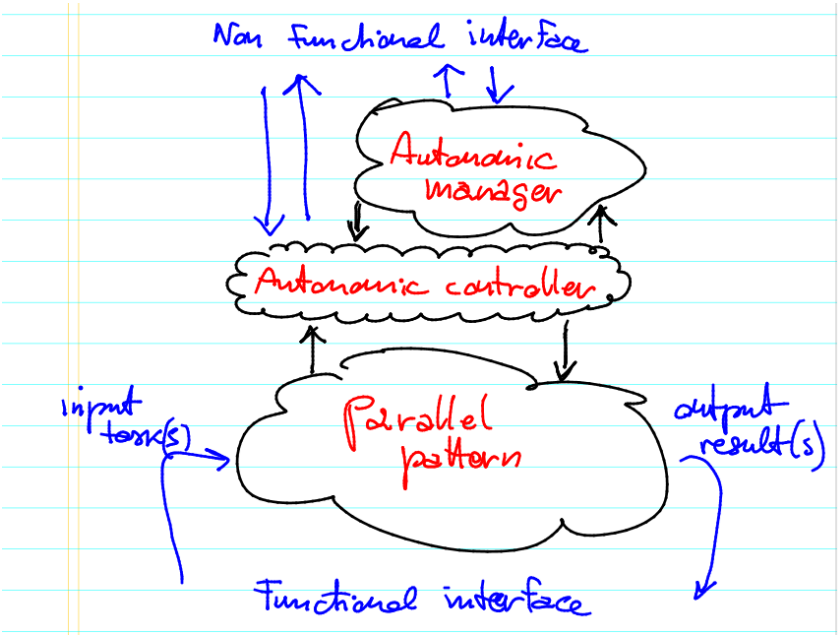


Figure 10.3 A behavioural skeleton (abstract view)

The concept behind behavioural skeleton is simple. By exposing the parallel computation structure to the autonomic manager, the manager may quite easily implement the adaptation schema discussed in Sec. 10.4. The complete separation of concerns between pattern implementation–taking care of the details related to parallelism exploitation–and manager–taking care of non functional features–allows very effective adaptation policies to be implemented. Eventually, the separation of concerns allows very clean and effective interfaces to be implemented exposing parallel pattern implementation (monitoring/triggering interface) and the parallel pattern mechanisms suitable to intervene on pattern implementation (actuator interface) to the autonomic manager.

Also, the complete separation of concerns between pattern implementation and pattern non functional management allows management policy reuse across different skeletons with similar implementations.

E.G.▷ As an example, once a proper performance management policy has been developed in an autonomic performance manager controlling a task farm emitter-workers-collector implementation, the very same policy could probably be reused in the implementation of an autonomic performance manager controlling an embarrassingly parallel skeleton, that will probably run a very similar scatter-workers-gather implementation. ——————— ■

A behavioural skeleton (see Fig. 10.3) is made out of separate and cooperating entities, both active and passive, namely:

Parallel pattern (active+passive) This is actually the parallel pattern implementation template , that is the part actually computing–in parallel–the function exposed to the user/programmer by the behavioural skeleton.

Autonomic manager (active+passive) This is the manger taking care of non functional features related to the pattern implementation. It implements the adaptation policies as outlined in Sec. 10.4.

Autonomic controller (passive) This component of the behavioural skeleton embodies the interfaces needed to operate on the parallel pattern execution, both the monitoring/triggering and the actuator ones.

Active parts of the behavioural skeleton are implemented with their own control flow (e.g. as threads, processes or components), while the passive parts are implemented in such a way their services may be started from an external, independent control flow. In particular:

- the autonomic controller is passive in that it only provides services modeled by the monitoring, triggering and actuator interfaces. Monitoring and actuator interfaces may be invoked from the behavioural skeleton manager active part. Triggering interface is usually invoked by the manager active part to register proper callbacks to be invoked when a given event happens in the parallel patter execution.
- the parallel pattern passive part provides the interfaces needed to operate the pattern, that is to submit task(s) to be computed and to retrieve the corresponding computed result(s). These interfaces are known as the behavioural skeleton *functional interfaces*.
- the autonomic manager passive part provides instead the *non functional* interfaces, that is those interfaces that may be used from outside to monitor behavioural skeleton execution (that is to retrieve measures and parameters relative to the behavioural skeleton execution progress) and/or to intervene on the behavioural skeleton manager policies (that is to re-define the management policies of the manager).

The functional and non functional interfaces of behavioural skeletons may be implemented in two different styles: in RPC/RMI style or in a more conventional[155] send/receive data flow channel style.

- In the former case, computing services offered by the behavioural skeleton may be invoked as we invoke a remote procedure or method. The invocation may be synchronous of asynchronous. If synchronous, the invocation is blocking and we'll eventually get the result of the service as result of the remote procedure/method invocation. If asynchronous, instead, we will immediately get an handle (usually a *future*) that represents the result that will be eventually computed by the behavioural skeleton. Once the result is actually needed the handle (future) *reification* may be invoked. This invocation is synchronous and will only terminate when the handle (future) is actually substituted by the actual computed value.
- In the latter case, instead, behavioural skeletons will implement ports (kind of communication channels) and services may be invoked sending data items to proper ports and receiving results from other ports.

The very first behavioural skeletons have been designed and implemented within the reference implementation of the GCM, the CoreGRID Grid Component Model. In that case:

- each behavioural skeleton was provided as a compound component
- the autonomic manager (AM) and the parallel pattern were components
- the autonomic controller (AC) was part of the interface of the parallel pattern component
- the behavioural skeleton non functional interface was the interface of exposed by the autonomic manager component

[155] at least from the parallel processing viewpoint

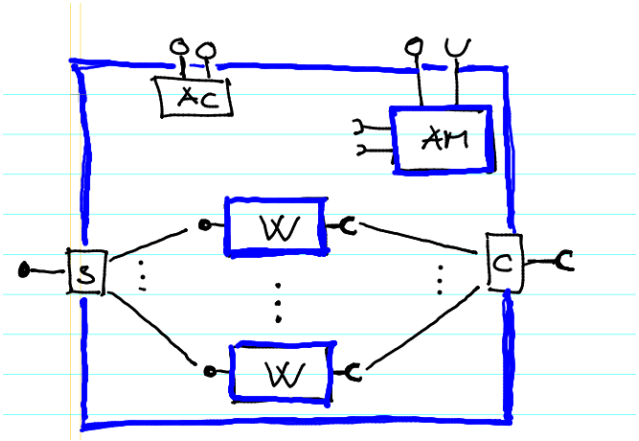


Figure 10.4 Functional replication behavioural skeleton

- the behavioural skeleton functional interface was the interface exposed by the parallel pattern component
- autonomic managers was only taking care of performance of the parallel patterns
- pipeline and functional replication behavioural skeletons were implemented. The functional replication behavioural skeleton could be specialized to implement both stream parallel and data parallel embarrassingly parallel computations.

10.5.1 Functional replication behavioural skeleton in GCM

Fig. 10.4 shows the structure of the functional replication behavioural skeleton. Functional replication behavioural skeleton (FRbs) operates as follows:

- the functional server port accepts input stream data items and schedules them for execution to the FRbs internal worker components.
  - If the FRbs is configured as a stream parallel behavioural skeleton, each input data item is scheduled one of the worker components in the FRbs. The scheduling policy may be customized by the programmer instantiating the FRbs. The default scheduling policy is "on demand": the task is scheduled to the first worker component asking for a new task to compute.
  - If the FRbs is configure as a data parallel behavioural skeleton, each input task is split into a set of partitions (one for each internal worker component) and this partition set is scattered to the set of internal worker components.
- the worker components implement a loop. The loop body starts asking for a new task to be computed, then proceeds computing the task and sending the results to the client port delivering the FRbs results. Possibly double or triple buffering techniques are used to hide communication latencies (see Sec. 8.3.1).
- the functional client port receives results from the internal worker components and delivers them to the FRbs output stream.
  - In case the FRbs is configured as a stream parallel behavioural skeleton, each item received from an internal worker component is delivered to the FRbs output

stream, possibly after a reordering phase aimed at maintaining the input/output ordering of tasks/results.

- In case the FRbs is configured as a data parallel behavioural skeleton, the client port waits–in a sort of barrier–a result item from each one of the internal worker components, then it rebuilds the global result and this result is eventually delivered to the FRbs output stream

- the autonomic controller AC implements the monitoring and actuator interfaces of the pattern. Monitoring interface provides methods to read the amount of tasks already computed (completed) by the FRbs, the current parallelism degree, the service time delivered by and the inter arrival time measured at the FRbs. Actuator interface basically provides two services: one to increase and one to decrease the FRbs parallelism degree.

- each FRbs is given a *performance contract* either by the user or by the upper level behavioural skeleton in the behavioural skeleton nesting. The performance contract establishes, in the form of a Service time, the non functional behaviour expected by the user and therefore the goal to achieve through autonomic management policies in the autonomic manager of the FRbs

- the autonomic manager includes a JBoss business rule engine[156]. Such engine processes a set of *pre-condition* $\rightarrow$ *action* rules. The set of rules is processed in steps:

  1. first, all the pre-conditions are evaluated. Pre-conditions, in our case, include formulas on the values provided by the autonomic controller monitoring interface. As an example, a pre-condition may be

     `AC.currentServiceTime()`$\leq$`X AND AC.parDegree==Y`

  2. then the rules with a true pre-condition are considered. The Rete algorithm is used to pick up one of these rules as the candidate rule
  3. the candidate rule is executed, that is action part is executed. The action part of the rule is made of a sequence of FRbs autonomic controller actuator interface calls.

  The autonomic manager of the GCM behavioural skeletons works in quantum deadlines. Each behavioural skeleton is associated with a quantum time. In a loop, a timer is started with this quantum time. When the timer elapses, the JBoss rule engine is run, and the loop is started again.

- Sample policies implemented in the FRbs manager include:

  - augmenting the parallelism degree in case the inter arrival time is smaller than the current service time
  - lowering the parallelism degree in case the current service time is much better than the user specified performance contract
  - avoiding taking an increase/decrease parallelism degree decision in case a decrease/increase decision as just be taken. This avoid fast fluctuation of the FRbs around a given parallelism degree value.

[156]The JBoss rule engine has been first used in a SCA based implementation of behavioural skeletons. The very first behavioural skeletons in GCM used plain Java code to implement the autonomic manager (policies)

This kind of behavioural skeleton has been demonstrated effective in different situations, including:

Computation hot spots In case a computation traverses an "hot spot", that is a situation where the currently available resources are not sufficient to exploit all the parallelism present, new resources are recruited and allocated to the computation (e.g. as new workers) in such a way the hot spot effects are reduced. The recruited resources are automatically released when the computation needs lower again after the end of the hot spot.

Target architecture overload In case a computation is executed on a target architecture whose nodes become overloaded by other applications, the autonomic manager succeeds recruiting more resources to the computation is such a way these new resources supply the computing power missing from the overloaded resources. As before, the recruited resources are automatically released when the target architecture nodes are anymore overloaded.

Initial parallelism degree setup Long running applications may be started with an arbitrary initial parallelism degree. The autonomic manager will recognize whether or not this parallelism degree supports the user supplied performance contract and intervene on the parallelism degree consequently. Eventually, the application will reach a state such that the measured service time matches the user performance contract.

Some further points are worth to be mentioned, concerning GCM behavioural skeleton implementation:

- the autonomic management policies implemented are *best effort*. In case the user asks for a contract that could not be achieved due to limitation of the target architecture or of the parallelization chosen for the application at hand, it will stop to the closest performance that can be achieved with the parallel pattern and management mechanisms available
- the autonomic management is actually `reactive`, in that it only reacts to situations, rather than trying to anticipate problems.

10.5.2 Hierarchical management

Up to now, we only considered original behavioural skeletons, as defined within CoreGRID and GridCOMP EU funded research projects. These behavioural skeletons were implemented in such a way that they could be functionally composed (e.g. a pipeline stage could be implemented with a FRbs) but no coordination among managers of different behavioural skeletons was supported.

Later on, the possibility to implement *hierarchical management* in behavioural skeleton nesting has been explored. The idea was to implement policies suitable to implement performance (the first non functional concern considered) as a whole in the overall nesting of behavioural skeletons used to implement the user application. This means:

- to have a single user defined performance contract. Formerly, each behavioural skeleton needed its own performance contract, not related to the contracts provided to the other behavioural skeletons appearing in the application.
- to be able to derive from the unique, application related user provided contract contracts and policies suitable to manage all the behavioural skeletons of the application
- to be able to coordinate manager policies according to the hierarchical composition of managers deriving from the hierarchical composition of behavioural skeletons in the user application.

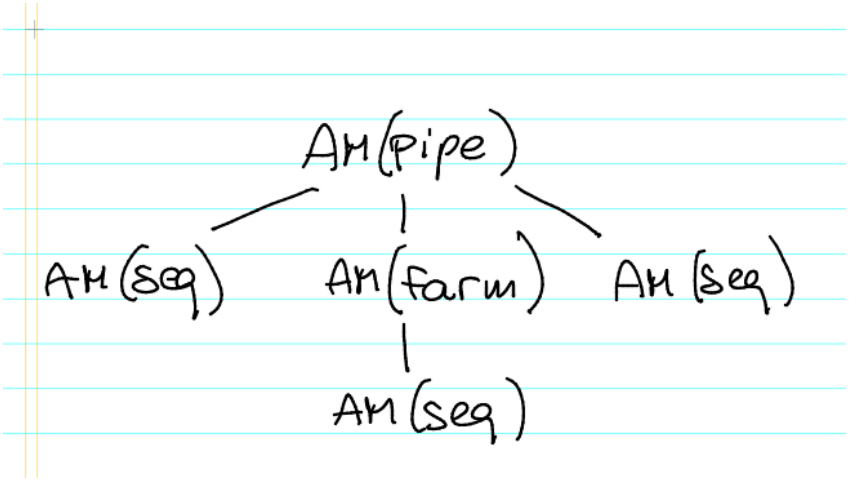


Figure 10.5 Sample manager tree

E.G.▷ Consider the manager tree of Fig. 10.5. This tree corresponds to the managers appearing in a behavioural skeleton tree (nesting) such as:

```
pipe(seq(C1),pipe(farm(seq(C2)),seq(C3)))
```

When designing coordinated, hierarchical autonomic management of performance in this behavioural skeleton composition we wish:

- to be able to supply a single performance contract related to the overall application. E.g. a single contract stating that the application must be implemented with a service time of at most t_s.
- to be able to derive, from the single performance contract, one "local" performance contract for each one of the managers in the tree. These local contract are the ones that, once satisfied, ensure the achievement of the overall, application performance contract.
- in case of failure of one of the managers, i.e. in case the manager does not succeed ensuring its local performance contract, the manager should inform its "parent manager" of the failure, in such a way more global adaptation actions can be taken, possibly involving issuing a new local contract for this manager.

■

We will now discuss the different problems related to hierachical, coordinated non functional feature management in behavioural skeletons.

10.5.2.1 Contract propagation The first problem is contract propagation. How can we derive local contracts for skeletons used as parameters of a parent skeleton, once the contract relative to the parent skeleton is known? In general, the process depends on both the type of the skeletons involved and in the type of non functional concern managed.

The specification of the contract propagation rules is therefore a task of the manager designer that uses knowledge and expertise from the parallel pattern designer.

The main rule to implement while designing contract propagation can be stated as follows:

for any skeleton $S(\Delta_1, \ldots, \Delta_k)$, given a contract $\mathcal{C}_S$ k contracts $\mathcal{C}_{S_1}, \ldots, \mathcal{C}_{S_k}$ have to be derived. Each new contract $\mathcal{C}_{S_i}$ will be a function of $\mathcal{C}_S$, type(S) and type(Δ_i) and it will be such that, if skeleton (manager of skeleton) Δ_i ensures $\mathcal{C}_{S_i}$, for any $i \in \{1, k\}$ then $\mathcal{C}_S$ is automatically ensured.

We assume, of course, that the user provided performance contract is actually the topmost skeleton initial contract.

E.G.▷ To exemplify the process, we consider again performance contracts in the form of required service times and and a (behavioural) skeleton framework supporting pipelines and farms, that is, stream parallel only skeletons.

The contract propagation rule is defined by induction on the skeleton type as follows:

Pipeline In a pipeline with service time contract t_S[157] the local contracts of all the pipeline stages are the same as the pipeline contract. This follows from the performance model of pipeline service time stating that the service time of a pipeline is the maximum among the service times of the pipeline stages.

Farm In a farm with service time contract t_s the service time of the workers is set to $t_s \times n_w$, with n_w being the number of workers in the farm. This follows from the performance model of the farm stating that the service time of the farm is the maximum among the emitter service time, the collector service time and the worker service time divided by the number of workers, under the assumption the the emitter and the collector are not bottlenecks in the farm implementation.

Seq A sequential portion of code wrapped in a sequential skeleton inherits the service time contract of its parent skeleton (as it happens for pipeline stages).

■

It is clear that although this rules must be eventually implemented by the manager designer, they definitely require all the knowledge available from the managed parallel pattern designer. The result of application of this contract propagation rule to the behavioural skeleton composition of Fig. 10.5 is the one illustrated in Fig. 10.6.

E.G.▷ Consider now the same skeleton framework, but assume managers must take care of computation security. In particular, assume that security in this contexts means ensuring confidentiality and integrity of data and or code processed and used by the application. In this case, the contract propagation rule may be simply stated as follows:

- the contract of the parent skeleton is inherited by nested skeletons.

This follows from the features of the particular security non functional concern considered. It is a binary property: either it holds or it does not hold. Therefore if the user asks the application to be secure, all the skeleton used must ensure "local" security. ■

10.5.2.2 Manager coordination The second relevant problem related to coordinated and hierarchical management of non functional features in a behavioural skeleton composition is even more important. It consists in the design of the coordinated management policies to be run among the different managers involved in the behavioural skeleton composition.

In this case we must devise proper policies to handle impossibility for a behavioural skeleton manager to ensure its local contract. This is the only case of interest as when all the managers succeed ensuring their local contract, the global contract is ensured "by construction", that is by the procedure used to derive the local contracts.

Once again, the handling of the manager fault in ensuring the local contract must be properly set up by the autonomic manager deisgner and requires specific knowledge relative to parallel pattern and to the particular non functional feature managed. The general policy to handle local failures can be stated as follows:

A manager failing to ensure the local contract keeps maintaining its current configuration and reports the local contract violation to the parent manager. "Keep the current

[157] that is required to guarantee at a service time $T_S(pipe) = t_s$

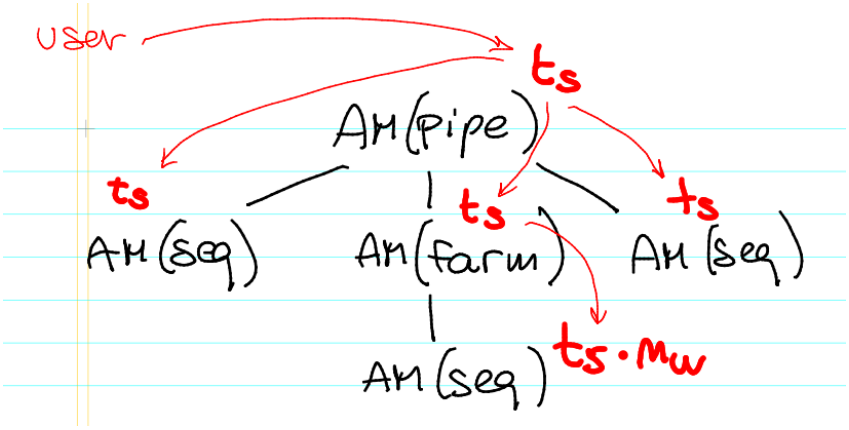


Figure 10.6 Sample performance contract propagation

configuration" means that no more adaptation decisions are taken up to the moment a new contract will be received from the parent manager. This "wait" state is called manager passive mode.

A manager receiving from a "son" manager a contract violation message will re-formulate its own management policy in such a way the sub-manager local contract violation could be accommodated. In case the re-formulation succeeds, the manager will deliver new performance contracts to all of its "son" managers. In case it does not succeed, the violation will be recursively reported to parent manager.

In case a contract violation is eventually reported by the top level manager, this is simply exposed to the user that may decide whether to continue in the execution of the "sub-performing" application or to stop it.

E.G.▷ To show how this policy may be implemented, we take again into account the behavioural skeleton composition of Fig. 10.5 and the same stream parallel behavioural skeleton framework considered when discussing contract propagation. Let us assume the manager of the first sequential skeleton (the first pipeline stage) may vary the rate producing result items onto the output stream, and let us assume an initial configuration of the program (that is an initial parameter n_w, the number of the workers in the second stage farm).

In case the processing elements that run the farm workers become overloaded because of some external program run concurrently with our skeleton application, the farm would eventually verify a service time no more ensuring the performance contract received.

The farm manager could apply the policy whose effect final effect is recruiting new resources and allocating more farm workers on these resources. If these resources may be actually recruited, the farm manager may locally re-ensure it contract. In case no new resources may be recruited[158], the performance manager of the farm will block any further adaptation process and communicate the local contract violation to the parent manager, that is to the manager of the pipeline skeleton.

The pipeline skeleton will therefore be made aware of a situation such that the first and the last stages perform better than the second one. As this is not useful at all, as the performance model of the pipeline states the pipeline service time is the maximum of the service times of its stages, the pipeline manager may resend to all its stages a new performance contract stating the service time should be the one currently provided by the second stage farm.

As a consequence, the first stage could decrease its delivery rate. Being a sequential stage, this does not actually save any processing element resource as it will happen if the first stage was a farm, as an example[159]. However, it may impact power consumption of the node or

[158] e.g. in case there are no additional resources available

[159] in case it was a farm, the decrease in the service time allowed by the new contract could lead to the release of resources used for one or more workers

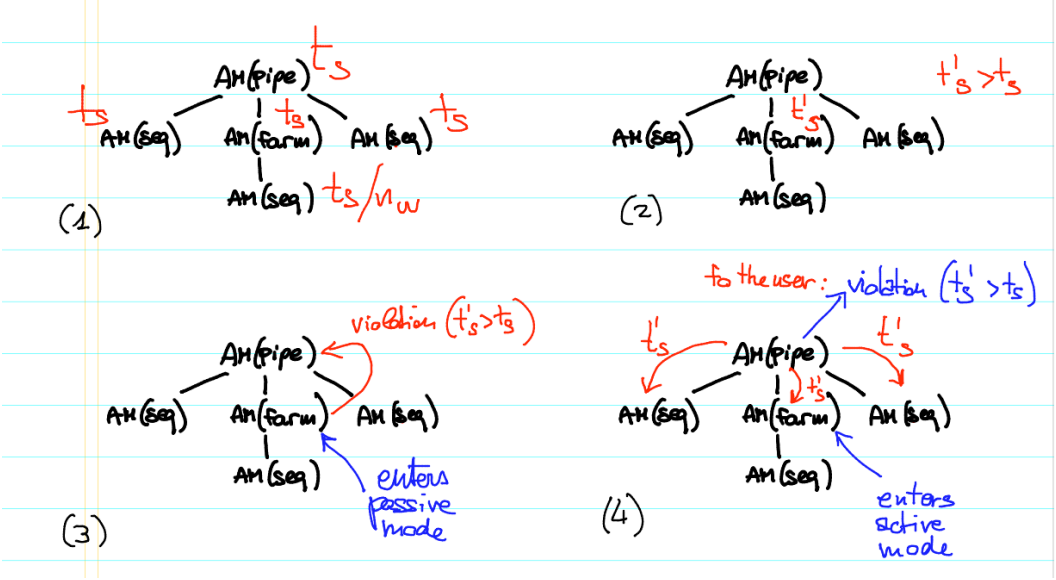


Figure 10.7 Hierarchical management of a contract violation

even this may allow to move the node on a less performant–and therefore less "expensive"–processing node[160].

In case the farm manager could not cooperate with its parent manager, instead, the violation of the contract would be confined to the farm and therefore no resource optimization could have been achieved.

It is worth pointing out that the violation eventually observed by the pipeline manager must be reported to the application user. The violation may be reported as a *non functional exception*. As for any exception, it can be "trapped" and managed by the user or explicitly ignored. ■

The example just discussed only details what can be done by the manager hierarchy in presence of a violation of a contract that cannot be solved. A much more interesting case is the one discussed below, concerning a situation when a local contract violation may actually be solved by the hierarchy of managers.

E.G.▷ We consider the same situation discussed in the last example. The user asked for a service time $T_S = t_s$. After a while, all the three stages satisfy their "local" contract that happens to be the same as the user supplied one, due to pipeline semantics.

Some overload on the node where the first stage is allocated makes the first stage unable to match its local contract anymore.

The first stage autonomic manager enters the passive mode and raises a violation exception to the pipeline manager. This happens as there is no way for a sequential code to improve its performance[161].

The violation of local contract by the first stage impacts also the second stage farm. In this case the farm will not be able to deliver the service time t_s required by its local contract as the inter arrival time of the tasks to be computed will be at most t'_s which is larger than t_s and no matter the number of workers included in the farm the farm service time will always be bound by this t'_s.

[160] the situation is shown in Fig.10.7

[161] we are taking the black-box approach here, that is we assume that no intervention is allowed on the sequential code wrapped in skeleton parameters.

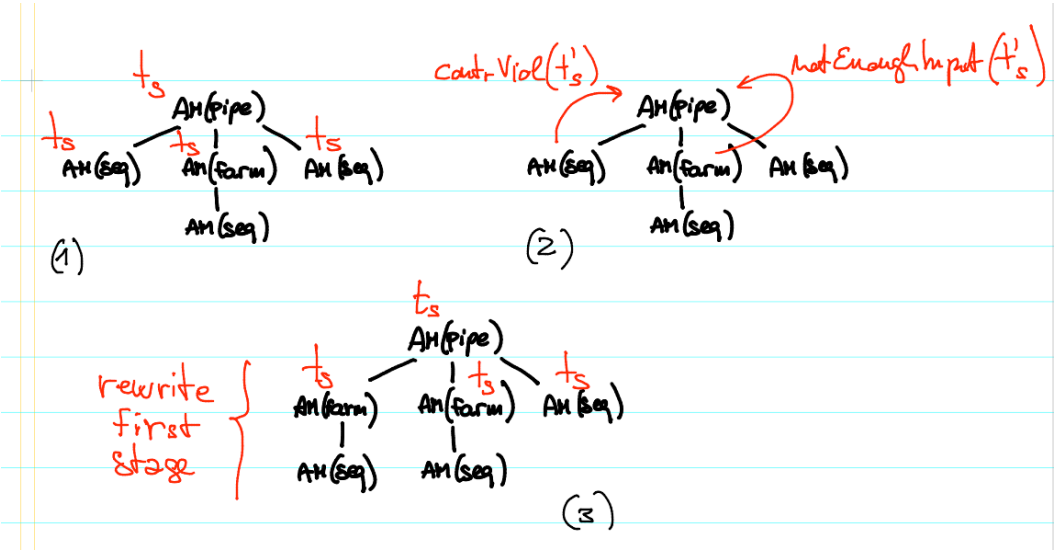


Figure 10.8 Skeleton tree rewriting in hierarchical management

Now the pipe manager will get the two violations from first and second stage. The first one will be a simple `ContractViolation`(t'_s) while the second one should be something different expressing the fact the violation is due to the behaviour of the modules external to the farm, such as a `NotEnoughInput`(t'_s). The pipe manager may therefore decide to improve the service time of the first stage by transforming this `seq(C1)` stage into a `farm(seq(C1))` stage. This eventually will lead to the full ensuring of the local contract at the first stage and, consequently, at the second stage also. This situation is depicted in Fig. 10.8. ■

10.5.3 Multi concern management

When considering behavioural skeletons and autonomic management of non functional features in parallel applications we should take into account that normally more than a single non functional concern is of interest relatively to parallel applications.

Notable non functional concerns to be taken into account, along with the main management goals are listed in Table 10.1. The question we must answer now is "can we design manager policies that allow more than a single non functional concern to be managed by the same manager hierarchy?" This would constitute a big improvement of the results discussed in the previous Section and relative to the autonomic management of a single non functional concern coordinated through a hierarchy of managers.

However, if we take again into account separation of concerns, another question comes immediately in mind: how may we force security and performance experts to work together to implement a single manager (hierarchy) dealing with both security *and* performance issues? It would be much better if complementary approach is followed: performance experts design performance policies and security experts design security policies. The two sets of policies are implemented in two independent manager hierarchies. The two hierarchies are somehow coordinated in such a way no conflicting "adaptation" decisions are taken.

This will of course a better solution:

- experts will only be concerned by their own field of interest issues, and therefore they will probably come out with excellent (the best possible?) policies.

Non functional concern	Main goals in management
Performance	Best performance independently of the resources used
Security	Code and data integrity and confidentiality
Fault tolerance	Tolerate limited amount of faults
Power management	Lowest power consumption

Table 10.1 Non functional concerns of interest

- independent managers may be built each monitoring (triggered by) non functional concern specific values (events) and operating on the parallel pattern by non functional concern specific actions. As the monitor/trigger and actuator interface are independent, we expect a much simpler implementation.

However, completely independent manager hierarchies are not a good solution, nor coordinating independently developed manager is a simple task.

E.G.▷ To illustrate the concept, assume two managers have been associated to each behavioural skeleton pattern, one taking care of performance and one taking care of security. The performance manger may decide a new resource is to be acquired to allocate a new worker that hopefully will improve some functional replication behavioural skeleton performance. The resource will be looked for, recruited, the worker allocated and eventually linked to the current computation. The computation happens to be the very same also controlled by the security managers. In case the new resource has been recruited from some "secure" domain (e.g. a cluster under the complete control of the user running the application, security requirements may be satisfied at no further cost. No "alien" users are supported by the cluster and therefore no (new) risks arise related to security of user application code and data. However, in case recruited resource belongs to some untrusted domain, or even in case it needs to be reached through some non secure interconnection link, then these nodes should not be used, in the perspective of the security manager of, if used, the communications to and from these nodes should be performed in some secure way. This means that in this case the decision taken by the performance manager is *in contrast* with the policies of the security manager. ■

E.G.▷ Another case illustrating the issues related to concurrent autonomic management of different non functional concerns is the following one. Assume two manager take care of performance and power management issues in a parallel computation. The power management will constantly look for more power efficient processing elements to allocate parallel computation activities. Notably, power efficient processing elements provide worst absolute performance than power hungry processing elements. If you execute the same code on a Xeon or on an Atom, you will probably eventually consume less power when using the Atom, but you also spent much more time waiting for your application to complete. Therefore the performance manager will constantly look for more powerful processing elements to allocate the parallel computation activities. If the two managers are run concurrently on the same computation this will result in one manager destroying the other manager decisions. In turn this represents a complete waste of resources (time and power). ■

In order to try to *orchestrate* different managers in such a way some globally "good" behaviour is achieved aiming at reaching the goals of *all* the managers involved, we can take two different ways:

W1 we use a *super manager*, that is a manager coordinating the decisions of the multiple manager hierarchies used in our application. The situation is the one depicted in

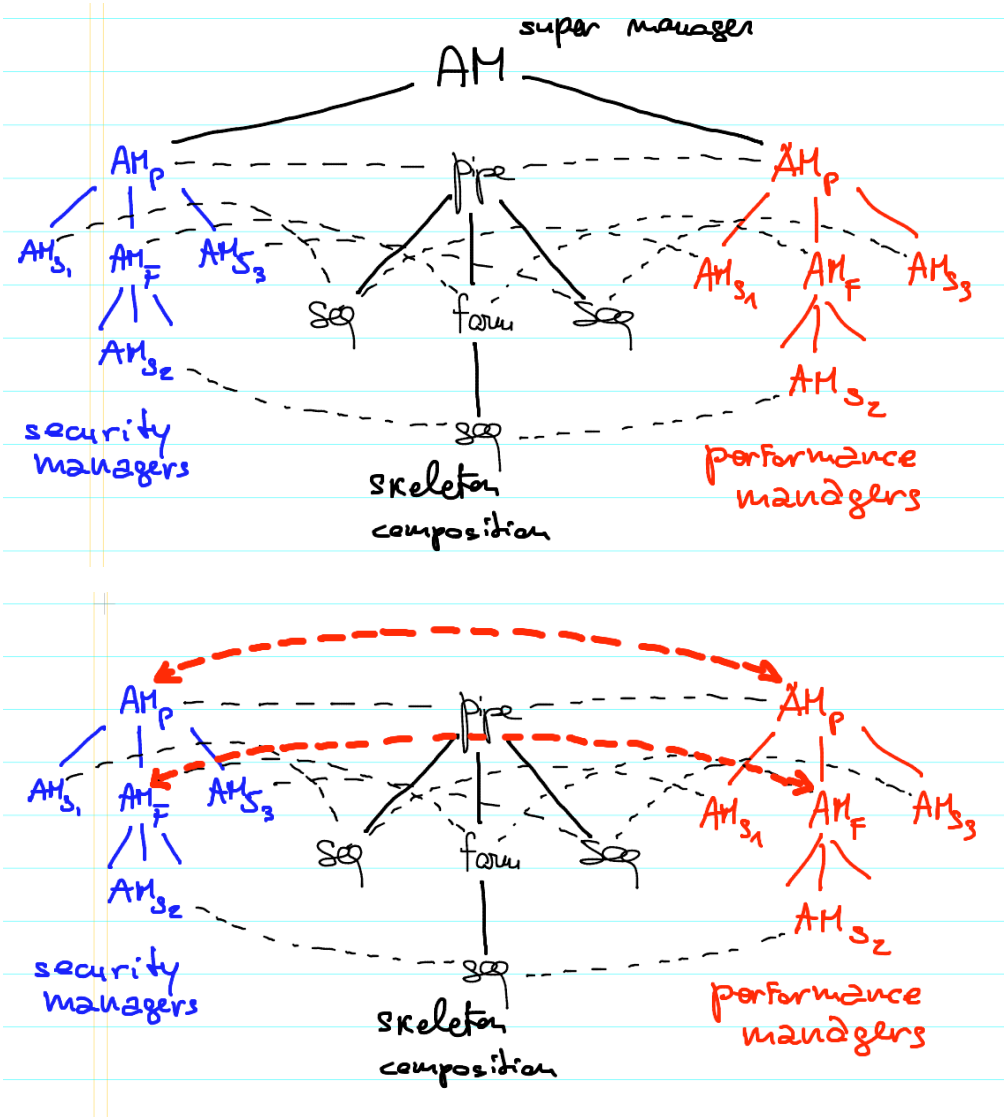


Figure 10.9 Alternative manager orchestration alternatives

Fig. 10.9 (upper part). In this case, the super manager should coordinate adaptation actions of the two manager hierarchies. This means that each manager hierarchy should "ask permission" to the super manager to commit any adaptation action planned. In turn, the super manager may ask consent to allow the commitment of a given planned action to the other manager hierarchy.

W2 we don't use a super manager, but we make managers instances relative to the same parallel pattern to coordinate each other (*layered coordination*). The situation is the one depicted in Fig. 10.9 (lower part). In this case, a manager deciding to take some adaptation action should "ask permission" to the other(s) manager(s) taking care of the same parallel pattern before committing the action execution.

In both cases, therefore, the key point consists in the *establishment of some mutual agreement protocol* in order to avoid that adaptation actions are taken that impair management policies of some of the involved managers (manager hierarchies).

10.5.4 Mutual agreement protocol

The mutual agreement protocol among a set $\mathcal{M}_1, \ldots, \mathcal{M}_m$ of managers should be designed such that:

- no adaptation actions are taken by any $\mathcal{M}_j$ that may impair the policies of another manager $\mathcal{M}_i$
- in case, the manager $\mathcal{M}_i$ whose policies may be impaired by a decision taken by manager $\mathcal{M}_j$ may grant $\mathcal{M}_j$ the permission to execute the planned action provided it is modified to accomplish $\mathcal{M}_i$ policies too.

In case we adopt a layered coordination approach, a simple mutual agreement protocol may therefore be designed as follows:

Decision step Manager $\mathcal{M}_i$ analyzes the current behaviour of the managed parallel pattern and decides an action A should be executed. This decision is taken on the exclusive basis of the policies implemented by manager $\mathcal{M}_i$.

Consensus "ask" step Manager $\mathcal{M}_i$ communicates the decision taken to other managers $\mathcal{M}_j$ ($j \neq i$) taking care of different non functional concerns for the same parallel pattern. The decision should be communicated is such a way the other managers could appreciate its effects. As the only common knowledge between managers is represented by the parallel structure of the entities participating to the parallel pattern computation, the decision may be suitably represented by the *changes in the current parallel activities graph*. Therefore the communication from manager $\mathcal{M}_i$ to the other managers should include information such as

$$\langle A, G, G' \rangle$$

where G represents current parallel activities graph and G' represents the graph eventually resulting from the implementation of action A.

Consensus "answer" step Each manager $\mathcal{M}_j$ receiving the message from manager $\mathcal{M}_I$ analyzes the effects of decision A according to its own management policies and sends an "answer" message to $\mathcal{M}_i$. The message may be one of three different messages:

1. OK
   This means that decision A does not anyway impact the management policies of

$\mathcal{M}_j$. As far as manager $\mathcal{M}_j$ is concerned, action A may be safely committed by manager $\mathcal{M}_i$.

2. NOK
   This means that decision A contradicts some management policy of $\mathcal{M}_j$. As far as manager $\mathcal{M}_j$ is concerned, action A *should not* be taken by manager $\mathcal{M}_i$.
3. *provide*(P)
   This is the most interesting case, actually. As far as manager $\mathcal{M}_j$ is concerned, action A *may be taken* by by manager $\mathcal{M}_i$ provided that the action is taken in such a way property P–of interest of manager $\mathcal{M}_j$–is granted.

Commit step Manager $\mathcal{M}_i$ gathers all the answers from the other managers and then:

- in case the answers are all of type OK, it commits the action, that is it goes on executing the planned action
- in case at least one answer of type NOK has been received, the action is aborted. Possibly, the policies that led to the decision of taking action A are given a lower priority, in such a way the very same decision is not chosen again in the immediate future
- in case some *provide*(P) answers has been received, $\mathcal{M}_i$ tries to modify action A in such a way property P is granted. This produces action A', as a result. If an A' satisfying property P (or the collection of $P_1, \ldots, P_k$ relative to k *provide* messages received) may be computed, it is eventually committed, otherwise the decision is aborted and the relative policy priorities lowered, as in the NOK case.

In case we adopted a super manager approach, instead, a similar protocol may have been implemented in the super manager. In that case, each local manager deciding to implement adaptation action A will communicate the action plan to its parent manager. Eventually, root manager will communicate the decision to the super manager. The super manager will implement the mutual agreement protocol as described above, querying the other top level managers and collecting answers, and eventually will communicate the decisions to the querying top level manager. This, in turn, will propagate the decision to the manager originally raising the request for agreement on action A.

E.G.▷ In order to better understand how this agreement protocol works, consider the following example. Two manager hierarchies take care of the same behavioural skeleton composition implementing a given parallel application. One manager hierarchy takes care of performance and one of security. Performance contracts are such that:

- a service time of at least t_s is required
- both code and data confidentiality and integrity is required

Let us suppose a farm performance manager decides it is under performing with respect to the contract received and therefore a new worker may be added. Therefore it enters the recruiting process. Suppose resource RW_{new} is found. If

$$G = (N, A)$$

with

$$N = \langle E, W_1, \ldots, W_{nw}, C \rangle$$

and

$$A = \langle \langle E, W_1 \rangle, \langle E, W_2 \rangle, \ldots, \langle E, W_{nw}, \rangle, \langle W_1, C \rangle, \langle W_2, C \rangle, \ldots \langle W_{nw}, C \rangle \rangle)$$

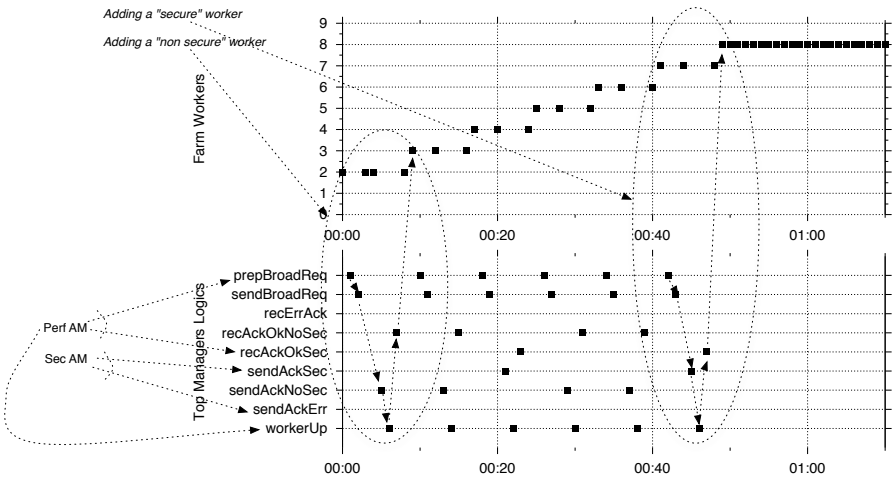


Figure 10.10 Sample trace of mutual agreement protocol operation

was the current parallel activities graph, than the graph resulting from the actual implementation of the planned action will be

$$G' = (A', N')$$

with

$$N' = N \cup \{RW_{new}\}$$

and

$$A' = A \cup \{\langle E, RW_{new}\rangle, \langle RW_{new}, C\rangle\}$$

At this point the farm performance manager may send to all the other farm managers the request

$$\langle addWorker, G, G'\rangle$$

Now three cases may be given:

1. the security manager, according to its policies and node security property knowledge decides it is sage to add the new resource RW_{new}, and therefore answers a OK message to the farm performance manager, or
2. it decides the new resource RW_{new} is "nasty" for same reason and therefore it answers a NOK message to the farm performance manager, or eventually
3. it decides the new resource may be used, provided secure data and code communications are used. In this case it answers a $provide(secureCodeData)$ message to the farm performance manager.

Depending on the kind of answer received, the farm performance manager:

1. commits the action, adding RW_{new} to the farm

2. aborts the action and manages to avoid in the near future to try to recruit again resource RW_{new}
3. manages to modify the action A in such a way *secureCodeData* is ensured. Now action A is the high level action *add worker*. The corresponding plan could be something as

   $$A = \langle deploy, start, link \rangle$$

   where the *deploy* action deploys the worker RTS code to resource NR_{new}, the *start* action starts a worker on that resource and eventually, the *link* action links the new worker to collector and emitter in such a way it an be used for the farm computation. Ensuring *secureCodeData* property, in this context, means different actions, including at least:
   - deploy code that eventually uses secure communication to interact with the emitter and collector processes (e.g. SSL sockets rather than plain TCP/IP ones)
   - use secure mechanisms to deploy the code to the remote resource

■

The mutual agreement protocol just discussed is a possible solution to the coordination of multiple managers in behavioural skeletons. It is relatively simple, and it has been demonstrated to be working with stream parallel behavioural skeleton applications with two managers taking care of performance and security. Fig. 10.10 is relative to the execution of a farm application whose performance contract was to have a constant parallelism degree equal to 8. The application was started with a parallelism degree equal to 2. The farm manager therefore started to plan adding more workers. Some of the resources recruited were judged "secure" by the security manager, and therefore they are added without taking any particular action to ensure security in communications, other were judged "not so secure" and therefore a *provide*(*secureCodeData*) was answered (*recAckOkSec* in the graph) such that the farm manager used secure worker set up (*sendSecAck* in the graph) to add the worker to the farm.

However, the protocol has also some flaws. First of all, the *provide*(P) messages violate somehow the separation of concerns principle. Our performance managed is forced to know about *secure* deployment and communications, which is not in general something he should know of, being a *performance* manager.

Second, in case more that a single *provide*(P) answer is collected the scheduling of the modifications of the planned action may actually impair the feasibility of the modification itself. More in detail, if P_1 and P_2 must be ensured ensuring first P_1 and then P_2 may have different implications and feasibility with respect to ensuring first P_2 and then trying to ensure P_1 on the resulting set of modified actions.

Last but not least, this protocol *de facto* gives priority to negative actions. The NOK messages always lead to the abortion of the planned action. This ensures the achievement of the policies of the "forbidding" manager–the one sending the NOK message–but may be it impairs the achievement of the management policies of the "forbidden" manager–the one receiving the NOK message–which could be in the situation of not having any more alternative actions to take to achieve its goals.

However, this represents the only real agreement protocol studied up to know.

10.5.5 Alternative multi concern management

Actually, multiple concern management has also been studied from a completely different perspective, in parallel programming contexts not using skeletons at all. The approach followed in that case can be summarized as follows.

We consider a number of functionally equivalent application implementations $A_1, \ldots, A_a$. Therefore A_i computes the same results of A_j but the results are computed differently as far as non functional behaviours are concerned.

Then we consider the non functional behaviour of interest $\mathcal{NF}_1, \ldots, \mathcal{NF}_{nf}$ and we assume a function $f_i(p_1, \ldots, p_p)$ estimating "how much" $\mathcal{NF}_i$ is being achieved when application A_k is executed with a given set of parameters $p_1, \ldots, p_p$ (application or target architecture related) is also available. We also assume that a linear combination of f_i exists

$$F(p_1, \ldots, p_p) = \sum_{i=1,a} a_i f_i(p_1, \ldots, p_p)$$

that evaluates the collective "goodness" our set of implementations, that is a weighted sum of the estimated of the goodness of the single non functional concerns achievements on certain application and target architecture parameters.

Eventually, at each adaptation iteration, we evaluate F for each A_j considering the current execution parameters $p_1, \ldots, p_p$ and we chose to adopt for the next step the application whose F gives the "best" value.

E.G.▷ In this case, taking into account our sample

```
pipe(seq(C1),pipe(farm(seq(C2),seq(C3)))
```

application and performance and security non functional concerns, we will have:

- $nf = 2$ and with $\mathcal{NF}_1 = performance$ and $\mathcal{NF}_2 = security$
- the set of equivalent applications including i) different versions of the original application differing in the parallelism degrees relative to the `farm`, ii) different versions of the original application modifies in such a way `seq(C1)` is rewritten as `farm(seq(C1))` differing in the $\langle nw_1, nw_2 \rangle$ parallelism degrees considered for the first and second stage farms, iii) different versions of this last version of the application with `seq(C3)` rewritten as `farm(seq(C3))` differing in the parallelism degrees $\langle nw_1, nw_3, nw_3 \rangle$, iv) different versions of a `farm(pipe(seq(C1),pipe(seq(c2),seq(C3)))` differing in the external farm parallelism degree, v) ...
- function f_1 evaluating performance of a given application. The function evaluates the application with respect to its parallel activities graph mapped on real resources. The function returns, as an example, the service time of the applications.
- function f_2 evaluating security of a given application. The function evaluates the application with respect to its parallel activities graph mapped on real resources. The function returns 1 in case of achievement of the security in the current application, or 0 otherwise.
- two weight parameters a_1 and a_2. The choice of these parameters is obviously critic. We have to consider that smaller service times are the better ones and that security is a "binary" valued function. As an example, we can pick $a_1 = 1$ and $a_2 = -k$ being k a value larger than the maximum service time possible.
- eventually, we can look for the smaller F among all the applications evaluated on the currently available parameters. In case A_j is the application delivering the best F and A_i is the current one, we should manage to run an adaptation step rewriting A_i as A_j. This may simply mean that we have to increase/decrease the parallelism degree of A_i, if A_i and A_j share the same skeleton tree but differ in the parallelism degree. It can also require a complete rewriting of the application (e.g. to change the application parallel patter from a pipe of farms to a farm of pipes). In this case the adaptation phase will obviously take longer and possibly require an application checkpoint-stop-transform-restart cycle.

■

It is evident that:

- the critical step in this case is the design of F: the choice of proper a_i as well as the choice of the way to evaluate "best" results (minimum, maximum)

- solutions like the one discussed above to handle "binary" concerns may be unfeasible when multiple binary concerns are taken into account, and in general deciding weights is again a critic issue

- at each adaptation step we must consider all the alternative possibilities. This space is huge. When adopting the mutual agreement protocol discussed in Sec. 10.5.4 instead, only the currently proposed application rewriting is to be considered and evaluated by the other managers.

All this points make this alternative multi concern management strategy slightly less effective than the one discussed in Sec. 10.5.4.

10.6 SKELETON FRAMEWORK EXTENDIBILITY

One of the major criticisms made to algorithmic skeleton frameworks was related to their *poor or absent* extendibility. Traditional algorithmic skeleton frameworks, in fact, do not allow users to intervene on the skeleton set neither to add new skeleton nor even to slightly modify the existing skeletons.

As a result, two kind of situations may arise:

- either the programmer succeeds modelling his/her application with (a composition of) the existing skeletons, or

- there is no way to write the application using the skeleton framework.

This is very bad, actually. The set of skeletons implemented in an algorithmic skeleton programming framework model most common and used parallelism exploitation patterns. However it is quite frequent that in order to achieve the very last bit of performance, programmers usually "fine tune" these patterns in such a way the computation at hand can be perfectly a very efficiently modelled. A programmer that could not use its favorite, super-optimized parallelism exploitation pattern would therefore dislike algorithmic skeleton frameworks.

Cole recognized this problem as one of the most important aspect impairing diffusion of algorithmic skeletons in the parallel programming community. In his "manifesto" paper [38], in fact, he lists four principles that must be ensured in order to allow widespread diffusion of the algorithmic skeleton concept (see Sec. 7.1). One of these principles exactly states that we must

> ***Integrate ad-hoc parallelism***. *Many parallel applications are not obviously expressible as instances of skeletons. Some have phases which require the use of less structured interaction primitives. For example, Cannon's well-known matrix multiplication algorithm invokes an initial step in which matrices are skewed across processes in a manner which is not efficiently expressible in many skeletal systems. Other applications have conceptually layered parallelism, in which skeletal behaviour at one layer controls the invocation of operations involving ad-hoc parallelism within. It is unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well-defined way.*

Now in order to "construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well-defined way" we may follow different ways. Each different approach must taken into account the way skeletons are provided to the user/programmer that is both the kind of skeletons provided *and* the implementation model chosen to support these skeletons.

We distinguish two alternative approaches, that are described in the following sections:

1. skeleton set extension in template based frameworks

2. skeleton set extension providing controlled user access to intermediate implementation layer

The two approaches basically differ in the level of access granted to the user/programmer to the internals of the skeleton framework.

10.6.1 Skeleton set extension in template based frameworks

When implementing a template based skeleton framework we basically have to implement a set of distinct tools:

- a compiler (or a library), parsing source code (implementing skeleton declaration calls) and building the abstract skeleton tree
- a template library (see Sec. 4.2.1)
- a compiler assigning templates to the skeletons in the skeleton tree and devising proper parameters to instantiate all the templates assigned (e.g. parallelism degree)
- a tool compiling the annotated skeleton tree and deploying the resulting object code onto the target architecture for execution
- a run time system to support execution on the target architecture.

In order to support skeleton set expandability we should guarantee the programmer with:

- the possibility to include new skeletons in the skeleton set
- the possibility to provide implementation templates for the new skeletons

Both points are quite complex to provide. The former requires full access to parsing facilities of the compiler or, in case of implementation of the skeleton framework through a library, the possibility to enter brand new entries in the skeleton library. In both cases, full access to internal skeleton tree data structures must be granted. The latter requires the possibility to access all the mechanisms of the target abstract machine used to implement the skeletons. If this target machine is C++ with POSIX threads and TCP/IP sockets, then the programmer should use these mechanisms to provide the new template(s) corresponding to the new skeleton he wants to add to the system. Moreover, as the new template should be nestable in other templates, full compliance to some general template composition model should be ensured by the new template. All in all, this is a quite huge effort to add a new skeleton into the system.

As far as we know, no algorithmic skeleton framework provides support to extension of the skeleton–and as a consequence, of the template–set as discussed above. Indeed, most algorithmic skeleton frameworks are template based and open source. Therefore nothing prohibits to dig the skeleton framework source code and to add new skeletons and templates. But the no framework provides support for this activity and actually internals of the framework implementation are in most cases completely undocumented. This means the programmer wishing to extend the skeleton set must first reverse engineer the framework implementation. This is probably much more than the work required to write a brand new, non skeleton based implementation of the particular parallelism exploitation pattern needed.

10.6.2 Skeleton set extension through intermediate implementation layer access

An alternative way of providing support for skeleton set extension may be designed when non template based implementations are used. If a macro data flow based implementation

```
Skeleton inc1 = new Inc();
Dest d = new Dest(0, 2 ,Mdfi.NoGraphId);
Dest[] dests = new Dest[1];
dests[0] = d;
Mdfi i1 = new Mdfi(manager,1,inc1,1,1,dests);
Skeleton sq1 = new Square();
Dest d1 = new Dest(0,Mdfi.NoInstrId, Mdfi.NoGraphId);
Dest[] dests1 = new Dest[1];
dests1[0] = d1;
Mdfi i2 = new Mdfi(manager,2,sq1,1,1,dests1);
MdfGraph graph = new MdfGraph();
graph.addInstruction(i1);
graph.addInstruction(i2);
ParCompute userDefMDFg = new ParCompute(graph);
```

Figure 10.11 Custom/user-defined skeleton declaration.

is used we can provide the user/programmer with a controlled access to the intermediate implementation layer–the macro data flow graph language. We must expose the user/programmer with the definition of "well formed MDFG", i.e. of a MDFG that may be used in a skeleton nesting. As discussed in Sec. 4.2.2 a macro data flow graph representing a skeleton is basically any MDFG with a single input token and a single output token. Then the user/programmer should be given a "language" suitable to express these well formed MDFG and a way to name the new MDFG modelling the particular parallelism exploitation pattern needed. The Muskel skeleton library [14] provides such a possibility, and we will described how this is supported in detail in the next section. For the moment being, we want to point out that the process providing skeleton set expandability is much simpler than the one discussed with template based implementations. All what the user needs to know is how to "describe" a well formed MDFG and how to name it in standard skeleton compositions. Then the MDF distributed interpreter takes care of executing the user supplied MDFG with the same efficiency achieved in the execution of MDFGs deriving from the compilation of standard, predefined skeletons.

controllare

10.6.3 User-defined skeletons in `muskel`

In order to introduce completely new parallelism exploitation patterns, `muskel` provides programmers with mechanisms that can be used to design arbitrary macro data flow graphs. A macro data flow graph can be defined creating some `Mdfi` (macro data flow instruction) objects and connecting them in a `MdfGraph` object.

For example, the code in Fig. 10.11 is the one needed to program a data flow graph with two instructions. The first computes the `inc1` `compute` method on its input token and delivers the result to the second instruction. The second computes the `sq1` `compute` method on its input token and delivers the result to a generic "next" instruction (this is modelled by giving the destination token tag a `Mdfi.NoInstrId` tag). The `Dest` type in the code represents the destination of output tokens as triples containing the graph identifier, the instruction identifier and the destination input token targeted in this instruction. Macro data flow instructions are built by specifying the manager they refer to, their identifier, the code executed (must be a `Skeleton` object) the number of input and output tokens and a vector with a destination for each of the output tokens.

We do not present all the details of arbitrary macro data flow graph construction here (a complete description is provided with the `muskel` documentation). The example is just to give the flavor of the tools provided in the `muskel` environment. Bear in mind that the simple macro data flow graph of Fig. 10.11 is actually the same macro data flow graph obtained by compiling a primitive `muskel` skeleton call such as: `Skeleton main = new Pipeline(new Inc(), new S`
More complex user-defined macro data flow graphs may include instructions delivering to-

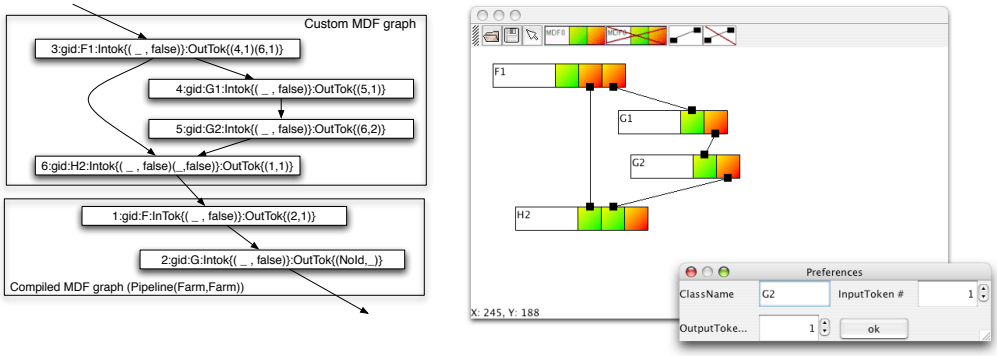


Figure 10.12 Mixed sample macro data flow graph (left): the upper part comes from a user-defined macro data flow graph (it cannot be derived using primitive `muskel` skeletons) and the lower part is actually coming from a three stage pipeline with two sequential stages (the second and the third one) and a parallel first stage (the user-defined one). GUI tool designing the upper graph (right).

kens to an arbitrary number of other instructions, as well as instructions gathering input tokens from several distinct other instructions. In general, the mechanisms of `muskel` permit the definition of any kind of graph with macro data flow instructions computing sequential (side effect free) code wrapped in a `Skeleton` class. Any parallel algorithm that can be modeled with a data flow graph can therefore be expressed in `muskel`[162]. Non deterministic MDFi are not yet supported (e.g. one that merges input tokens from two distinct sources) although the firing mechanism in the interpreter can be easily adapted to support this kind of macro data flow instructions. Therefore, new skeletons added through the macro data flow mechanism always model pure functions. skeletons in `muskel` programs gives the programmer the possibility to develop the same parallel programs he could develop using more classic programming environments (e.g. MPI), but those explicitly using data

`MdfGraph` objects are used to create new `ParCompute` objects. `ParCompute` objects can be used in any place where a `Skeleton` object is used. Therefore, user-defined parallelism exploitation patterns can be used as pipeline stages or as farm workers, for instance. The only limitation on the graphs that can be used in a `ParCompute` object consists in requiring that the graph has a unique input token and a unique output token.

When executing programs with user-defined parallelism exploitation patterns the process of compiling skeleton code to macro data flow graphs is slightly modified. When an original `muskel` skeleton is compiled, the standard MDF compilation (see Sec. 4.2.2) process is applied. When a user-defined skeleton is compiled, the associated macro data flow graph is directly taken from the `ParCompute` instance variables where the graph supplied by the user is maintained. Such a graph is linked to the rest of the graph according to the rules appropriate to the skeleton where the user-defined skeleton appears.

To show how the whole process works, let us suppose we want to pre-process each input task in such a way that for each task t_i a new task

$$t'_i = h_1(f_1(t_i), g_2(g_1(f_1(t_i))))$$

is produced. This computation cannot be programmed using the stream parallel skeletons currently provided by `muskel`. In particular, current pre-defined skeletons in `muskel` allow only processing of one input to produce one output, and therefore there is no way

[162] Note, however, that common, well know parallel application skeletons are already modelled by pre-defined `muskel Skeletons`.

to implement the graph described here. In this case we wish to process the intermediate results through a two-stage pipeline to produce the final result. To do this the programmer can set up a new graph using code similar to the one shown in Fig. 10.11 and then use that new `ParCompute` object as the first stage of a two-stage pipeline whose second stage happens to be the post processing two-stage pipeline. When compiling the whole program, the outer pipeline is compiled first. As the first stage is a user-defined skeleton, its macro data flow graph is directly taken from the user-supplied one. The second stage is compiled according to the standard (recursive) procedure (such as the one described in Sec. 4.2.2) and eventually the (unique) last instruction of the first graph is modified in such a way that it sends its only output token to the first instruction in the second stage graph. The resulting graph is outlined in Fig. 10.12 (left).

Making good use of the mechanisms allowing definition of new data flow graphs, the programmer can arrange to express computations with arbitrary mixes of user-defined data flow graphs and graphs coming from the compilation of structured, stream parallel skeleton computations. The execution of the resulting data flow graph is supported by the `muskel` distributed data flow interpreter in the same way as the execution of any other data flow graph derived from the compilation of a skeleton program. At the moment the `muskel` prototype allows user-defined skeletons to be used as parameters of primitive `muskel` skeletons, but not vice versa. We are currently working to extend `muskel` to alow the latter. There is no conceptually difficult step behind. In order to allow primitive `muskel` skeleton usage as code to be executed in an instruction of a user-defined macro data flow graph it is sufficient to compile "on the fly" the primitive skeleton and include in the user-defined macro data flow graph the result (i.e. the macro data flow graph) of this compilation.

While the facility to include user-defined skeletons provides substantial flexibility, we recognize that the current way of expressing new macro data flow graphs is error prone and not very practical. Therefore we have designed a graphic tool that allows users to design their macro data flow graphs and then compile them to actual Java code as required by `muskel` and shown above. Fig. 10.12 (right) shows the interface presented to the user. In this case, the user is defining the upper part of the graph in the left part of the same Figure. It is worth pointing out that all that is needed in this case is to connect output and input token boxes appropriately, and to configure each MDFi with the name of the sequential `Skeleton` used. The smaller window on the right lower corner is the one used to configure each node in the graph (that is, each MDFi). This GUI tool produces an XML representation of the graph. Then, another Java tool produces the correct `muskel` code implementing the macro data flow graph as a `muskel ParCompute` skeleton. As a result, users are allowed to extend, if required, the skeleton set by just interacting with the GUI tool and "compiling" the graphic MDF graph to `muskel` code by clicking on one of the buttons in the top toolbar.

As a final example, consider the code of Fig. 10.13. This code outlines how a new `Map2` skeleton, performing in parallel the same computation on all the portions of an input vector, can be defined and used. It is worth pointing out how user-defined skeletons, once properly debugged and fine-tuned, can simply be incorporated in the `muskel` skeleton library and used seamlessly, as the primitive `muskel` ones, but for the fact that (as shown in the code) the constructor needs the manager as a parameter. This is needed so as to be able to link together the macro data flow graphs generated by the compiler and those supplied by the user. data flow graph creation to the moment the graph needs to be instantiated after the arrival of a new task to compute, as at that time all the information necessary to perform graph "conjunction" is available. It is worth noting that skeletons such as a general form of `Map` are usually provided in the fixed skeleton set of any skeleton system and users usually do not need to implement them. However, as `muskel` is an experimental skeleton system, we concentrate the implementation efforts on features such as the autonomic managers, portability, security and expandability rather than providing a complete skeleton set. As

```
public class Map2 extends ParCompute {

    public Map2(Skeleton f, Manager manager) {
        super(null);
        // first build the empty graph
        program = new MdfGraph();
        // build the emitter instruction
        Dest [] dds1 = new Dest[2];
        dds1[0]=new Dest(0,2);
        dds1[1]=new Dest(0,3);
        Mdfi emitter =
            new Mdfi(manager, 1,
            new MapEmitter(2), 1, 2, dds1);
        // add it to the graph
        program.addInstruction(emitter);
        // build first half map Skeleton node
        Dest [] dds2 = new Dest[1];
        dds2[0] = new Dest(0,4);
        Mdfi if1 = new Mdfi(manager,2, f, 1, 1, dds2);
        program.addInstruction(if1);
        // build second half map Skeleton node
        Dest []dds3 = new Dest[1];
        dds3[0] = new Dest(1,4);
        Mdfi if2 = new Mdfi(manager,3, f, 1, 1, dds3);
        program.addInstruction(if2);
        Dest[] ddslast = new Dest[1];
        ddslast[0] = new Dest(0,Mdfi.NoInstrId);
        Mdfi collector = new Mdfi(manager,4,new
                    MapCollector(), 2, 1, ddslast);
        program.addInstruction(collector);
        return;
    }
    ...
}
```

```
public class SampleMap {
    public static void main(String[] args) {
        Manager manager =
            new Manager();
        Skeleton worker = new Fdouble();
        Skeleton main =
            new Map2(worker,manager);

        InputManager inManager =
            new DoubleVectIM(10,4);
        OutputManager outManager =
            new DoubleVectOM();

        ParDegree contract =
            new ParDegree(10);
        manager.setInputManager(inManager);
        manager.setOutputManager(outManager);
        manager.setContract(contract);
        manager.setProgram(main);

        manager.compute();
    }
}
```

Figure 10.13 Introducing a new, user-defined skeleton: a map working on vectors and with a fixed, user-defined parallelism degree.

a consequence, `muskel` has no predefined map skeleton and the example of user defined skeleton just presented suitably illustrates the methodology used to expand the "temporary" restricted skeleton set of the current version of `muskel` depending on the user needs. The `Map2` code shown here implements a "fixed parallelism degree" *map*, that is the number of "workers" used to compute in parallel the skeleton does not depend on the size of the input data. It is representative of a more general `Map`skeleton taking a parameter specifying the number of workers to be used. However, in order to support the implementation of a map skeleton with the number of workers defined as a function of the input data, some kind of support for the dynamic generation of macro data flow graphs is needed, which is not present in the current `muskel` prototype. provide users a more handy way to build the user-defined data flow graphs. With the tool, graphs can be built on the screen and all the work needed to arrange `Dest` code in the instructions, which is definitely the more cumbersome and error prone stuff related to graph designing, can simply be generated with a few mouse clicks connecting with arrows source and destination tokens.

CHAPTER 11

STRUCTURED SKELETON/PATTERN DESIGN

After having considered all the different aspects related to structured parallel programming, we consider an alternative approach to the design of algorithmic skeletons/parallel design patterns. In Chap. 8, we considered different aspects related to the design of efficient implementation templates suitable to implement different kind of skeletons on different kind of target architectures. The general methodology suggested was

- to start considering alternative template implementations,
- to devise proper performance models,
- to evaluate the different alternative, in order to individuate the "better" one, and eventually
- to proceed in the optimized implementation of the "best" template schema.

We move a step forward here, by investigating the possibility to to adopt a more structured and formal approach in part of the steps mentioned above. In particular, we propose to design skeleton implementations by composing parallel building blocks from a limited (RISC-like) set of parallel building blocks, to proceed optimizing the parallel building blocks expressions deriving from building block compositions applying formally proven rewritings and eventually by implementing the resulting, optimized parallel building block expressions exploiting the better mechanisms available to implement the building block on the target architecture at hand.

11.1 RISC-pb^{2}l

When implementing a set of orchestrated concurrent activities, we have to cope with different aspects, namely: the proper re-use of existing code (either sequential or parallel),

the set up of the concurrent activities and, last but not least, the communication and synchronization among the different concurrent activities.

In this chapter we discuss the design, implementation and usage of a parallel building block library (RISC-pb^{2}l), that is a collection building blocks suitable to support the implementation of any parallel pattern/algorithmic skeleton. Being the three kind of concerns listed above the ones we have to deal with, the parallel building block library may be naturally designed as the union of three different kinds of building blocks:

Wrappers Those building blocks used to wrap sequential or parallel portions of code, in such a way the portions of code may be suitably used as parameters of the templates/skeletons.

Functionals Those building blocks implementing the collection of concurrent activities *computing* the results out of the input data.

Combinators Those building blocks implementing *communications* and *synchronizations* among the computing concurrent activities.

In the following sections, we first outline the different building blocks included in RISC-pb^{2}l.

11.1.1 Wrappers

Wrapper building blocks are used to encapsulate portions of existing, debugged and optimized code in such a way that code may be reused within a parallel computation implemented using RISC-pb^{2}l. The main aim is therefore to wrap portion of code in such a way they may be used as *functions* with clearly identified input and output parameters.

It is expected that the wrapped code implements *stateless* computations. This is necessary to allow wrapped code portions to be replicated or migrated across the different processing elements used to run the parallel application. If the wrapped portions of code were not stateless the hidden state may impair the possibility to replicate or move the concurrent activities running the wrapped code.

RISC-pb^{2}l includes two distinct wrapper building blocks:

Sequential code wrapper This is a wrapper implementing a "function" f out of a sequential portion of code *code* such that the function f has type $f : \alpha \rightarrow \beta$. In case the wrapped code accepts more inputs (deliver different outputs) α (β) may be a structured type.

E.G.▷ Consider a code computing the Inner Product of two vectors. The input parameters will be the two vectors and the output parameter will be the inner product value. Therefore we will have

$$\alpha = \texttt{struct}\{float[], float[]\}$$

and

$$\beta = float$$

■

RISC-pb^{2}l sequential wrappers are denoted by

$$((f))$$

were f is the function implemented by the wrapped code.

Parallel code wrapper This is a wrapper similar to the sequential wrapper, but for the fact that the wrapped code is parallel. The parallel code should be stateless, as it is

required in case of sequential code wrapper. Furthermore, the wrapped parallel code should not interfere with the management of concurrent activities proper of RISC-pb^{2}l building blocks. RISC-pb^{2}l parallel wrappers are denoted by

$$(\!| f |\!)$$

were f is the function implemented by the wrapped code.

All wrappers eventually implement activities that take a single (possibily structured) input to return a single (possibily) structured output.

11.1.2 Functionals

Functional bulding blocks are used to implement actual parallel computations. In principle, sequential computations are implemented executing the code exported through wrappers, while parallel computations are orchestrated using functionals.

Different kinds of functionals are included in RISC-pb^{2}l:

Parallel string of concurrent activities This is the functional where a number of concurrent activities compute in parallel. In case the concurrent activities are all of the same kind, we refer to the parallel string of concurrent activities as "parallel" building block and we denote this parallel building block as

$$[|\ \Delta\ |]_n$$

where Δ is the single activity (either sequential or parallel) which is replicated n times in the parallel building block.

E.G.▷ Assume we have some code wrapped into a sequential building block $((filter_x))$ and computing a filter $filter_x$ over an input image to produce an output image. Then

$$[|\ ((filter_x))\ |]_k$$

represents a string of k different concurrent activities each running the wrapped filter code. It's worth noting (see Sec. 11.1.1 above) that in case the $filter_x$ code was stateful the replication of the $((filter_x))$ building blocks could not be arbitrarily implemented as replication of the stateful computation may lead to incorrect results. —— ■

In case the concurrent activities parameter of the parallel string are different, which is denoted in RISC-pb^{2}l as

$$[|\ \Delta_1, \ldots, \Delta_n\ |]$$

the parallel string building block is referred to as "misd" parallel command. In both cases, the parallel or misd building block accepts n inputs, processes each input with one of the n component building blocks and returns n results. The kind of parallelism exploited by this building blocks is the execution of n *independent* parallel activities.

Pipeline This is the functional orchestrating the computation of other building blocks as they were stages of a single computation. The pipeline building block is denoted in RISC-pb^{2}l as

$$\Delta_1 \bullet \Delta_2 \bullet \ldots \bullet \Delta_n$$

The input data is passed to Δ_1, each result computed by Δ_i on the input got from Δ_{i-1} is passed to Δ_{i+1} and eventually the result of the pipeline is produced by the last stage Δ_n.

E.G.▷ In case we want to apply two filters to an image, and we have available the sequential code implementing each one of the filters, we can use a pipeline such as

$$((filter_a)) \bullet ((filter_b))$$

■

The kind of parallelism exploited by the pipeline building block is a set of n *dependent* parallel activities where each parallel activity waits for the input from its "previous" parallel activity and delivers output to its "next" parallel activity, apart from the first and the last activity that process input data and output final result, respectively.

Reduce This is the functional computing a single result out of a number of input outputs. It uses a single component building block, which is assumed to implement a function f accepting k inputs and producing a single result. A tree is built with nodes computing f such that the inputs represents the leaves and the root of the tree delivers the final result. In principle, there are no particular constrains on the function f. However, in case the function is associative and commutative, different kind of more aggressive optimizations may be performed while implementing the reduce building block. The tree of the reduce building block may be built using an arbitrary number of levels l with $l \geq 1$. As a special case, trees with $l = 1$ may be used. In that case, the building block acts as "gather" (that is an *n-to-one*) communication, with some computation (the function f) applied on all the input items. The reduce building block is denoted by

$$(_f \triangleright)$$

The amount of inputs in the reduce node is actually defined by the features of the building block feeding the reduce.

E.G.▷ Assume we want to compute the sum of all the floating points produced by a parallel building block $[|\ \Delta\ |]_{nw}$. We may compose the parallel building block with a reduce in a pipeline as follows:

$$[|\ \Delta\ |]_{nw} \bullet (_+ \triangleright)$$

In this case, the number of levels in the reduce three will be $log_2(nw)$. In case we have available a function *sum* computing the sum of an arbitrary number of floats, we can use a

$$[|\ \Delta\ |]_{nw} \bullet (_{sum} \triangleright)$$

and in this case the reduce tree consists in a single level. ■

Spread The spread building block works as a "reverse" reduce. It computes a number of outputs out of a single input. It uses a single component building block, which is assumed to compute a function f computing k output items out of a single input item. A tree is built using this component as node, with l levels ($l \geq 1$). As a special case, the tree may consists in a single level. In that case, it may work like a collective operation (a broadcast, a scatter or a unicast) depending on the semantics of the function f. The number of levels in the tree is determined by the arity of the function f and by the inputs required by the following building block. If the spread is used to feed a parallel building block with nw workers, such as in

$$(_f \triangleleft) \bullet [|\ \Delta\ |]_{nw}$$

then the spread tree will be build in such a way the number of "leave" nodes is nw/k.

E.G.▷ Assume we want to compute a stateful function on all the items appearing onto an input stream. Assume that the function is computed in parallel using a string of workers $[|\ ((fun))\ |]_n$ where worker i computes the function over all the input elements x such as for some function $hash$ $hash(x) = i$. To schedule the input items to the proper workers, we can use a spread building block where the function f is defined as follows: for each input x, compute $hash(x) = i$ and direct x (unchanged) to the i-th output arc. In this case the spread tree has just one level, of course, as the function has output arity equal to n. The overall computation may be therefore expressed as a

$$(_f\triangleleft) \bullet [|\ ((fun))\ |]_n$$

■

11.1.3 Combinators

Combinator building blocks are used to route data in the places were they have to be computed by functional building blocks. In particular, having defined functionals including the parallel string of workers building block, the combinator building block set includes components routing a single input to n outputs and vice-versa. The combinator set also include a building block routing back results to inputs, such that recursive computations may be easily modelled. The RISC-pb^{2}l combinator building block set therefore includes:

One-to-N This combinator has one input and n outputs. It takes a parameter specifying the kind of *policy* P used to move the input data to the outputs. It is represented as

$$\triangleleft_P$$

and P may be:

scatter meaning that the input data item is assumed to be a collection (e.g. a vector), it is split in partitions and each partition is delivered to one of the inputs.

broadcast meaning that each input data item is copied to all the outputs, and

unicast(Pol) meaning that each input data item is directed to one and only one of the outputs, according to the policy Pol. Pol may be $Auto$, RR or $userdef(f)$. In the first case, the input data is directed to the output connecting the building block to an *idle* resource[163]. In case of a RR policy, a round robin policy is applied. In case of a $userdef(f)$, for each input x $i = f(x)$ is computed and the input is directed to output $i\%n$ (n being the number of outputs).

As for the case of the spread building block, the number of outputs is determined by the number of inputs of the building block following the one-to-n building block.

N-to-one This combinator is used to route inputs coming from n sources to just a single output. It takes as a parameter a routing policy P and it is represented as

$$\triangleright_P$$

P may be:

gather in this case any item appearing on one of the inputs is immediately routed to the output

gatherall in this case n items (one per input) are awaited, and when all these items are available a collection (e.g. a vector) hosting all the data is delivered on the output.

[163] i.e. to the output were a "task request" message has been received

reduce(op) in this case *op* is a binary, associative and commutative operator and the $\triangleright_{reduce(op)}$ behaves as a $\triangleright_{gatherall}$ but for the fact that the reduce with operator *op* is computed on the resulting collection and the reduce value is delivered in output instead of the collection.

It is worth noting that both the gatherall and the reduce cases act as a barrier with respect to the computations performed by the previous building block.

feedback This combinator is used to route back outputs to inputs. It takes as a parameter a boolean function c and a building block Δ and it is represented as:

$$\overleftarrow{(\Delta)}_c$$

Any input is delivered to Δ. Any output y computed by Δ is used to compute $c(y)$: in case the result is true, y is routed back and becomes another input for Δ, otherwise it is delivered as one of the feedback component outputs.

E.G.▷ In case we want to program a map, we can use the RISC-pb^{2}l expression

$$\triangleleft_{scatter} \bullet [|\ ((())f)\ |]_n \triangleright_{gatherall}$$

In this case, the scatter will partition the input collection in as many pieces as the number of the workers in the parallel building block and deliver one partition per worker. The gatherall will wait for all the workers to have completed their work on the assigned partition and then collect all the results and deliver the collection of the results. ——————— ■

11.1.4 Legal compositions of RISC-pb^{2}l components

We already used expression defined as compositions of building blocks while explaining the building block informal semantics in the previous section. We consider now the "legal" compositions, that is we defined the grammar generating the legal expressions of building blocks.

First of all we classify the building block in terms of their input and output *arity*, defining the following classes of building blocks:

$$\begin{array}{lcl}
\Delta^n & ::= & [|\ \Delta\ |]_n \quad | \quad [|\ \Delta_1, \ldots, \Delta_n\ |] \quad | \quad \overleftarrow{(\Delta^n)}_{cond} \quad | \quad \Delta^n \bullet \Delta^n \\
\Delta^{1n} & ::= & \triangleleft_{Pol} \quad | \quad (_f\triangleleft) \\
\Delta^{n1} & ::= & \triangleright_{Pol} \quad | \quad (_g\triangleright)
\end{array}$$

The superscript define input and output arity. In case of Δ^n both input and output arity are equal to n. In case of a superscript such as nm, n represents the input arity and m represents the output arity.

The first class Δ^n represents the class of building blocks with n inputs and n outputs. $[|\ \Delta\ |]_n$ and $[|\ \Delta_1, \ldots, \Delta_n\ |]$ obviously belong to the class. The last two terms are worth more words. The $\overleftarrow{(\Delta^n)}_c$ wraps all the outputs of the inner Δ^n component to the (corresponding) inputs of the same component (n distinct feedback channels, in this case). This means the condition c is evaluated *in parallel* over all the outputs of Δ^n. The $\Delta^n \bullet \Delta^n$ pipeline actually connects each one of the n outputs of the first component directly to the corresponding input of the second component.

Once this classes have been defined, the grammar expressing the legal RISC-pb^{2}l expression is given by

$$\Delta \quad ::= \quad ((code)) \quad | \quad (|\ code\ |) \quad | \quad \Delta \bullet \Delta \quad | \quad \overleftarrow{(\Delta)}_{cond} \quad | \quad \Delta^{1n} \bullet \Delta^n \bullet \Delta^{n1}$$

Therefore

$$(_f\triangleleft) \bullet [|\ ((f))\ |]_n$$

is a legal expression while

$$(_f\triangleleft) \bullet \triangleleft_{Scatter}$$

is not. In particular, the second expression does not respect input-output arities as it requires to address n outputs from the spread building block to the single input of the scatter connector.

11.2 IMPLEMENTING CLASSICAL SKELETONS/PARALLEL DESIGN PATTERNS

In this Section, we show how classical parallel patterns may be easily modelled using RISC-pb^{2}l component expressions.

11.2.1 Farm

A classic farm operating over streams is modelled by

$$Farm(\Delta) = \triangleleft_{unicast(auto)} \bullet [|\ \Delta\ |]_{n_w} \bullet \triangleright_{Gather}$$

In this case, the first scatter building block just delivers any of the items appearing on single input to one of[164] the workers that declared to be available[165] to process a new input item. Each worker in the parallel string of workers $[|\ \Delta\ |]_{n_w}$ processes all the input items delivering the corresponding results to the gather building block. The gather building block, in turn, collects inputs from any of its input channels and delivers the inputs (with no reordering) to the output stream.

11.2.2 Map

A map processing all the items of a collection by means of the function implemented by some sequential code f is modelled by

$$Map(f) = \triangleleft_{scatter} \bullet [|\ ((f))\ |]_{n_w} \bullet \triangleright_{Gatherall}$$

In this case, the first scatter just splits the collection in input into a set of n_w partitions, delivers each partition to the workers. The workers compute f over all the items of the partition and deliver the result to the gatherall, which implements a barrier: it awaits for one item from each of the input channels and delivers as a result the collection of the received items onto the output channel.

11.2.3 Divide and conquer

A divide and conquer may be modelled by a three stage pipeline where:

- the first stage, proceeds computing the divide phase. Each time a task has to be divided, it is associated a unique *id*, used to identify all the generated subtasks
- the second stage computes the base cases

[164] because of the *unicast* policy
[165] because of the *auto* specialization of the *unicast* policy

- the third stage directs already computed partial results to workers reconstructing other partial results (or the final result) by applying the conquer function.

$$D\&C(\Delta_{div}, \Delta_{isBaseCase}, \Delta_{baseCase}, \Delta_{conquer}) = \overbrace{(\lhd_{unicast(auto)} \bullet [|\ \Delta_{div}\ |]_{n_w} \bullet \rhd_{gather})}^{not(\Delta_{isBaseCase})} \bullet [|\ \Delta_{baseCase}\ |]_{n'_w} \bullet \overbrace{((hash\lhd) \bullet [|\ \Delta_{conquer}\ |]_{n''_w} \bullet \rhd_{gather})}^{not(lastTask)}$$

The interesting part here is the way the partial results are conquered. In fact, the *hash* function in the spread of the third stage is in charge of directing all the items with the same tag (unique *id*) to the same worker. The worker will keep storing the items with the same tag up to the point it gets the last one: at this point it may compute the conquer and deliver the result.

11.3 RISC-pb²l REWRITING RULES

RISC-pb²l expressions may be rewritten in such a way different compositions of RISC-pb²l building blocks may be used to model (and therefore implement) the same parallel pattern.

The following rules represent a small subset of the rewriting rules that can be demonstrated correct over RISC-pb²l expressions:

R.1	$[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n$	$\equiv$	$[|\ \Delta_1 \bullet \Delta_2\ |]_n$
R.2	$[|\ \Delta_1\ |]_n \rhd_{gather} \bullet \lhd_{unicast(\ldots)} [|\ \Delta_2\ |]_n$	$\equiv$	$[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n$
R.3	$[|\ \Delta_1\ |]_n \rhd_{gatherall} \bullet \lhd_{scatter} [|\ \Delta_2\ |]_n$	$\equiv$	$[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n$
R.4	$\overbrace{(\lhd_{scatter} \bullet [|\ \Delta_1\ |]_n \bullet \rhd_{gather})}^{cond}$	$\equiv$	$\overbrace{([|\ \Delta_1\ |])}^{cond}$
R.5	$((f)) \bullet ((g))$	$\equiv$	$((f \circ g))$

Each one of the rules may be informally verified by comparing the outputs produced by the left and right hand side expressions.

E.G.▷ Consider the first rule R.1 : $[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n \equiv [|\ \Delta_1 \bullet \Delta_2\ |]_n$. The left hand side expression takes in input a vector of n values[166] and applies to each element x_i of that vector the function computed by Δ_1 (let's call it f_1). Then the item is sent to the second stage that in turn computes the function denoted by Δ_2 (let's call it f_2). Therefore, after receiving an input such as $x = \langle x_1, \ldots x_n \rangle$ the left hand side expression computes $\langle f_2(f_1(x_1)), \ldots, f_2(f_1(x_n)) \rangle$. The right hand side, upon the reception of the same input $x = \langle x_1, \ldots x_n \rangle$, computes for each x_i the function computed by the pipeline of Δ_1 and Δ_2, which happens to be the same $f_2(f_1(x_i))$ computed in the first stage. ■

Rewriting rules may applied iteratively to the same RISC-pb²l expression to achieve more complex rewritings or, alternatively, much more simple expressions.

E.G.▷ Suppose we have a sequence of maps, implemented according to the schema discussed in Sec. refsec:riscpbb:map. The RISC-pb²l expression modelling the application will be a

$$\lhd_{scatter} \bullet [|\ \Delta_1\ |]_n \bullet \rhd_{gatherall} \bullet \lhd_{scatter} \bullet [|\ \Delta_2\ |]_n \bullet \rhd_{gatherall}$$

By applying R.3 above we can get a simpler expression:

$$\lhd_{scatter} \bullet [|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n \bullet \rhd_{gatherall}$$

This expression, in turn, may be rewritten using R.1 as

$$\lhd_{scatter} \bullet [|\ \Delta_1 \bullet \Delta_2\ |]_n \bullet \rhd_{gatherall}$$

[166] same parallelism degree n in both parallel building blocks of the pipe

Eventually, in case the two workers are sequential wrappers (that is $\Delta_i = f_i \; i \in [1,2]$), we can apply rule R.5 and therefore we obtain:

$$\triangleleft_{scatter} \bullet [|\ ((f_1 \circ f_2))\ |]_n \bullet \triangleright_{gatherall}$$

The whole rewriting process actually implements the map fusion rule described in Chap. 10 Eq. 10.1, read from left to right. ■

11.4 IMPLEMENTING RISC-pb^{2}l

The correct and efficient implementation of RISC-pb^{2}l building blocks and of the relative composition mechanisms will ensure the correct and efficient implementation of patterns build on top of RISC-pb^{2}l and therefore of the applications whose parallelism exploitation has been delegated to those patterns.

Therefore the RISC-pb^{2}l building blocks should be implemented as efficiently as possible and, in particular, their implementation should make the better usage of all the notable features of target architecture (both hw and sw).

The efficiency in the implementation of the RISC-pb^{2}l components may be achieved following two distinct but coordinated principles:

- the implementation of each building block should be designed and implemented in such a way that the introduction of any unnecessary overhead is avoided, and
- the implementation of recurring compositions of building blocks should be carefully optimized, possibly restructuring the building block composition or even implementing a completely different pattern, provided it implements the very same building block semantics.

Here we only outline some techniques used to design suitable, correct and efficient implementation of RISC-pb^{2}l, along with some example/use cases better motivating these techniques.

11.4.1 Avoid introducing unnecessary overheads

We consider some techniques suitable to avoid introduction of unnecessary overheads.

11.4.1.1 Copies vs. pointers In case of shared memory architectures, communications should be implemented "by reference" in all those cases where there is no risk to re-write "alive" data[167]. Taking into account that the RISC-pb^{2}l semantics is "data-flow"[168], the semantics associated to passing a parameter from one building block to the "following" one is such that it is guaranteed the first building block should not access any more the parameter after delivering it to the following block, indeed.

E.G.▷ Consider the broadcast combinator $\triangleleft_{broadcast}$. It may be implemented differently in case of share memory multicores and COW/NOW:

multicore the outputs will simply host a pointer to the shared data structure;

COW/NOW the outputs will host a copy of the relevant partition of the input data structures.

■

[167] that is, data whose current value is still subject to be read by some concurrent activity in the program
[168] functionals (combinators) compute results (deliver outputs) as soon as enough inputs are available

11.4.1.2 Scatter implementation The scatter combinator may be implemented quite efficiently on moder multicore architectures by adopting the following strategy:

- first, the exact amount of data to be directed to the n outputs of the $\lhd_{scatter}$ combinator are computed
- then, instead of actually splitting the input data into the n different slices computed in the previous step, a data structure is built hosting:
  1. a pointer to the input data structure, and
  2. a (logical) iterator navigating exacting those elements that belong to the slice.

E.G.▷ Consider the scatter of vector data (`float x[N]`, as an example). Let us suppose the output of the $\lhd_{scatter}$ is directed to some $[|\ldots|]_{n_w}$ with $N = k \times n_w$. In this case, the partition P_i to be directed to output i may be computed as the part of the vector from $i * k$ to $i * (k + 1) - 1$. Therefore we can use a record such as a `struct { float * v; int from; int to; }` to represent any of the partitions to be directed to the $\lhd_{scatter}$ outputs. Let's suppose that the computation to be performed on the partitions is a simple $f(x)$ for all the items of the vector in the partition[169]. Worker j and $j + 1$ will end up accessing $x[j * (k + 1) - 1]$ and $x[j * (k + 1)]$ respectively. These two items will mostly likely belong to the very same cache line. Therefore any modification of one of the two elements will trigger a cache coherence protocol to maintain consistency in between the cache line used by the thread executing the worker j and the cache line used by the thread executing worker $j + 1$. In case the modification is repeated in time (e.g. in case of iterative computation) this may lead to a non negligible overhead. By simply padding the vector elements in such a way two vector elements never belong to the same cache line[170] we can avoid any unnecessary cache coherence protocol runs. ■

11.4.2 Optimizing notable compositions

Particular attention has to be used while implementing notable compositions of building blocks.

First of all, if any rewriting rule may be used to "simplify" the RISC-pb^{2}l component composition, that rewriting rule has to be applied. This is the case of rule R.2 in Sec. 11.3. If the rule applies (left to right), we can get rid of two connectors and therefore simplify the overall implementation of the left hand side expression by implementing the right hand size one, actually.

Second, different optimizations may be designed even for those building block composition non corresponding to one of the rules listed in Sec. 11.3. These optimizations should be designed:

- when it is evident that a particular sub-expression of the general building block expression introduced some kind of overhead
- by taking into account the semantics of the involved building blocks.

We illustrate this second point with an example taken from [3]. A BSP (Bulk Synchronous Parallel) [85] pattern is built of a sequence of "super-steps" where each superstep computes in parallel a number of concurrent activities, each ending up with some communications

[169] therefore we are taking into account the scatter building block used to implement a $map(f)$ actually
[170] this is highly inefficient with respect to the memory footprint of the application, of course

directed to the parallel concurrent activities of the next super step. The BSP pattern may be modelled using RISC-pb^{2}l by the following building block expression:

$$\Delta^i_{SS} = \underbrace{([|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |]}_{Compute + Prepare\ Msg} \bullet \underbrace{\rhd_{Gatherall} \bullet (_{routeToDest}\lhd))}_{Communication + Barrier}$$

and the whole BSP computation may be expressed as

$$BSP(k) = \Delta^1_{SS} \bullet \ldots \bullet \Delta^k_{SS}$$

where the different Δ^a_b represent the b-th concurrent activity of the a-th super step, and $routeToDest$ is the function routing input message to the output used as input of the target concurrent activity in the next superstep.

When considering the composition of the Δ^i_{SS} phases (see Fig. 11.1 left) we can clearly recognize the presence of a bottleneck and, at the same time, a discrepancy w.r.t. the way communications are implemented within BSP. By collecting all messages in a single place through the $\rhd_{Gatherall}$ and then applying the $(_{RouteToDest}\lhd)$, a) the constraint of not having more than h communications (incoming or outcoming)[171] incident upon the same node is violated and b) the node gathering the messages from the super-steps constitutes a bottleneck.

The factorization of components in the RISC-pb^{2}l set, however, provides suitable tools to cope with this kind of situation. An expression such as

$$\rhd_{Gatherall} \bullet (_f\lhd)$$

actually routes to the $(_f\lhd)$ messages from the n sources of the $\rhd_{Gatherall}$ leaving those messages unchanged. Under the hypothesis that the $(_f\lhd)$ only routes those messages to their correct destinations among the m outputs of the $(_f\lhd)$ tree–that is, it does not process the messages "as a whole"–the $\rhd_{Gatherall} \bullet (_f\lhd)$ expression may clearly be transformed into an equivalent form were a distinct $(_f\lhd)$ is used to route messages at each of the inputs of the original $\rhd_{Gatherall}$ tree. All the n $(_f\lhd)$ trees will share the m outputs of the original, single $(_f\lhd)$. The original and optimized communication patterns for the two cases are shown in Fig. 11.1 right. In terms of RISC-pb^{2}l components, this optimizations may be expressed as

$$\rhd_{Gatherall} \bullet (_f\lhd) \equiv [|\ (_f\lhd)\ |]_n \qquad \texttt{(Opt1)}$$

where all the i^{th} outputs of the n $(_f\lhd)$ are assembled in the single i^{th} output of the $[|\ (_f\lhd)\ |]_n$.

It is worth noting that the optimization just outlined a) removes the original bottleneck, b) ensures the BSP $h-$relation and, last but not least, c) may be introduced in a completely automatic way *any time* we recognize that the (stateless) function f only processes a single message at a time in a $(\rhd_{Gatherall} \bullet (_f\lhd))$ expression. This is exactly in the spirit of RISC-pb^{2}l design: the system programmer responsible for providing an implementation for a given parallel pattern may use the RISC-pb^{2}l building blocks and rely on the optimizations, separately designed by the RISC-pb^{2}l engineers, to achieve efficiency.

The optimized version of the communications in the BSP skeleton as depicted in Fig. 11.1 (right) actually removed the synchronization of the $\rhd_{Gatherall}$ of the original expression. Therefore we should add an explicit synchronization immediately before the conclusion of the generic BSP superstep Δ^i_{SS}:

$$[|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |] \bullet$$
$$[|\ (_{routeToDest}\lhd)_1, \ldots, (_{routeToDest}\lhd)_n\ |] \bullet$$
$$\rhd_{Gatherall} \bullet \lhd_{Scatter}$$

[171] In order to satisfy the h-relation required by the BSP model

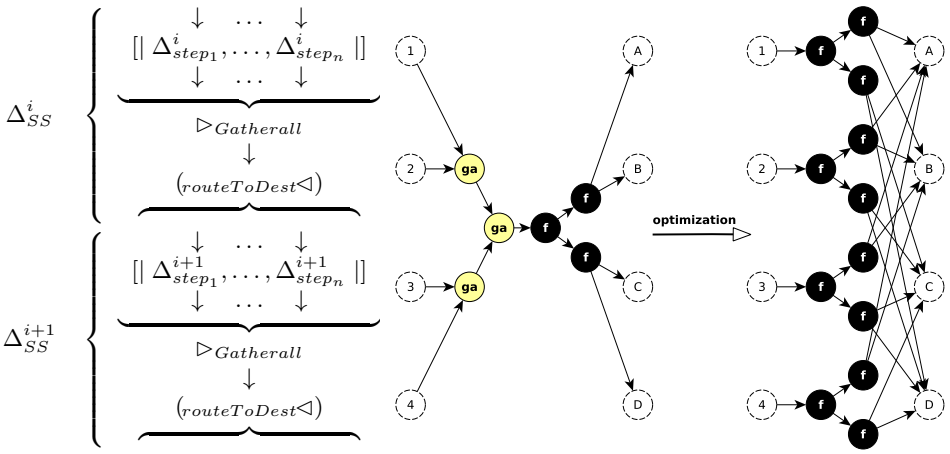


Figure 11.1 RISC-pb^{2}l modelling of the BSP superstep sequence (left). Optimizing the $\triangleright_{Gatherall} \bullet ({}_f\triangleleft)$ by $[|\ ({}_f\triangleleft)_1, \ldots, ({}_f\triangleleft)_n\ |]$ (f being $routeToDest$, ga being $Gatherall$) (right).

(the final $\triangleleft_{Scatter}$ is needed to re-establish the correct arities after the $\triangleright_{Gatherall}$ has collapsed its n inputs into a single output). The $\triangleright_{Gatherall} \bullet \triangleleft_{Scatter}$ implements the barrier and, in order not to introduce inefficiencies (bottlenecks) and violations of the BSP model features (h-relation), it has to be substituted by an efficient implementation of a barrier, if available, with no data routing at all. As a consequence, the BSP super-step may simply be rewritten as

$$[|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |] \bullet [|\ ({}_{routeToDest}\triangleleft)_1, \ldots, ({}_{routeToDest}\triangleleft)_n\ |] \bullet ((barrier))$$

11.4.3 Implementing RISC-pb^{2}l with FastFlow

The RISC-pb^{2}l components may be easily implemented in terms of FastFlow components (see Appendix B). The following table summarizes the correspondences between RISC-pb^{2}l and FastFlow components:

RISC-pb^{2}l	FastFlow
$((f))$	`ff_node` The `svc` method of the FastFlow `ff_node` may be used to wrap any kind of sequential C/C++ code. The responsibility to guarantee the wrapped code behaves as a function is in charge of the programmer.
$(\mid f \mid)$	`ff_node` The `svc` method may be also used to wrap some parallel code, e.g. offloading code to GPUs or using some OpenMP directive
$\lhd_{Pol}$, $(_f\lhd)$	`ff_farm` emitter, with proper `svc` methods implementing Pols and fs. The difference in between the $\lhd_{Pol}$ and the $(_f\lhd)$ implementation mainly relies in the amount of parallelism to be exploited. While the $\lhd_{Pol}$ may usually be implemented by a single thread (a farm Emitter), the $(_f\lhd)$ may require to use both the farm emitter and a string of concurrent activities (farm workers) to implement the spread with the intended parallelism degree.
$\rhd_{Pol}$, $(_f\rhd)$	`ff_farm` collector, with proper `svc` methods implementing Pols and fs (the considerations listed for the $\lhd_{Pol}/(_f\lhd)$ case hold in this case too)
$\overleftarrow{(\Delta)}_c$	The `add_feedback` method for farms and pipes implements exactly the RISC-pb^{2}l $\overleftarrow{(\Delta)}_c$ connector
$[\mid \Delta \mid]_n$, $[\mid \Delta_1, \ldots, \Delta_n \mid]$	`add_workers` in `ff_farm`, possibly sub classing different worker classes
$\Delta_1 \bullet \ldots \bullet \Delta_n$	`ff_pipeline`

CHAPTER 12

SKELETON SEMANTICS

We already mentioned that algorithmic skeleton semantics has traitionally been provided as a formal functional semantics plus an informal parallel semantics. This is in fact what we adopted in Chap. 3 to introduce the skeleton framework used in this book.

In this Chapter we discuss an alternative, more formal approach to define algorithmic skeleton semantics. The approach discussed here is mostly taken from [13]. In that work, authors were referring to the Lithium skeleton framework [19]. Here we update the semantics definition in such a way a skeleton system similar to the one introduced in Chap. 3 is modeled.

12.1 FORMAL DEFINITION OF THE SKELETON FRAMEWORK

We consider a skeleton framework providing both task parallel and data parallel skeletons. All the skeletons process a stream (finite or infinite) of input tasks to produce a stream of results. All the skeletons are assumed to be stateless, that is static variables are forbidden the sequential portions of code wrapped into sequential skeleton. No concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user[172]. All our skeletons are fully nestable. Each skeleton has one or more parameters that model the computations encapsulated in the related parallelism exploitation pattern. Our skeleton framework manages the usual primitive types (integers, floats, strings, vectors, arrays) plus two new types: streams and tuples. Streams are denoted by angled braces and tuples by "$\phi\ \flat$" braces.

$$\begin{array}{ll} \mathit{value} ::= \mathit{primitive\ data\ value} & \mathit{stream} ::= \langle\ \mathit{values}\ \rangle \mid \langle \epsilon \rangle \\ \mathit{values} ::= \mathit{value} \mid \mathit{value}\ ,\ \mathit{values} & \mathit{tuple}_k ::= \phi\ \mathit{stream}_1, \cdots, \mathit{stream}_k\ \flat \end{array}$$

[172]Such as external "servers" encapsulating shared data structures.

A stream represents a sequence (finite or infinite) of values of the same type, whereas the tuple is a parametric type that represents a (finite, ordered) set of streams. Actually, streams appearing in tuples are always *singleton streams*, i.e. streams holding a single value. The set of skeletons (Δ) provided by our framework is defined as follows:

$$\begin{array}{rcl} \Delta & ::= & \mathsf{seq}\ f \mid \mathsf{comp}\ \Delta_1\ \Delta_2 \mid \\ & & \mathsf{farm}\ \Delta \mid \mathsf{pipe}\ \Delta_1\ \Delta_2 \mid \\ & & \mathsf{map}\ f_{\mathsf{d}} \Delta\ f_{\mathsf{c}} \mid \mathsf{d\&c}\ f_{\mathsf{tc}} f_{\mathsf{d}} \Delta\ f_{\mathsf{c}} \mid \\ & & \mathsf{while}\ f_{\mathsf{tc}} \Delta \end{array}$$

where sequential functions (such as f, g) with no index have type $\alpha \rightarrow \beta$ with α and β primitive types as defined above and indexed functions (such as $f_{\mathsf{c}}, f_{\mathsf{d}}, f_{\mathsf{tc}}$) have the following types: f_{c} :$tuple_k \rightarrow stream$; f_{d} :$stream \rightarrow tuple_k$; f_{tc} :$\alpha \rightarrow boolean$. In particular, $f_{\mathsf{c}}, f_{\mathsf{d}}$ represent families of functions that enable the splitting of a stream in k-tuples of singleton streams and vice-versa.

Our skeletons can be considered as a pre-defined higher-order functions. Intuitively, seq skeleton just wraps sequential code chunks within the structured parallel framework; farm and pipe skeletons model embarrassingly parallel and pipeline computations, respectively; comp models pipelines with stages serialized on the same processing element (PE); map models data parallel computations: f_d decomposes the input data into a set of possibly overlapping data subsets, the inner skeleton computes a result out of each subset and the f_c function rebuilds a unique result out of these results; d&c models Divide&Conquer computations: input data is divided into subsets by f_d and each subset is computed recursively and concurrently until the f_{tc} condition does not hold true. At this point results of subcomputations are conquered via the f_c function. while skeleton model indefinite iteration.

A skeleton applied to a stream is called a *skeletal expression*. Expressions are defined as follows:

$$\begin{array}{rcl} exp & ::= & \Delta\ stream \mid \Delta\ exp \mid \\ & & \mathcal{R}_\ell\ exp \mid f_{\mathsf{c}}\ (\alpha\, \Delta)\ f_{\mathsf{d}}\ stream \end{array}$$

The execution of a skeleton program consists in the evaluation of a Δ *stream* expression.

12.2 OPERATIONAL SEMANTICS

We describe the semantics of our skeleton framework by means of a LTS. We define the *label* set as the string set augmented with the special label "$\perp$". We rely on labels to distinguish both streams and transitions. Input streams have no label, output stream are $\perp$ labeled. Labels on streams describe where data items are mapped within the system, while labels on transitions describe where they are computed. The operational semantics of our skeleton framework is described in Figure 12.1. The rules of the semantics may be grouped in two main categories, corresponding to the two halves of the figure. Rules 1–7 describe how skeletons behave with respect to a stream. These rules spring from the following common schema:

$$\mathsf{Skel}\ params\ \langle x, \tau \rangle_{\ell_1} \xrightarrow{\ell_1} \mathcal{F} \langle x \rangle_{\ell_2} :: \mathsf{Skel}\ params\ \langle \tau \rangle_{\ell_3}$$

where $\mathsf{Skel} \in [\mathsf{seq}, \mathsf{farm}, \mathsf{pipe}, \mathsf{comp}, \mathsf{map}, \mathsf{d\&c}, \mathsf{while}]$, and the infix stream constructor $\langle \sigma \rangle_{\ell_i} :: \langle \tau \rangle_{\ell_j}$ is a non strict operator that sequences skeletal expressions. In general, $\mathcal{F}$ is a function appearing in the *params* list. For each of these rules a couple of *twin rules* exists (not shown in Figure 12.1):

α) $\mathsf{Skel}\ params\ \langle x, \tau \rangle \xrightarrow{\perp} \langle \epsilon \rangle_\perp :: \mathsf{Skel}\ params\ \langle x, \tau \rangle_\perp$

ω) $\mathsf{Skel}\ params\ \langle x \rangle_{\ell_1} \xrightarrow{\ell_1} \mathcal{F} \langle x \rangle_{\ell_2}$

1. $\mathsf{seq}\ f\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \langle f\,x\rangle_\ell :: \mathsf{seq}\ f\ \langle\tau\rangle_\ell$
2. $\mathsf{farm}\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta\ \langle x\rangle_{\mathcal{O}(\ell,\mathrm{x})} :: \mathsf{farm}\ \Delta\ \langle\tau\rangle_{\mathcal{O}(\ell,\mathrm{x})}$
3. $\mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta_2\ \mathcal{R}_{\mathcal{O}(\ell,\mathrm{x})}\ \Delta_1\ \langle x\rangle_\ell :: \mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
4. $\mathsf{comp}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta_2\ \Delta_1\ \langle x\rangle_\ell :: \mathsf{comp}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
5. $\mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} f_\mathsf{c}\ (\alpha\ \Delta)\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$
6. $\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$

$$\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle y\rangle_\ell = \begin{cases} \Delta\ \langle y\rangle_\ell & \textit{iff}\ \ (f_\mathsf{tc}\ y) \\ f_\mathsf{c}\,(\alpha\,(\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}))\ f_\mathsf{d}\ \langle y\rangle_\ell & \textit{otherwise} \end{cases}$$

7. $\mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \begin{cases} \mathsf{while}\ f_\mathsf{tc}\ \Delta\ (\Delta\ \langle x\rangle_\ell :: \langle\tau\rangle_\ell) & \textit{iff}\ \ (f_\mathsf{tc}\ x) \\ \langle x\rangle_\ell ::\ \mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$

$$\langle\sigma\rangle_{\ell_1} :: \langle\tau\rangle_{\ell_2} \xrightarrow{\perp} \langle\sigma,\tau\rangle_\perp \quad \textit{join} \qquad \langle\epsilon\rangle_\perp :: \langle\tau\rangle_{\ell_2} \xrightarrow{\perp} \langle\tau\rangle_{\ell_2} \quad \textit{join}_\epsilon$$

$$\frac{\Delta\ \langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_{\ell_3}}{\mathcal{R}_\ell\ \Delta\langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_\ell}\ \textit{relabel} \qquad \frac{E_1 \xrightarrow{\ell} E_2}{\Delta\ E_1 \xrightarrow{\ell} \Delta\ E_2}\ \textit{context}$$

$$\frac{f_\mathsf{d}\,\langle x\rangle_\ell \xrightarrow{\ell} \langle\!\langle\, \langle y_1\rangle_\ell, \cdots \langle y_n\rangle_\ell \,\rangle\!\rangle \quad \Delta\langle y_i\rangle_\ell \xrightarrow{\ell} \langle z_i\rangle_\ell \quad f_\mathsf{c} \langle\!\langle\, \langle z_1\rangle_\ell, \cdots \langle z_n\rangle_\ell \,\rangle\!\rangle \xrightarrow{\ell} \langle z\rangle_\ell,\ i = 1..n}{f_\mathsf{c}\ (\alpha\Delta)\ f_\mathsf{d}\,\langle x\rangle_\ell \xrightarrow{\ell} \langle z\rangle_\ell}\ \textit{dp}$$

$$\frac{E_i \xrightarrow{\ell_i} E_i' \quad \forall i\ 1 \le i \le n \quad \wedge \quad \exists i,j\ 1 \le i,j \le n,\ \ell_i = \ell_j \Rightarrow i = j}{\langle\nu\rangle_\perp :: E_1 :: \cdots E_n :: \Gamma \xrightarrow{\perp} \langle\sigma\rangle_\perp :: E_1' :: \cdots E_n' :: \Gamma}\ \textit{sp}$$

Figure 12.1 *Lithium* operational semantics. $x, y \in \textit{value}$; $\sigma, \tau \in \textit{values}$; $\nu \in \textit{values} \cup \{\epsilon\}$; $E \in \textit{exp}$; $\Gamma \in \textit{exp}^*$; $\ell, \ell_i, \ldots \in \textit{label}$; $\mathcal{O} : \textit{label} \times \textit{value} \to \textit{label}$.

these rules manage the first and the last element of the stream respectively. Each triple of rules manages a stream as follows: the stream is first labeled by a rule of the kind α). Then the stream is unfolded in a sequence of singleton streams and the nested skeleton is applied to each item in the sequence. During the unfolding, singleton streams are labeled according to the particular rule policy, while the transition is labeled with the label of the stream before the transition (in this case ℓ_1). The last element of the stream is managed by a rule of the kind ω). Eventually resulting skeletal expressions are joined back by means of the :: operator.

Let us show how rules 1–7 work with an example. We evaluate

$$\mathsf{farm}\ (\mathsf{seq}\ \mathrm{f})$$

on the input stream

$$\langle x_1, x_2, x_3\rangle$$

At the very beginning only the 2_α can be applied. It marks the begin of stream by introducing $\langle\epsilon\rangle_\perp$ (empty stream) and labels the input stream with $\perp$. Then rule 2 can be applied:

$$\langle\epsilon\rangle_\perp :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_1, x_2, x_3\rangle_\perp \xrightarrow{\perp} \langle\epsilon\rangle_\perp :: \mathsf{seq}\ f\ \langle x_1\rangle_0 :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_2, x_3\rangle_0$$

The head of the stream has been separated from the rest and has been re-labeled (from $\perp$ to 0) according to the $\mathcal{O}(\perp, x)$ function. Inner skeleton (seq) has been applied to this singleton stream, while the initial skeleton has been applied to the rest of the stream in a recursive fashion. The re-labeling function $\mathcal{O} : \textit{label} \times \textit{value} \to \textit{label}$ (namely the *oracle*) is an external function with respect to the LTS. It would represent the (user-defined) data

mapping policy. Let us adopt a two-PEs round-robin policy. An oracle function for this policy would cyclically return a label in a set of two labels. In this case the repeated application of rule 1 proceeds as follows:

$\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f\ \langle x_1\rangle_0 :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_2, x_3\rangle_0 \xrightarrow{0} \xrightarrow{1} \langle\epsilon\rangle_\perp :: \mathsf{seq}\ f\ \langle x_1\rangle_0 :: \mathsf{seq}\ f\ \langle x_2\rangle_1 :: \mathsf{seq}\ f\ \langle x_3\rangle_0$

The oracle may have an internal state, and it may implement several policies of label transformation. As an example the oracle might always return a fresh label, or it might make decisions about the label to return on the basis of the x value. As we shall see, in the former case the semantics models the maximally parallel computation of the skeletal expression.

Observe that using only rules 1–7 (and their twins) the initial skeleton expression cannot be completely reduced (up to the output stream). Applied in all the possible ways, they lead to an aggregate of expressions *(exp)* glued by the :: operator.

The rest of the work is carried out by the six rules in the bottom half of figure 12.1. There are two main rules (*sp* and *dp*) and four auxiliary rules (*context*, *relabel*, $join_\epsilon$ and *join*). Such rules define the order of reduction along aggregates of skeletal expressions. Let us describe each rule:

sp (stream parallel) rule describes evaluation order of skeletal expressions along sequences separated by :: operator. The meaning of the rule is the following: suppose that each skeletal expression in the sequence may be rewritten in another skeletal expression with a certain labeled transformation. Then all such skeletal expressions can be transformed in parallel, provided that they are adjacent, that the first expression of the sequence is a stream of values, and that all the transformation labels involved are pairwise different.

dp (data parallel) rule describes the evaluation order for the $f_\mathsf{c}\ (\alpha\,\Delta)\ f_\mathsf{d}$ *stream* expression. Basically the rule creates a tuple by means of the f_d function, then requires the evaluation of all expressions composed by applying Δ onto all elements of the tuple. All such expression are evaluated in one step by the rule (apply-to-all). Finally, the f_c gathers all the elements of the evaluated tuple in a singleton stream. Labels are not an issue for this rule.

relabel rule provides a relabeling facility by evaluating the meta-skeleton $\mathcal{R}_\ell$. The rule does nothing but changing the stream label. Pragmatically, the rule imposes to a PE to send the result of a computation to another PE (along with the function to compute).

context rules establishes the evaluation order among nested expressions, in all other cases but the ones treated by *dp* and *relabel*. The rule imposes a strict evaluation of nested expressions (i.e. evaluated the arguments first). The rule leaves unchanged both transition and stream labels with respect to the nested expression.

join rule does the housekeeping work, joining back all the singleton streams of values to a single output stream of values. $join_\epsilon$ does the same work on the first element of the stream.

Let us consider a more complex example. Consider the semantics of

$\mathsf{farm}\ (\mathsf{pipe}\ f_1\ f_2)$

evaluated on the input stream

$\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle$

$$\mathsf{farm}\ (\mathsf{pipe}\ (\mathsf{seq}\ f_1)\ (\mathsf{seq}\ f_2))\ \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle \tag{12.1}$$

$$\langle\epsilon\rangle_\perp :: \mathsf{farm}\ (\mathsf{pipe}\ (\mathsf{seq}\ f_1)\ (\mathsf{seq}\ f_2))\ \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle_\perp \tag{12.2}$$

$$\begin{array}{l}\langle\epsilon\rangle_\perp :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_1\rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_2\rangle_1 ::\\ \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_3\rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_4\rangle_1 ::\\ \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_5\rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_6\rangle_1 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_7\rangle_0\end{array} \tag{12.3}$$

$$\begin{array}{l}\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_1\rangle_0 :: \mathsf{seq}\ f_2\ \mathcal{R}_{12}\ \mathsf{seq}\ f_1\ \langle x_2\rangle_1 :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_3\rangle_0 ::\\ \mathsf{seq}\ f_2\ \mathcal{R}_{12}\ \mathsf{seq}\ f_1\ \langle x_4\rangle_1 :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_5\rangle_0 ::\\ \mathsf{seq}\ f_2\ \mathcal{R}_{12}\ \mathsf{seq}\ f_1\ \langle x_6\rangle_1 :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_7\rangle_0\end{array} \tag{12.4}$$

$$\begin{array}{l}\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f_2\ \langle f_1\ x_1\rangle_{02} :: \mathsf{seq}\ f_2\ \langle f_1\ x_2\rangle_{12} :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_3\rangle_0 ::\\ \mathsf{seq}\ f_2\ \mathcal{R}_{12}\ \mathsf{seq}\ f_1\ \langle x_4\rangle_1 :: \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_5\rangle_0 :: \mathsf{seq}\ f_2\ \mathcal{R}_{12}\ \mathsf{seq}\ f_1\ \langle x_6\rangle_1 ::\\ \mathsf{seq}\ f_2\ \mathcal{R}_{02}\ \mathsf{seq}\ f_1\ \langle x_7\rangle_0\end{array} \tag{12.5}$$

$$\begin{array}{l}\langle\epsilon\rangle_\perp :: \langle f_2\,(f_1\ x_1)\,\rangle_{02} :: \langle f_2\,(f_1\ x_2)\,\rangle_{12} :: \langle f_2\,(f_1\ x_3)\,\rangle_{02} ::\\ \langle f_2\,(f_1\ x_4)\,\rangle_{12} :: \langle f_2\,(f_1\ x_5)\,\rangle_{02} :: \langle f_2\,(f_1\ x_6)\,\rangle_{12} :: \langle f_2\,(f_1\ x_7)\,\rangle_{02}\end{array} \tag{12.6}$$

$$\langle f_2\,(f_1\ x_1),\ f_2\,(f_1\ x_2)\ f_2\,(f_1\ x_3),\ f_2\,(f_1\ x_4)\ f_2\,(f_1\ x_5),\ f_2\,(f_1\ x_6),\ f_2\,(f_1\ x_7)\,\rangle_\perp \tag{12.7}$$

Figure 12.2 The semantic of a *Lithium* program: a complete example.

Let us suppose that the oracle function returns a label chosen from a set of two labels. Pragmatically, since farm skeleton represent the replication paradigm and pipe skeleton the pipeline paradigm, the nested form farm (pipe f_1 f_2) basically matches the idea of a multiple pipeline (or a pipeline with multiple independent channels). The oracle function defines the parallelism degree of each paradigm: in our case two pipes, each having two stages. As shown in Figure 12.2, the initial expression is unfolded by rule 2_α ((12.1) $\rightarrow$ (12.2)) then reduced by many applications of rule 2 ((12.2) $\rightarrow^*$ (12.3)). Afterward the term can be rewritten by rule 3 ((12.3) $\rightarrow^*$ (12.4)). At this point, we can reduce the formula using the *sp* rule. *sp* requires a sequence of adjacent expressions that can be reduced with differently labeled transitions. In this case we can find just two different labels $(0, 1)$, thus we apply *sp* to the leftmost pair of expressions as follows:

$$\dfrac{\dfrac{\dfrac{\mathsf{seq}\ f_1\ \langle x_1\rangle_0 \xrightarrow{0} \langle f_1\ x_1\rangle_0}{\mathcal{R}_{02}\,\mathsf{seq}\ f_1\ \langle x_1\rangle_0 \xrightarrow{0} \langle f_1\ x_1\rangle_{02}}\ \textit{relabel}}{\mathsf{seq}\ f_2\,\mathcal{R}_{02}\,\mathsf{seq}\ f_1\ \langle x_1\rangle_0 \xrightarrow{0} \mathsf{seq}\ f_2\ \langle f_1\ x\rangle_{02}}\ \textit{context} \qquad \dfrac{\dfrac{\dfrac{\mathsf{seq}\ f_1\ \langle x_2\rangle_1 \xrightarrow{1} \langle f_1\ x_2\rangle_1}{\mathcal{R}_{12}\,\mathsf{seq}\ f_1\ \langle x_2\rangle_1 \xrightarrow{1} \langle f_1\ x_2\rangle_{12}}\ \textit{relabel}}{\mathsf{seq}\ f_2\,\mathcal{R}_{12}\,\mathsf{seq}\ f_1\ \langle x_2\rangle_1 \xrightarrow{1} \mathsf{seq}\ f_2\ \langle f_1\ x\rangle_{12}}\ \textit{context}}{\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f_2\,\mathcal{R}_{02}\,\mathsf{seq}\ f_1\ \langle x_1\rangle_0 :: \mathsf{seq}\ f_2\,\mathcal{R}_{12}\,\mathsf{seq}\ f_1\ \langle x_2\rangle_1 :: \mathsf{seq}\ f_2\,\mathcal{R}_{02}\,\mathsf{seq}\ f_1\ \langle x_3\rangle_0 :: \cdots \xrightarrow{\perp}}\ \textit{sp}$$
$$\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f_2\langle f_1\ x_1\rangle_{02} :: \mathsf{seq}\ f_2\ \langle f_1\ x_2\rangle_{12} :: \mathsf{seq}\ f_2\,\mathcal{R}_{02}\,\mathsf{seq}\ f_1\ \langle x_3\rangle_0 :: \cdots$$

Observe that due to the previous reduction ((12.4) $\rightarrow$ (12.5) in Figure 12.2) two new stream labels appear (02 and 12). Now, we can repeat the application of *sp* rule as in the previous step. This time, it is possible to find four adjacent expression that can be rewritten with (pairwise) different labels $(0, 1, 02, 12)$. Notice that even on a longer stream this oracle function never produces more than four different labels, thus the maximum number of skeletal expressions reduced in one step by *sp* is four. Repeating the reasoning the we can completely reduce the formula to a sequence of singleton streams ((12.5) $\rightarrow^*$ (12.6)), that, in turn can be transformed by many application of the *join* rule ((12.6) $\rightarrow^*$ (12.7)).

Let us analyze the whole reduction process: from the initial expression to the final stream of values we applied three times the *sp* rule. In the first application of *sp* we transformed two skeletal expression in one step; in the second application four skeletal expressions have been involved; while in the last one just one expression has been reduced[173].

[173]Second and third *sp* application are not shown in the example.

The succession of *sp* applications in the transition system matches the expected behavior for the double pipeline paradigm. The first and last applications match pipeline start-up and end-up phases. As expected the parallelism exploited is reduced with respect to the steady state phase, that is matched by the second application. A longer input stream would rise the number of *sp* applications, in fact expanding the steady state phase of modeled system.

12.3 PARALLELISM AND LABELS

The first relevant aspect of the proposed schema is that the functional semantics is independent of the labeling function. Changing the oracle function (i.e. how data and computations are distributed across the system) may change the number of transitions needed to reduce the input stream to the output stream, but it cannot change the output stream itself.

The second aspect concerns parallel behavior. It can be completely understood looking at the application of two rules: *dp* and *sp* that respectively control data and stream parallelism. We call the evaluation of either *dp* or *sp* rule a *par-step*. *dp* rule acts as an apply-to-all on a tuple of data items. Such data items are generated partitioning a single task of the stream by means of an user-defined function. The parallelism comes from the reduction of all elements in a tuple (actually singleton streams) in a single *par-step*. A single instance of the *sp* rule enable the parallel evolution of adjacent terms with different labels (i.e. computations running on distinct PEs). The converse implication also holds: many transitions of adjacent terms with different labels *might* be reduced with a single *sp* application. However, notice that *sp* rule may be applied in many different ways even to the same term. In particular the number of expressions reduced in a single *par-step* may vary from 1 to n (i.e. the maximum number of adjacent terms exploiting different labels). These different applications lead to both different proofs and parallel (but functional) semantics for the same term. This degree of freedom enables the language designer to define a (functionally confluent) family of semantics for the same language covering different aspects of the language. For example, it is possible to define the semantics exploiting the maximum available parallelism or the one that never uses more than k PEs.

At any time, the effective parallelism degree in the evaluation of a given term in a *par-step* can be counted by inducing in a structural way on the proof of the term. Parallelism degree in the conclusion of a rule is the sum of parallelism degrees of the transitions appearing in the premise (assuming one the parallelism degree in rules 1–7). The parallelism degree counting may be easily formalized in the LTS by using an additional label on transitions.

The LTS proof machinery subsumes a generic skeleton implementation: the input of skeleton program comes from a single entity (e.g. channel, cable, etc.) and at discrete time steps. To exploit parallelism on different tasks of the stream, tasks are spread out in PEs following a given discipline. Stream labels trace tasks in their journey, and *sp* establishes that tasks with the same label, i.e. on the same PE, cannot be computed in parallel. Labels are assigned by the oracle function by rewriting a label into another one using its own internal policy. The oracle abstracts the mapping of data onto PEs, and it can be viewed as a parameter of the transition system used to model several policies in data mapping. As an example a farm may take items from the stream and spread them in a round-robin fashion to a pool of workers. Alternatively, the farm may manage the pool of workers by divination, always mapping a data task to a free worker (such kind of policy may be used to establish an upper bound in the parallelism exploited). The label set effectively used in a computation depends on the oracle function. It can be a statically fixed set or it can change cardinality during the evaluation. On this ground, a large class of implementations may be modeled.

Labels on transformations are derived from label on streams. Quite intuitively, a processing element must known a data item to elaborate it. The re-labeling mechanism enables

to describe a data item re-mapping. In a *par-step*, transformation labels point out which are the PEs currently computing the task.

12.4 HOW TO USE THE LABELED TRANSITION SYSTEM

The semantics discussed in section 12.2 can be used to prove the correctness of our skeleton programs, as expected. Here we want to point out how the transition system can be used to evaluate analogies and differences holding between different skeleton systems and to evaluate the effect of rewriting rules on a skeleton program.

Concerning the first point, let us take into account, as an example, the skeleton framework introduced by Darlington's group in mid '90. In [46] they define a farm skeleton as follows: *"Simple data parallelism is expressed by the* FARM *skeleton. A function is applied to each element of a list, each element of which is thought of as a task. The function also takes an environment, which represents data which is common to all tasks. Parallelism is achieved by distributing the tasks on different processors.*

$\mathsf{FARM} : (\alpha \to \beta \to \gamma) \to \alpha \to (\, \{\!|\, \beta \,|\!\} \to \{\!|\, \gamma \,|\!\}\,)$
$\mathsf{FARM\ f\ env} = \mathsf{map\ (f\ env)}$
where $\mathsf{map\ f\ x} = \mathsf{f\ x}$ *and* $\mathsf{map\ f\ x::rx} = \mathsf{(f\ x)::(map\ f\ rx)}$".

Darlington's FARM skeleton happens to have the same name as the farm skeleton developed by our group [23, 24] and by other groups ([77, 60]) but it has a slightly different semantics. It actually exploits data parallelism, in that parallelism comes from parallel computation of items belonging to the same task (input data item), i.e. the list x parameter, whereas other farm skeletons express embarrassingly parallel computations exploiting parallelism in the computation of disjunct, completely independent task items. This can be easily expressed using our semantics by observing that the clause:

$$\mathsf{FARM}\ f_c\ \Delta\ f_d\ \langle x \rangle_\ell \xrightarrow{\ell} f_c\ (\alpha\ \Delta)\ f_d\ \langle x \rangle_\ell$$

exactly models both the functional and the parallel behavior of *Lithium* map provided that f_d is the function taking a list and returning a tuple of singletons, f_c is the function taking a tuple of singletons and returning a list and that Δ is actually a sequential function f. However, in order to make the FARM being "stream aware", we must rather consider a clause such as:

$$\mathsf{FARM}\ f_c\ \Delta\ f_d\ \langle x, \tau \rangle_\ell \xrightarrow{\ell} f_c\ (\alpha\ \Delta)\ f_d\ \langle x \rangle_\ell :: \mathsf{FARM}\ f_c\ \Delta\ f_d\ \langle \tau \rangle_\ell$$

recursive on the stream parameter. At this point we should immediately notice that this is the same clause modeling our map skeleton (cf. rule 5 in Figure 12.1).

Concerning the possibility of using the LTS to prove correctness of skeleton rewriting rules, let us take as an example the equivalence "$\mathsf{farm}\ \Delta \equiv \Delta$" (see [4, 7]). The equivalence states that farms (stream parallel ones, not the Darlington's ones) can be inserted and removed anywhere in a skeleton program without affecting the functional semantics of the program. The equivalence looks like to be a really simple one, but it is fundamental in the process of deriving the skeleton "normal form" [4] (see Sec. 10.3. The normal form is a skeletal expression that can be automatically derived from any other skeletal expression. It computes the same results of the original one but uses less resources and delivers better performance.

The validity of these equivalences can be evaluated using the labeled transition system and the effects on parallelism degree in the execution of left and right hand side programs

can be evaluated too. The validity of the equivalence can be simply stated by taking into account rule 2 in Figure 12.1. It simply unfolds the stream and applies Δ to the stream items, as we can expect for Δ applied to the same stream, apart for the labels. As the labels do not affect the functional semantics, we can derive that the equivalence "$\mathsf{farm}\ \Delta\ \equiv\ \Delta$" actually holds for any skeleton tree Δ. We can say more, however. Take into account expression (12.1) in Figure 12.2. We derived its semantics and we concluded that four different (non-bottom) labels can be obtained: $00, 01, 10, 11$. Rewriting (12.1) by using the equivalence from left to right we get: "$\mathsf{pipe}\ (\mathsf{seq}\ f_1)\ (\mathsf{seq}\ f_2)\ \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle$". From this expression we can derive exactly the same output but holding only two distinct labels: $0, 1$. These two labels happen to be the two labels modeling the two stages of the original pipeline. As a conclusion we can state that:

1. the equivalence "$\mathsf{farm}\ \Delta\ \equiv\ \Delta$" holds and functional semantics is not affected by it;
2. parallel semantics is affected by the equivalence in that the parallelism degree changes.[174]

The skeleton framework considered here only provides *stateless* skeletons, as well as most of the other existing skeletons systems [23, 45, 77, 71]. A discussion about the opportunity of introducing a state in a functional framework, as well as the effectiveness of different approaches to implement it, goes beyond our aims. However, we want to point out that the proposed operational semantics may effectively support both global and local state concept with just few adjustments.

As labels can be used to represent the places where computations actually happen, they can also be used to model/drive global and local state implementation. In particular, they can be used to have a precise idea of where the subjects and objects of state operations (the process/thread issuing the state operation and the process(es)/thread(s) actually taking care of actually performing the operation) are running. This allows a precise modeling of the operations needed to implement skeleton state.

[174]This just one case, of course, in order to complete the proof all the different possibles Δ must be considered. However the simple case of $\Delta=\ \mathsf{pipe}$ should made clear how the full proof may work.

CHAPTER 13

SURVEY OF EXISTING SKELETON FRAMEWORKS

In this Chapter we present a quick survey of existing skeleton frameworks. We do not pretend to be exhaustive nor to discuss all the features of the skeleton frameworks we present. However, we give an overview of the most important features of these frameworks. First, we describe the criteria used to classify and comment the different skeleton frameworks, then we will discuss each skeleton framework in detail. Frameworks will be presented as belonging to different classes distinguishing the kind of "sequential" language used as the "host" language of the skeleton framework. Eventually, we will shortly discuss some programming environments that are not "true" skeleton based programming environments, although they support a limited amount of features typical of skeleton frameworks.

13.1 CLASSIFICATION

We discuss in this Section the main features we will consider when presenting the different skeleton frameworks. Some features are more related to the abstract programming model supported, others are more related to the implementation features of the programming framework or the mechanisms and policies provided to support/target different architectures. All contribute to differentiate–or to make more similar–the different programming environments discussed.

13.1.1 Abstract programming model features

These are the features we will take into account when surveying the skeleton frameworks more related to the *programming model presented to the framework users/programmers.*

Supported skeleton classes. Different skeleton classes may be supported, including stream parallel, data parallel and control parallel skeletons.

State. Algorithmic skeletons are usually stateless, but some stateful kind of skeletons may also be supported.

Extendability. Different levels of expandability are supported, ranging from no expandability at all to limited or unlimited expandability.

Semantics. Formal or informal/semi-formal semantics may be defined relatively to the algorithmic skeletons included in the framework.

Code reuse. Different code reuse options may be provided. Reuse of sequential, functional code written in a single sequential programming language within a sequential skeleton wrapping is supported by all the frameworks. Several frameworks support more than a single sequential programming language. A few of them also support the usage of parallel languages/libraries in a `seq` skeleton.

Autonomic management of non functional features. Management of non functional features may be supported or not supported at all. In case of support, different levels of support may be provided, ranging from mechanisms to support user directed management up to complete autonomic management of non functional features.

Fault tolerance. Fault tolerance may be supported natively by the skeleton programming framework or by the abstract machine targeted when compiling skeletons either in a template assembly or in MDGI to be executed within a distributed MDF interpreter. More likely, the skeleton framework may not support any kind of fault tolerance, neither at the mechanism nor at the policy level.

Support for application fine tuning. Some frameworks may support particular mechanisms aimed at driving/assisting users/programmers in the fine tuning of a skeleton application. Most likely, fine tuning is related to non functional properties tuning (e.g. performance tuning).

13.1.2 Implementation related features

These are the features more related to the way the skeleton programming framework is actually implemented on top of some other hardware virtualization layer.

Implementation model: user interface. This is the kind of interface presented to the user/programmer. It may be some kind of library interface or a new language syntax, or a mix of the two.

Implementation model: internals. This is the kind of implementation model chosen to implement the skeleton based programming framework: template based, macro data flow based, other.

"Host" language. The "host" language in a skeleton framework is the sequential language used to express the sequential portions of code or whose data types have been inherited to define the data types of the skeleton programming framework.

"Object code" abstract machine. This is the abstract machine targeted by the skeleton framework compiling tools back end or used to write the libraries implementing the framework. It could be a true (parallel) abstract machine, but most likely it could be made out

of some kind of sequential programming environment paired with some kind of communication/synchronization library.

OSS. The skeleton based programming framework may be licensed as open source, freeware or even as copyrighted code. This mainly impacts diffusion of the programming framework.

13.1.3 Architecture targeting features

These are those features that are more related to the kind of target architectures eventually supported by the skeleton programming framework.

Multi/many core support. Most of the existing skeleton based programming frameworks have been developed to target distributed architectures such as COWs, NOWs or GRIDs. Some have been re-designed to be able to efficiently support also multi/many core architectures.

Heterogeneous architecture targeting. The simplest skeleton based programming environments only support homogeneous networks/clusters/assemblies of processing elements. Is is more and more important to be able to target with the same efficiency also architectures hosting different kinds of processing elements.

13.2 C/C++ BASED FRAMEWORKS

13.2.1 P3L

The Pisa Parallel Programming Language (P3L) has been developed in early '90 at the Hewlett Packard Pisa Science Center in conjunction with the Dept. of Computer Science of the Univ. of Pisa (I) [23]. It has been the first complete skeleton based programming framework available after algorithmic skeletons had been introduced.

P3L supported stream, data and control parallel skeletons. All the P3L skeletons were stateless, although extensions were designed[175] that allowed skeletons to share some kind of state. The skeleton set of P3L was fixed: no possibility was given to the user to extend it. It included pipeline, farm, map, reduce, geometric[176] and loop skeletons. The semantics of the algorithmic skeletons provided was given in terms of HOF relatively to functional semantics only. Parallel semantics was given informally, instead. The sequential wrapping skeleton of P3L supported different host languages. Anacleto, the open source P3L compiler only allowed C and C++ to be used in sequential skeletons. However SkIE, the "industrial" version of P3L [24] also supported Fortran, HPF and eventually Java. The sequential skeleton was conceived in such a way libraries and further object code modules may be named to be linked with the sequential portions of code wrapped in the sequential skeleton.

P3L did not provide any kind of non functional feature management, nor it supported any kind of automatic fault tolerance. It was designed as a new language[177] and implemented with a template based set of tools. The first version of the compiling tools, completed in 1991, were written in Prolog and generated Occam code for Transputer based architectures only. Later on the open source P3L compiler was written in C and targeted MPI COW/NOW architectures. SkIE was also written in C/C++ and targeted MPI

[175] never implemented
[176] a particular skeleton modelling data parallel computation, possibly with stencils
[177] a coordination language, actually. Most of the syntax was inherited from C/C++. Skeleton declarations closely looked like C/C++ function declarations.

COW/NOW architectures, as well as Meiko/CS Transputer MPP. All these version only target homogeneous collections of processing elements. Non homogeneous collections of processing elements could be targeted exploiting MPI portability, but no specific implementation/optimizations had been designed to support heterogeneity.

P3L represents a milestone in the algorithmic skeleton framework scenario for at least two different reasons:

- it constituted the "proof-of-concept" environment demonstrating that algorithmic skeleton frameworks were feasible and could achieve performance figures comparable to those achieved when running hand written parallel code
- P3L introduced the "two-tier" composition model (see Sec. 3.4.3) that later on has been adapted by other skeleton based programming frameworks (e.g. Muesli [33]).

P3L is no longer maintained although an ML version of P3L (OcamlP3L, see Sec. 13.4.2) is maintained in France.

13.2.2 Muesli

Muesli[178] has been introduced in early '2000 at the Univ. of Muenster (D) by the group of Prof. Kuchen [60]. Kuchen built Muesli on the experience gained with Skil [27] in the late '90s.

Muesli represents a very nice implementation of an algorithmic skeleton framework on top of an existing sequential programming environment. It provides stream and data parallel skeletons as C++ template classes. In order to use a skeleton, you must simply instantiate an object in the corresponding class. The C++ technique and tools have been properly stressed to provide a very clean implementation of the skeletons. By the way the usage made of C++ features requires some care when compiling the Muesli library. Muesli skeletons are stateless and there is no way to expand the skeleton set. Code reuse is guaranteed by the adoption of C++ as the host programming language. Muesli does not support non functional feature management nor fine grain application tuning or fault tolerance.

The Muesli programming framework is provided as a C++ library. The skeleton tree is build instantiating objects in the template classes provided by Muesli and then a specific method of the top level skeleton may be used to start the actual computation. The implementation is template based, although one template per skeleton is only provided. Muesli eventually generates code for MPI COW/NOW architectures. Recently, it has been improved in such a way both multi core architectures and COW/NOW hosting multicore workstations may be targeted. Targeting of multi cores is achieved by generating OpenMP code in addition to the MPI code when multi cores are addressed [35]. We can therefore state that Muesli supports both multi/many core architectures and heterogeneous[179] architectures.

The big merit of Muesli is its ability to provide algorithmic skeletons with the OO C++ typical programming techniques. This, in turn, makes conventional programmers more comfortable with skeleton based parallel programming.

13.2.3 SkeTo

SkeTo[180] is being maintained by the Tokio University [67, 68]. It has been introduced in early '00s. Although in some intermediate version SkeTo supported also stream parallel

[178] http://www.wi.uni-muenster.de/PI/forschung/Skeletons/
[179] to a limited amount, at least
[180] http://www.ipl.t.u-tokyo.ac.jp/sketo/

skeletons, it is a framework mainly providing data parallel skeletons. Its current version (1.10) only provides data parallel skeletons operating on arrays (vectors) and lists. Skeletons are stateless and the skeleton set could not be extended by the user/programmer. The semantics of the skeletons is formally given in terms of homomorphisms and homomorphism transformations according to the founding Birt-Meertens theory. In Sketo C and C++ code can be reused to implement the sequential skeletons. There is no management of either non functional features or fault tolerance and fine tuning of skeleton applications is completely in charge of the user/programmer.

SkeTo is provided as a C/C++ library and the skeletons are implemented using templates. The framework targets any C/MPI abstract machine and therefore some partial support for heterogeneous architectures is achieved via MPI portability. It is OSS distributed under the open source MIT license. Actually, the main interesting and distinguishing point of SkeTo consists in providing the skeletons as library calls to be spread across "sequential" code. Differently from other frameworks, where parallel program are entirely build around skeletons, SkeTo authors propose to use their skeletons as plain library calls issued within sequential C/C++ programs that–as a side effect–actually improve application performance.

At the moment begin, there is no explicit support for multi/many core architectures, nor explicit support for heterogeneous architectures, but the one provided by MPI.

13.2.4 FastFlow

FastFlow[181] is a stream parallel programming framework introduced in late '00s and maintained by the Dept. of Computer Science of the Univ. of Torino and Pisa by the group of Dr. Aldinucci [18, 20]. FastFlow is a layered programming framework and stream parallel skeletons happen to be the abstractions provided by the upper layer. Lower layers provide much more elementary cooperation mechanisms such as single producer single consumer data flow channels[182] and memory allocators.

FastFlow is implemented as a C++ library and is basically a template based framework. C/C++ code may be reused to program the sequential parts of the skeleton application. A limited form of expandability is granted by the possibility offered to the user/programmer to extend the base skeleton classes provided by the library and by using the very efficient FastFlow communication mechanisms to implement new skeletons by writing the corresponding template in a proper C++ template class. No support for non functional feature management, fault tolerance or application fine tuning is present at the moment (version 1.0).

The abstract machine architecture targeted by FastFlow is any Posix compliant cache coherent shared memory architecture, such as those currently supported by Intel and AMD. Therefore it fully supports multi/many core architectures while not supporting at all COW/NOW or heterogeneous architectures.

The main strengths of FastFlow lie in the excellent efficiency while dealing with fine grain computations–derived from the very efficient lock and fence free[183] implementation of the base communication mechanisms–and in the possibility to exploit data parallelism as well by implementing data parallel computations through stream parallel skeletons as detailed in Sec. 8.2. FastFlow is distributed under OSS license LGPL.

[181] http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:architecture

[182] as well as single producer multiple consumer and multiple producer single consumer

[183] actually, the implementation is not such in case of PowerPC, due to the different architecture of the memory hierarchy. In case of Intel and AMD processors, the implementation is lock and fence free according to the classical semantics of the term: normal operation incurs in no fences and locks when normal operation is performed (i.e. read from non empty and write to non full queue).

13.2.5 SkePu

SkePu[184] is a skeleton programming framework targeting multicore and heterogeneous architectures (multicores with one or more GPUs). It is provided as a C++ template library. Two versions of the library exists, at the moment being:

- SkePu 0.7, including data parallel skeletons only, and
- SkePu 0.7 with StarPu integration, including data parallel skeletons and a task farm skeleton.

The SkePu framework includes data parallel skeletons modelling map, reduce, stencil ("mapoverlap" according to SkePu maintainers), scan and mapreduce skeletons. The StarPu version also includes a task farm skeleton. The implementation of the farm task parallel skeleton is build on top of the StartPu li brary developed by the INRIA researchers of Bordeaux[185]. The library is maintained by the group of Prof. Christoph Kessler at the Univ. of Linkoping, Sweden. Efficiency has been demonstrated with different kind of data parallel programs on multicores and on machines equipped with one or more GPUs [57, 47]. The skeletons provided within SekPu are implemented as operators on data parallel containers. SkePu provides a Vector and a Matrix parallel data container. The `skepu::Vector` class provides the same interface of the `std::Vector` class. The `skepu::Matrix` class is slightly different and also provides collective "primitive" operations such as adding a constant to all the matrix elements, much in the style of the APL[186] operators. SkePu is a very compact and quite engineered library. Is has been experimented to implement data parallel skeletons in FastFlow[187] and belongs to the class of GPU enabled skeleton libraries, that includes Muesli[188] and SkeCL [83].

13.2.6 ASSIST

ASSIST[189] (A Software development System based on Integrated Skeleton Technology) has been developed in early '00s at the Univ. of Pisa by the group of Prof. Vanneschi. ASSIST is a coordination language providing a programming environment rather different from the environments provided by the other skeleton frameworks. It basically provides a single, highly configurable parallel skeleton–the `parmod`–that can be used to instantiate nodes in an arbitrary data flow graph of parmods. Parmods may be used to model both stateful and stateless parallel computations and expandability of the skeleton set is partially guaranteed by the possibility to use any combination of the parmod parameters to implement parallel patterns not originally used in ASSIST. Sequential code reused is allowed for C/C++. Some support for fault tolerance and autonomic management of non functional features is included as well. Actually, ASSIST hosted in 2005 the very first autonomic reconfiguration support in a skeleton based framework [9]. An ASSIST program was able to perform an autonomic reconfiguration of its parallelism degree when changes in the performances of the targeted processing nodes were observed.

ASSIST is exposed to the user as a new programming language. The semantics is provided informally through textual description of the functional and parallelism aspects. The implementation is based on templates and it is written in C/C++. It targets COW/NOW architectures compiling code for a POSIX TCP/IP abstract machine. ASSIST supports

[184] http://www.ida.liu.se/ chrke/skepu/
[185] http://runtime.bordeaux.inria.fr/StarPU/
[186]
[187] http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spm/skepu_ff
[188] http://www.wi.uni-muenster.de/pi/forschung/Skeletons/
[189] http://www.di.unipi.it/Assist.html

heterogeneous architecture targeting although it does not provide any special support for multi/many core nodes.

The main strength of ASSIST consists in its unique and configurable skeleton that can be used to model a variety of different situations. It is distributed under GPL OSS license.

13.3 JAVA BASED FRAMEWORKS

13.3.1 Lithium/Muskel

Lithium has been introduced in early '00s [19]. Later on a subset of Lithium has been re-implemented and the name has been changed in Muskel[190], due to the poor maintainability of the original Lithium code. Muskel only supports stateless stream parallel skeletons (pipeline and farms). The semantics of Muskel has been first provided informally and then in formal way through labeled transition systems [13][191]. Muskel supports both autonomic management of non functional feature and some limited fault tolerance is provided as well. In particular, Muskel introduced the *autonomic manager* concept that has been later on inherited by other skeleton programming frameworks. The autonomic manager of Muskel is able to ensure performance contracts in the form of required parallelism degree and can solve problems related to the failure of nodes used to implement the skeleton application.

Muskel is implemented as a Java library. Reuse of Java code is therefore supported in the sequential parts of the parallel application. It is implemented using macro data flow technology[192] and provides possibility to expand the skeleton set providing a MDFG implementation of the new skeleton as detailed in Sec. 10.6. No fine tuning support is provided to the Muskel user/programmer, however.

Muskel supports automatic normal form reduction (see Sec. 10.3). It targets COW/NOWs supporting Java/RMI. It is distributed under GPL OSS code license. Recently, multi/many core support has been introduced [17].

The main strength of Muskel consists in the support of seamless extension of the skeleton set as well as of automatic normal form execution of stream parallel applications.

13.3.2 Calcium

Calcium[193] has been introduced in late '00s by M. Leyton. Calcium provides stream, data and control parallel stateless skeletons. The semantics of Calcium has been formally provided [28]. Calcium provides no support for skeleton set expandability. It allows reuse of Java code in sequential "muscles"–sequential portions of code in Calcium jargon. Calcium first provided support for application fine tuning, as mentioned in Sec. 5.4.3. Users get reports about the possible bottlenecks in their applications along hints on how to solve the problem [30]. On the other side Calcium does not provide any particular support for autonomic management of non functional features.

Calcium is built on top of the ProActive middleware[194]. It is provided as a Java library and the implementation is based on macro data flow, although in Calcium stacks are used instead of MDFG to compile skeletons. It may target two different abstract architectures, namely ProActive on COW/NOW architectures and ProActive on shared memory multicores. The targeting is explicitly programmed in the source code by using either the distributed or the multicore Calcium "execution environment". As a result, Calcium

[190] http://backus.di.unipi.it/ marcod/wiki/doku.php?id=muskel
[191] Actually this is the semantics exposed in Chap. 12
[192] Lithium has been the first skeleton programming framework using this technology
[193] http://docs.huihoo.com/proactive/3.2.1/Calcium.html
[194] http://proactive.inria.fr/

both supports multi/many core–through execution environment choice–and heterogeneous architectures–exploiting ProActive features. Calcium does not explicitly support fault tolerance, although ProActive natively supports it.

Calcium is released as part of the ProActive middleware suite under INRIA GPL OSS license.

The main innovative features of Calcium consists in the stack based implementation and in the fine tuning support exposing the user/programmer with possible sources of inefficiencies monitored in current application execution.

13.3.3 Skandium

Skandium[195] represents a kind of evolution of the Calcium framework. It is always maintained by M. Leyton, the creator of Calcium, that now moved to the NIC labs[196] in Chile.

Skandium offers the very same features of Calcium, with a few notable differences:

- it is a full Java implementation, that is it does not require any kind of additional library or middleware
- it only supports multi/many core architectures, exploiting Java thread facilities
- it provides controlled access to exceptions raised by the implementation in such a way they can be understood as referred to the skeleton by the user/programmer [62]
- it supports a quite unusual `fork` skeleton representing "data parallelism where a data element is subdivided, each sub-element is computed with a different code, and then results are merged."

13.4 ML BASED FRAMEWORKS

13.4.1 Skipper

Skipper [80] has been developed by J. Serot in late '90s. It represented the evolution of previous skeleton frameworks contributed by the same author [79]. Skipper includes both stream parallel and data parallel stateless skeletons. Being mainly aimed at the support of vision applications, the data parallel skeletons included in Skipper were quite different from common data parallel skeletons. In particular, a *Split-Compute-Merge* and a *Data Farming* data parallel skeletons were included in the skeleton set. The semantics of the skeletons was given in terms of functions (Ocaml code) for the functional part and informally for the parallel semantics. Reuse of sequential Ocaml code was allowed but no expandability of the skeleton set nor autonomic management of non functional features or fault tolerance was supported.

Skipper was implemented as an Ocaml library according to the macro data flow implementation model [40]. The abstract machine targeted was COW/NOW Posix Ocaml architecture. There was no explicit support for multicore architectures nor particular support for heterogeneous architectures, but those regularly supported by the Ocaml programming environment. Skipper has been demonstrated effective to support a variety of computer graphics applications, including "artificial vision" applications. It was released under OSS license.

Skipper is no more supported but it has been mentioned here as it has been important for two main reasons:

[195] http://skandium.niclabs.cl/
[196] http://www.niclabs.cl/

- it first adopted the macro data flow implementation model outside the research framework where this model has been designed, and
- it was explicitly targeting a specific application domain (graphics and vision) and therefore it represents a quite unusual, in some sense *non general purpose* skeleton framework.

13.4.2 OcamlP3L

OcamlP3L[197] [41, 39, 36] has been designed in mid '90s as an Ocaml library implementation of P3L (the skeleton framework discussed in Sec. 13.2.1). OcamlP3L supported the skeletons supports in P3L, that is both stream parallel and data parallel skeletons. However, differently from P3L it is implemented as a library rather than as a new language. The skeleton set provided cannot be easily expanded. Adding a new skeleton requires a simple extension of the Ocaml syntax for skeletons but also the full implementation of a template implementing the skeleton. The functional semantics of the OcamlP3L skeletons is given in term of HOF, while their parallel semantics is given informally. OcamlP3L allow reuse of code written in Ocaml wrapped into OcamlP3L sequential skeleton. It does not support autonomic management of non functional feature nor fault tolerance.

The implementation of OcamlP3L follows the template model. Templates are written using POSIX processes/threads and TCP/IP sockets. Therefore any COW/NOW supporting the Ocaml programming environment may be targeted and heterogeneous architectures are also supported exploiting the Ocaml bytecode virtual machine.

OcamlP3L does not support specifically multi core architectures, even if multi core nodes may *de facto* used as multiple processing elements anyway[198]. It is releases under OSS license. From 2010, OcamlP3L is no longer maintained and CamlP3L[199] is maintained instead.

The main contribution of OcamlP3L was the usage of functional closure transmission across distributed Ocaml interpreters used to specialize the different processing elements used in such a way they behave as specific processes of the specific templates used to compile the skeletons of the user application.

13.5 OTHER FUCNTIONAL FRAMEWORKS

13.5.1 skel (Erlang)

skel is a library written in pure Erlang and developed and maintained within the STREP FP7 project ParaPhrase[200]. The library provides stream and data parallel skeletons similar to the ones provided in FastFlow (Sec. 13.2.4). The skeleton nesting framework is implemented through the concept of "workflow", that is of an expression of (possibly nested) skeleton components, operating onto some input list to produce an output result list.

The library is a pure Erlang library and, as such, runs naturally on shared memory multicores. There is in principle the possibility to run skeletons across different Erlang interpreters running on different, networked multicores, although no primitive support is provided at the moment[201] for this.

An extension of the **skel** library has been developed that supports GPUs through SkePu (Sec. 13.2.5). The support is limited, in that the kernels processing data on the GPU must

[197] http://ocamlp3l.inria.fr/eng.htm
[198] but this is true for most skeleton based frameworks, actually
[199] http://camlp3l.inria.fr/eng.htm
[200] `http://www.paraphrase-ict.eu`
[201] as of 2014

be written according to the (C/C++) syntax of the macros provided by SkePu and the only data type supported is the list of floats[202]. This extension is available on the project web site (follow "Deliverables" tab and look for D2.7 or D2.4 software releases) and is currently in the process of being more carefully integrated in **skel** using a native support for OpenCL kernels in Erlang developed by the ESL ParaPhrase project partners.

The **skel** library demonstrated pretty good scalability on state-of-the-art multicore architectures (e.g. AMD Magny Cours or Intel Sandy/Ivy bridge).

13.6 COMPONENT BASED FRAMEWORKS

13.6.1 GCM Behavioural skeletons

Behavioural skeletons (BS) have been introduced in late '00s[203] by the researchers at the CoreGRID Programming model Institute[204] [10, 11, 44]. A reference implementation was produced by the EU funded STREP project GridCOMP[205]. Behavioural skeletons provide stream parallel and data parallel components. In the last version of their implementation, farm, pipeline and data parallel (with or without stencil) were provided. Those skeletons were equipped (therefore the name "behavioural" skeletons) by autonomic managers taking care of skeleton non functional features as described in Sec. 10.5.

The behavioural skeletons included in the reference implementation are stateless. They can be arbitrary composed. The non functional concern managed in an autonomic way is performance. The semantics is provided on an informal basis. Code reuse is guaranteed as any GCM[25] component may be used as "sequential" building block of BS components and Java code may be re-used to model sequential parts of the components. There is no explicit support for fault tolerance in BS. Application fine tuning is supported via the possibility to customize the JBoss rules constituting the "program" of the autonomic managers associated to the skeletons.

The abstract machine targeted is the ProActive 4.0.1[206] middleware platform that runs on most distributed architecture, including COW/NOWs and GRIDs. There is no explicit support for multi/many core architectures. The implementation of BS is template based and the host language is definitely Java. The BS implementation is released under LGPL OSS.

The main strengths of BS can be individuated in two aspects:

- On the one hand, in the autonomic manager implementation careful and appropriate design, including programmability through abstract rule in the business rule engine included in each one of the BS managers. The autonomic manager design in BS opened perspective in the management of several different and important non functional concerns, including performance, security, fault tolerance and power management.

- On the other hand, in the evolution of the manager concept in such a way that both hierarchies of autonomic managers all taking care of the same non functional concern [15] and multiple (hierarchies of) autonomic managers taking care of different non functional concerns can be used [16].

[202] this means kernels may only process list of floats to produce floats (reduce) or lists of floats (map)
[203] http://gridcomp.ercim.eu/content/view/26/34/
[204] http://www.coregrid.net
[205] http://gridcomp.ercim.eu/
[206] http://proactive.inria.fr/

13.6.2 LIBERO

LIBERO[207] is a *Lightweight Implementation of BEhaviouRal skeletOns* [6, 5]. It has been developed by the designers of the behavioural skeletons discussed in Sec. 13.6.1 to make available a much more compact and manageable framework to support experimentation with BS than the one provided by the reference implementation on top of ProActive/GCM.

LIBERO supports stream parallel only skeletons, at the moment, namely pipeline and farm. It has been designed and implemented to make easy the introduction of other behavioural skeletons, but no facility is provided to user/programmer to support expandability. The semantics of the skeletons is given on an informal basis. Being the whole implementation in Java, Java code re-used is ensured in sequential skeleton wrappings. At the moment there is no support for application fine tuning nor to fault tolerance although fault tolerance policies can be seamlessly integrated in manager rules. LIBERO uses the same JBoss business rule engine used in reference implementation of BS on top of ProActive/GCM, actually. LIBERO does not implement traditional component, however. Rather it uses Java POJOs as components and relies on a small RMI based run time to support deployment of these POJO components and to support component life cycle operations.

LIBERO provides behavioural skeletons as a Java library and targets any architecture supporting Java, RMI and TCP/IP sockets. It is distributed as open source. As in BS reference implementation, there is no explicit support for multi/many core architectures.

The main strength of LIBERO is in its lightweight implementation providing anyway a complete behavioural skeleton framework. LIBERO design has been conceived to support expandability, as stated above, and this also constitutes a strength of this framework even if the expandability is not actually exposed to the user/programmer[208].

13.7 *QUASI*-SKELETON FRAMEWORKS

We just outline here some notable programming frameworks that have been developed completely outside the algorithmic skeleton community, that are very popular and that somehow support a little bit of the skeleton technologies.

13.7.1 TBB

TBB is the Threading Building Block library by Intel[209]. The library is aims at supporting multi-core programmers in *taking advantage of multi-core processor performance without having to be a threading expert*.

In the initial versions of the library a few entries could be regarded as a kind of algorithmic skeletons, namely those using thread to express parallelism in loop iterations and therefore implementing a kind of `map` skeleton and those modelling variable length pipelines (`tbb::pipeline`).

From version 3.0 (released in May 2010), a set of *design patterns* have been included in TBB. These design patterns include a *reduction*, a *divide and conquer* and a *wavefront* pattern. Some of these patterns may be implemented using primitive skeletons library calls of TBB. As an example, the reduction pattern implementation is described in terms of the `tbb::parallel_reduce` library entry. Implementations of other patterns is described in terms of other basic TBB mechanisms (library entries) which look like more as templates (or templates building blocks) than as skeletons if our perspective is adopted. As an example,

[207] http://backus.di.unipi.it/ marcod/wiki/doku.php?id=libero#libero
[208] but this is a consequence of the design of LIBERO that has been conceived as an experimental BS framework to test improvements and advanced in BS implementation.
[209] http://threadingbuildingblocks.org

these mechanisms include `tbb::parallel_invoke` to fork a new thread computing a function/procedure call or the `tbb::parallel_for` to compute for iterations in parallel.

The interested reader may refer to the documentation appearing on the "Intel Threading Building Blocks 3.0 for Open Source web site"[210].

13.7.2 TPL

Microsoft provides different tools to support parallel programming in the .NET framework, including the Task Parallel Library (TPL). TPL provides some support for creating and executing concurrent tasks (suitable to support stream and control parallel skeletons) as well as for executing in parallel loop iterations (suitable to implement data parallel skeletons)[211]. The kind of support provided by TPL looks like more basic than the one provided by TBB although some clear functional style inspiration of the interface seems to promise an higher level of abstraction with respect to TBB C++ style methods and classes.

13.7.3 OpenMP

OpenMP is the *de facto* standard as far as multi core programming frameworks are concerned[212]. Despite the fact OpenMP only provides loop parallelization and some kind of task parallelism, it has gained an outstanding popularity as it is provided as an annotation (pragma) extension of C++ (and FORTRAN) languages. Parallelizing a loop using OpenMP is as trivial as adding a line such as

```
#pragma omp parallel for
```

immediately before the `for` statement of the loop. However, writing *efficient* OpenMP programs is much more difficult, as it requires deep understanding of all the additional parameters of the OpenMP pragmas as well as a complete understanding of the concurrent accesses performed on shared data structures.

This notwithstanding OpenMP may be used to implement both stream parallel and data parallel programs with a moderate programming effort. Fine tuning–especially the tuning activities related to performance tuning–are much more difficult than having an OpenMP program running.

[210] http://threadingbuildingblocks.org/documentation.php
[211] http://msdn.microsoft.com/en-us/library/dd460717.aspx
[212] http://openmp.org/wp/

APPENDIX A

STRUCTURED PARALLEL PROGRAMMING IN EU PROJECTS

EU (European Union) has funded and currently funds projects aimed at advancing the technology and research results in Europe in several different fields, including ICT.

In the recent years, three framework programs have been active:

- **FP6**, Sixth Framework Programme covering Community activities in the field of research, technological development and demonstration for the period 2002 to 2006,
- **FP7**, the Seventh Framework Programme for Research and Technological Development, covering the next period (2007-2013), and
- **H2020**, Horizon 2020, covering the period 2014–2020, which–according to the official EU documents–"is the financial instrument implementing the Innovation Union, a Europe 2020 flagship initiative aimed at securing Europe's global competitiveness".

All those programs run a number of different "call for proposals" where researcher in academia and industry submitted project proposals fitting the specific objectives of the call. The EU officers, with the help of qualified scientific reviewers, select the best project submitted for each call and these projects are then funded with public (i.e. tax) money from the EU. The features of funded project varies (in size and kind) and typically included at least three specific types of projects:

- STREP (Specific Targeted Research Project) projects, including a small number of partners (between 5 and 10, usually), lasting 3 years and receiving funds in the 2-4 million euro range,
- IP (Integrated Project) projects, with larger consortium (10-15 partners) and budget (up to 10 million euros), and

- NOE (Network Of Excellence) projects, with even larger number of partners and budget.

The first two instruments are aimed at funding research and development activities directly producing some kind of concrete result/advance in the specific call field(s), while the NOE instrument is more aimed at creating and enforcing synergies in between the existing research groups in Europe on specific themes.

Within these frameworks, different projects have been funded that investigated, improved or simply adopted structured parallel programming ideas and tools. In this chapter, we want to briefly outline a small number of these projects that directly involved the authors of these notes and were/are related to structured parallel programming themes. In particular, we will consider:

- *CoreGRID*[1], an NOE funded in 2005-2009 and involving more that 40 research institutions in Europe active on GRID research themes,
- *GridCOMP*[2], a spin off STREP project of CoreGRID, funded in 2006-2009 to design and implement a parallel programming framework targeting grid (distributed) architectures,
- *ParaPhrase*[3], a STREP project funded in 2011-2013 to investigate the possibility to implement efficient parallel programming frameworks exploiting design patterns and skeletons and targeting CPU+GPU heterogeneous architectures, and
- *REPARA*[4], a STREP project funded in 2013-2016 to investigate the possibility to parallelize regular C++ code such that performance *and* power consumption are improved while targeting architectures sporting multicore CPUs, GPUs, FPGAs and, possibly, DSPs.

While the concrete results achieved in these projects can be appreciated looking at the relevant publications usually accessible through the project web sites, in the following sections we'll give an overview of the main features of these project, in particular of the features related to structured parallel programming.

A.1 COREGRID

CoreGRID was a NOE and its activities were structured in "Institutes" (AKA work packages). The activities of each institute were related to a different set of aspects among those related to computational grids:

- **Knowledge & Data Management**
- **Programming Model**

[1] http://coregrid.ercim.eu/mambo/
[2] http://gridcomp.ercim.eu/
[3] http://paraphrase-ict.eu/
[4] http://repara-project.eu/

- **Architectural Issues: Scalability, Dependability, Adaptability**
- **Grid Information, Resource and Workflow Monitoring Services**
- **Resource Management and Scheduling**
- **Grid Systems, Tools and Environments**

The Programming model Institute was aimed to

> to deliver a definition of a lightweight, component programming model that can be usefully exploited to design, implement and run grid applications. We will work on the hypothesis that the component based programming model's main aim is to efficiently face the new challenges in terms of programmability, interoperability, code reuse and efficiency that mainly derive from the features that are peculiar to GRID, namely, heterogeneity and dynamicity. GRID programmability, in particular, represents a big challenge. GRID programs cannot be written using normal programming models and tools, unless the programmer is willing to pay a high price in terms of programming, program debugging and program tuning efforts. In the design and implementation of the new component based programming model we will therefore try to move as much as possible these activities away from the programmer.

In particular, the specific goals of the Institute were stated as:

- Definition of a programming model suitable for the implementation of the single component model
- Definition of the component system in such a way that interoperability with existing, commonly agreed standards is guaranteed
- Definition of a component composition such that complex, multidisciplinary applications can be written as composition of building block components, possibly obtained by suitably wrapping existing code. Component composition must support and guarantee that scalability is achieved
- Definition of clear component and component composition semantics allowing a user to prove the correctness of the transformation and/or improvement techniques derived and designed for the lightweight component model
- Definition of performance/cost models allowing the development of tools reasoning about component and component composition programs.

Within the institute activities, the general idea of Behavioural skeletons discussed in Sec. 10.5 has been designed along with a the general features of a specific component model–the GCM, Grid Component Model–aimed at providing grid application programmers the features listed above.

The research teams involved in the Institute activities included Univ. of Pisa (Institute leader, I), INRIA (F), Univ. of Passau (D), Muenster (D), Lisboa (P), Sannio (I), Queen's Univ. of Belfast (UK), Vrie Univ. of Amsterdam (NL), Imperial College London (UK), EIA-FR (CH), HRLS (D).

After conceiving the general features of both the component model (GCM) and of the behavioural skeletons (BS), members of the institute submitted a project proposal specifically aimed at providing a reference implementation of GCM and BS. The project–GridCOMP–was eventually funded and within its activities both GCM and BS reference implementations were developed (see Sec. A.2 below). It's wort pointing out that both ideas somehow spread from a huge research community, after convenient discussions and investigations related to the identification of the specific features to be included. Also, it's worth pointing out this has been the first point where component models and structured programming have been

identified as the main feasible models to target grids and, consequently, distributed architectures (COW/NOW).

A.2 GRIDCOMP

GridCOMP is a Specific Targeted Research Project on Grid programming with components, funded by the European Commission in the sixth Framework Programme from June 2006 to February 2009. The main objective of GridCOMP was the design and implementation of a component based framework suitable to support the development of efficient grid applications: the Grid Component Model (GCM). The framework implements a kind of "invisible grid" concept as it properly abstracts all those specific grid related implementation details that usually require high programming efforts to handle.

GridCOMP provides the reference implementation of the GCM. This prototype features a component framework allowing creation of remote components and their remote access in a transparent manner supporting collective communications. This prototype also includes a deployment framework providing interoperability with several grid schedulers and middlewares.

The *Non Functional Component Features* (see Fig. A.1) provide a prototype of behavioural skeletons modelling common parallelism exploitation patterns and implementing an autonomic manager which takes care of ensuring user-supplied performance contracts (SLAs).

The behavioural skeletons themselves are provided as composite GCM components. Users can instantiate such composites by providing other components to specify the functional part of the code. Behavioural skeletons will enable users to develop and deploy efficient grid parallel applications in a sensibly shorter time with respect to the time required to develop and deploy applications with similar efficiency built from scratch by the application programmers. Actually, GCM composite components implementing behavioural skeletons automatically deploy on target architecture while optimising usage of the existing resources.

The initial set of behavioural skeletons developed within GridCOMP, implement parameter sweeping, master/worker as well as several types of common data parallel patterns. A new component-oriented development methodology has been created. This methodology enables users to build large-scale Grid systems by integrating independent and possibly distributed software components, via well-defined interfaces, into higher level composite components. The main benefit from such an approach is improved productivity. As an implementation of this methodology, the Grid integrated development environment (GIDE) has been built. It is tightly integrated with Eclipse software framework and was designed to empower the end user with all the tools necessary to compose, deploy, monitor, and steer Grid applications.

Four use cases have been developed and promote the capabilities of the GridCOMP framework. They represent test cases as well as demo applications with respect to the GridCOMP component framework. They also point out the advanced features provided by the framework and, through their complete documentation, illustrate how they can be exploited in real world scenarios. The use cases represent a jump start for people new to

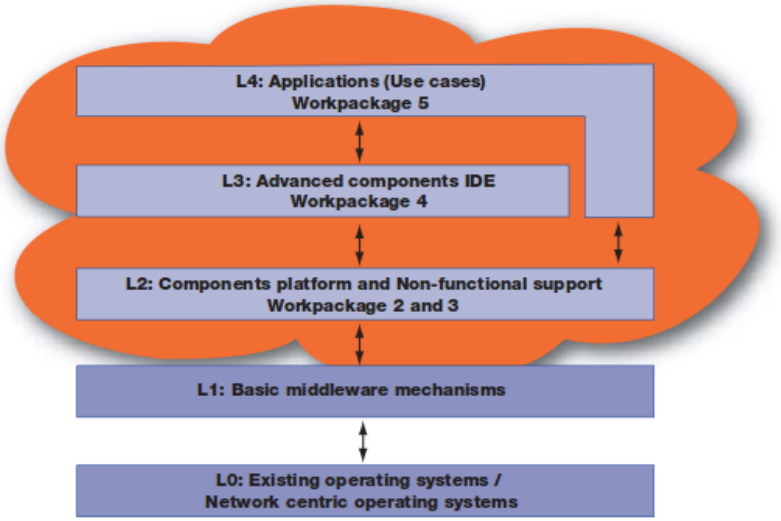


Figure A.1 Overall structure of the GridCOMP project

GCM and are thus vital for the success of the framework. Industrial partners make use of their respective use cases to highlight the benefits of GridCOMP, both internally and to their customers. GridCOMP also disseminated the GCM through many channels. Training materials introducing applications and frameworks produced within GridCOMP have been published on-line. Additionally, GridCOMP held several successful workshops and tutorials in Europe and China in order to get users familiar with the GCM platform and to ensure the impact of the results to a worldwide audience via our international partners. In the meantime, standardisation of the GCM definition in four different standards made headway. Different items have been officially accepted as ETSI standards including "GCM Interoperability Deployment", "GCM Interoperability Application Description", "GCM Fractal ADL" and "GCM Management API'.

Partners involved in the Non Functional Features work package of the project implemented several distinct behavioural skeletons. A behavioural skeleton is a composite GCM component modelling a well-known parallelism exploitation pattern (such as task farm (see Fig. A.2) or data parallel patterns) that also provides the user with an autonomic manager taking care of ensuring a user supplied performance contract. GCM programmers can use the skeleton by simply providing the worker component (i.e., the component computing a single task in the task farm or a partition of the final result in a data parallel pattern) and the performance contract. Performance contracts include requiring throughput not smaller than a given constant and/or not higher than another user-supplied constant and can be roughly intended as first order logic formulas over execution parameters. Autonomic managers within the GCM composites implement a classical control cycle: the current composite component behaviour is monitored and in case a violation of the user contract is detected, proper corrective actions are planned and executed. The control cycle executed within the autonomic manager allows component execution to adapt to varying features of both the Grid target architecture and the application requirements themselves. Experiments on Grid5000 with both synthetic applications and real use cases demonstrated both task farm and data parallel behavioural skeleton (and consequently composite autonomic manager) functionality and efficiency. Partners of the use case work package were able to use these behavioural skeletons while programming the use case prototypes.

The behavioural skeleton approach adopted within the work package can be clearly extended in such a way that i) multiple non-functional features are concurrently managed,

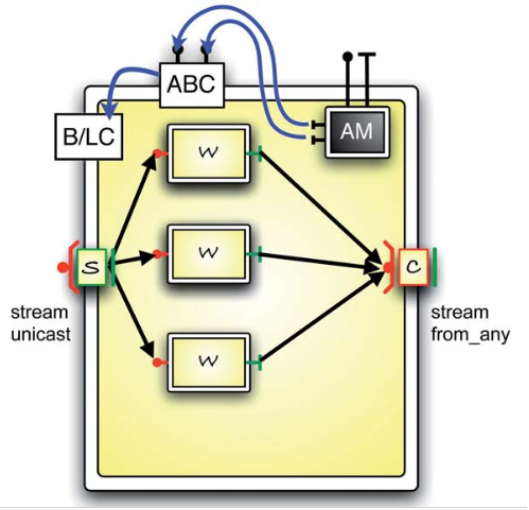


Figure A.2 Task farm behavioural skeleton in GridCOMP

and ii) hierarchical composition of autonomic managers is exploited to achieve autonomic management of large Grid applications programmed using a nesting/composition of several distinct behavioural skeletons. While the latter point has already been partially investigated within the Non Functional Com- ponent Features work package and preliminary results have already been achieved, the former topic still needs consistent research activity. The availability of a GCM programming framework supporting both features will provide users with advanced environments where all the relevant, non functional features are handled within the environment itself, much in the perspective of a cloud architecture for component based applications.

A.3 PARAPHRASE

ParaPhrase is a STREP FP7 project funded by EU in the period 2011-2014 (`http://www.paraphrase-ict.eu`).

The ParaPhrase project aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to meet application needs and hardware availability (see Fig. A.3). The project exploits new developments in the implementation of parallel patterns that will allow us to express a variety of parallel algorithms as compositions of lightweight software components forming a collection of virtual parallel tasks. Components from multiple applications are instantiated and dynamically allocated to the available hardware resources through a simple and efficient software virtualisation layer. In this way, the project promotes adaptivity, not only at an application level,

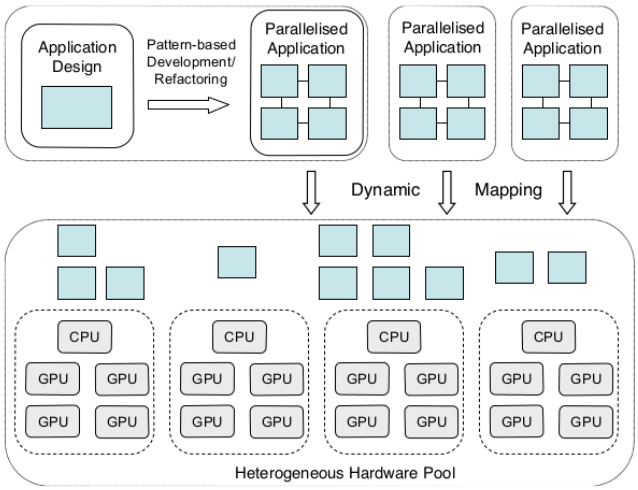


Figure A.3 Overall view of the ParaPhrase

but also at a system level. Finally, it develops virtualisation abstractions across the hardware boundaries, allowing components to be dynamically re-mapped to either CPU/GPU resources on the basis of suitability and availability in a simple and straightforward way.

Within ParaPhrase, a methodology has been defined to support the development of parallel applications targeting heterogeneous parallel architectures including multi-core CPUs and many-core GPUs. The methodology, outlined in Fig. A.4, may be summarized as follows:

- programmers introduce parallelism in (existing) sequential code by using proper refactoring tools introducing parallel patterns;
- the parallel patterns are implemented with (compiled to) algorithmic skeletons;
- different versions of the algorithmic skeleton implementation exist targeting different hardware;
- proper tools pick up the correct implementation of the skeletons, depending on the features of the application at hand and of the target hardware, and
- dynamically remap and reconfigure the application such that the available hardware resources may be better exploited in the different phases of the application runs.

The general ParaPhrase parallel programming methodology has been implemented within the project following two distinct tracks:

- on the one hand, C/C++ code has been considered for the sequential portions of the parallel applications and patterns have been implemented through proper calls to the FastFlow (see Sec. 13.2.4[5]) open source skeleton framework
- on the other hand, Erlang code has been considered for the business logic of the parallel application(s) and the **skel** library (see Sec. 13.5.1[6]) has been used to implement the parallel patterns introduced through refactoring.

[5]available at `http://calvados.di.unipi.it/`
[6]available at `https://github.com/ParaPhrase/skel`

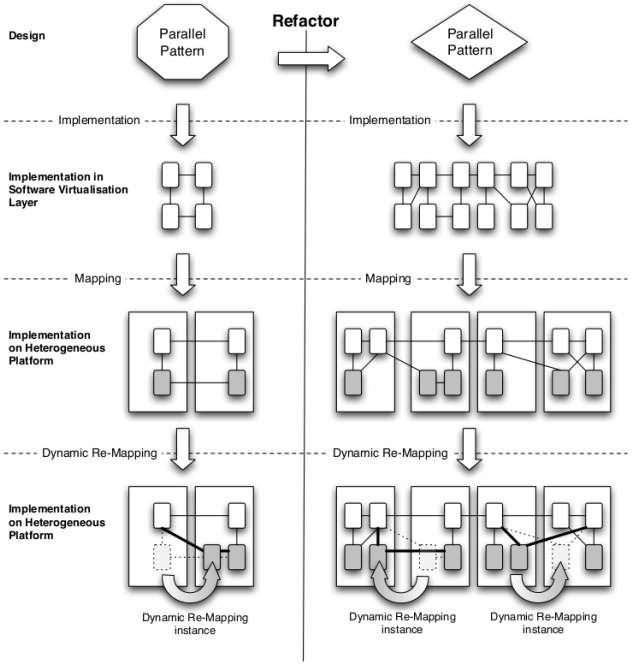


Figure A.4 Overall view of the ParaPhrase methodology

The different concepts used and exploited in the ParaPhrase methodology have been summarized in different parts of these notes. In particular:

- the patterns considered within ParaPhrase to parallelize applications are *de-facto* a subset of the patterns considered in Chap. 6. Some additional patterns derived from algorithmic skeletons as discussed in Chap. 3 have also been taken into account.

- the implementation of patterns exploits algorithmic skeletons (Chap. 3) implemented according to a template based approach (Chap. 8).

- Performance models such as those introduced in Chap. 5 are used to devise proper mapping and re-mapping strategies when implementing patterned parallel applications on heterogeneous architectures.

- Refactoring rules such as the rewriting rules discussed in Sec. 10.1 have been implemented both to introduce parallelism (e.g. the farm introduction rule) or to improve parallelism (e.g. map fusion rule).

- Last but not least, principles from the Behavioural skeleton ideas have been used to implement part of the FastFlow mechanisms used to i) target different kind of processing resources within the same application [76] and ii) to support adaptivity in FastFlow skeletons (e.g. supporting dynamic adjustment of number of workers in a farm/map skeleton).

The ParaPhrase methodology has been exploited to develop and implement different applications, kernels and use cases within the project "Use cases" work package, including Haar transform, Molecular dynamics, Waste water processing, Video noise processing and Discrete optimization applications.

A.4 REPARA

REPARA is an FP7 STREP project funded for the period 2013-2016 (`http://www.repara-project.eu`). The REPARA project joins forces of experts in software engineering methodology, development tools, computer hardware design and analysis, all working hand-in-hand with industrial end-users to achieve a unified programming model for heterogeneous computers developing also the required automated software support tools. Relative to the base line of a sequential algorithm executed on a current general-purpose processor, REPARA expects to achieve at least a 50% reduction of energy consumption combined with a performance improvement of at least by a factor of two. REPARA will also allow for an increased productivity realizing designs in half of the development time that would be required using non-unified programming methods for the different components of a heterogeneous system. Combined, REPARA will lead to fourfold gain in efficiency (quality gain per development time) for energy savings and performance. These objectives will be verified in 5 real-world use cases in the domains of railway, healthcare, industrial maintainance and robotics. Achieving such ambitious objectives will create opportunities for the involved contractors and the European citizens on various strands. The industrial contractors EVOPRO and IXION are active in the targeted use cases and will earn improved competitiveness over other players in their respective markets turning the REPARA results into higher profits and increased employment. The 5 academic contractors will not only strengthen their scientific reputation as leading experts in the field, but their tools will facilitate them to also commercially exploit their research efforts through tool licensing and industry-academia follow-on projects. Finally, European citizens will profit directly from safer rail transport, more sophisticated health care and a reduced power bill. Last but not least, the 50% reduction of energy consumption will have considerable environmental impact given that 2% of the world-wide CO2 footprint was caused by server infrastructures in 2007 and is increasing year by year.

REPARA concept is simple and, in the meanwhile, effective: standard C++ code is transformed to conveniently expose those parts that may benefit from parallel execution on different target resources. Then the application code is partitioned, with different parts targeting CPU cores, GPUs, FPGAs or DSPs and all together contributing to the computation of the final application result(s). Specific targeting of heterogeneous resources is driven by proper performance and power consumption models. Hardware targeting is implemented through a structured parallel run time support based on FastFlow (Sec. 13.2.4).

The project is just started, but despite the fact that most of the work has still to be performed, different activities in the project may be reconducted to structured parallel programming techniques. IN particular:

- code transformation and application partitioning will rely on refactoring techniques such as those already adopted in ParaPhrase (Sec. sec:paraphrase). Moreover, effort is made to recognize "structured parallel" kernels within the original code and to transform therefore the original code into code calling proper algorithmic skeletons implemented in the run time abstracting the heterogenous resources available.

- patterns and skeletons are considered–besides being considered suitable ways to express parallelism in the source code–as convenient ways to implement the execution

of the kernels from the partitioned applications onto the heterogeneous parallel hardware.

APPENDIX B

FASTFLOW

B.1 DESIGN PRINCIPLES

FastFlow[1] has been designed to provide programmers with efficient parallelism exploitation patterns suitable to implement (fine grain) stream parallel applications. In particular, FastFlow has been designed

- to promote high-level parallel programming, and in particular skeletal programming (i.e. pattern-based explicit parallel programming), and
- to promote efficient programming of applications for multi-core.

The whole programming framework has been incrementally developed according to a layered design on top of Pthread/C++ standard programming framework and targets shared memory multicore architectures (see Fig. B.1).

A first layer, the **Simple streaming networks** layer, provides lock-free Single Producers Single Consumer (SPSC) queues on top of the Pthread standard threading model.

A second layer, the **Arbitrary streaming networks** layer, provides lock-free implementations for Single Producer Multiple Consumer (SPMC), Multiple Producer Single Consumer (MPSC) and Multiple Producer Multiple Consumer (MPMC) queues on top of the SPSC implemented in the first layer.

Eventually, the third layer, the **Streaming Networks Patterns** layer, provides common stream parallel patterns. The primitive patterns include pipeline and farms. Simple

[1]see also the FastFlow home page at `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`

specialization of these patterns may be used to implement more complex patterns, such as divide and conquer, map and reduce patterns.

Parallel application programmers are assumed to use FastFlow directly exploiting the parallel patterns available in the Streaming Network Patterns level. In particular:

- defining sequential concurrent activities, by sub classing a proper FastFlow class, the `ff_node` class, and
- building complex stream parallel patterns by hierarchically composing sequential concurrent activities, pipeline patterns, farm patterns and their "specialized" versions implementing more complex parallel patterns.

The `ff_node` sequential concurrent activity abstraction provide suitable ways to define a sequential activity processing data items appearing on a single input channel and delivering the related results onto a single output channel. Particular cases of `ff_nodes` may be simply implemented with no input channel or no output channel. The former is used to install a concurrent activity *generating* an output stream (e.g. from data items read from keyboard or from a disk file); the latter to install a concurrent activity *consuming* an input stream (e.g. to present results on a video or to store them on disk).

The pipeline pattern may be used to implement sequences of streaming networks $S_1 \rightarrow \ldots \rightarrow S_k$ with S_k receiving input from S_{k-1} and delivering outputs to S_{k+1}. S_i may be either a sequential activity or another parallel pattern. S_1 must be a stream generator activity and S_k a stream consuming one.

The farm pattern models different embarrassingly (stream) parallel constructs. In its simplest form, it models a master/worker pattern with workers producing no stream data items. Rather the worker consolidate results directly in memory. More complex forms including either an emitter, or a collector of both an emitter and a collector implement more sophisticated patterns:

- by adding an emitter, the user may specify policies, different from the default round robin one, to schedule input tasks to the workers;

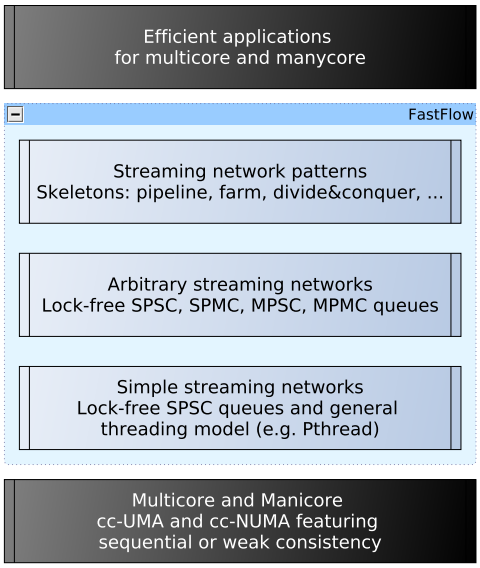


Figure B.1 Layered FastFlow design

- by adding a collector, the user may use worker actually producing some output values, which are gathered and delivered to the farm output stream. Different policies may be implemented on the collector to gather data from the worker and deliver them to the output stream.

In addition, a feedback channel may be added to a farm, moving output results back from the collector (or from the collection of workers in case no collector is specified) back to the emitter input channel. The feedback channel may only be added to the farm/pipe *at the root of the skeleton tree.*

Specialized version of the farm may be used to implement more complex patterns, such as:

- divide and conquer, using a farm with feedback loop and proper stream items tagging (input tasks, subtask results, results)
- MISD (multiple instruction single data, that is something computing $f_1(x_i), \ldots, f_k(x_i)$ out of each x_i appearing onto the input stream) pattern, using a farm with an emitter implementing a broadcast scheduling policy
- map, using an emitter partitioning an input collection and scheduling one partition per worker, and a collector gathering sub-partitions results from the workers and delivering a collection made out of all these results to the output stream.

It is worth pointing out that while using plain pipeline and farms (with or without emitters and collectors) actually can be classified as "using skeletons" in a traditional skeleton based programming framework, the usage of specialized versions of the farm streaming network can be more easily classified as "using skeleton templates", as the base features of the FastFlow framework are used to build new patterns, not provided as primitive skeletons[2].

Concerning the usage of FastFlow to support parallel application development on shared memory multicores, the framework provides two abstractions of structured parallel computation:

- a "skeleton program abstraction" which is used to implement applications completely modelled according to the algorithmic skeleton concepts. When using this abstraction, the programmer write a parallel application by providing the business logic code, wrapped into proper `ff_node` subclasses, a skeleton (composition) modelling the parallelism exploitation pattern of the application and a single command starting the skeleton computation and awaiting for its termination.
- an "accelerator abstraction" which is used to parallelize (and therefore accelerate) only some parts of an existing application. In this case, the programmer provides a skeleton (composition) which is run on the "spare" cores of the architecture and implements a parallel version of the business logic to be accelerated, that is the computing a given $f(x)$. The skeleton (composition) will have its own input and output channels. When an $f(x)$ has actually to be computed within the application, rather than writing proper code to call to the sequential f code, the programmer may insert code asynchronously "offloading" x to the accelerator skeleton. Later on, when the result of $f(x)$ is to be used, some code "reading" accelerator result may be used to retrieve the accelerator computed values.

This second abstraction fully implements the "minimal disruption" principle stated by Cole in his skeleton manifesto [38], as the programmer using the accelerator is only required to

[2]Although this may change in future FastFlow releases, this is the current situation as of FastFlow version 1.1

program a couple of `offload/get_result` primitives in place of the single $\ldots = f(x)$ function call statement (see Sec. B.7).

B.2 INSTALLATION

Before entering the details of how FastFlow may be used to implement efficient stream parallel (and not only) programs on shared memory multicore architectures, let's have a look at how FastFlow may be installed[3].

The installation process is trivial, actually:

1. first, you have to download the source code from SourceForge (`http://sourceforge.net/projects/mc-fastflow/`)
2. then you have to extract the files using a `tar xzvf fastflow-XX.tgz` command, and
3. eventually, you should use the top level directory resulting from the *tarxzvf* command as the argument of the `-I` flag of `g++`.

As an example, the currently available version (1.1) is hosted in a `fastflow-1.1.0.tar.gz` file. If you download it and extract files to your home directory, you should compile FastFlow code using the flags `g++ -I $HOME/fastflow-1.1.0 -lpthread` in addition to any other flags needed to compile your specific code. Sample `makefiles` are provided both within the `fastflow-1.1.0/tests` and the `fastflow-1.1.0/examples` directories in the source distribution.

B.3 TUTORIAL

As all programming frameworks tutorials, we start with a *Hello world* code. In order to implement our hello world program, we use the following code:

```
1  #include <iostream>
2  #include <ff/pipeline.hpp>
3
4  using namespace ff;
5
6  class Stage1: public ff_node {
7  public:
8
9      void * svc(void * task) {
10         std::cout << "Hello world" << std::endl;
11         return NULL;
12     }
13 };
14
15 int main(int argc, char * argv[]) {
16
17     ff_pipeline pipe;
18     pipe.add_stage(new Stage1());
19
20     if (pipe.run_and_wait_end()<0) {
21         error("running pipeline\n");
```

[3]We only detail instructions needed to install FastFlow on Linux/Unix/BSD machines here. A Windows port of FastFlow exist, that requires slightly different steps for the installation.

```
22          return -1;
23      }
24
25      return 0;
26  }
```

ffsrc/helloworldSimple.cpp

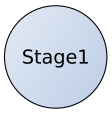
Line 2 includes all what's needed to compile a FastFlow program just using a pipeline pattern and line 4 instruct compiler to resolve names looking (also) at `ff` namespace. Lines 6 to 13 host the application business logic code, wrapped into a class sub classing `ff_node`. The `void * svc(void *)` method[4] wraps the body of the concurrent activity resulting from the wrapping. It is called every time the concurrent activity is given a new input stream data item. The input stream data item pointer is passed through the input `void *` parameter. The result of the single invocation of the concurrent activity body is passed back to the FastFlow runtime returning the `void *` result. In case a `NULL` is returned, the concurrent activity actually terminates itself. The application main only hosts code needed to setup the FastFlow streaming network and to start the skeleton (composition) computation: lines 17 and 18 declare a pipeline pattern (line 17) and insert a single stage (line 18) in the pipeline. Line 20 starts the computation of the skeleton program and awaits for skeleton computation termination. In case of errors the `run_and_wait_end()` call will return a negative number (according to the Unix/Linux syscall conventions).

When the program is started, the FastFlow RTS accomplishes to start the pipeline. In turn the first stage is started. As the first stage `svc` returns a `NULL`, the stage is terminated immediately after by the FastFlow RTS.

If we compile and run the program, we get the following output:

```
1 ffsrc$ g++ -lpthread -I
    /home/marcod/Documents/Research/CodeProgramming/fastflow-1.1.0
    helloworldSimple.cpp -o hello
2 ffsrc$ ./hello
3 Hello world
4 ffsrc$
```

There is nothing parallel here, however. The single pipeline stage is run just once and there is nothing else, from the programmer viewpoint, running in parallel. The graph of concurrent activities in this case is the following, trivial one:



A more interesting "HelloWorld" would have been to have a two stage pipeline where the first stage prints the "Hello" and the second one, after getting the results of the computation of the first one, prints "world". In order to implement this behaviour, we have to write two sequential concurrent activities and to use them as stages in a pipeline. Additionally, we have to send something out as a result from the first stage to the second stage. Let's assume we just send the string with the word to be printed. The code may be written as follows:

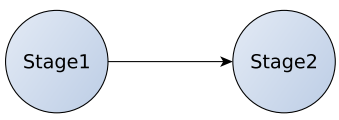
```
1 #include <iostream>
2 #include <ff/pipeline.hpp>
3
4 using namespace ff;
```

[4] we use the term `svc` as a shortcut for "service"

```
5
6  class Stage1: public ff_node {
7  public:
8
9      void * svc(void * task) {
10         std::cout << "Hello " << std::endl;
11         char * p = (char *) calloc(sizeof(char),10);
12         strcpy(p,"World");
13         sleep(1);
14         return ((void *)p);
15     }
16 };
17
18 class Stage2: public ff_node {
19 public:
20
21     void * svc(void * task) {
22       std::cout << ((char *)task) << std::endl;
23       free(task);
24       return GO_ON;
25     }
26 };
27
28 int main(int argc, char * argv[]) {
29
30     ff_pipeline pipe;
31     pipe.add_stage(new Stage1());
32     pipe.add_stage(new Stage2());
33
34     if (pipe.run_and_wait_end()<0) {
35         error("running pipeline\n");
36         return -1;
37     }
38
39     return 0;
40 }
```

ffsrc/hello2stages.cpp

We define two sequential stages. The first one (lines 6–16) prints the "Hello" message, the allocates some memory buffer, store the "world" message in the buffer and send its to the output stream (return on line 14). The `sleep` on line 13 is here just for making more evident the FastFlow scheduling of concurrent activities. The second one (lines 18–26) just prints whatever he gets on the input stream (the data item stored after the `void * task` pointer of `svc` header on line 21), frees the allocated memory and then returns a `GO_ON` mark, which is intended to be a value interpreted by the FastFlow framework as: "I finished processing the current task, I give you no *result* to be delivered onto the output stream, but please keep me alive ready to receive another input task". The `main` on lines 28–40 is almost identical to the one of the previous version but for the fact we add two stages to the pipeline pattern. Implicitly, this sets up a streaming network with `Stage1` connected by a stream to `Stage2`. Items delivered on the output stream by `Stage1` will be read on the input stream by `Stage2`. The concurrent activity graph is therefore:



If we compile and run the program, however, we get a kind of unexpected result:

```
ffsrc$ g++ -lpthread -I
    /home/marcod/Documents/Research/CodeProgramming/fastflow-1.1.0
    hello2stages.cpp -o hello2stages
ffsrc$ ./hello2stages
Hello
WorldHello

Hello World

Hello World

Hello World

^C
ffsrc$
```

First of all, the program keeps running printing an "Hello world" every second. We in fact terminate the execution through a CONTROL-C. Second, the initial sequence of strings is a little bit strange[5].

The "infinite run" is related to way FastFlow implements concurrent activities. Each `ff_node` is run as many times as the number of the input data items appearing onto the output stream, unless the `svc` method returns a `NULL`. Therefore, if the method returns either a task (pointer) to be delivered onto the concurrent activity output stream, or the `GO_ON` mark (no data output to the output stream but continue execution), it is re-executed as soon as there is some input available. The first stage, which has no associated input stream, is re-executed up to the moment it terminates the `svc` with a `NULL`. In order to have the program terminating, we therefore may use the following code for `Stage1`:

```
class Stage1: public ff_node {
public:

  Stage1() { first = (1==1); }

  void * svc(void * task) {
    if(first) {
      std::cout << "Hello " << std::endl;
      char * p = (char *) calloc(sizeof(char),10);
      strcpy(p,"World");
      sleep(1);
      first = 0;
      return ((void *)p);
    } else {
      return NULL;
    }
  }
private:
  int first;
};
```

If we compile and execute the program with this modified `Stage1` stage, we'll get an output such as:

[5]and depending on the actual number of cores of your machine and on the kind of scheduler used in the operating system, the sequence may vary a little bit

```
ffsrc$ g++ -lpthread -I
    /home/marcod/Documents/Research/CodeProgramming/fastflow-1.1.0
    hello2terminate.cpp -o hello2terminate
ffsrc$ ./hello2terminate
Hello
World
ffsrc$
```

that is the program terminates after a single run of the two stages. Now the question is: why the second stage terminated, although the `svc` method return value states that more work is to be done? The answer is in the stream semantics implemented by FastFlow. FastFlow streaming networks automatically manage `end-of-streams`. That is, as soon as an `ff_node` returns a `NULL`–implicitly declaring he wants to terminate its output stream, the information is propagated to the node consuming the output stream. This nodes will therefore also terminate execution–without actually executing its `svc` method–and the end of stream will be propagated onto its output stream, if any. Therefore `Stage2` terminates after the termination of `Stage1`.

The other problem, namely the appearing of the initial 2 "Hello" strings apparently related to just one "world" string is related to the fact that FastFlow does not guarantee any scheduling semantics of the `ff_node svc` executions. The first stage delivers a string to the second stage, then it is executed again and again. The `sleep` inserted in the first stage prevents to accumulate too much "hello" strings on the output stream delivered to the second stage. If we remove the `sleep` statement, in fact, the output is much more different: we will see on the input a large number of "hello" strings followed by another large number of "world" strings. This because the first stage is enabled to send as much data items on the output stream as of the capacity of the SPSC queue used to implement the stream between the two stages.

B.3.1 Generating a stream

In order to achieve a better idea of how streams are managed within FastFlow, we slightly change our `HelloWorld` code in such a way the first stage in the pipeline produces on the output stream n integer data items and then terminates. The second stage prints a "world -i-" message upon receiving each i item onto the input stream.

We already discussed the role of the return value of the `svc` method. Therefore a first version of this program may be implemented using as the `Stage1` class the following code:

```
#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

class Stage1: public ff_node {
public:

  Stage1(int n) {
    streamlen = n;
    current = 0;
  }

  void * svc(void * task) {
    if(current < streamlen) {
      current++;
      std::cout << "Hello number " << current << " " << std::endl;
      int * p = (int *) calloc(sizeof(int),1);
```

```
19        *p = current;
20        sleep(1);
21        return ((void *)p);
22      } else {
23        return NULL;
24      }
25    }
26  private:
27    int streamlen, current;
28  };
29
30  class Stage2: public ff_node {
31  public:
32
33      void * svc(void * task) {
34        int * i = (int *) task;
35        std::cout << "World -" << *i << "- " << std::endl;
36        free(task);
37        return GO_ON;
38      }
39  };
40
41  int main(int argc, char * argv[]) {
42
43      ff_pipeline pipe;
44      pipe.add_stage(new Stage1(atoi(argv[1])));
45      pipe.add_stage(new Stage2());
46
47      if (pipe.run_and_wait_end()<0) {
48          error("running pipeline\n");
49          return -1;
50      }
51
52      return 0;
53  }
```

ffsrc/helloStream.cpp

The output we get is the following one:

```
1  ffsrc$ g++ -lpthread -I
       /home/marcod/Documents/Research/CodeProgramming/fastflow-1.1.0
       helloStream.cpp -o helloStream
2  ffsrc$ ./helloStream 5
3  Hello number 1
4  Hello number 2World - 1-
5
6  Hello number World -32 -
7
8  World -3- Hello number
9  4
10 Hello number 5World - 4-
11
12 World -5-
13 ffsrc$
```

However, there is another way we can use to generate the stream, which is a little bit more "programmatic". FastFlow makes available an `ff_send_out` method in the `ff_node`

class, which can be used to direct a data item onto the concurrent activity output stream, without actually using the `svc return` way.

In this case, we could have written the `Stage` as follows:

```
class Stage1: public ff_node {
public:

  Stage1(int n) {
    streamlen = n;
    current = 0;
  }

  void * svc(void * task) {
    while(current < streamlen) {
      current++;
      std::cout << "Hello number " << current << " " << std::endl;
      int * p = (int *) calloc(sizeof(int),1);
      *p = current;
      sleep(1);
      ff_send_out(p);
    }
    return NULL;
  }
private:
  int streamlen, current;
};
```

In this case, the `Stage1` is run just once (as it immediately returns a `NULL`. However, during the single run the `svc while` loop delivers the intended data items on the output stream through the `ff_send_out` method. In case the sends fill up the SPSC queue used to implement the stream, the `ff_send_out` will block up to the moment `Stage2` consumes some items and consequently frees space in the SPSC buffers.

B.4 MORE ON `FF_NODE`

The `ff_node` class actually defines three distinct virtual methods:

```
public:
    virtual void* svc(void * task) = 0;
    virtual int   svc_init() { return 0; };
    virtual void  svc_end()  {}
```

The first one is the one defining the behaviour of the node while processing the input stream data items. The other two methods are automatically invoked once and for all by the FastFlow RTS when the concurrent activity represented by the node is started (`svc_init`) and right before it is terminated (`svc_end`).

These virtual methods may be overwritten in the user supplied `ff_node` subclasses to implement initialization code and finalization code, respectively. Actually, the `svc` method *must* be overwritten as it is defined as a pure virtual method.

We illustrate the usage of the two methods with another program, computing the Sieve of Eratosthenes. The sieve uses a number of stages in a pipeline. Each stage stores the first integer it got on the input stream. Then is cycles passing onto the output stream only the input stream items which are not multiple of the stored integer. An initial stage injects in

the pipeline the sequence of integers starting at 2, up to n. Upon completion, each stage has stored a prime number.

We can implement the Eratostheness sieve with the following FastFlow program.

```
#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

class Sieve: public ff_node {
public:

  Sieve() { filter = 0; }

  void * svc(void * task) {
    unsigned int * t = (unsigned int *)task;

    if (filter == 0) {
      filter = *t;
      return GO_ON;
    } else {
      if(*t % filter == 0)
        return GO_ON;
      else
        return task;
    }
  }

  void svc_end() {
    std::cout << "Prime(" << filter << ")\n";
    return;
  }


private:
  int filter;
};

class Generate: public ff_node {
public:

  Generate(int n) {
    streamlen = n;
    task = 2;
    std::cout << "Generate object created" << std::endl;
    return;
  }


  int svc_init() {
    std::cout << "Sieve started. Generating a stream of " << streamlen
        <<
      " elements, starting with " << task << std::endl;
    return 0;
  }

  void * svc(void * tt) {
    unsigned int * t = (unsigned int *)tt;

```

```
55      if(task < streamlen) {
56        int * xi = (int *) calloc(1, sizeof(int));
57        *xi = task++;
58        return xi;
59      } else {
60        return NULL;
61      }
62    }
63  private:
64    int streamlen;
65    int task;
66  };
67
68  class Printer: public ff_node {
69
70    int svc_init() {
71      std::cout << "Printer started " << std::endl;
72      first = 0;
73    }
74
75    void * svc(void *t) {
76      int * xi = (int *) t;
77      if (first == 0) {
78        first = *xi;
79      }
80      return GO_ON;
81    }
82
83    void svc_end() {
84      std::cout << "Sieve terminating, prime numbers found up to " <<
            first
85          << std::endl;
86    }
87
88  private:
89    int first;
90  };
91
92  int main(int argc, char * argv[]) {
93    if (argc!=3) {
94      std::cerr << "use: "  << argv[0] << " nstages streamlen\n";
95      return -1;
96    }
97
98    ff_pipeline pipe;
99    int nstages = atoi(argv[1]);
100   pipe.add_stage(new Generate(atoi(argv[2])));
101   for(int j=0; j<nstages; j++)
102     pipe.add_stage(new Sieve());
103   pipe.add_stage(new Printer());
104
105   ffTime(START_TIME);
106   if (pipe.run_and_wait_end()<0) {
107     error("running pipeline\n");
108     return -1;
109   }
110   ffTime(STOP_TIME);
111
112   std::cerr << "DONE, pipe  time= " << pipe.ffTime() << " (ms)\n";
```

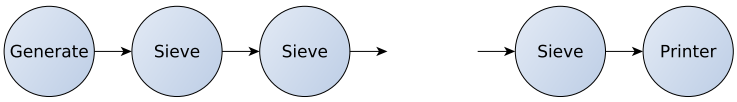
```
113   std::cerr << "DONE, total time= " << ffTime(GET_TIME) << " (ms)\n";
114   pipe.ffStats(std::cerr);
115   return 0;
116 }
```

ffsrc/sieve.cpp

The `Generate` stage at line 35–66 generates the integer stream, from 2 up to a value taken from the command line parameters. It uses an `svc_init` just to point out when the concurrent activity is started. The creation of the object used to represent the concurrent activity is instead evidenced by the message printed in the constructor.

The `Sieve` stage (lines 6–28) defines the generic pipeline stage. This stores the initial value got from the input stream on lines 14–16 and then goes on passing the inputs not multiple of the stored values on lines 18–21. The `svc_end` method is executed right before terminating the concurrent activity and prints out the stored value, which happen to be the prime number found in that node.

The `Printer` stage is used as the last stage in the pipeline (the pipeline build on lines 98–103 in the program `main`) and just discards all the received values but the first one, which is kept to remember the point where we arrived storing prime numbers. It defines both an `svc_init` method (to print a message when the concurrent activity is started) and an `svc_end` method, which is used to print the first integer received, representing the upper bound (non included in) of the sequence of prime numbers discovered with the pipeline stages. The concurrent activity graph of the program is the following one:



The program output, when run with 7 `Sieve` stages on a stream from 2 to 30, is the following one:

```
1  ffsrc$ ./sieve 7 30
2  Generate object created
3  Printer started
4  Sieve started. Generating a stream of 30 elements, starting with 2
5  Prime(2)
6  Prime(3)
7  Prime(5)
8  Prime(7)
9  Prime(Prime(Sieve terminating, prime numbers found up to 1317)
10 )
11 19
12 Prime(11)
13 DONE, pipe  time= 0.275 (ms)
14 DONE, total time= 25.568 (ms)
15 FastFlow trace not enabled
16 ffsrc$
```

showing that the prime numbers up to 19 (excluded) has been found.

B.5 MANAGING ACCESS TO SHARED OBJECTS

Shared objects may be accessed within FastFlow programs using the classical `pthread` concurrency control mechanisms. The FastFlow program is actually a multithreaded code using the `pthread` library, in fact.

We demonstrate how access to shared objects may be ensured within a FastFlow program forcing mutual exclusion in the access to the `std::cout` file descriptor. This will be used to have much nicer strings output on the screen when running the Sieve program illustrated in the previous section.

In order to guarantee mutual exclusion on the shared `std::cout` descriptor we use a `pthread_mutex_lock`. The lock is declared and properly initialized as a static, global variable in the program (see code below, line 7). Then each one of the writes to the `std::cout` descriptor in the concurrent activities relative to the different stages of the pipeline are protected through a `pthread_mutex_lock / pthread_mutex_unlock` "brackets" (see line 29–31 in the code below, as an example).

```
1  #include <iostream>
2  #include <ff/pipeline.hpp>
3  #include <pthread.h>
4
5  using namespace ff;
6
7  static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
8
9  class Sieve: public ff_node {
10 public:
11
12   Sieve() { filter = 0; }
13
14   void * svc(void * task) {
15     unsigned int * t = (unsigned int *)task;
16
17     if (filter == 0) {
18       filter = *t;
19       return GO_ON;
20     } else {
21       if(*t % filter == 0)
22   return GO_ON;
23       else
24   return task;
25     }
26   }
27
28   void svc_end() {
29     pthread_mutex_lock(&lock);
30     std::cout << "Prime(" << filter << ")\n";
31     pthread_mutex_unlock(&lock);
32     return;
33   }
34
35
36 private:
37   int filter;
38 };
39
40 class Generate: public ff_node {
41 public:
42
43   Generate(int n) {
44     streamlen = n;
45     task = 2;
46     pthread_mutex_lock(&lock);
47     std::cout << "Generate object created" << std::endl;
```

```
48      pthread_mutex_unlock(&lock);
49      return;
50    }
51
52
53    int svc_init() {
54      pthread_mutex_lock(&lock);
55      std::cout << "Sieve started. Generating a stream of " << streamlen
            <<
56        " elements, starting with " << task << std::endl;
57      pthread_mutex_unlock(&lock);
58      return 0;
59    }
60
61    void * svc(void * tt) {
62      unsigned int * t = (unsigned int *)tt;
63
64      if(task < streamlen) {
65        int * xi = (int *) calloc(1, sizeof(int));
66        *xi = task++;
67        return xi;
68      } else {
69        return NULL;
70      }
71    }
72  private:
73    int streamlen;
74    int task;
75  };
76
77  class Printer: public ff_node {
78
79    int svc_init() {
80      pthread_mutex_lock(&lock);
81      std::cout << "Printer started " << std::endl;
82      pthread_mutex_unlock(&lock);
83      first = 0;
84    }
85
86    void * svc(void *t) {
87      int * xi = (int *) t;
88      if (first == 0) {
89        first = *xi;
90      }
91      return GO_ON;
92    }
93
94    void svc_end() {
95      pthread_mutex_lock(&lock);
96      std::cout << "Sieve terminating, prime numbers found up to " <<
            first
97          << std::endl;
98      pthread_mutex_unlock(&lock);
99    }
100
101 private:
102   int first;
103 };
104
```

```
105 int main(int argc, char * argv[]) {
106   if (argc!=3) {
107     std::cerr << "use: "  << argv[0] << " nstages streamlen\n";
108     return -1;
109   }
110
111   ff_pipeline pipe;
112   int nstages = atoi(argv[1]);
113   pipe.add_stage(new Generate(atoi(argv[2])));
114   for(int j=0; j<nstages; j++)
115     pipe.add_stage(new Sieve());
116   pipe.add_stage(new Printer());
117
118   ffTime(START_TIME);
119   if (pipe.run_and_wait_end()<0) {
120     error("running pipeline\n");
121     return -1;
122   }
123   ffTime(STOP_TIME);
124
125   std::cerr << "DONE, pipe  time= " << pipe.ffTime() << " (ms)\n";
126   std::cerr << "DONE, total time= " << ffTime(GET_TIME) << " (ms)\n";
127   pipe.ffStats(std::cerr);
128   return 0;
129 }
```

ffsrc/sievelock.cpp

When running the program, we get a slightly different output than the one we obtained when the usage of `std::cout` was not properly regulated:

```
1  ffsrc$ ./a.out 7 30
2  Generate object created
3  Printer started
4  Sieve started. Generating a stream of 30 elements, starting with 2
5  Prime(2)
6  Prime(5)
7  Prime(13)
8  Prime(11)
9  Prime(7)
10 Sieve terminating, prime numbers found up to 19
11 Prime(3)
12 Prime(17)
13 DONE, pipe  time= 58.439 (ms)
14 DONE, total time= 64.473 (ms)
15 FastFlow trace not enabled
16 ffsrc$
```

The strings are printed in clearly separated lines, although some apparently unordered string sequence appears, which is due to the FastFlow scheduling of the concurrent activities *and* to the way locks are implemented and managed in the `pthread` library.

It is worth pointing out that

- FastFlow ensures correct access sequences to the shared object used to implement the streaming networks (the graph of concurrent activities), such as the SPSC queues used to implement the streams, as an example.

- FastFlow stream semantics guarantee correct sequencing of activation of the concurrent activities modelled through `ff_nodes` and connected through streams. The stream implementation actually ensures *pure data flow* semantics.
- any access to any user defined shared data structure must be protected with either the primitive mechanisms provided by FastFlow (see Sec. B.5) or the primitives provided within the `pthread` library.

B.6 MORE SKELETONS: THE FastFlow FARM

In the previous sections, we used only pipeline skeletons in the sample code. Here we introduce the other primitive skeleton provided in FastFlow, namely the `farm` skeleton.

The simplest way to define a farm skeleton in FastFlow is by declaring a `farm` object and adding a vector of *worker* concurrent activities to the `farm`. An excerpt of the needed code is the following one

```
#include <ff/farm.hpp>

using namespace ff;

int main(int argc, char * argv[]) {

  ...
  ff_farm<> myFarm;
  std::vector<ff_node *> w;
  for(int i=0;i<nworkers;++i)
    w.push_back(new Worker);
  myFarm.add_workers(w);
  ...
```

This code basically defines a farm with `nworkers` workers processing the data items appearing onto the farm input stream and delivering results onto the farm output stream. The scheduling policy used to send input tasks to workers is the default one, that is round robin one. Workers are implemented by the `ff_node` `Worker` objects. These objects may represent sequential concurrent activities as well as further skeletons, that is either pipeline or farm instances.

However, this farm may not be used alone. There is no way to provide an input stream to a FastFlow streaming network but having the first component in the network generating the stream. To this purpose, FastFlow supports two options:

- we can use the farm defined with a code similar to the one described above as the second stage of a pipeline whose first stage generates the input stream according to one of the techniques discussed in Sec. B.3.1. This means we will use the farm writing a code such as:

```
  ...
  ff_pipeline myPipe;

  myPipe.add_stage(new GeneratorStage());
  myPipe.add_stage(myFarm);
```

- or we can provide an `emitter` and a `collector` to the farm, specialized in such a way they can be used to produce the input stream and consume the output stream of the farm, respectively, while inheriting the default scheduling and gathering policies.

The former case is simple. We only have to understand why adding the farm to the pipeline as a pipeline stage works. This will discussed in detail in Sec. B.8. The latter case is simple as well, but we discuss it through some more code.

B.6.1 Farm with emitter and collector

First, let us see what kind of objects we have to build to provide the farm an `emitter` and a `collector`. Both `emitter` and `collector` must be supplied as `ff_node` subclass objects. If we implement the `emitter` just providing the `svc` method, the tasks delivered by the `svc` on the output stream either using a `ff_send_out` or returning the proper pointer with the `svc return` statement, those elements will be dispatched to the available workers according to the default round robin scheduling. An example of `emitter` node, generating the stream of tasks actually eventually processed by the farm `worker` nodes is the following one:

```
1  class Emitter: public ff_node {
2  public:
3      Emitter(int n) {
4          streamlen = n;
5          task = 0;
6      };
7
8      void * svc(void *) {
9          sleep(1);
10         task++;
11         int * t = new int(task);
12         if (task<streamlen)
13             return t;
14         else
15             return NULL;
16     }
17
18 private:
19     int streamlen;
20     int task;
21 };
```

In this case, the node `svc` actually does not take into account any input stream item (the input parameter name is omitted on line 5). Rather, each time the node is activated, it returns a task to be computed using the internal `ntasks` value. The task is directed to the "next" worker by the FastFlow farm run time support.

Concerning the `collector`, we can also use a `ff_node`: in case the results need further processing, they can be directed to the next node in the streaming network using the mechanisms detailed in Sec. B.3.1. Otherwise, they can be processed within the `svc` method of the `ff_node` subclass.

As an example, a `collector` just printing the tasks/results he gets from the workers may be programmed as follows:

```
1  class Collector: public ff_node {
2  public:
3      void * svc(void * task) {
4          int * t = (int *)task;
5          std::cout << "Collector got " << *t << std::endl;
6          return GO_ON;
7      }
8  };
```

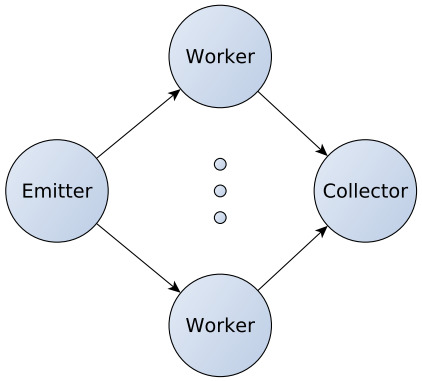
With these classes defined and assuming to have a worker defined by the class:

```
class Worker: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        (*t)++;
        return task;
    }
};
```

we can define a program processing a stream of integers by increasing each one of them with a farm as follows:

```
int main(int argc, char * argv[]) {
    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);

    ff_farm<> farm;

    Emitter E(streamlen);
    farm.add_emitter(&E);

    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i)
      w.push_back(new Worker);
    farm.add_workers(w);

    Collector C;
    farm.add_collector(&C);

    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
        return -1;
    }
    return 0;
}
```

The concurrent activity graph in this case is the following one:



When run with the first argument specifying the number of workers to be used and the second one specifying the length of the input stream generated in the collector node, we get the expected output:

```
ffsrc$ ./a.out 2 10
Collector got 2
Collector got 3
Collector got 4
Collector got 5
Collector got 6
Collector got 7
Collector got 8
Collector got 9
Collector got 10
ffsrc$
```

B.6.2 Farm with no collector

We move on considering a further case: a farm with emitter but no collector. Having no collector the workers may not deliver results: all the results computed by the workers must be consolidated in memory. The following code implements a farm where a stream of tasks of type `TASK` with an integer tag `i` and an integer value `t` are processed by the worker of the farm by:

- computing `t++` and
- storing the result in a global array at the position given by the tag `i`.

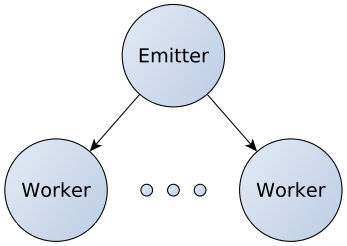
Writes to the global result array need not to be synchronized as each worker writes different positions in the array (the `TASK` tags are unique).

```
#include <vector>
#include <iostream>
#include <ff/farm.hpp>

static int * results;

typedef struct task_t {
    int i;
    int t;
} TASK;

using namespace ff;

class Worker: public ff_node {
public:
    void * svc(void * task) {
        TASK * t = (TASK *) task;
        results[t->i] = ++(t->t);
        return GO_ON;
    }
};


class Emitter: public ff_node {
public:
    Emitter(int n) {
```

```
27          streamlen = n;
28          task = 0;
29      };
30
31      void * svc(void *) {
32          task++;
33          TASK * t = (TASK *) calloc(1,sizeof(TASK));
34          t->i = task;
35          t->t = task*task;
36          if (task<streamlen)
37              return t;
38          else
39              return NULL;
40      }
41
42  private:
43      int streamlen;
44      int task;
45  };
46
47
48  int main(int argc, char * argv[]) {
49
50      int nworkers=atoi(argv[1]);
51      int streamlen=atoi(argv[2]);
52      results = (int *) calloc(streamlen,sizeof(int));
53
54      ff_farm<> farm;
55
56      Emitter E(streamlen);
57      farm.add_emitter(&E);
58
59      std::vector<ff_node *> w;
60      for(int i=0;i<nworkers;++i)
61          w.push_back(new Worker);
62      farm.add_workers(w);
63
64      std::cout << "Before starting computation" << std::endl;
65      for(int i=0; i<streamlen; i++)
66          std::cout << i << " : " << results[i] << std::endl;
67      if (farm.run_and_wait_end()<0) {
68          error("running farm\n");
69          return -1;
70      }
71      std::cout << "After computation" << std::endl;
72      for(int i=0; i<streamlen; i++)
73          std::cout << i << " : " << results[i] << std::endl;
74      return 0;
75  }
```

ffsrc/farmNoC.cpp

The `Worker` code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:



The main program prints the results vector before calling the FastFlow `start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc$ farmNoC 2 10
Before starting computation
0 : 0
1 : 0
2 : 0
3 : 0
4 : 0
5 : 0
6 : 0
7 : 0
8 : 0
9 : 0
After computation
0 : 0
1 : 2
2 : 5
3 : 10
4 : 17
5 : 26
6 : 37
7 : 50
8 : 65
9 : 82
ffsrc$
```

Besides demonstrating how a farm without collector may compute useful results, the program of the last listing also demonstrates how complex task data structures can be delivered and retrieved to and from the FastFlow streaming network streams.

B.6.3 Specializing the scheduling strategy in a farm

In order to select the worker where an incoming input task has to be directed, the FastFlow farm uses an internal `ff_loadbalancer` that provides a method `int selectworker()` returning the index in the worker array corresponding to the worker where the next task has to be directed. This method cannot be overwritten, actually. But the programmer may subclass the `ff_loadbalancer` and provide his own `selectworker()` method and pass the new load balancer to the farm emitter, therefore implementing a farm with a user defined scheduling policy.

The steps to performed in this case are exemplified with the following, relevant portions of code.

First, we subclass the `ff_loadmanager` and provide our own `setworker()` method:

```
class my_loadbalancer: public ff_loadbalancer {
```

```
protected:
    // implement your policy...
    inline int selectworker() { return victim; }

public:
    // this is necessary because ff_loadbalancer has non default
        parameters....
    my_loadbalancer(int
        max_num_workers):ff_loadbalancer(max_num_workers) {}

    void set_victim(int v) { victim=v;}

private:
    int victim;
};
```

Then we create a farm with specifying the new load balancer class as a type parameter:

```
ff_farm<my_loadbalancer> myFarm(...);
```

Eventually, we create an emitter that within its `svc` method invokes the `set_victim` method right before outputting a task towards the worker string, either with a `ff_send_out(task)` or with a `return(task)`. The emitter is declared as:

```
class myEmitter: public ff_node {

  myEmitter(ff_loadbalancer * ldb) {
    lb = ldb;
  }

  ...

  void * svc(void * task) {
    ...
    workerToBeUsed = somePolicy(...);
    lb->set_victim(workerToBeUsed);
    ...
    ff_send_out(task);
    return GO_ON;
  }

  ...
private:
  ff_loadbancer * lb;
}
```

and inserted in the farm with the code

```
myEmitter emitter(myFarm.getlb());
myFarm.add_emitter(emitter);
```

What we get is a farm where the worker to be used to execute the task appearing onto the input stream is decided by the programmer through the proper implementation of `my_loadbancer` rather than being decided by the current FastFlow implementation.

Two particular cases specializing the scheduling policy in different way by using FastFlow predefined code are illustrated in the following two subsections.

B.6.3.1 Broadcasting a task to all workers FastFlow supports the possibility to direct a task to all the workers in a farm. It is particularly useful if we want to process the task by workers implementing different functions. The broadcasting is achieved through the declaration of a specialized load balancer, in a way very similar to what we illustrated in Sec. B.6.3.

The following code implements a farm whose input tasks are broadcasted to all the workers, and whose workers compute different functions on the input tasks, and therefore deliver different results on the output stream.

```
#include <iostream>
#include <ff/farm.hpp>
#include <ff/node.hpp>
#include <math.h>
#include <pthread.h>

using namespace std;
using namespace ff;


static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
#define WRTLOCK(cmd) {pthread_mutex_lock(&lock); {cmd}
    pthread_mutex_unlock(&lock);}

// should be global to be accessible from workers
#define MAX 4
int x[MAX];

class WorkerPlus: public ff_node {
  int svc_init() {
    WRTLOCK( cout << "Worker initialized" << endl; )
      return 0;
  }

  void * svc(void * in) {
    int *i = ((int *) in);
    int ii = *i;
    int * newi = new int();
    *newi = ++ii;
    WRTLOCK(cout << "WorkerPLus got " << ii << " and computed " <<
        *newi << endl ;)
      return (void *)newi;
  }
};

class WorkerMinus: public ff_node {
  int svc_init() {
    WRTLOCK(cout << "Worker initialized" << endl; )
      return 0;
  }

  void * svc(void * in) {
    int *i = ((int *) in);
    int ii = *i;
    int * newi = new int();
    *newi = --ii;
    WRTLOCK(cout << "WorkerPLus got " << ii << " and computed " <<
        *newi << endl ;)
      return (void *)newi;
```

```
47    }
48  };
49
50  class my_loadbalancer: public ff_loadbalancer {
51  public:
52    // this is necessary because ff_loadbalancer has non default
        parameters....
53    my_loadbalancer(int
        max_num_workers):ff_loadbalancer(max_num_workers) {}
54
55    void broadcast(void * task) {
56      ff_loadbalancer::broadcast_task(task);
57    }
58  };
59
60  class Emitter: public ff_node {
61  public:
62    Emitter(my_loadbalancer * const lb):lb(lb) {}
63    void * svc(void * task) {
64      lb->broadcast(task);
65      return GO_ON;
66    }
67  private:
68    my_loadbalancer * lb;
69  };
70
71  class Collector: public ff_node {
72  public:
73    Collector(int i) {}
74    void * svc(void * task) {
75      WRITLOCK(cout << "Got result " << * ((int *) task) << endl;)
76        return GO_ON;
77    }
78  private:
79    int itemsize; // needed for TBB's allocators
80  };
81
82
83
84  #define NW 2
85
86  int main(int argc, char * argv[])
87  {
88    ffTime(START_TIME);
89
90    cout << "init " << argc  << endl;
91    int nw = (argc==1 ? NW : atoi(argv[1]));
92
93    cout << "using " << nw << " workers " << endl;
94
95    // init input (fake)
96    for(int i=0; i<MAX; i++) {
97      x[i] = (i*10);
98    }
99    cout << "Setting up farm" << endl;
100   // create the farm object
101   ff_farm<my_loadbalancer> farm(true,nw);
102   // create and add emitter object to the farm
103   Emitter E(farm.getlb());
```

```
104   farm.add_emitter(&E);
105   cout << "emitter ok "<< endl;
106
107
108   std::vector<ff_node *> w;    // prepare workers
109   w.push_back(new WorkerPlus);
110   w.push_back(new WorkerMinus);
111   farm.add_workers(w);         // add them to the farm
112   cout << "workers ok "<< endl;
113
114   Collector C(1);
115   farm.add_collector(&C);
116   cout << "collector ok "<< endl;
117
118   farm.run_then_freeze();      // run farm asynchronously
119
120   cout << "Sending tasks ..." << endl;
121   int tasks[MAX];
122   for(int i=0; i<MAX; i++) {
123     tasks[i]=i;
124     farm.offload((void *) &tasks[i]);
125     sleep(1);
126     WRITLOCK( cout << "offloaded " << tasks[i] << endl; )
127   }
128   farm.offload((void *) FF_EOS);
129
130   cout << "Waiting termination" << endl;
131   farm.wait();
132
133   cout << "Farm terminated after computing for " << farm.ffTime() <<
          endl;
134
135   ffTime(STOP_TIME);
136   cout << "Spent overall " << ffTime(GET_TIME) << endl;
137
138 }
```

ffsrc/ff_misd.cpp

At lines 50–58 a `ff_loadbalancer` is defined providing a `broadcast` method. The method is implemented in terms of an `ff_loadbalancer` internal method. This new loadbalancer class is used as in the case of other user defined schedulers (see Sec. B.6.3) and the emitter eventually uses the load balancer `broadcast` method *instead* of delivering the task to the output stream (i.e. directly to the string of the workers). This is done through the `svc` code at lines 63–66.

Lines 109 and 110 are used to add two different workers to the farm.

The rest of the program is standard, but for the fact the resulting farm is used as an accelerator (lines 118–131, see Sec. B.7).

B.6.3.2 Using autoscheduling FastFlow provides suitable tools to implement farms with "auto scheduling", that is farms where the workers "ask" for something to be computed rather than accepting tasks sent by the emitter (explicit or implicit) according to some scheduling policy. This scheduling behaviour may be simply implemented by using the `ff_farm` method `set_scheduling_ondemand()`, as follows:

```
1 ff_farm myFarm(...);
2 myFarm.set_scheduling_ondemand();
3 ...
4 farm.add_emitter(...);
```

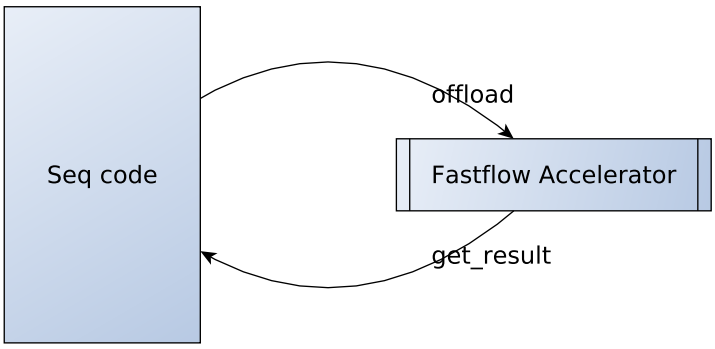


Figure B.2 FastFlow accelerator

```
5 ...
```

The scheduling policy implemented in this case is an approximation of the auto scheduling, indeed. The emitter simply checks the length of the SPSC queues connecting the emitter to the workers, and delivers the task to the first worker whose queue length is less or equal to 1. To be more precise, FastFlow should have implemented a request queue where the workers may write tasks requests tagged with the worker id and the emitter may read such request to choose the worker where the incoming tasks is to be directed. This is not possible as of FastFlow 1.1 because it still doesn't allow to read from multiple SPSC queues preserving the FIFO order.

B.7 FastFlow AS A SOFTWARE ACCELERATOR

Up to know we just showed how to use FastFlow to write a "complete skeleton application", that is an application whose complete flow of control is defined through skeletons. In this case the `main` of the C/C++ program written by the user is basically providing the structure of the parallel application by defining a proper FastFlow skeleton nesting and the commands to start the computation of the skeleton program and to wait its termination. All the business logic of the application is embedded in the skeleton parameters.

Now we want to discuss the second kind of usage which is supported by FastFlow, namely FastFlow accelerator mode. The term "accelerator" is used the way it used when dealing with hardware accelerators. An hardware accelerator–a GPU or an FPGA or even a more "general purpose" accelerator such as Tilera 64 core chips, Intel Many Core or IBM WireSpeed/PowerEN–is a device that can be used to compute particular kind of code faster that the CPU. FastFlow accelerator is a software device that can be used to speedup skeleton structured portions of code using the cores left unused by the main application. In other words, it's a way FastFlow supports to accelerate particular computation by using a skeleton program and offloading to the skeleton program tasks to be computed.

The FastFlow accelerator will use $n-1$ cores of the n core machine, assuming that the calling code is not parallel and will try to ensure a $n-1$ fold speedup is achieved in the computation of the tasks offloaded to the accelerator, provide a sufficient number of tasks are given to be computed.

Using FastFlow accelerator mode is not too much different from using FastFlow to write an application only using skeletons (see Fig. B.2). In particular, the following steps must be followed:

- A skeleton program has to be written, using the FastFlow skeletons (or their customized versions), computing the tasks that will be given to the accelerator. The

skeleton program used to program the accelerator is supposed to have an input stream, used to offload the tasks to the accelerator.

- Then, the skeleton program must be run using a particular method, different from the `run_and_wait_end` we have already seen, that is a `run_then_freeze()` method. This method will start the accelerator skeleton program, consuming the input stream items to produce either output stream items or to consolidate (partial) results in memory. When we want to stop the accelerator, we will deliver and end-of-stream mark to the input stream.
- Eventually, we must wait the computation of the accelerator is terminated.

A simple program using FastFlow accelerator mode is shown below:

```
#include <vector>
#include <iostream>
#include <ff/farm.hpp>
#include <time.h>

using namespace ff;

int * x;
int nworkers = 0;

class Worker: public ff_node {
public:

  Worker(int i) {
    my_id = i;
  }

  void * svc(void * task) {
    int * t = (int *)task;
    x[my_id] = *t;
    return GO_ON;
  }
private:
  int my_id;
};

int main(int argc, char * argv[]) {

  if (argc<3) {
    std::cerr << "use: "
        << argv[0]
        << " nworkers streamlen\n";
    return -1;
  }

  nworkers=atoi(argv[1]);
  int streamlen=atoi(argv[2]);

  x = (int *) calloc(nworkers,sizeof(int));
  for(int i=0; i<nworkers; i++)
    x[i] = 0;

  ff_farm<> accelerator(true);

  std::vector<ff_node *> w;
```

```
46    for(int i=0;i<nworkers;++i)
47      w.push_back(new Worker(i));
48    accelerator.add_workers(w);
49
50    if (accelerator.run_then_freeze()<0) {
51      error("running farm\n");
52      return -1;
53    }
54
55    for(int i=0; i<=streamlen; i++) {
56      int * task = new int(i);
57      accelerator.offload(task);
58    }
59    accelerator.offload((void *) FF_EOS);
60    accelerator.wait();
61
62    for(int i=0; i<nworkers; i++)
63      std::cout << i << ":" << x[i] << std::endl;
64
65    return 0;
66  }
```

ffsrc/acc.cpp

We use a farm accelerator. The accelerator is declared at line 43. The "true" parameter is the one telling FastFlow this has to be used as an accelerator. Workers are added at lines 45–48. Each worker is given its id as a constructor parameters. This is the same as the code in plain FastFlow applications. Line 50 starts the skeleton code in accelerator mode. Lines 55 to 58 offload tasks to be computed to the accelerator. These lines could be part of any larger C++ program, indeed. The idea is that whenever we have a task ready to be submitted to the accelerator, we simply "offload" it to the accelerator. When we have no more tasks to offload, we send and end-of-stream (line 59) and eventually we wait for the completion of the computation of tasks in the accelerator (line 60).

This kind of interaction with an accelerator not having an output stream is intended to model those computations than consolidate results directly in memory. In fact, the `Worker` code actually writes results into specific position of the vector `x`. Each worker writes the task it receives in the i-th position of the vector, being i the index of the worker in the farm worker string. As each worker writes a distinct position in the vector, no specific synchronization is needed to access vector positions. Eventually the last task received by worker i will be stored at position i in the vector.

We can also assume that results are awaited from the accelerator through its output stream. In this case, we first have to write the skeleton code of the accelerator in such a way an output stream is supported. In the new version the accelerator sample program below, we add a collector to the accelerator farm (line 45). The collector is defined as just collecting results from workers and delivering the results to the output stream (lines 18–24). Once the tasks have been offloaded to the accelerator, rather waiting for accelerator completion, we can ask computed results as delivered to the accelerator output stream through the `bool load_result(void **)` method (see lines 59–61).

```
1  #include <vector>
2  #include <iostream>
3  #include <ff/farm.hpp>
4  #include <time.h>
5
6  using namespace ff;
7
8  class Worker: public ff_node {
```

```
9  public:
10
11   void * svc(void * task) {
12     int * t = (int *)task;
13     (*t)++;
14     return task;
15   }
16 };
17
18 class Collector: public ff_node {
19 public:
20     void * svc(void * task) {
21         int * t = (int *)task;
22         return task;
23     }
24 };
25
26
27 int main(int argc, char * argv[]) {
28
29   if (argc<3) {
30     std::cerr << "use: "
31         << argv[0]
32         << " nworkers streamlen\n";
33     return -1;
34   }
35
36   int nworkers=atoi(argv[1]);
37   int streamlen=atoi(argv[2]);
38
39   ff_farm<> accelerator(true);
40
41   std::vector<ff_node *> w;
42   for(int i=0;i<nworkers;++i)
43     w.push_back(new Worker());
44   accelerator.add_workers(w);
45   accelerator.add_collector(new Collector());
46
47   if (accelerator.run_then_freeze()<0) {
48     error("running farm\n");
49     return -1;
50   }
51
52   for(int i=0; i<=streamlen; i++) {
53     int * task = new int(i);
54     accelerator.offload(task);
55   }
56   accelerator.offload((void *) FF_EOS);
57
58   void * result;
59   while(accelerator.load_result(&result)) {
60     std::cout << "Got result :: "<<  (*((int *)result)) << std::endl;
61   }
62   accelerator.wait();
63
64   return 0;
65 }
```

ffsrc/accColl.cpp

The `bool load_result(void **)` methods synchronously await for one item being delivered on the accelerator output stream. If such item is available, the method returns "true" and stores the item pointer in the parameter. If no other items will be available, the method returns "false".

An asynchronoud method is also available `bool load_results_nb(void **)`. In this case, if no result is available at the moment, the method returns a "false" value, and you should retry later on to see whether a result may be retrieved.

It is worth pointing out that in order to be able to use the `load_result` functions the accelerator must be equipped with a `Collector`. A simple collector such as the

```
class Collector: public ff_node {
  void * svc (void * task) {
    return task;
  }
};
```

is sufficient to the purpose.

B.8 SKELETON NESTING

In FastFlow skeletons may be arbitrarily nested. As the current version only supports farm and pipeline skeletons, this means that:

- farms may be used as pipeline stages, and
- pipelines may be used as farm workers.

There are no limitations to nesting, but the following one :

- skeletons using the `wrap_around` facility (see also Sec. B.9) cannot be used as parameters of other skeletons.

  As an example, you can define a farm with pipeline workers as follows:

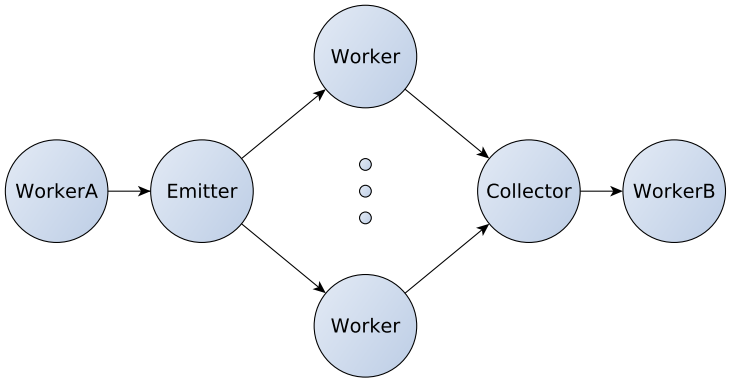
```
    ff_farm<> myFarm;

    std::vector<ff_node *> w;
    for(int i=0; i<NW; i++)
      ff_pipeline * p = new ff_pipeline;
      p->add_stage(new S1());
      p->add_stage(new S2());
      w.push_back(p);
    }
    myFarm.addWorkers(w);
```

  or we can use a farm as a pipeline stage by using a code such as:

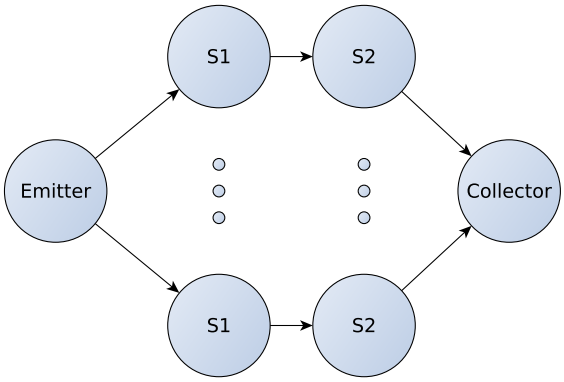
```
    ff_pipeline * p = new ff_pipeline;
    ff_farm <>    f = new ff_farm;

    ...

    f.addWorkers(w);

    ...

```

```
10      p->add_stage(new SeqWorkerA());
11      p->add_stage(f);
12      p->add_stage(new SeqWorkerB());
```

The concurrent activity graph in this case will be the following one:



while in the former case it will be such as



B.9 FEEDBACK CHANNELS

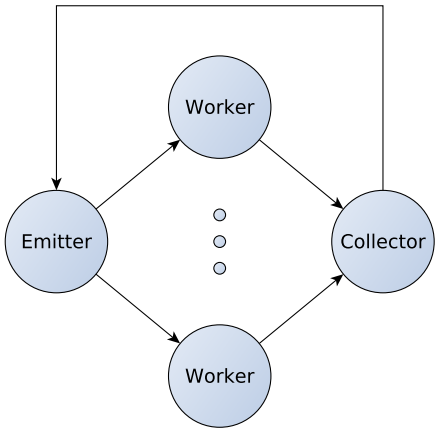
In some cases, it will be useful to have the possibility to route back some results to the streaming network input stream. As an example, this allows to implement divide and conquer using farms. Task injected in the farm are split by the workers and the resulting splitted tasks are routed back to the input stream for further processing. Tasks that can be computed using the base case code, are computed instead and their results are used for the conquer phase, usually performed in memory.

All what's needed to implement the feedback channel is to invoke the `wrap_around` method on the interested skeleton. In case our applications uses a farm pattern as the outermost skeleton, we may therefore add the method call after instantiating the farm object:

```
1 ff_farm<> myFarm;
2 ...
3 myFarm.add_emitter(&e);
4 myFarm.add_collector(&c);
5 myFarm.add_workers(w);
6
7 myFarm.wrap_aroud();
```

```
8 ...
```

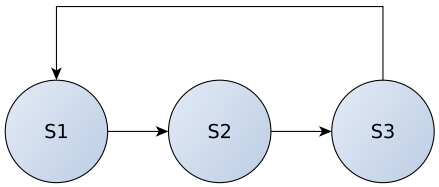
and this will lead to the concurrent activity graph



The same if parallelism is expressed by using a pipeline as the outermost skeleton:

```
1 ff_pipeline myPipe;
2
3 myPipe.add_stage(s1);
4 myPipe.add_stage(s2);
5 myPipe.add_stage(s3);
6 ...
7 myPipe.wrap_around();
8 ...
```

leading to the concurrent activity graph:



As of FastFlow 1.1, the only possibility to use the feedback channel provided by the `wrap_around` method is relative to the outermost skeleton, that is the one with no input stream. This because at the moment FastFlow does not support merging of input streams. In future versions this constrain will be possibly eliminated.

B.10 INTRODUCING NEW SKELETONS

Current version of FastFlow (1.1) only supports stream parallel pipeline and farm skeletons. However, the skeletons themselves may be used/customized to serve as "implementation templates"[6] for different kinds of skeletons. The FastFlow distribution already includes sample applications where the farm with feedback is used to support divide&conquer applications. Here we want to discuss how a data parallel *map* skeleton may be used in FastFlow, exploiting the programmability of farm skeleton emitter and collector.

[6]according to the terminology used in the algorithmic skeleton community

B.10.1 Implementing a Map skeleton with a Farm "template"

In a pure map pattern all the items in a collection are processed by means of a function f. If the collection was

$$x = \langle x_1, \ldots, x_m \rangle$$

then the computation

$$map\ f\ x$$

will produce as a result

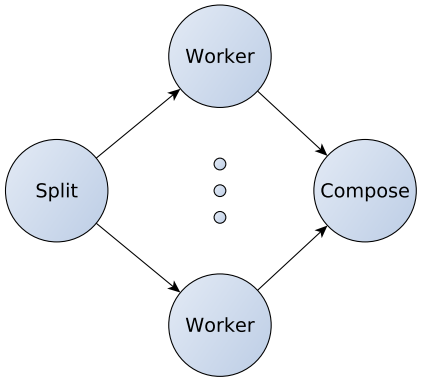
$$\langle f(x_1), \ldots, f(x_m) \rangle$$

In more elaborated map skeletons, the user is allowed to define a set of (possibly overlapping) partitions of the input collection, a function to be applied on each one of the partitions, and a strategy to rebuild–from the partial results computed on the partitions–the result of the map.

As an example, a matrix multiplication may be programmed as a map such that:

- the input matrixes A and B are considered as collections of rows and columns, respectively
- a set of items $\langle A_{i,*}, B_{*,j} \rangle$–the $i-th$ row of A and the $j-th$ column of B–are used to build the set of partitions
- an inner product is computed on each $\langle A_{i,*}, B_{*,j} \rangle$: this is $c_{i,j}$ actually
- the C matrix ($C = A \times B$) is computed out of the different $c_{i,j}$.

If we adopt this second, more general approach, a map may be build implementing a set of concurrent activities such as:



where the `Split` node create the partitions and delivers them to the workers, the workers compute each $c_{i,j}$ and deliver the results to the `Compose`. The `Compose` eventually rebuilds the full C out of the n^2 $c_{i,j}$.

We can therefore program the whole map as a FastFlow farm. After defining proper task, subtask and partial result data structures:

```
1 // a task requires to compute the matrix multiply C = A x B
2 // we assume square matrixes, for the sake of simplicity
3 typedef struct {
4   int n;
5   float **a;
6   float **b;
```

```
  float **c;
  int tag;     // used to gather partial results from the same task
} TASK;

// a subtask is the computation of the inner product or A, row i, by
    B, col j
typedef struct {
  int i,j;
  TASK * t;
} SUBTASK;

// a partial result is the i,j item in the result matrix
typedef struct {
  int i,j;
  float x;
  TASK * t;
} PART_RESULT;
```

we define the emitter to be used in the farm as follows:

```
// this node is used to generate the task list out of the initial data
// kind of user defined iterator over tasks
class Split: public  ff_node {
   void * svc(void * t) {
     TASK * task = (TASK *) t;      // tasks come in already allocated
     for(int i=0; i<task->n; i++)
       for(int j=0; j< task->n; j++)  {
         // SUBTASKe are allocated in the splitter and destroyed in
             the worker
         SUBTASK * st = (SUBTASK *) calloc(1,sizeof(SUBTASK));
         st->i = i;
         st->j = j;
         st->t = task;
         ff_send_out((void *)st);
       }
     return GO_ON;
   }
};
```

Basically, the first time the emitter is called, we generate all the tasks relative to the different $\langle A_{i,*}, B_{*,j}\rangle$. These tasks are directed to the workers, that will compute the different $c_{i,j}$ and direct the `PART_RESULT` to the collector. The worker `ff_node` will therefore be programmed as:

```
class Worker: public ff_node {
public:

    void * svc(void * task) {
        SUBTASK * t = (SUBTASK *) task;
        float * x = new float(0.0);
        for(int k=0; k<(t->t)->n; k++) {
          *x = *x + (t->t->a)[t->i][k] * (t->t->b)[k][t->j];
        }
        // prepare the partial result to be delivered
        PART_RESULT * pr = (PART_RESULT *)
            calloc(1,sizeof(PART_RESULT));
        pr->i = t->i;
```

```
13          pr->j = t->j;
14          pr->t = t->t;
15          pr->x = *x;
16          // the subtask is no more useful, deallocate it
17          free(task);
18          // return the partial result
19          return pr;
20      }
21  };
```

The collector will be defined in such a way the different partial results computed by the workers are eventually consolidated in memory. Therefore each $c_{i,j}$ received is stored at the correct entry of the C matrix. The pointer of the result matrix is in fact a field in the TASK data strcuture and $c_{i,j}$, i and j are fields of teh PART_RESULT data structure. The code for the collector is therefore:

```
1  class Compose: public ff_node {
2  public:
3      Compose() {
4        n = 0;
5        for(int i=0; i<MAXDIFF; i++)
6          tags[i] = 0;
7      }
8
9      void * svc(void * t) {
10       PART_RESULT * r = (PART_RESULT *) t;
11       TASK * tt = r->t;
12       // consolidate result in memory
13       ((r->t)->c)[r->i][r->j] = r->x;
14       tags[((r->t)->tag)%MAXDIFF]++;
15       if(tags[((r->t)->tag)%MAXDIFF] == ((r->t)->n)*((r->t)->n)) {
16         tags[((r->t)->tag)%MAXDIFF]  = 0;
17         free(t);
18         return(tt);
19       } else {
20         free(t);
21         return GO_ON;
22       }
23     }
24 private:
25   int n;
26   int tags[MAXDIFF];
27 };
```

The tags here are used to deliver a result on the farm output stream (i.e. the output stream of the collector) when exactly n^2 results relative to the same input task have been received by the collector. A MAXDIFF value is used assuming that no more than MAXDIFF different matrix multiplication tasks may be circulating at the same time in the farm, due to variable time spent in the computation of the single $c_{i,j}$.

With these classes, our map may be programmed as follows:

```
1      ff_farm<> farm(true);
2
3      farm.add_emitter(new Split());        // add the splitter emitter
4      farm.add_collector(new Compose());    // add the composer collector
5      std::vector<ff_node *> w;             // add the convenient # of
           workers
```

```
6       for(int i=0;i<nworkers;++i)
7           w.push_back(new Worker);
8       farm.add_workers(w);
```

It is worth pointing out that:

- the kind of knowledge required to write the `Split` and `Compose` nodes to the application programmer is very application specific and not too much related to the implementation of the map
- this implementation of the map transforms a data parallel pattern into a stream parallel one. Some overhead is paid to move the data parallel sub-tasks along the streams used to implement the farm. This overhead may be not completely negligible
- a much coarser grain implementation could have been designed assuming that the `Split` node outputs tasks representing the computation of a whole $C_{i,*}$ row and modifying accordingly the `Worker`.
- usually, the implementation of a map data parallel pattern generates as many subtasks as the amount of available workers. In our implementation, we could have left to the `Split` node this task, using the FastFlow primitive mechanisms to retrieve the number of workers actually allocated to the farm[7] and modifying accordingly both the `Worker` and the `Compose` code.

Also, the proposed implementation for the map may be easily encapsulated in a proper `ff_map` class:

```
1  class ff_map {
2  public:
3
4    // map constructor
5    // takes as parameters: the splitter,
6    // the string of workers and the result rebuilder
7
8    ff_map(ff_node * splt, std::vector<ff_node *> wrks, ff_node * cmps) {
9
10       exec.add_emitter(splt);      // add the splitter emitter
11       exec.add_collector(cmps);    // add the composer collector
12       exec.add_workers(wrks);      // add workers
13   }
14
15     operator ff_node*() {          // (re)define what's returned when
16     return (ff_node*)&exec;        // asking a pointer to the class
           object
17     }
18
19 private:
20   ff_farm<> exec;                  // this is the farm actually used to
       compute
21 };
```

With this definition, the user could have defined the map (and added the map stage to a pipeline) using the following code:

[7] this is the `getnworkers` method of the farm loadbalancer.

```
1  std::vector<ff_node *> w;
2  for(int i=0;i<nworkers;++i)
3    w.push_back(new Worker);
4  ff_map myMap(new Split(), w, new Compose());
5
6  ...
7
8  myPipe.add_stage(myMap);
```

B.11 PERFORMANCE

Up to now we only discussed how to use FastFlow to build parallel programs, either applications completely coded with skeletons, or FastFlow software accelerators. We want to shortly discuss here the typical performances improvements got through FastFlow.

In skeleton application or in software accelerator, using a FastFlow farm would in general lead to a performance increase proportional to the number of workers used (that is to the parallelism degree of the farm). This unless:

- we introduce serial code fragments–in this case the speedup will be limited according to the Amdahl law–or
- we use more workers than the available tasks
- or eventually the time spent to deliver a task to be computed to the worker and retrieving a result from the worker are higher than the computation time of the task.

This means that if the time spent to compute m tasks serially is T_{seq}, we can expect the time spent computing the same m tasks with an nw worker farm will be more or less $\frac{T_{seq}}{nw}$. It is worth pointing out here that the *latency* relative to the computation of the single task does not decrease w.r.t. the sequential case.

In case a k stage FastFlow pipeline is used to implement a parallel computation, we may expect the overall service time of the pipeline is $T_S = max\{T_{S_1}, \ldots, T_{S_k}\}$. As a consequence, the time spent computing m tasks is approximately $m \times T_S$ and the relative speedup may be quantified as $\frac{m \times \sum_{i=1}^{k} T_{S_i}}{m \times max\{T_{S_1}, \ldots, T_{S_k}\}} = \frac{\sum_{i=1}^{k} T_{S_i}}{max\{T_{S_1}, \ldots, T_{S_k}\}}$. In case of balanced stages, that is pipeline stages all taking the same time to compute a task, this speedup may be approximated as k, being $\sum_{i=1}^{k} T_{S_i} = k \times T_{S_1}$ and $max\{T_{S_1}, \ldots, T_{S_k}\} = T_{S_1}$.

B.12 RUN TIME ROUTINES

Several utility routines are defined in FastFlow. We recall here the main ones.

- `virtual int  get_my_id()`
  returns a virtual id of the node where the concurrent activity (its `svc` method) is being computed
- `const int ff_numCores()`
  returns the number of cores in the target architecture
- `int ff_mapThreadToCpu(int cpu_id, int priority_level=0)`
  pins the current thread to `cpu_id`. A priority may be set as well, but you need root rights in general, and therefore this should non be specified by normal users

- `void error(const char * str, ...)`
  is used to print error messages

- `virtual bool ff_send_out(void * task,`
  `unsigned intretry=((unsigned int)-1),`
  `unsigned int ticks=(TICKS2WAIT))`
  delivers an item onto the output stream, possibly retrying upon failre a given number of times, after waiting a given number of clock ticks.

- `double ffTime()`
  returns the time spent in the computation of a farm or of pipeline, including the `svc_init` and `svc_end` time. This is method of both classes pipeline and farm.

- `double ffwTime()`
  returns the time spent in the computation of a farm or of pipeline, in the `svc` method only.

- `double ffTime(int tag)`
  is used to measure time in portions of code. The `tag` may be: `START_TIME`, `STOP_TIME` or `GET_TIME`

- `void ffStats(std::ostream & out)`
  prints the statistics collected while using FastFlow. The program must be compiled with `TRACE_FASTFLOW` defined, however.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AC	Autonomic Controller
AM	Autonomic Manager
AOP	Aspect Oriented Programming
BS	Behavioural Skeletons
CISC	Complex Instruction Set Computer
COW	Cluster Of Workstations
DNS	Domain Name Service
EU	European Union
FPGA	Field Programmable Gate Array
FRbs	Functional Replication Behavioural Skeleton
GCM	Grid Component Model
GP-GPU	General Purpose Graphic Process Unit
HOF	High Order Functions
HPF	High Performance Fortran
JVM	Java Virtual Machine
LTS	Labelled Transition System
MDF	Macro Data Flow
MDFG	Macro Data Flow Graph
MDFi	Macro Data Flow Instruction
MIMD	Multiple Instruction Multiple Data
MPI	Message Passing Interface

MPP	Massively Parallel Processor
NFS	Network File System
NOW	Network Of Workstations
NUMA	Non-Uniform Memory Access
OO	Object Oriented
OSS	Open Source Software
P3L	Pisa Parallel Processing Language
PE	Processing Element
POJO	Plain Old Java Object
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTS	Run Time Support
SIMD	Single Instruction Multiple Data (programming model)
SMP	Symmetric Multi Processor
SSI	Single System Image
TBB	Thread Building Block (library)
T_C	Completion time
T_S	Service time
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word

INDEX

REFERENCES

1. E. Agerbo and A. Cornils. How to preserve the benefits of Design Patterns. In *Proceedings of OOPSLA'98*, pages 134–143, 1998.
2. Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and Guillermo Leguizamon. MALLBA a software library to design efficient optimisation algorithms. *Intl. Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.
3. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. 2013. Paris, to appear.
4. M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED Intl. Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED/ACTA press.
5. M. Aldinucci, M. Danelutto, P. Kilpatrick, and V. Xhagijka. LIBERO: a framework for autonomic management of multiple non-functional concerns. In *Proceedings of Euro-Par 2010 workshops*, number 6586 in LNCS. Springer Verlag, 2011.
6. M. Aldinucci, M. Danelutto, P. Kilpatrick, and V. Xhagjika. LIBERO: a LIghtweight BEhaviouRal skeletOn framework, 2007. available at `http://compass2.di.unipi.it/TR/Files/TR-10-07.pdf.gz`.
7. M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.
8. Marco Aldinucci, Francoise André, Jérémy Buisson, Sonia Campa, Massimo Coppola, Marco Danelutto, and Corrado Zoccolo. Parallel program/component adaptivity management. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 89–96, Germany, December 2005. John von Neumann Institute for Computing.
9. Marco Aldinucci, Carlo Bertolli, Sonia Campa, Massimo Coppola, Marco Vanneschi, Luca Veraldi, and Corrado Zoccolo. Self-configuring and self-optimizing grid components in the GCM model and their ASSIST implementation. In *Proc of. HPC-GECO/Compframe (held in conjunction with HPDC-15)*, pages 45–52, Paris, France, June 2006. IEEE.
10. Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Peter Kilpatrick, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons for component autonomic management

on grids. In Marco Danelutto, Paraskevi Frangopoulou, and Vladimir Getov, editors, *Making Grids Work*, CoreGRID, chapter Component Programming Models, pages 3–16. Springer, August 2008.

11. Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In Didier El Baz, Julien Bourgeois, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, February 2008. IEEE.
12. Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.
13. Marco Aldinucci and Marco Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.
14. Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.
15. Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In *Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, Rome, Italy, May 2009. IEEE.
16. Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic managenemt of multiple non-functional concerns in behavioural skeletons. In *Grids, P2P and Services Computing (Proc. of the CoreGRID Symposium 2009)*, CoreGRID, Delft, The Netherlands, August 2009. Springer.
17. Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Skeletons for multi/many-core systems. In *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 265–272, Lyon, France, September 2009. IOS press.
18. Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 09, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 273–280. IOS press, September 2009.
19. Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.
20. Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with fastflow. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, February 2010. IEEE.
21. Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.
22. P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. of Euro-Par 1996*, pages 601–614. Springer-Verlag, 1996.
23. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.
24. B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.
25. Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2):5–24, 2009.

26. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of 11th Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 761–770, Lisboa, Portugal, August 2005. Springer.

27. G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

28. Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:45–53, 2008.

29. Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81, Rennes, France, 2007. Springer-Verlag.

30. Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer, 2007.

31. C. Chambers, B. Handerson, and J. Vlissides. A Debate on Lnaguage and Tool Support for Design Patterns. In *Proceedings of POPL 2000*.

32. S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proceedings of the PCW'97*, 1997. Camberra, Australia.

33. P. Ciechanowicz, M. Poldner, and H. Kuchen. The Munster skeleton library Muesli – a comprehensive overview. In *ERCIS Working paper*, number 7. ERCIS – European Research Center for Information Systems, 2009.

34. Philipp Ciechanowicz and Herbert Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-Core Computer Architectures. In *Proc. of the 10th Intl. Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, Los Alamitos, CA, USA, September 2010. IEEE.

35. Philipp Ciechanowicz and Herbert Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *HPCC*, pages 108–113. IEEE, 2010.

36. François Clément, V. Martin, A. Vodicka, Roberto Di Cosmo, and Pierre Weis. Domain decomposition and skeleton programming with ocamlp3l. *Parallel Computing*, 32(7-8):539–550, 2006.

37. Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

38. Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

39. Roberto Di Cosmo, Zheng Li, Susanna Pelagatti, and Pierre Weis. Skeletal parallel programming with ocamlp3l 2.0. *Parallel Processing Letters*, 18(1):149–164, 2008.

40. Rémi Coudarcher, Jocelyn Sérot, and Jean-Pierre Dérutin. Implementation of a skeleton-based parallel programming environment supporting arbitrary nesting. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, HIPS '01, pages 71–85, London, UK, 2001. Springer-Verlag.

41. M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.

42. M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

43. Marco Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

44. Marco Danelutto and Giorgio Zoppi. Behavioural skeletons meeting services. In *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, volume 5101 of *LNCS*, pages 146–153, Krakow, Poland, June 2008. Springer.

45. J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf,

editors, *Proc. of the Parallel Architectures and Langauges Europe (PARLE'93)*, number 694 in LNCS. Springer-Verlag, June 1993.

46. J. Darlington and H. W. To. Building parallel applications without programming. In *Leeds Workshop on Abstract Parallel Machine Models 93*, 1993.
47. Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *Proc. ParCo-2011 Int. Conference on Parallel Computing*, Sept. 2011. Ghent, Belgium.
48. E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
49. Auto-vectorization in GCC, 2013. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.
50. D. Goswami, A. Singh, and B. R. Preiss. Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons. In *Proceedings of the ISCOPE'99 conference*, pages 130–141. Springer Verlag, 1999. LNCS No. 1732.
51. N. Gunn. Using the Concurrency and Coordination Runtime, 2010. available at `http://www.infoq.com/articles/Using-CCR;jsessionid=9B1FE895F8A97410B7E1BEC40D18B33A`.
52. C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.
53. Infiniband web page, 2011. `http://www.infinibandta.org/`.
54. Intel. Intel Threading Building Blocks Design Patterns, 2010. available at `http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Design_Patterns.pdf`.
55. Intel. An Introduction to Vectorization with the Intel(C) C++ Compiler, 2013. `http://d3f8ykwhia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf`.
56. C. W. Kessler. Symbolic Array Data Flow Analysis and Pattern Recognition in Numerical Codes. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 57–68. Birkhauser, April 1994.
57. Christoph W. Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 1403–1408. IEEE, 2012.
58. Khronos Compute Working Group. *OpenCL*, November 2009. `http://www.khronos.org/opencl/`.
59. David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *Proc. of the 6th Intl. Symposium on Memory Management (ISMM)*, pages 103–104, New York, NY, USA, 2007.
60. Herbert Kuchen. A skeleton library. In B. Monien and R. Feldman, editors, *Proc. of 8th Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 620–629, Paderborn, Germany, August 2002. Springer.
61. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.
62. Mario Leyton, Ludovic Henrio, and José M. Piquer. Exceptions for algorithmic skeletons. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 14–25. Springer, 2010.
63. Mario Leyton and José M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 289–296, Washington, DC, USA, 2010. IEEE Computer Society.
64. Keith Ansel Marzullo. *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging)*. PhD thesis, Stanford, CA, USA, 1984. AAI8506272.

65. B. L. Massingill, T. G. Mattson, and B. A Sanders. A Pattern Language for Parallel Application Languages. Technical Report TR 99-022, Univeristy of Florida, CISE, 1999.

66. B. L. Massingill, T. G. Mattson, and B. A. Sanders. A Pattern Language for Parallel Application Programs. In Bode, Ludwig, Karl, and Wismuller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 678–681. Springer Verlag, August 2000.

67. Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proc. of the 1st Intl. Conference on Scalable Information Systems*, page 13, New York, NY, USA, 2006. ACM.

68. Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A Fusion-Embedded Skeleton Library. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 644–653. LNCS Springer, 2004.

69. Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, first edition, 2004.

70. S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling. From Patterns to Frameworks to Parallel Programs. submitted to Journal of Parallel and Distributed Computing, December 2000.

71. S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating parallel program frameworks from parallel design p atterns. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Proc of. 6th Intl. Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 95–105. Springer, August 2000.

72. M.P.I.Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

73. Myricom. Myrinet overview, 2011. `http://www.myri.com/myrinet/overview/`.

74. Ocaml web page, 2011. `http://caml.inria.fr/ocaml/`.

75. Michael Poldner and Herbert Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.

76. T. Serban, Marco Danelutto, and Peter Kilpatrick. Autonomic scheduling of tasks from data parallel patterns to cpu/gpu core mixes. In *HPCS*, pages 72–79. IEEE, 2013.

77. J. Serot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipp er project. *Parallel Computing*, 28(12):1685–1708, 2002.

78. J. Serot, D. Ginhac, and J.P. Derutin. SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In *Proceedings of the 5th International Parallel Computing Technologies Conference (PaCT-99)*, September 1999.

79. Jocelyn Sérot. Embodying parallel functional skeletons: An experimental implementation on top of mpi. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 629–633. Springer, 1997.

80. Jocelyn Sérot, Dominique Ginhac, and Jean-Pierre Dérutin. Skipper: A skeleton-based parallel programming environment for real-time image processing applications. In Victor E. Malyshkin, editor, *PaCT*, volume 1662 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 1999.

81. SkeTo project home, 2011. `http://www.ipl.t.u-tokyo.ac.jp/sketo/`.

82. Sourceforge. *FastFlow project*, 2009. `http://mc-fastflow.sourceforge.net/`.

83. Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *IPDPS Workshops*, pages 1176–1182. IEEE, 2011.

84. Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 666–677. Springer-Verlag, 2009.

85. Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

86. M. Vanneschi. Course Notes: High Performance Computing Systems and Enabling Platforms, 2010. Department of Computer Science, University of Pisa, available at `http://www.di.unipi.it/~vannesch/ASE2007-08/ASE07_08%20-%202.html`.

87. Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.