ASSIST

Lorenzo Anardu 442078

14/05/2010

ASSIST is a skeleton-based general purpouse parallel programming environment developed by Computer Science department in Pisa.

1 Skeleton Programming Model

Skeleton programming is a style of computer programming based on simple high-level program structures.
The principle upon which this programming model is based is contained in the Murray Cole's PhD thesis: useful patterns of parallel computation and interaction can be packaged up as 'framework/second order/template' constructs (parameterized by user's pieces of code). Such constructs are 'skeletons', in that they have structure but lack detail.
This means that some mechanism to allow the user to specify the parameters (such as sequential code) must be provided.

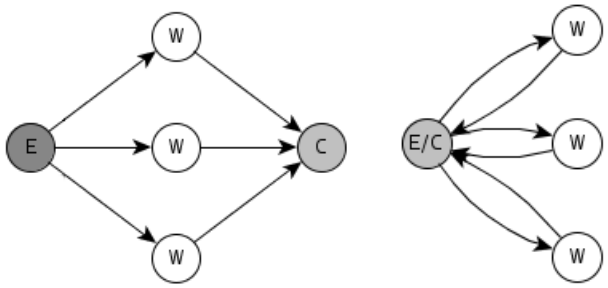

Figure 1: Possible implementation templates of the Farm skeleton.

In a skeleton programming model the parallelism is not arbitrarily expressable, it is limited to some exploitable parallelism patterns.
Each parallelism pattern has its own semantics and its own implementation. The semantics of the parallelism pattern is the 'skeleton', the particular implementation is the 'implementation template'. A single skeleton can have multiple implementation templates, e.g. the farm skeleton can be implemented in different ways, as shown in Figure 1.
The skeleton approach allows to separate functional behaviour, provided by the user with parameters, from non-functional behaviour, encoded by the skeleton systrem developer in the implementation template.
It is possible to combine the single skeletons for exploiting more complex parallelism forms in which the parallelism is divided in many levels. Over the years has established the approach to compose few simple basic skeletons rather than provide many detailed skeletons optimized for special cases.

ASSIST is an atypical skeleton system: unlike prior skeleton systems, provides the possibility to use behaviours different from the standard one (e.g. exceptions in the standard behaviour), this feature implies an incomplete compositionality of the skeletons.

2 ASSIST skeleton system

In ASSIST the skeleton approach was extended with some ideas from other prototypes previously developed in Pisa

P3L a classical skeleton system: C-based implementation;

SkIE a classical skeleton system: multilanguage support;

Muskel a data-flow skeleton system: customizable skeletons;

and some target from scientific and industrial needs.

The system has been designed with the aim of providing:

- high-level programmability and productivity in software development;
- high performance and portability of applications;
- software reuse and integration between ASSIST programs and other parallel applications.

ASSIST aims to separate the concerns between application programmer, i.e. the user, and system programmer, i.e. the implementor.
Another aim was to provide sufficient expressive power to code all possible parallelism patterns in a modular and portable way.
ASSIST provides the option to write skeleton-parallel applications interacting with components and the option to export parallel programs as components, for this reason ASSIST provides multilanguage support for C, C++ and Fortran-77.

The framework's implementation is structured onto three layers, providing compilation tools, deployment tools and a multitarget runtime.
The ASSIST compiler has been designed to be modular and extendable. The compilation process advances through completely separated phases, uses portable file formats for intermediate results.
Deployment tools provided with ASSIST are able to work either on both high performance architectures, such as clusters, and normal workstations.
The runtime is called multitarget because it can handle multiple target architectures (with different hardware, different memory management mechanisms, different Operative Systems, etc). A first degree of portability is ensured by the use of ACE communication library which is supported by all POSIX compliant systems.
Another important feature of ASSIST environment is the dinamic adaptivity: the application must be able to automatically react to a number of conditions in order to obtain a fixed performance.

In ASSIST the fundamental interaction is the data-flow one: all the skeletons written with ASSIST interact through typed data streams. The user's code encapsulation is provided through interfaces around code modules which indicates input and output parameters of the module. Upon these interfaces a stub is generated which permits to "concatenate" the module with the rest of the parallel program.

3 ASSIST constructs

The main interaction mechanism in ASSIST is represented by typed streams. When, in particular cases, this mechanism is not sufficient it is complemented by shared memory data structures. The shared memory mechanism is emulated if it is not provided natively by the underlying architecture. This does not mean that the main memory can be used as the main interaction mechanism (such as Java threads), because it would bring bad performances: the shared memory feature is provided just for the management of few cases in which stream interaction is not sufficient.

All the ASSIST basic construct support multilanguage code encapsulation:

Seq represents the simplest case of a sequential code module;

Generic stands for Generic Graph, represents all computational models based on graphs;

ParMod stands for Parallel Module, represents a configurable parallel skeleton.

3.1 Streams and data types

As said before streams are the main interaction mechanism. Streams in ASSIST are typed: packets belonging to a certain stream have the same ASSIST type.

ASSIST data types correspond to CORBA data types. In this way it is possible to easily exchange data between processes running on different machines. ASSIST types adopt a c-like syntax, all ASSIST data types are serializable structures (this implies that is impossible use pointers or recursive structures such as linked lists) and there is a predefined multilanguage equivalence (ASSIST types are mapped onto language types).

Streams are entirely managed by the ASSIST runtime: user code does not deal with problems of input and output data, the module is executed when input data is available and the output data is delivered to the proper receiver in a data-flow approach.
Data transmission is performed in different ways depending on the kind of machines considered:

- for identical machines data is sent in binary format;
- for different machines data is sent in XDR format.

3.2 Sequential code modules

The Seq construct defines a simple code module which can be executed on any resource.

Any sequential module has a specific interface indicating the type of input and output parameters of the module. The sequential code is contained by a module named proc.
In pratice the definition of a sequential module consists of 2 parts:

1. module's interfaces definition, through seq construct;
2. code definition, through proc construct.

The proc construct is a general use construct in ASSIST , it is used also to specify sequential parts of the parallel modules.
proc specifies the language used to write the code module and the actual code behaviour, as shown in Figure2.

In the example of Figure 2 the first three lines define a sequantial module operating on an input stream x and putting results on an output stream y. For each element of the stream is executed the

```
My_seq_module(    input_stream long x
                  output_stream long y)
  { f(in x out y); }

proc f(in long a out long b)
  inc<“myHeader.cpp”,
      “mySource.cpp”>
  path<“/home/marcod/myIncludes”>
  obj<“myObjectCode.o”>
  src<“mySource.c”>
$c++{  /* here goes your code…*/ }c++$
```

Figure 2: A simple seq module.

function f. In particular, while the module works on streams, the function defined by proc works on single packets of the stream.
The parameters and the behaviour of the function are defined by the subsequent proc module. The module can refer to sources, headers, external object-code files and sequential libraries indipendently of the language.

3.3 Generic construct

The generic construct represents all possible graph-based computational modules. All communication channels are directed ones.

It is possible to create pipelines and DAG (Directed Acyclic Graphs) as special cases of graphs containing loops. If the user creates a graph of modules containing cycles it is of to the user to avoid infinite loops problem.
The generic construct allows to define unconstrained data-flow graphs containing sequential modules, parallel modules and also nested generic: from this standpoint the generic construct is completely compositional.

As in the seq construct each module specifies its functional interfaces, indicating input and output streams.
Generic construct provides multiple input and output streams. In case of multiple input streams it is possible to have both deterministic or nondeterministic reading behaviour, while in case of output streams the value is sent to all streams.

This construct provides also a shared status between modules, represented by unsynchronized shared variables. This means that is in charge to the user's code to correctly use these variables without inconstistencies.
The shared variables access is implemented this way for mantaining a data-flow behaviour of the modules: synchronization mechanisms cause performance degradations.

In the example of Figure 3 is shown the declaration of a generic graph. In a first phase the types and the names of the streams are declared, subsequently the declared streams are “mapped” on the modules.
Each module in the example can be a Seq module, a ParMod or a Generic graph (in the example it is not specified).

min 54.30 (grafi diretti e gestione cicli)

```
generic main (. . .)
{
    stream   int s13;
    stream   int [N][N] s23;
    stream   int [N][N] s34;
    stream   int s25;

   M1 (ouput_stream s13);
   M2 (output_stream s23, s25);
   M3    (input_stream    s13,    s23;
   output_stream s34);
   M4 (input_stream s34);
   M5 (input_stream s25);
}
```

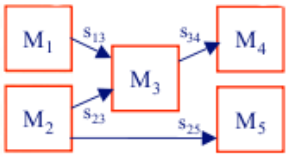

Figure 3: A Generic graph module.

3.4 ParMod construct

ParMod is the generic parallel module construct. This construct allows structured way of defining parallel computations abstracting from the underlying architecture: the module is structured in terms of logical parallel activities, called *Virtual Processors*, which are able to share data. The ParMod definition also specifies the cooperation with the "outside":

- specifying the interfaces and the interaction between modules;
- specifying how the interactions via shared memory are performed.

This construct provides more expressiveness than other constructs, and previous skeleton frameworks. Tuning the parameters of the ParMod the user can express both classical skeletons (such as Farm, Map, etc) and special cases (mixed skeletons behaviour).

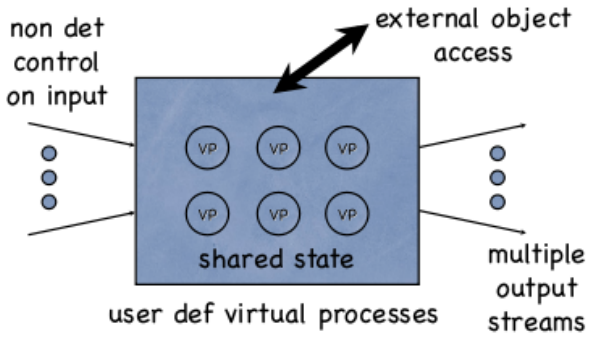

Figure 4: ParMod abstract schema.

As shown in Figure 4 the ParMod can be thought as a set o Virtual Processors operating on a set of heterogeneous input streams and with a set of output streams. Unlike Generic construct, the ParMod can choose the output stream in which send a particular packet in a nondeterministic way. Another important fact shown in the schema is that ParMod has escapes to acceed external objects (such as shared memory, a remote server, etc).