The ASSIST Programming Environment

Massimo Coppola

06/07/2007 - Pisa, Dipartimento di Informatica

Within the Ph.D. course “Advanced Parallel Programming” by M. Danelutto

With contributions from the whole Research Group on Parallel Architectures

Summary

- The ASSIST skeleton system
- Basic Concepts
  - Code encapsulation
  - Execution
- Syntax and Semantics
  - The ASSIST Constructs
  - Parallelism expression
- Advanced features
  - Run-time, Dynamic Adaptivity
  - Heterogeneous Platforms
  - Component orientation
- Future Extensions

The ASSIST skeleton system

- Skeleton-based parallel programming environment
  - Compilation, deployment, execution
  - Targets scientific and industrial needs
    - High performance
    - Programmability, portability, interoperability, time-to-market,
- Developed with ideas from other prototypes
  - P3L (classical skeletons, C -based implementation)
  - SkIE (classical skeletons, multi language)
  - Muskel (data-flow Java based)
- Paradigm shift: “modable” skeletons, “escapes”

The ASSIST skeleton system

- Separation of concerns
  - Application programmer vs. system programmer
- Expressive power
  - Most current patterns available through parmod
- Code reuse
  - C, C++, F77 (Java)
- External objects/library access support
  - Provide escapes to unstructured / external resources
- Layered implementation
  - Compiler, deploy tools, multitarget run time
- Multiple target architectures
  - Different CPUs/Memory & different Operating Systems

Basic Concepts

- Data flow-interaction
  - Typed streams
- Code encapsulation
  - Well defined interfaces around code modules
- Which parallel skeletons?
  - Flexible, extendable approach
- Execution
- Deployment tools
- Adaptivity

The ASSIST Constructs

- Streams
  - main interaction mechanism
  - complemented by shared memory data structures
- Seq
  - the simplest case of a code module
  - multi language code encapsulation
- Generic
  - pipeline, DAG, generic graph task parallelism
- Parmod
  - Multiple-pattern parallel skeleton

Streams and data types

- Interaction mechanism among modules
- Typed (stream packets are ASSIST types)
- ASSIST types = CORBA types
  - C-like syntax
  - serializable data structures
  - predefined inter-language equivalence
  - technology ages fast…
- Stream management within the run-time
  - Implementation details are hidden
  - exploit binary or XDR formats

Sequential code modules

- Seq defines a simple code module
  - To run sequentially on any single computing resource
  - With specified interface: type of input and output parameters
  - Code defined by a **proc** section
- Proc specifies sequential code behavior
  - General use in ASSIST (also within parallel modules)
  - **Which** language
  - What is the actual code, and what are its interfaces

The *seq* and *proc* Constructs

```
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”>
$c++{  /* here goes your code…*/ }c++$
```

can include/link:

source, headers, externally generated object-code files and sequential libraries

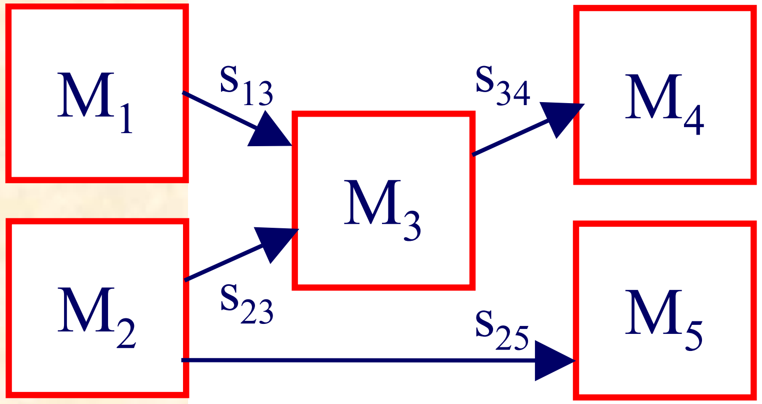
also different source languages

The *generic* Construct

- Short for generic graph
  - can have loops
  - pipeline and direct acyclic graphs (DAG) as special cases
- Allows to define unconstrained data-flow graphs of modules
  - Sequential, parallel modules and nested generic
- Each module represented by its functional interfaces
  - input and output streams
  - Multiple inputs and outputs: supports nondeterministic behavior
- Data-flow + shared status
  - Unsynchronized shared var.s (rely on program structure!)

Generic example

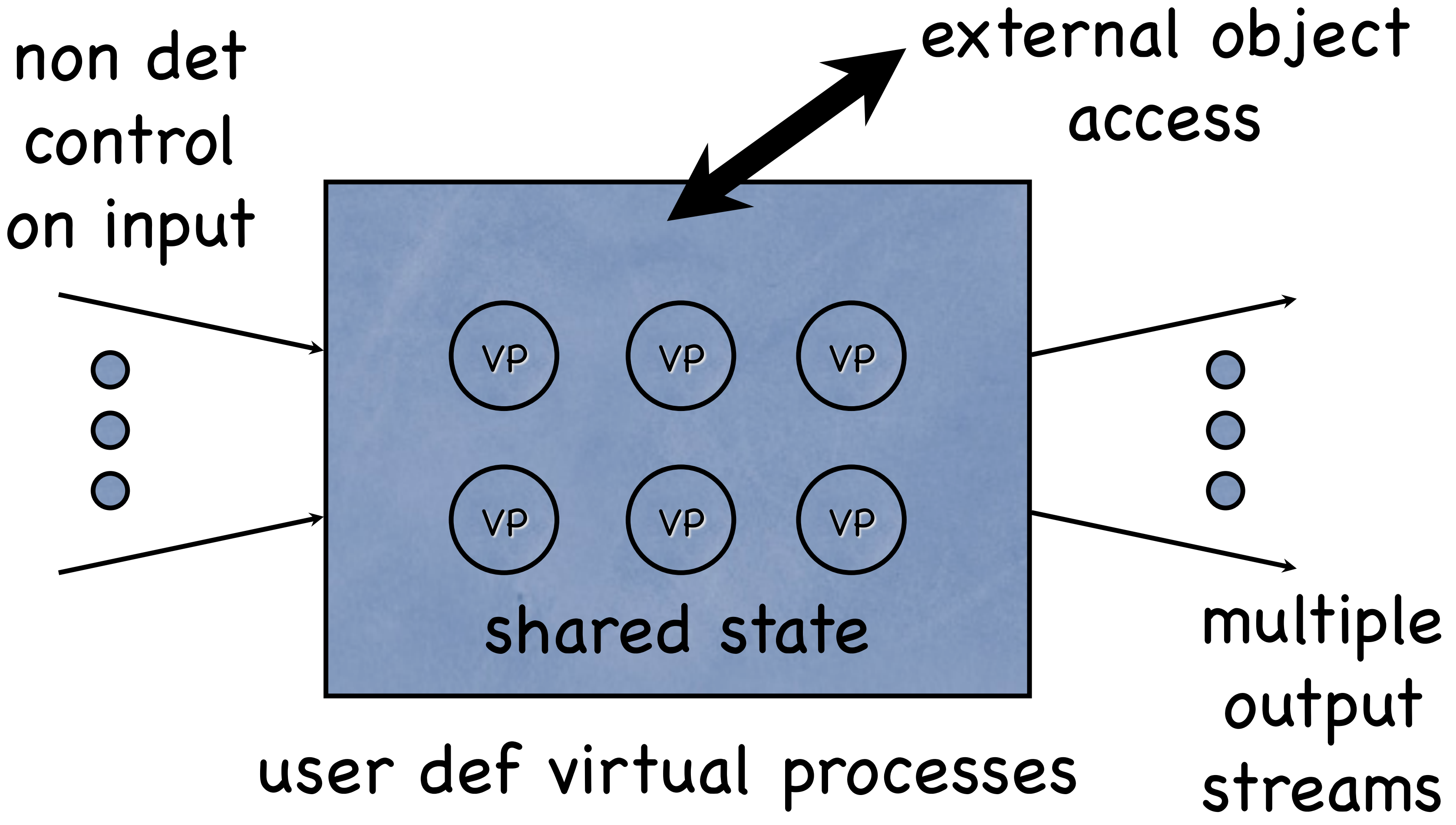
```
generic main (. . .)
{
     stream   int s13;
     stream   int [N][N] s23;
     stream   int [N][N] s34;
     stream   int s25;

    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);
}
```


$$\begin{tikzcd}
M_1 \arrow[dr, "s_{13}"] & & M_4 \\
& M_3 \arrow[ur, "s_{34}"] & \\
M_2 \arrow[ur, "s_{23}"'] \arrow[rr, "s_{25}"] & & M_5
\end{tikzcd}$$

parmod = generic PARallel MODule

- structured way of defining parallel computations
- abstracting away from actual mechanisms
  - logical parallel activities
  - logical data sharing
  - specification of cooperation with the "outside"
- syntax special cases : farm, map, ….
- + expressiveness = deal with special cases
  - classical skeletons are enough, usually
  - mix skeleton behaviors / switch among them

parmod abstract schema



parmod detailed

Drawn from a presentation by
Marco Danelutto

parmod overall

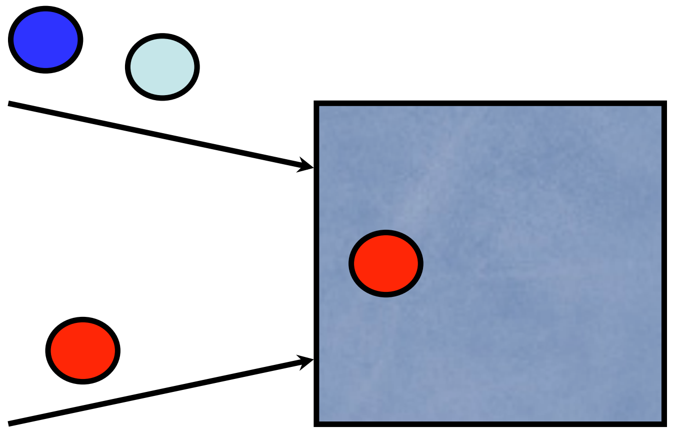
- process (multiple) input stream(s) of data
- produce (multiple) output stream(s)
- Streams
  - data flow semantics (sort of one way comms)

parmod Interface

- Parallel Computation
  - **Virtual Processes** (VP) express computation grain
  - VP eventually map to physical resources (automatic!)
- Parmod *minimal* syntax / semantics
  - bring data to VPs
  - define how VP cooperate
  - bring results out

Non-deterministic input control

Multiple data-flow inputs:
(how) do we choose?



- boolean guards
  - accessible and modifiable
- priorities
- input guards
- data availability

when satisfied, trigger virtual process(es)

Nondeterminism: input section

- non deterministic input control
  - set of data-flow input streams to choose from
- input section handles:
  - Priorities
  - Boolean Guards (enable input streams on expression)
  - Stream combinations (*f* needs both *A* and *B* to compute…)
- data from streams is *distributed* to
  - virtual processes or parmod state
  - **Distributions**: broadcast, unicast, scatter, multicast
- data availabilty triggers virtual processor execution (à la Data Flow)

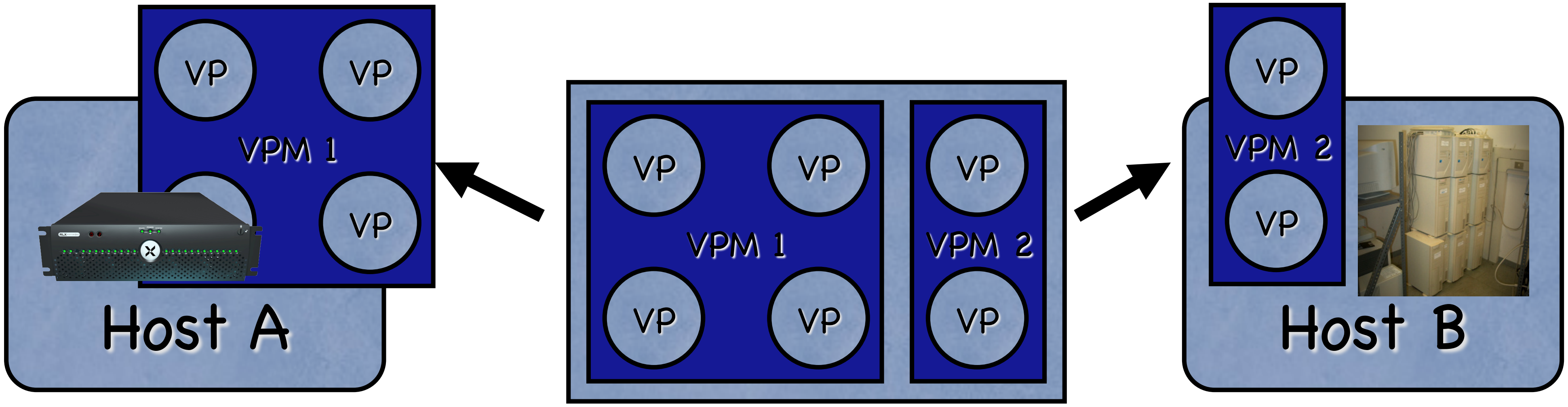
VP : logically parallel activity

- Concept of virtual process:
  - a logically concurrent/parallel activity
  - with a name
    - there is a topology arranging VPs
    - topology can be exploited to define the computation
  - can perform different functions
    - selects according to its state and inputs
    - sequential code modules encapsulated in a **proc**
- Computation is described in terms of code & data dependencies
  - VP possibly sharing state with the other activities

VP : logical and actual machines

At execution time:

- VP mapped to Virtual Processes Manager (VPM)
- VPM mapped to physical processing resources
  - Mapping performed by tools
  - Mapping can change at run-time (dynamic reconfiguration)



VP naming

- Topology = VP naming scheme
  - array: topology array [i:N] myVP;
    - processors name after indexes of a (multidimensional) array
    - topology array [i:N] [j:M] [k:O] myVP
  - none: topology none myVP;
    - none= no naming, anonymous processes (task farm)
    - can still express many different computation schemes
  - one: topology one myVP;
    - one single (seq) process, but all parmod features
    - e.g. multiple in/out, non deterministic input control

Parmod internal state

- attributes = variables (typed, structured)
- can be logically distributed on VPs
  - match attribute structure on parmod's topology
- owner-computes rule
- compiler + run time support ensure (safe) accessibility
- implemented through AdHOC
  - independent shared-memory support

Parmod distributions

- state to VPs
- input data to VPs and state
- scatter, broadcast, multicast + scheduled
- scheduled
  - computed on the basis of the input data

Parmod application code

- associated to virtual processes
  - to all or to subsets (using naming)
- Call through the *proc* code in C, C++, F77
  - Java soon ...
- possibility to introduce parmod iterations
  - for, while statements
- Input data triggers code execution
- Barriers can be automatically inserted
  - take care of data-parallel synchronizations

Parmod output section

- Simple syntax for simple cases
  - output parameters of virtual processes simply delivered to output streams
- User control for more complex cases
  - assist_out(stream, object)
  - recompose data structures out of VP results
  - insert arbitrary proc (attributes, guards)
- Multiple output stream handling

external objects

- run time code access to invoke external services
  - e.g. CCM, WS, AdHOC, shared objects, ...
- proc code can access these services under complete user control
  - sort of ESCape to structured parallelism ...

Examples of structured patterns (parmod subcases)

- task farm
  - topology none, distribution on-demand, collect from any
- “dedicated” task farm
  - topology array, distribution scheduled
- (embarrassingly) data parallel
  - topology array, tree
- fixed/variable stencil data parallel
  - topology array, tree
- Custom schemes
  - topology array, tree + non det input section+ state + multiple VP proc + code within output section

parmod examples

Multi-language support and application structure

- ASSIST `astcc` is a front-end compiler
  - Employs several other compilers as back-ends
  - Run-time support code, and final linker: C++
- Compiler and sub-compiler configuration
  - The **ast_rc** XML file defines
    paths, flags, compilers, linker to exploit
- Compiled application is a set of executables
  - Application structure is a directory tree
  - Compact form (**.aar** archive)
  - Structure encoded in a flexible XML format

Application Description

ALDL=Application Level Description Language

- Application-level information
  - Structure and parameters (e.g. degree of parallelism)
  - Application executables
  - Run-time support processes
- Process-level information
  - Architecture, OS
  - HW/SW resources: memory, CPU, libraries…
  - Input and output files
- Run-time parameters
  - E.g. TCP ports, or network configuration

All information gathered by the compiler

(simple) ALDL fragment

```
<?xml version="1.0" ?>
<aldl:application xmlns = "urn:aldl-assist" xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance" xmlns:aldl = "http://
     www.isti.cnr.it/schemas/aldl/" xmlns:ns1 = "http://www.isti.cnr.it/schemas/assist/" xmlns:tns = "urn:aldl-assist"
     targetNamespace = "urn:aldl-assist" xsi:schemaLocation = "http://www.isti.cnr.it/schemas/aldl/ xml/aldl.xsd http://
     www.isti.cnr.it/schemas/assist/ xml/assist.xsd">
 <aldl:requirement name = "libraries">
  <ns1:lib fileName = "libACE.so.5" fileSystemName = "/tmp" arch = "i686" executable = "no">
   <ns1:source url = "file:///home/pascucci/Assist/utils/ACE-5.5/lib/libACE.so.5"/>
  </ns1:lib>
  <ns1:lib fileName = "libm.so.6" fileSystemName = "/tmp" arch = "i686" executable = "no">
   <ns1:source url = "file:///lib/tls/libm.so.6"/>
  </ns1:lib>
  <ns1:lib fileName = "libxml2.so.2" fileSystemName = "/tmp" arch = "i686" executable = "no">
   <ns1:source url = "file:///home/pascucci/Assist/utils/libxml2-2.6.27/lib/libxml2.so.2"/>
  </ns1:lib>
 </aldl:requirement>
 <aldl:requirement name = "ND000__leggiConfiguration">
  <ns1:executable master = "yes" strategy = "no" arch = "i686">/home/pascucci/Assist/compiledAssist/testKMeans//bin//
     i686-pc-linux-gnu/ND000__leggi</ns1:executable>
 </aldl:requirement>
 <aldl:requirement name = "ND001__kmeansConfigurationIsm">
  <ns1:executable master = "no" strategy = "no" arch = "i686">/home/pascucci/Assist/compiledAssist/testKMeans//bin//
     i686-pc-linux-gnu/ND001__kmeans_ism</ns1:executable>
 </aldl:requirement>
 <aldl:requirement name = "ND001__kmeansConfigurationOsm">
  <ns1:executable master = "no" strategy = "no" arch = "i686">/home/pascucci/Assist/compiledAssist/testKMeans//bin//
     i686-pc-linux-gnu/ND001__kmeans_osm</ns1:executable>
```

Generic execution requirements

Process spec.

Per-process run-time support configuration

Deployment : the GEA loader

- Exploits the ALDL description
- Goto GEA presentation

Slides will be merged on the web site :-)

ASSIST deployment issues

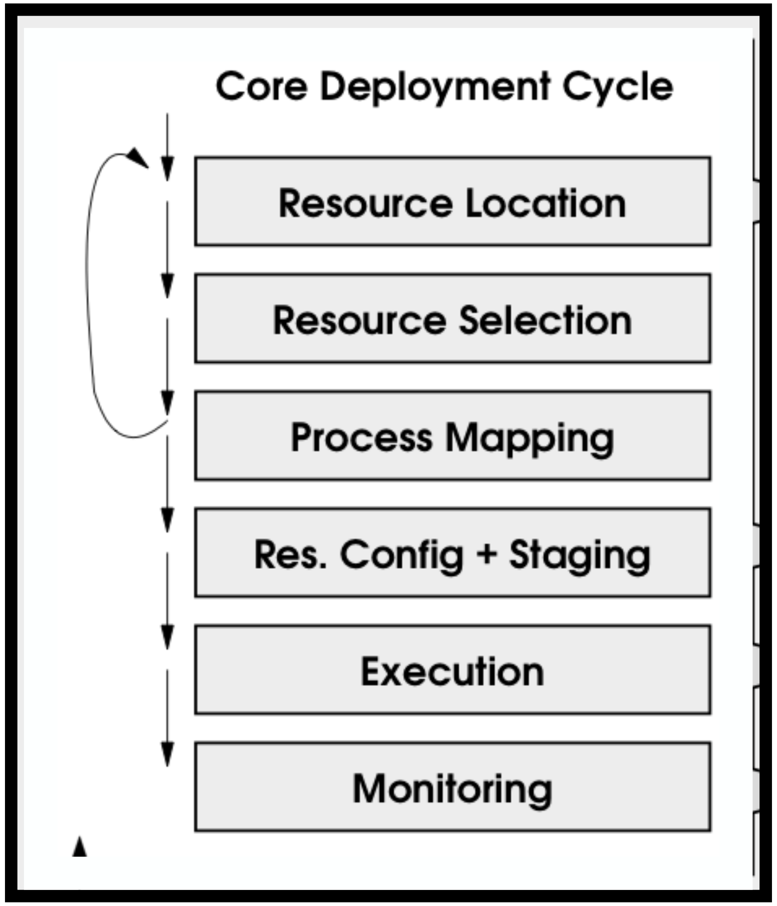
- Parallel modules
  - Reconfigure exploiting QoS models and goals
- Run-time support of the language
  - Multi-architecture (heterogeneous OS, HW)
  - On physical system (no virtual machine)
- Deployment on clusters and Grids
  - SSH, Globus
- Extendable to a component model
  - Grid.it project

GEA, the Grid Execution Agent

- Tool to automatically deploy ASSIST application
  - Implemented in Java for maximum portability
- Provides abstraction of a Grid computing platform (GAM) together with the ASSIST run-time
- Applications = multi-architecture “parallel executable” archives
  - Executable availability directs matchmaking and staging
- Essential use of the ALDL application description language
  - ALDL descriptors are compiler generated
  - do not depend on the source language

GEA Core Deployment Schema

- Deploy ASSIST applications
- Exploit High-level, structured ALDL application description
- Satisfy resource constraints
  - Static and Dynamic
  - HW, SW
  - Aggregate
- Several translation steps
- Finally exploit middleware
  - broker, allocation, staging, network configuration
- GEA provides
  - Filtering, matchmaking, mapping



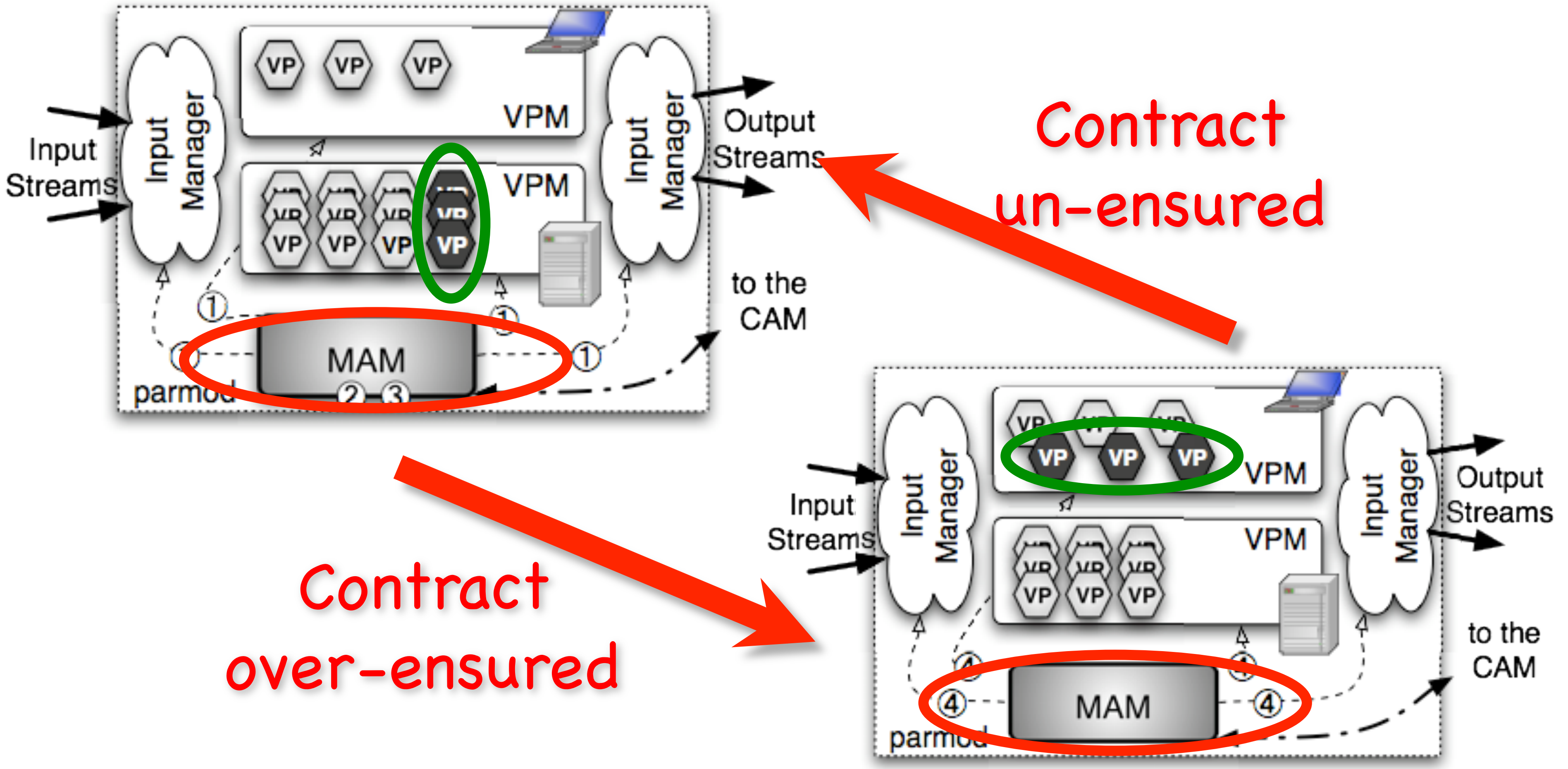
Dynamic Adaptivity

- Change resource configuration
  - Relocate processes and/or computations
  - According to environment changes
- Complex task:
  - Stop/synchronize processes
  - Exchange status / change configuration
  - Restart
- Hard to do with low-level MPI_Send() …
- Easy with ASSIST high-level code
  - Compiler knows relevant program properties (structure!!)
  - All necessary protocols are built into the run-time

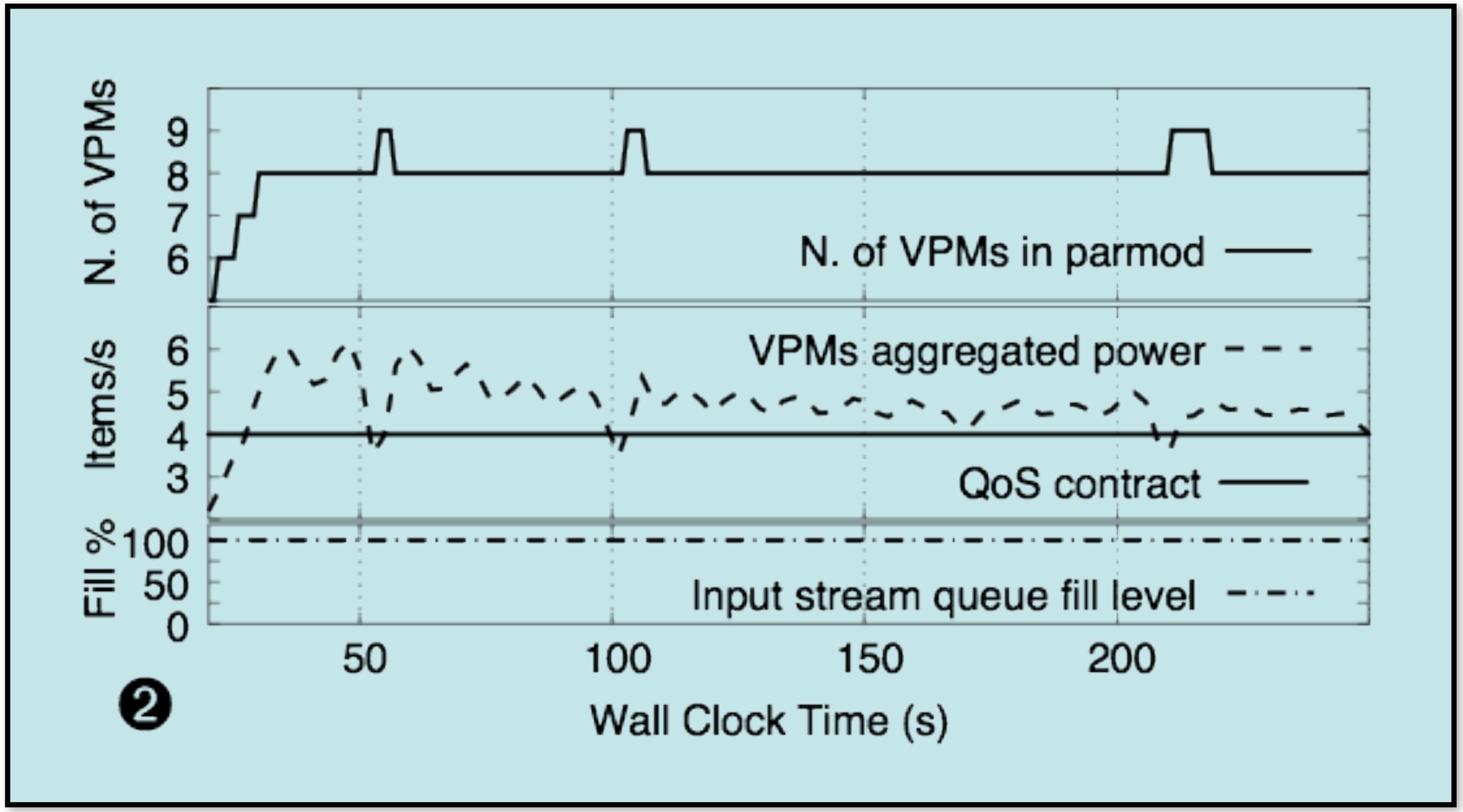
Dynamic Adaptivity

- ASSIST exploits structure-information
  - Avoid unnecessary synchronizations
  - Avoid state propagation when not useful
- Farm skeletons do not almost need to synch
  - Single stream communication can be controlled
  - Overhead is minimal
- Data parallel is reconfigurable too
  - Need to redistribute the computation
  - Same interface to add/reduce resources, and redistribute the load
- Reconfiguration either
  - User-driven
  - Based on **autonomic** control

Autonomic control



Autonomic control (2)



Adaptivity example

- Adaptivity -- p. 91

Heterogeneous Platforms

Multi-phase compilation

Conclusions

- Parallel program made up by modules
  - Declared, data-flow stream interfaces
  - Unconstrained graph
- Seq modules to encapsulate seq code
  - Multi-language, code reuse
- Parmod to express parallel activities
  - High level powerful syntax
  - Skeleton oriented, shared memory available
- Run-time exploits structure information
  - Low-level details hidden to programmers
  - Automatic mapping over platforms
  - Dynamic reconfiguration
  - Portability, performance, efficiency, load balancing…

Thanks for your attention

- Web site: **`www.di.unipi.it://Assist.html`**
- Information