Structured programming models targeting heterogeneous architectures

Marco Danelutto

with M. Aldinucci, P. Kilpatrick

Dept. of Computer Science, University of Pisa, Italy

HeteroPar 2010, Ischia
Contents

1 Hetereogeneity dimensions
2 A ”Manager first” approach
3 Structuring management
4 Increasing efficiency: structuring applications
5 Experimental results
6 Problems & Perspectives
Progress ...

1. Hetereogeneity dimensions
2. A "Manager first" approach
3. Structuring management
4. Increasing efficiency: structuring applications
5. Experimental results
6. Problems & Perspectives
Dimensions in heterogeneity

Hardware

- **PE** → distinct CPUs, or same CPU with different parameters (\#Cores, Cache, Cache coherence, Clock, Ram, Network, ...)

- **Processor** → CPU, CPU+GPU(s), CPU+Hw coproc, CPU+programmable Hw coproc (FPGAs)

- ***** → Clouds
**Dimensions in heterogeneity**

**Hardware**

- **PE** → distinct CPUs, or same CPU with different parameters (#Cores, Cache, Cache coherence, Clock, Ram, Network, ...)
- **Processor** → CPU, CPU+GPU(s), CPU+Hw coproc, CPU+programmable Hw coproc (FPGAs)
- **Clouds** → changes in the programming model(s)
Dimensions in heterogeneity (2)

Software

- **Operating system** →
  Different implementation of the same API or different, functionally equivalent APIs

- **Operating system** →
  Shared resources (dynamic load)

- **Application** →
  Different needs at different application stages/execution times/data inputs
Dimensions in heterogeneity (2)

Software

- **Operating system** →
  Different implementation of the same API or different, functionally equivalent APIs

- **Operating system** →
  Shared resources (dynamic load)

- **Application** →
  Different needs at different application stages/execution times/data inputs

- mostly **dynamic** → need for *adaptation*
Scenario

Targeting heterogeneous hardware:

- different (CPU, RAM, clock, network connection) processing elements
- different processors (family, vendor, with or without co-processors, ...)
Scenario

Targeting heterogeneous hardware:
- different (CPU, RAM, clock, network connection) processing elements
- different processors (family, vendor, with or without co-processors, ...)

Given a problem:
- a number of different, functionally equivalent software implementation exists,
- solve the problem while targeting different hardware with different efficiencies.
→ w.r.t. different non functional concerns
Use case scenario (def-1)

Image processing:

- stream of images (e.g. from a satellite)
- to be processes by a three stage filter
- stage 2 and 3 are data parallel
Image processing:

- *stream* of images (e.g. from a satellite)
- to be processes by a three stage filter
- stage 2 and 3 are data parallel

- a delay while processing the single image is accepted
- but no image should be “lost” (i.e. the inter-arrival time *has to be* the service time)
Use case scenario (def-2)

Functional concerns:
- algorithms to be used,
- qualitative parallel structuring
Use case scenario (def-2)

Functional concerns:
- algorithms to be used,
- qualitative parallel structuring

Non functional concerns:
- heterogeneity (e.g. targeting different hardware with different object code)
- performance (e.g. devising the number of parallel workers in stages 2 and 3)
- fault tolerance (e.g. substitute fault nodes + rescheduling aborted computations)
Classical approaches

General purpose strategies

- Implementation → user responsibility
- Application programmer forced to learn/apply techniques proper of system programmers
- Poor separation of concerns
- Code intermingling
- Hard to debug
- Poor maintainability
- Poor code re-use

Excess of responsibility on programmers

M. Danelutto (www.di.unipi.it/~marcod)
Classical approaches

General purpose strategies

Implementation $\rightarrow$ user responsibility

- Application programmer forced to learn/apply techniques proper of system programmers
  - poor separation of concerns
- Code intermingling
  - hard to debug
  - poor maintainability
  - poor code re-use
Classical approaches

General purpose strategies

Implementation → user responsibility

- Application programmer forced to learn/apply techniques proper of system programmers
  - poor separation of concerns
- Code intermingling
  - hard to debug
  - poor maintainability
  - poor code re-use

Excess of responsibility on programmers
Use case scenario (asm level)

Up to programmer:

- code to compile & deploy *proper* code on all the workstations and to start, stop (checkpoint?) application
  - script code
- code to set up communication framework, implement communications (with marshalling), suspend/resume, . . .
  - code intermingling with the application functional code
Use case scenario (asm level)

Up to programmer:
- code to compile & deploy proper code on all the workstations and to start, stop (checkpoint?) appl
  - script code
- code to set up communication framework, implement communications (with marshalling), suspend/resume, . . .
  - code intermingling with the application functional code
- in case of fault of a node
  - compile & deploy proper code to another WS item link it to comm framework & start it
Use case scenario (asm level)

Up to programmer:
- code to compile & deploy *proper* code on all the workstations and to start, stop (checkpoint?) application
  - script code
- code to set up communication framework, implement communications (with marshalling), suspend/resume, . . .
  - code intermingling with the application functional code
- in case of fault of a node
  - compile & deploy proper code to another WS item, link it to communication framework & start it
- in case of overload of a node:
  - move the node computation to another node, or split computation of that node
Progress ...

1. Hetereogeneity dimensions

2. A "Manager first" approach

3. Structuring management

4. Increasing efficiency: structuring applications

5. Experimental results

6. Problems & Perspectives
A “manager first approach”

Applications built out of two components:

1. the application (functional code)
2. the manager (the non functional management code)
A “manager first approach”

Applications built out of two components:
1. the application (functional code)
2. the manager (the non functional management code)

Application $\rightarrow$ in charge to the “applicative” programmer
- exploit domain specific knowledge
A “manager first approach”

Applications built out of two components:
1. the application (functional code)
2. the manager (the non functional management code)

Application → in charge to the “applicative” programmer
- exploit domain specific knowledge

Manager → in charge to the “system” programmer
- exploit system specific knowledge
- reusable, parametric, customizable
Non functional manager

NF MGR → even more important than application

Takes care of non functional concerns (fault tolerance, security, performance, heterogeneity, power saving)

To ensure user (application programmer) specified non functional goals
Non functional manager

NF MGR $\rightarrow$ even more important than application

Takes care of non functional concerns (fault tolerance, security, performance, heterogeneity, power saving)

To ensure user (application programmer) specified non functional goals

Relying on:

- possibility to “observe” system (hw&sw)
- possibility to “act” on the system (hw&sw)
Non functional manager

Implementing a *general adaptation* strategy:
Non functional manager

Implementing a general adaptation strategy:

- **STEP 1 (startup)**
  - “Observe” the system
  - figure out ”best” solution
  - apply it

- **STEP 2 (steady state)**
  - Monitor the system in case adapt loop monitoring

M. Danelutto (www.di.unipi.it/~marcod)

HeteroPar 2010, Ischia
Non functional manager

Implementing a *general adaptation* strategy:

- **STEP 1 (startup)**
  - “Observe” the system
  - figure out ”best” solution
  - apply it

- **STEP 2 (steady state)**
  - Monitor the system
  - in case adapt
  - loop monitoring
Non functional manager

Implementing a general adaptation strategy:

- **STEP 1 (startup)**
  - “Observe” the system
  - figure out ”best” solution
  - apply it

- **STEP 2 (steady state)**
  - Monitor the system
  - in case adapt
  - loop monitoring

- **STEP 1** $\rightarrow$ static management (of heterogeneity)
- **STEP 2** $\rightarrow$ dynamic management (of heterogeneity)
Base assumptions

Component based application & framework

- different “functional” components implementing (parts of) algorithms used
- interchangeable alternative implementations
- interfaces completely specified
- *use and provide* interfaces specified

- support for hierarchical components (composite components → first class components)
  - to model complex application patterns
Enforcing separation of concerns

Application programmer:

- Functional concerns:
  alternative algorithms and implementations
- Non functional concerns:
  abstract specification of expectations
Enforcing separation of concerns

Application programmer:

- Functional concerns: alternative algorithms and implementations
- Non functional concerns: abstract specification of expectations

System programmer:

- Run time support implementation & optimization
- Customizable non functional concern management infrastructure
User & system views (ideal)

The application programmer:

- Selects proper algorithm(s)/implementation(s)
- Instantiates management framework
- Provides non functional concern goals/strategies
User & system views (ideal)

The application programmer:
- Selects proper algorithm(s)/implementation(s)
- Instantiates management framework
- Provides non functional concern goals стрategies

The system programmer:
- Implements general purpose, customizable, non functional management framework
User & system views (ideal)

The application programmer:
- Selects proper algorithm(s)/implementation(s)
- Instantiates management framework
- Provides non functional concern goals/strategies

The system programmer:
- Implements general purpose, customizable, non functional management framework

The NF Management framework:
- Reads system NF requirements
- Best effort ensures user NF requirements
1. Heterogeneity dimensions
2. A "Manager first" approach
3. Structuring management
4. Increasing efficiency: structuring applications
5. Experimental results
6. Problems & Perspectives
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
2. **Structure**: hierarchical vs. peer-to-peer

M. Danelutto (www.di.unipi.it/~marcod)
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
2. **Structure**: hierarchical vs. peer-to-peer
3. **Concerns**: single vs. multiple
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
2. **Structure**: hierarchical vs. peer-to-peer
3. **Concerns**: single vs. multiple
   - multiple: independent vs. reciprocally dependent
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
2. **Structure**: hierarchical vs. peer-to-peer
3. **Concerns**: single vs. multiple
   - multiple: independent vs. reciprocally dependent
4. **System perception**: event triggers vs. polling
Structuring management

Dimensions in management of Non Functional Features

1. **Behaviour**: reactive vs. proactive
2. **Structure**: hierarchical vs. peer-to-peer
3. **Concerns**: single vs. multiple
   - multiple: independent vs. reciprocally dependent
4. **System perception**: event triggers vs. polling
5. **System influence**: RISC vs. CISC actuators
Separation of concerns within management:

- **Control**
  - as flexible as possible, under user control or “compiled” from user abstract specification(s)
Separation of concerns within management:

- **Control**
  - as flexible as possible, under user control or “compiled” from user abstract specification(s)

- **System perception**
  - comprehensive, reconfigurable set of hardware & software sensors reporting suitable measures of system and application behaviour
  - comprehensive, reconfigurable set of trigger events used to trigger management activities
Structuring management (2)

Separation of concerns within management:

- **Control**
  - as flexible as possible, under user control or “compiled” from user abstract specification(s)

- **System perception**
  - comprehensive, reconfigurable set of hw & sw sensors reporting suitable measures of system and application behaviour
  - comprehensive, reconfigurable set of trigger events used to trigger management activities

- **System influence**
  - comprehensive, reconfigurable set of actuators operating on system and application configuration & behaviour
**Set of rules**

name:: trigger, condition, priority → action

- **trigger**: event that causes rule evaluation
- **condition**: action is executed if the condition holds true (possibly uses system monitored values)
- **priority**: priority of the rule (to be considered in case more than a single rule triggers)
- **action**: (sequence of) actuator(s) operating on the system
Management: system perception

**Set of sensors**

\[ \text{Type}_i \quad X_i \]

- **Type}_i**  
  type of the sensor

- \( X_i(p_1, \ldots p_{k_i}) \)  
  value of variable \( X_i \) as currently observed in the system with given parameters

E.g.
- boolean hasGPU(131.114.3.17),
- string cpuName(nodeId),
- float serviceTime(pgmId,nodeId)
Management: system perception

**Set of sensors**

<table>
<thead>
<tr>
<th>Type&lt;sub&gt;i&lt;/sub&gt;</th>
<th>X&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
</table>

Type<sub>i</sub>  
- type of the sensor

X<sub>i</sub>(p<sub>1</sub>, . . . p<sub>k<i></sub>)  
- value of variable X<sub>i</sub> as currently observed in the system with given parameters

E.g.:

- boolean hasGPU(131.114.3.17),
- string cpuName(nodeld),
- float serviceTime(pgmId,nodeld)
Management: system perception

Set of triggers

\[ T_i(p_1, \ldots p_{k_i}) \]

- \( T_i \): trigger event \( T_i \)
- \( p_j \): parameter specifying the event
Management: system perception

Set of triggers

$T_i(p_1, \ldots p_{k_i})$

- $T_i$: trigger event $T_i$
- $p_j$: parameter specifying the event

E.g.:
- `dataAvailableOnInputChannel(ChA)`
- `newUserRequirement(X)`
- `networkDown(netSegmentId)`
Management: system influence

Set of actuators

\[ A_i(p_1, \ldots, p_{k_i}) \]

- \( A_i \): action \( A_i \)
- \( p_j \): action parameter
Management: system influence

Set of actuators

\[ A_i(p_1, \ldots, p_{k_i}) \]

- \( A_i \) action \( A_i \)
- \( p_j \) action parameter

E.g.:
- deployApplication(applId, nodeId),
- setUpCommCh(procIdA, nodeIdA, procIdB, NodeIdB),
- execute(applId, nodeId), callMethod(...)
Management components

Active part: control + events
own threads of control

Passive part: sensors + actuators
invoked from control
Management dimensions (revisited)

Behaviour

- **reactive** → rules (conditionally) react to events
- **proactive** → triggering: unused, actuators executed as result of policies, learning exploited
Management dimensions (revisited)

Behaviour

- *reactive* → rules (conditionally) react to events
- *proactive* → triggering: unused, actuators executed as result of policies, learning exploited

Structure

- *hierarchical* → communicating managers, top level policies subsume lower level ones
- *peer-to-peer* → full autonomic control, condition and events also from other peer managers
Management dimensions (revisited)

Concerns

- *single* $\rightarrow$ single set of rules
- *multiple* $\rightarrow$ independent set of rules, conditions and events used to orchestrate “independent” decisions
Management dimensions (revisited)

Concerns

- single $\rightarrow$ single set of rules
- multiple $\rightarrow$ independent set of rules, conditions and events used to orchestrate “independent” decisions

System perception

- system + user defined triggers and sensors
Management dimensions (revisited)

Concerns

- *single* → single set of rules
- *multiple* → independent set of rules, conditions and events used to orchestrate “independent” decisions

System perception

- system + user defined triggers and sensors

System influence (operation)

- CISC → large set of single, comprehensive actuators
- RISC → small set of compact composable actuators
  + structured composition rules
Use case scenario (HLL level)

Use case →
  generic application (components + assembly code)

NF MGR framework →
  provides triggers, sensors, actuators and \textit{rule engine}

Application programmer:
  \begin{itemize}
  \item provides functional components + assembly code
  \item implements NF management providing proper rules
  \end{itemize}
Use case scenario (HLL level)

Use case →
generic application (components + assembly code)

NF MGR framework →
provides triggers, sensors, actuators and rule engine

Application programmer:
- provides functional components + assembly code
- implements NF management providing proper rules

Sample rule:
```
componentFailed(worker), notEndComputation() →
(findResource(R), hasGPU(R), compileGPU(...),
deploy(...), schedule(...)) ORELSE (findResource(R),
compile(...), deploy(...), schedule(...))
```
Use case scenario (HLL level)

Pros & cons

- rules $\rightarrow$ full responsibility on application programmer
- effectiveness of rules depend on suitability of trigger/sensor/action sets
Use case scenario (HLL level)

Pros & cons

- rules → full responsibility on application programmer
- effectiveness of rules depend on suitability of trigger/sensor/action sets
+ code managing heterogeneity/NF concerns separated from application code
+ implementation of NF management mechanisms in charge of system programmers
+ application programmer responsibility for management policies
1. Hetereogeneity dimensions
2. A ”Manager first” approach
3. Structuring management
4. Increasing efficiency: structuring applications
5. Experimental results
6. Problems & Perspectives
Improving management

“Limit” programmer possibilities

- small set of composable application building blocks
  - covering major cases
  - specialized through functional code (component) params
- each building block
  - with passive management components
- control: rules orchestrating possibilities offered by sensors and actuators
Improving management

“Limit” programmer possibilities

- small set of composable application building blocks
  - covering major cases
  - specialized through functional code (component) params

- each building block
  - with passive management components

- control: rules orchestrating possibilities offered by sensors and actuators

Programming

Specializing building block by functional code + orchestrating management
Structuring application code

Algorithmic skeletons (design patterns): *known programming patterns, parametric, composable, with clear functional and non functional semantics*
Structuring application code

Algorithmic skeletons (design patterns): known programming patterns, parametric, composable, with clear functional and non functional semantics

Each skeleton:

- comes with proper implementation strategies
  - suitable to target different architectures/processors/coprocessors
- comes with proper management strategies
  - related to different concerns
  - including sensors & actuators
### Sample skeletons/patterns

<table>
<thead>
<tr>
<th>Task Farm</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Param</strong></td>
<td>worker component + data/result types</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>increase/decrease parDegree</td>
</tr>
<tr>
<td><strong>Sensors</strong></td>
<td>serviceTime $T_S(\ldots)$, interarrivalTime $T_A(\ldots)$, ...</td>
</tr>
<tr>
<td><strong>Impl.</strong></td>
<td>different implementations targeting different hw with their own abstract performance models</td>
</tr>
</tbody>
</table>
### Sample skeletons/patterns

<table>
<thead>
<tr>
<th>Task Farm</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Param</strong></td>
<td>worker component + data/result types</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>increase/decrease parDegree</td>
</tr>
<tr>
<td><strong>Sensors</strong></td>
<td>serviceTime $T_S(...)$, interarrivalTime $T_A(...)$, ...</td>
</tr>
<tr>
<td><strong>Impl.</strong></td>
<td>different implementations targeting different hw with their own abstract performance models</td>
</tr>
</tbody>
</table>

**Sample skeleton application**

```
pipeline(seq(f), farm(seq(w)), seq(g))
```
Sample policies

Farm, performance (CISC actions):
\[ T_A() < T_S() \rightarrow increaseParDegree() \]
Sample policies

Farm, performance (CISC actions):
\[ T_A() < T_S() \rightarrow increaseParDegree() \]

Pipeline performance (CISC actions):
\[ T_S(S_i) + T_S(S_{i+1}) < T_S(S_{i+2}) \rightarrow collapse(S_i, S_{i+1}) \]
Sample policies

Farm, performance (CISC actions):
\[ T_A() < T_S() \rightarrow increaseParDegree() \]

Pipeline performance (CISC actions):
\[ T_S(S_i) + T_S(S_{i+1}) < T_S(S_{i+2}) \rightarrow collapse(S_i, S_{i+1}) \]

Farm, heterogeneity (RISC actions):
\[ T_A() < T_S()\& available(PE_k)\& hasGPU(PE_k) \rightarrow makeGPU(workerCode); deploy(\ldots); schedule(\ldots) \]
\[ T_A() < T_S()\& available(PE_k) \rightarrow compile(workerCode); deploy(\ldots); schedule(\ldots) \]
Behavioural skeletons

**Behavioural skeleton**: co-design of parallelism exploitation pattern & autonomic management of NF features in a (composite) GCM component

- GCM (Grid Component Model) implementation
- within CoreGRID (EU FP6 NoE) and GridCOMP (EU FP6 STREP)
- components: Seq, Farm, Pipeline, with Autonomic managers (AM)
- AM: JBoss rule engine → control, sensors & actuators → GCM (Fractal) controller
- hierarchical NF concern management (tree of skeleton components)
Behavioural skeleton concept

- Functional concern
  - Algorithmic skeleton
  - Params...
- Non functional concern
  - Autonomic manager
  - System programmer concern
  - Application programmer concern
  - Working Application

M. Danelutto (www.di.unipi.it/~marcod)
Contracts

User supplied non functional “constrain”

- directed to top level manager
- node X splits its contract into sub contracts directed to its components nodes (e.g. pipeline contract split into stage contracts)
- abstract syntax from user $\rightarrow$ compiled to suitable JBoss rules
Hierarchical management

Node $X$ in skeleton tree with ancestor $Y$

- runs its own manager (active mode)
- ensures best effort contract propagated from root node (from user supplied contract)

Contract violation & active/passive MGR state

- Contract violation
  - Perform corrective action
- Receive contract
  - Start control loop
- Contract violation & no corrective policy applies
  - Report violation
Hierarchical composition
pipe(seq,farm(seq),seq)

Input tasks come to S and are dispatched to W according to the policy of S (scatter, unicast, multicast, broadcast). W compute results that are collected by C (gather, reduce). AM monitors execution, having received an SLA contract. In case of contract violation, it can either take local corrective actions or report violation to upper level manager (or to the user).
Multi-concern management

- Independent manager (hierarchies) relative to concerns $C_a$ and $C_b$
Multi-concern management

- Independent manager (hierarchies) relative to concerns $C_a$ and $C_b$
Multi-concern management

- Independent manager (hierarchies) relative to concerns $C_a$ and $C_b$

- “Common knowledge” to interact $\rightarrow$ annotated computation graph $\mathcal{G} = \langle N, A, L \rangle$ ($N$: nodes, $A$: arcs, $L \in \text{string}^*$ ($N \cup A$: labels)

- Consensus needed to avoid decision $D$ and $D^{-1}$
Consensus protocol

- $MGR_a$ fires rule $R_i$ leading to actions $a_1; \ldots ; a_n$
- actions cause $G \rightarrow G_{new}$
- $G_{new}$ broadcasted to all managers
- each manager analyses $G_{new}$ and answers
  - OK: assess commitment of actions leading to $G_{new}$
  - NOK: $G_{new}$ unfeasible w.r.t. the managed concern
  - provide(Property): $G_{new}$ feasible provided Property
- answers collected
  - all OK: commit rule executing actions $a_1; \ldots ; a_n$
  - at least one NOK: abort rule and reduce its priority
  - foreach provide(Property): modify $a_i$'s accordingly and eventually commit
Multi-concern management

Each “original” rule compiled to set of rules:

\[
\begin{align*}
\text{trigger}_i, \text{cond}_j & \rightarrow a_1; \ldots; a_n \\
\downarrow & \\
\text{trigger}_i, \text{cond}_j & \rightarrow \text{askConsensus}(G_{\text{new}}) \\
\text{allManagerAgree} & \rightarrow a_1, \ldots, a_n \\
\text{oneManagerDisagree} & \rightarrow \text{lowerRulePriority}, \text{abortRule} \\
\text{someManagerRequires}(\text{Prop}) & \& \\
\text{eqEnsuring}(\text{Prop}, \langle a'_1; \ldots; a'_{n'} \rangle, \langle a_1; \ldots; a_n \rangle) & \rightarrow a'_1; \ldots; a'_{n'}
\end{align*}
\]

with $G_{\text{new}}$ new computation graph resulting from $a_1; \ldots; a_n$ taken...
Use case scenario (BS level)

The applicative programmer:

- provides functional components
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly $\rightarrow$ pipe(C1, farm(W), C3)

... and all the rest is in charge of the compiler/run time
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly → pipe(C1, farm(W), C3)
- provides contracts →

... and all the rest is in charge of the compiler/run time
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly $\rightarrow$ pipe(C1, farm(W), C3)
- provides contracts $\rightarrow$
  - $Ts \in [low, high] \rightarrow AM_{perf}$
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly \[\rightarrow \text{pipe}(C1, \text{farm}(W), C3)\]
- provides contracts \[\rightarrow \]
  - \(Ts \in [\text{low}, \text{high}] \rightarrow AM_{\text{perf}}\)
  - \(\text{secureComm(on)} \rightarrow AM_{\text{sec}}\)

... and all the rest is in charge of the compiler/run time
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly → pipe(C1, farm(W), C3)
- provides contracts →
  - $Ts \in [\text{low}, \text{high}] \rightarrow AM_{perf}$
  - $secureComm(\text{on}) \rightarrow AM_{sec}$

... and all the rest is in charge of the compiler/run time
Use case scenario (BS level)

The applicative programmer:

- provides functional components
- provides assembly → pipe(C1, farm(W), C3)
- provides contracts →
  - $Ts \in [\text{low}, \text{high}] \rightarrow AM_{\text{perf}}$
  - $\text{secureComm(on)} \rightarrow AM_{\text{sec}}$

... and all the rest is in charge of the compiler/run time
Progress ...

1. Hetereogeneityit dimensions
2. A ”Manager first” approach
3. Structuring management
4. Increasing efficiency: structuring applications
5. Experimental results
6. Problems & Perspectives
BS Prototypes

Two prototypes available:

- **GMC BS:**
  - developed within GridCOMP, runs on top of ProActive/GCM (4.x), provides Functional Replication and Pipeline BS, supports hierarchical management of performance, available from http://gridcomp.ercim.org

- **LIBERO:**
  - developed within a UNIPI project, runs on top of plain Java (1.6), provides Farm, Pipeline, supports multiple managers per node (in addition to hierarchical management), available from our group at UNIPI
Experimental results

Hierarchical management of a $\text{pipe(seq,farm(seq),seq)}$ medical image processing application (GCM/BS, GridCOMP final review 2009)
Experimental results

Hierarchical management of a pipe(seq,farm(seq),seq) medical image processing application (GCM/BS, GridCOMP final review 2009)

dynamic heterogeneity
Experimental results

Hierarchical management of a pipe(seq,farm(seq),seq) medical image processing application (GCM/BS, IPDPS’09)
Experimental results

Hierarchical management of a $\text{pipe}(\text{seq}, \text{farm(\text{seq})}, \text{seq})$ medical image processing application (GCM/BS, IPDPS’09)

static heterogeneity
Experimental results

Multiple concern management (synthetic \texttt{pipe(seq,farm(seq),seq)} application): performance & security management (LIBERO, CoreGRID WS 2010)

![Diagram showing experimental results with event logs and timelines for security AM and performance AM.]
Experimental results

Multiple concern management (synthetic \texttt{pipe(seq,farm(seq),seq)} application): performance & security management (LIBERO, CoreGRID WS 2010)
Experimental results

In both cases

Limited intervention by application programmer

- provide worker and stage components (functional)
- provide XML assembly (using provided pipe and farm tags)
- provide contracts:
  - first case: the expected throughput range \([minThroughput, maxThroughput]\)
  - second case: \textit{idem}
Experimental results

Processing elements with GPUs (nVIDIA/CUDA)

- “function” code provided through architecture neutral C/C++ macros
- rules to decide whether GPU or CPU should be targeted
  - based on observed computational grain
  - on trivial arithmetic ($\forall i : x_i + +$) nVidia GTX285 GPU (240 cores) outperforms Xeon 5220 at 8K items
- farm increaseParalellismDegree rule:
  
  $T_A < T_S \land \text{available(Node}_i\text{)} \land \text{hasCudaGPU(Node}_i\text{)} \rightarrow$

  $\text{compileCuda(workerCode)}; \text{deploy(Node}_i, \ldots); \text{schedule(Node}_i)$
Experimental results

Processing elements with GPUs (nVIDIA/CUDA)

- “function” code provided through architecture neutral C/C++ macros
- rules to decide whether GPU should be targeted
  - based on observed computational grain
  - on trivial arithmetic ($\forall i: x_i + +$) nVidia GTX285 GPU (240 cores) outperforms Xeon 5220 at 8K items

- farm in ParallelismDegree rule:
  
  $$TA < TS \land \text{available}(Node_i) \land \text{hasCudaGPU}(Node_i) \rightarrow$$
  
  compileCuda(workerCode); deploy(Node_i, ...);
  schedule(Node_i)
Experimental results

Processing elements with GPUs (nVIDIA/CUDA)

- \( \text{pipe}(\text{map(seq)}, \text{map(seq)}) \rightarrow \text{map}_{\text{CUDA}}(\text{pipe(seq,seq)}) \)

- optimize data transfers to/from GPU memory
  - \( \text{CPU}_M \rightarrow \text{GPU}_M; \text{compute}; \text{GPU}_M \rightarrow \text{CPU}_M; \text{CPU}_M \rightarrow \text{GPU}_M; \text{compute}; \text{GPU}_M \rightarrow \text{CPU}_M \text{ vs. } \text{CPU}_M \rightarrow \text{GPU}_M; \text{compute}; \text{compute}; \text{GPU}_M \rightarrow \text{CPU}_M \)

- no programmer intervention in case node has \( \text{hasGPU}() \)
Progress ...

1. Heterogeneity dimensions
2. A "Manager first" approach
3. Structuring management
4. Increasing efficiency: structuring applications
5. Experimental results
6. Problems & Perspectives
Open problems

- multi concern management: more sophisticated coordination strategies
Open problems

- multi concern management: more sophisticated coordination strategies
- user contract compilation & spreading through sub-components
Open problems

- multi concern management: more sophisticated coordination strategies
- user contract compilation & spreading through sub-components
- control structuring alternatives
Open problems

- multi concern management: more sophisticated coordination strategies
- user contract compilation & spreading through sub-components
- control structuring alternatives
- action “structured” language (Appel)
Open problems

- multi concern management: more sophisticated coordination strategies
- user contract compilation & spreading through sub-components
- control structuring alternatives
- action “structured” language (Appel)
- proactive management support
Perspectives

- System knowledge
- Optimization techniques
- Control theory
- Learning

NFC
- Management (adaptive)

High level programming abstractions

Tools
- IDE, compiler, RTS

Domain specific expertise

"Abstract program"

Efficient / portable / parallel executable
Any questions?
mailto://marcod@di.unipi.it

public static void main(...) {

    Skeleton stage1 = new Stage1(...);
    Skeleton stage2 = new Farm(new Worker(...));
    Skeleton stage3 = new Stage3(...);

    Skeleton main =
        new Pipe(new Pipe(stage1, stage2), stage3);

    InputStreamManager inputData =
        new FileInputManager("datain.dat");
    OutputStreamManager outputResults =
        new ConsoleOutputManager();

    Manager mgr = new Manager(main, new ParDegree(5),
        inputData, outputResults);

    mgr. compute();

    return;