Skeletons for multi/many core systems

Marco Aldinucci, Univ. of Torino, I
Marco Danelutto, Univ. of Pisa, I
Peter Kilpatrick, Queen’s Univ. of Belfast, UK
Motivation

- Skeleton technology recognized successful
  - Muesli, SkeTo, ASSIST, Muskel, Mallba, GCM-BS

- Multi-core and many core recent development → parallel programs needed
Motivation

• Skeleton technology recognized successful

• *Muesli, SkeTo, ASSIST, Muskel, Mallba, GCM-BS*

• Multi-core and many core recent development ➔ *parallel programs needed*
Skeleton concept

- Know, efficient, reusable, parametric parallelism exploitation pattern

- Skeleton framework
  - application programmers use skeletons, system designers implement them (separation of concerns)
  - nesting (often) allowed (two tier model)
  - portability across target architectures
  - performance portability
Skeleton concept

- Know, efficient, reusable, parametric parallelism exploitation pattern
- Skeleton framework
  - application programmers use skeletons, system designers implement them (separation of concerns)
  - nesting (often) allowed (two tier model)
  - portability across target architectures
  - performance portability
Skeleton concept

- Know, efficient, reusable, parametric parallelism exploitation pattern
- Skeleton framework
  - application programmers use skeletons, system designers implement them (separation of concerns)
  - nesting (often) allowed (two tier model)
  - portability across target architectures
  - performance portability
Skeletons @ work

- Application programmer uses available skeletons
  - specialized through application code
- then
  - compiles
  - and runs the program
- no need for target hw expertise
History

late '80
Cole PhD thesis, introduction of the concept

'90
First skeleton frameworks
(P3L Pisa, Imperial College, Skil Muenster)

'00
Assessed skeleton frameworks
eSkel Edinburg, Muesli Muenster, SkeTo
Tokio, ASSIST Pisa, Muskel Pisa, Calcium
Sophia Antipolis, Mallba La Laguna
History

Languages

P3L

SkIE

ASSIST

Libraries

SkeLib

OcamlP3L

Lithium

Muskel

GCM Behavioural Skeletons
History

Languages

- P3L
- SkIE
- ASSIST
  - Two tier nesting
  - Dynamic adaptation

Libraries

- Skeleton library
- SkeLib
- OcamlP3L
- Lithium
- Muskel
- GCM Behavioural Skeletons
  - Macro data flow impl.
  - Application manager
  - Per skeleton autonomic management

Dynamic adaptation

Application manager

Per skeleton autonomic management

Two tier nesting

Macro data flow impl.
Muskel

• Macro data flow implementation model (1999)
• Stream parallel skeletons (pipe, farm)
  • map recently added
• Application manager
  • with user supplied contracts and autonomic behaviour
• User defined skeletons
  • same efficiency of system provided skeletons
Muskel (user view)

- **Skeleton function wrapper**
  - `Object compute(Object)`
- **Skeletons ➔ program**
  - `Farm(Skeleton w), Pipe(Skeleton a, Skeleton b)`
- **Manager**
  - **params**: I/O, program, contract, parallel execution
- **User**
  - defines skeleton programs
  - invokes execution through Manager
import muskel.*;
public class MySkelProgram {
    public static void main(String[] args) {

        Skeleton stage1 = new Stage1();
        Skeleton worker = new Worker();
        Skeleton stage2 = new Farm(worker);
        Skeleton main = new Pipeline(stage1, stage2);

        InputManager inputStream = new InputStreamMgr(args[0]);
        OutputManager outputStream = new ConsoleOutMgr();

        PerfContract parDegree =
            new ParDegree(Integer.parseInt(args[1]));

        Manager mgr =
            new Manager(main, inputStream, outputStream, parDegree);

        mgr.compute();
    }
}
Muskel (macro data flow)

• Skeleton program ➔ macro data flow graph

• Input task ➔ MDF graph instance

• Execution ➔ distributed macro data flow interpreter

• get fireable instruction (from any graph) ➔ execute ➔ store back result tokens ➔ loop
Muskel (macro data flow)

- Skeleton program ➔ macro data flow graph
- Input task ➔ MDF graph instance
- Execution ➔ distributed macro data flow interpreter

- get fireable instruction (from any graph) ➔ execute ➔ store back result tokens ➔ loop

Skeleton stage1 = new Stage1();
Skeleton worker = new Worker();
Skeleton stage2 = new Farm(worker);
Skeleton main = new Pipeline(stage1,stage2);
Muskel (macro data flow)

- Skeleton program ➞ macro data flow graph
- Input task ➞ MDF graph instance
- Execution ➞ distributed macro data flow interpreter
- get fireable instruction (from any graph) ➞ execute ➞ store back result tokens ➞ loop

```java
Skeleton stage1 = new Stage1();
Skeleton worker = new Worker();
Skeleton stage2 = new Farm(worker);
Skeleton main = new Pipeline(stage1,stage2);
```
Muskel (macro data flow)

- Skeleton program ➔ macro data flow graph
- Input task ➔ MDF graph instance
- Execution ➔ distributed macro data flow interpreter

- get fireable instruction (from any graph) ➔ execute ➔ store back result tokens ➔ loop

Skeleton stage1 =
new Stage1();
Skeleton worker =
new Worker();
Skeleton stage2 =
new Farm(worker);
Skeleton main   =
new Pipeline(stage1,stage2);
Muskel (execution)
Muskel: Orc modeling

system(pgm, tasks, contract, G, t) ≡
  taskpool.add(tasks) | discovery(G, pgm, t) | manager(pgm, contract, t)

discovery(G, pgm, t) ≡
  (\exists_{g \in G} \ (if \ remw \neq \ false \ => \ rworkerpool.add(remw)
    where \ remw :\in\ (g.can.execute(pgm) | Rtimer(t) => let(false) ) ) )
  => discovery(G, pgm, t)

manager(pgm, contract, t) ≡
  | i \in [1..contract]: (rworkerpool.get > remw > ctrlthread_i(pgm, remw, t))
  | monitor

ctrlthread_i(pgm, remw, t) ≡ taskpool.get > tk >
  ( if valid => resultpool.add(r) => ctrlthread_i(pgm, remw, t)
  | if \neg valid => ( taskpool.add(tk)
  | \alarm.put(i) => c_i.get > w > ctrlthread_i(pgm, w, t) ) )

where (valid, r) :\in
  ( remw(pgm, tk) > r > let(true, r) | Rtimer(t) => let(false, 0) )

monitor ≡ alarm.get > i > rworkerpool.get > remw > c_i.put(remw)
  => monitor
3 parallel activities in the system: task repository, discovery and manager

\[
\text{system}(\text{pgm}, \text{tasks}, \text{contract}, G, t) \triangleq \\
\text{taskpool}.\text{add}(\text{tasks}) \mid \text{discovery}(G, \text{pgm}, t) \mid \text{manager}(\text{pgm}, \text{contract}, t)
\]

\[
\text{discovery}(G, \text{pgm}, t) \triangleq \\
(\{g \in G \mid \text{if } \text{remw} \neq \text{false} \implies \text{rworkerpool}.\text{add}(\text{remw})\\
\quad \text{where } \text{remw} :\in (g.\text{can\_execute}(\text{pgm}) \mid \text{Rtimer}(t) \implies \text{let}(\text{false}) ) ) )
\implies \text{discovery}(G, \text{pgm}, t)
\]

\[
\text{manager}(\text{pgm}, \text{contract}, t) \triangleq \\
| i \leq i \leq \text{contract}.: (\text{rworkerpool}.\text{get} > \text{remw} > \text{ctrlthread}_i(\text{pgm}, \text{remw}, t)) \\
\mid \text{monitor}
\]

\[
\text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \triangleq \text{taskpool}.\text{get} > \text{tk} > \\
( \text{if } \text{valid} \implies \text{resultpool}.\text{add}(r) \implies \text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \\
\mid \text{if } \neg \text{valid} \implies (\text{taskpool}.\text{add}(\text{tk}) \\
\quad \mid \text{alarm}.\text{put}(i) \implies c_i.\text{get} > w > \text{ctrlthread}_i(\text{pgm}, w, t) ) )
\quad \text{where } (\text{valid}, r) :\in \\
(\text{remw}(\text{pgm}, \text{tk}) > r > \text{let}(\text{true}, r) \mid \text{Rtimer}(t) \implies \text{let}(\text{false}, 0) )
\]

\[
\text{monitor} \triangleq \text{alarm}.\text{get} > i > \text{rworkerpool}.\text{get} > \text{remw} > c_i.\text{put}(\text{remw})
\implies \text{monitor}
\]
discovery adds resources to the resource repository (infinite loop)

3 parallel activities in the system: task repository, discovery and manager

\[ \text{system}(\text{pgm}, \text{tasks}, \text{contract}, \text{G}, t) \triangleq \]
\[ \text{taskpool.add(tasks)} | \text{discovery}(\text{G}, \text{pgm}, t) | \text{manager}(\text{pgm}, \text{contract}, t) \]

\[ \text{discovery}(\text{G}, \text{pgm}, t) \triangleq \]
\[ (\exists g \in \text{G} \ ( \text{if } \text{remw} \neq \text{false} \Rightarrow \text{rworkerpool.add(remw)}) \]
\[ \text{where } \text{remw} : \in ( \text{g.can_execute}(\text{pgm}) | \text{Rtimer}(t) \Rightarrow \text{let(false)} ) ) \]
\[ \Rightarrow \text{discovery}(\text{G}, \text{pgm}, t) \]

\[ \text{manager}(\text{pgm}, \text{contract}, t) \triangleq \]
\[ | i \ 1 \leq i \leq \text{contract} : (\text{rworkerpool.get} > \text{remw} > \text{ctrlthread}_i(\text{pgm}, \text{remw}, t)) \]
\[ | \text{monitor} \]

\[ \text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \triangleq \text{taskpool.get} > \text{tk} > \]
\[ ( \text{if } \text{valid} \Rightarrow \text{resultpool.add}(r) \Rightarrow \text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \]
\[ | \text{if } \neg \text{valid} \Rightarrow ( \text{taskpool.add(tk)} \]
\[ | \text{alarm.put}(i) \Rightarrow \text{c}_i.get > w > \text{ctrlthread}_i(\text{pgm}, w, t) ) ) \]
\[ \text{where } (\text{valid, r}) : \in \]
\[ ( \text{remw}(\text{pgm}, \text{tk}) > r > \text{let(true, r)} | \text{Rtimer}(t) \Rightarrow \text{let(false, 0)} ) \]

\[ \text{monitor} \triangleq \text{alarm.get} > i > \text{rworkerpool.get} > \text{remw} > \text{c}_i.put(\text{remw}) \]
\[ \Rightarrow \text{monitor} \]
Muskel: Orc modeling

3 parallel activities in the system: task repository, discovery and manager

discovery adds resources to the resource repository (infinite loop)

manager forks control threads (user required #)

\[
\text{system}(\text{pgm}, \text{tasks}, \text{contract}, G, t) \triangleq \\
\text{taskpool}.\text{add}(\text{tasks}) \mid \text{discovery}(G, \text{pgm}, t) \mid \text{manager}(\text{pgm}, \text{contract}, t)
\]

\[
\text{discovery}(G, \text{pgm}, t) \triangleq \\
(\forall g \in G \:\text{if } \text{remw} \neq \text{false} \Rightarrow \text{rworkerpool}.\text{add}(\text{remw}) \\
\text{where } \text{remw} :\in (\ g.\text{can}_\text{execute}(\text{pgm}) \mid \text{Rtimer}(t) \Rightarrow \text{let}(\text{false}) \ ) ) ) \\
\Rightarrow \text{discovery}(G, \text{pgm}, t)
\]

\[
\text{manager}(\text{pgm}, \text{contract}, t) \triangleq \\
| i \; 1 \leq i \leq \text{contract}.:\text{(rworkerpool}.\text{get} > \text{remw} > \text{ctrlthread}_i(\text{pgm}, \text{remw}, t)) \\
\mid \text{monitor}
\]

\[
\text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \triangleq \text{taskpool}.\text{get} > \text{tk} > \\
(\text{if } \text{valid} \Rightarrow \text{resultpool}.\text{add}(r) \Rightarrow \text{ctrlthread}_i(\text{pgm}, \text{remw}, t) \\
\mid \text{if } \neg \text{valid} \Rightarrow (\text{taskpool}.\text{add}(\text{tk}) \\
\mid \text{alarm}.\text{put}(i) \Rightarrow \text{c}_i.\text{get} > \text{w} > \text{ctrlthread}_i(\text{pgm}, \text{w}, t)) ) \\
\text{where } (\text{valid}, r) :\in \\
(\text{remw}(\text{pgm}, \text{tk}) > r > \text{let}(\text{true}, r) \mid \text{Rtimer}(t) \Rightarrow \text{let}(\text{false}, 0) )
\]

\[
\text{monitor} \triangleq \text{alarm}.\text{get} > i > \text{rworkerpool}.\text{get} > \text{remw} > \text{c}_i.\text{put}(\text{remw}) \\
\Rightarrow \text{monitor}
\]
Muskel: Orc modeling

3 parallel activities in the system: task repository, discovery and manager

discovery adds resources to the resource repository (infinite loop)

manager forks control threads (user required #)

a control thread looks for a fireable MDF instr, executes it remotely and in case of fault it stores back the instruction and a new control thread is forked

\[\text{system}(\text{pgm}, \text{tasks}, \text{contract}, G, t) \triangleq \]
\[\text{taskpool.add}(\text{tasks}) \mid \text{discovery}(G, \text{pgm}, t) \mid \text{manager}(\text{pgm}, \text{contract}, t)\]

\[\text{discovery}(G, \text{pgm}, t) \triangleq \]
\[\left(\begin{array}{c}
\forall g \in G \left( \text{if } \text{remw} \neq \text{false} \Rightarrow \text{rworkerpool.add}(\text{remw})
\text{ where } \text{remw} : \in \left( \text{g.can.execute}(\text{pgm}) \mid \text{Rtimer}(t) \Rightarrow \text{let(false)} \right) \right)
\Rightarrow \text{discovery}(G, \text{pgm}, t)
\end{array}\right)\]

\[\text{manager}(\text{pgm}, \text{contract}, t) \triangleq \]
\[\left(\begin{array}{c}
i \leq i \leq \text{contract} : (\text{rworkerpool.get} > \text{remw} > \text{ctlthread}_i(\text{pgm}, \text{remw}, t))
\mid \text{monitor}
\end{array}\right)\]

\[\text{ctlthread}_i(\text{pgm}, \text{remw}, t) \triangleq \text{taskpool.get} > \text{tk} >
\left(\begin{array}{c}
\text{if } \text{valid} \Rightarrow \text{resultpool.add}(r) \Rightarrow \text{ctlthread}_i(\text{pgm}, \text{remw}, t)
\mid \text{if } \neg \text{valid} \Rightarrow (\text{taskpool.add}(\text{tk})
\mid \text{alarm.put}(i) \Rightarrow \text{c}_i.get > w > \text{ctlthread}_i(\text{pgm}, w, t))
\end{array}\right)\]

\[\text{where } (\text{valid}, r) : \in\]
\[\left(\begin{array}{c}
\text{remw}(\text{pgm}, \text{tk}) > r > \text{let(true, r)} \mid \text{Rtimer}(t) \Rightarrow \text{let(false, 0)}
\end{array}\right)\]

\[\text{monitor} \triangleq \text{alarm.get} > i > \text{rworkerpool.get} > \text{remw} > \text{c}_i.put(\text{remw})
\Rightarrow \text{monitor}\]
Muskel: Orc modeling

3 parallel activities in the system: task repository, discovery and manager

discovery adds resources to the resource repository (infinite loop)

manager forks control threads (user required #)

a control thread looks for a fireable MDF instr, executes it remotely and in case of fault it stores back the instruction and a new control thread is forked
Muskel going MC

- Multi-core
  - single chip, shared memory hierarchy, private caches
- Single multi-core
  - minor changes:
    - control thread operates on RemoteInterpreter Objects rather than looking up RMI objects
    - #core discovery through /proc inspection
Modeling

• The original discovery:

\[
\text{discovery}(G, pgm, t) \triangleq \\
\left( \| g \in G \left( \text{if } remw \neq \text{false} \implies rworkerpool.add(remw) \right) \right) \\
\text{where } remw \in (g\text{.can}\_\text{execute}(pgm) \mid Rtimers(t) \implies \text{let(false) } ) \\
\implies \text{discovery}(G, pgm, t)
\]

• is substituted by:

\[
\text{discovery}(G, pgm, t) \triangleq \\
\left( |i=1,n_{core} \text{let(worker.new())} \right) > w > \text{rmworkerpool.add(w)}
\]

• an object (rather than a remote object) is eventually used by control threads
Scalability
Grain

![Graph showing completion time and efficiency vs distributed MDF interpreter instances for different tasks and efficiency levels.](image-url)
Integer vs. FP

Efficiency vs. Distributed MDF interpreter instances

- Dual Qquad core (integer computation)
- Dual quad core (floating point computation)
- Efficiency (ideal)
Muskel going MC-NOW

- Discovery process modified:
  - PEs announce IP, MDF interpreter version, \textbf{core} \#

- Manager modified:
  - \textbf{cores} recruited up to ParDegree
    - several policies used (e.g. all cores first, round robin on MC-PEs, ...)
  - RMI native multi threading exploited on Remote Interpreters (stateless!)
Modeling

- Original discovery process

\[ \forall g \in G \ ( \text{if } \text{remw} \neq \text{false} \Rightarrow \text{rworkerpool.add(remw)} \] 
where \( \text{remw} :\in (g.\text{can.execute(pgm)}|\text{Rtimer}(t) \Rightarrow \text{let(false)}) \)

- is changed to:

\[ (\forall g \in G \ ( \text{if } \text{remw} \neq \text{false} \Rightarrow (\forall i \in [1,\#\text{ncore}] \text{rworkerpool.add(remw)} ) \] 
where \( (\text{remw}, \#\text{ncore}) :\in (g.\text{can.execute(pgm)}|\text{Rtimer}(t) \Rightarrow \text{let(false)}) \) )

- core number of remote nodes is reported to the manager
MC-NOW scalability
MC-NOW scalability

Completion time (measured)
Completion time (ideal)
Efficiency (measured)
Efficiency (ideal)

MC-NOW scalability

use all cores of a discovered machine before moving to the next one
MC-NOW scalability

![Graph showing the scalability of MC-NOW with respect to the number of cores. The x-axis represents the number of cores, ranging from 0 to 16. The y-axis represents the completion time (in seconds) for 1K tasks. Multiple lines depict different completion times and efficiencies for varying task lengths (1x, 16x, 128x). The graph illustrates the efficiency and completion time improvements as the number of cores increases.]
MC-NOW scalability

![Graph showing completion time and efficiency against number of cores]

- **Completion time (task len=1x) (measured)**
- **Completion time (task len=16x) (measured)**
- **Completion time (task len=128x) (measured)**
- **Efficiency (task len=1x)**
- **Efficiency (task len=16x)**
- **Efficiency (task len=128x)**
- **Efficiency (ideal)**

Use available cores from different machines in a round robin way.
Why does it work?

- Medium/Coarse grain
  - tens of milliseconds of computation per in/out packet
  - control threads do not compete too much to access task pool

- Communication/serialization hiding
  - parallelism “in excess”

- Skeleton structure
  - much more knowledge available than with traditional, “free hand” parallel programming
Conclusions

- Feasibility of porting skeleton technology to multi (many?) core systems
  - seamless, not the same for other programming models
- Assessed by experimental results
  - on single multi core system, on MC COW/NOWs
- Implementation supported by ORC modeling
  - significantly simplifies the programming effort
Future work

- Experimenting different recruitment policies
- *user configurable through Manager*
- Porting experience to many core processing elements
  - *many core programming model harder than multi core*
  - *but most of the advantages in the muskel approach are preserved*
Thank you for your attention

Any questions?