Skeletons from grids to multicores

Marco Danelutto

Dept. of Computer Science, University of Pisa, Italy

October 2010, Amsterdam
Current situation

Complex algorithms

Parallel software

Emergencies

Compiler & RTS

Hardware (parallel)

Knowledge

Coding

User

Pressure
Vision

Complex algorithms

(tools)

High level specification

knowledge

Emergencies

Pressure

Compiler & RTS

Parallel software

Adaptive

Hardware (parallel)

Emergencies

Non-functional concerns

User

Conclusions

Multicores

Scenario

Urgencies

Hardware (parallel)

Multicores

Scenario

FastFlow

Experimental results

Offloading

Non functional concerns

Scenario

Behavioural skeletons

Hierarchical composition

Multiple concern management

Conclusions

Introduction

Scenario

Urgencies

Multicores

Scenario

FastFlow

Experimental results

Offloading
Structured parallel programming

Skeletons from grids to multicores

M. Danelutto

Introduction
Scenario
Urgencies

Multicores
Scenario
FastFlow
Experimental results
Offloading

Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management

Conclusions
Structured parallel programming

Algorithmic skeletons

- Cole 1988 → common, parametric, reusable parallelism exploitation pattern
- languages & libraries since ’90 (P3L, Skil, eSkel, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, BS, ...)
- high level parallel abstractions (parallel programming community)
  - hiding most of the technicalities related to parallelism exploitation
  - directly exposed to application programmers
Structured parallel programming

Algorithmic skeletons

- Cole 1988 → common, parametric, reusable parallelism exploitation pattern
- languages & libraries since ’90 (P3L, Skil, eSkel, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, BS, …)
- high level parallel abstractions (parallel programming community)
  - hiding most of the technicalities related to parallelism exploitation
  - directly exposed to application programmers

Parallel design patterns

- design patterns à la Gamma book
  - name, problem, solution, use cases, etc.
- concurrency, algorithms, implementation, mechanisms
Parallelism

- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details
Parallelism
- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details

New architectures
- *Heterogeneous* in Hw & Sw
- *Multicore* NUMA, cache coherent architectures
Parallelism

- parallelism exploitation patterns shared among applications
- separation of concerns:
  - system programmers $\rightarrow$ efficient implementation of parallel patterns
  - application programmers $\rightarrow$ application specific details

New architectures

- *Heterogeneous* in Hw & Sw
- *Multicore* NUMA, cache coherent architectures

Further non functional concerns

- security, fault tolerance, power management, ...
Urgencies

→ Targeting multi/many cores
  - different implementation issues and solutions
  - completely different computational grains to be addressed

→ Targeting non functional concerns
  - autonomic management of independent non functional concerns
  - co-management of different non functional concerns
→ Targeting multi/many cores
- different implementation issued and solutions
- completely different computational grains to be addressed

→ Targeting non functional concerns
- autonomic management of independent non functional concerns
- co-management of different non functional concerns

Structured parallel programming models
- can be exploited to address both issues
- synergies among the solutions may be exploited as well
Targeting multicores
Targeting multi/many cores

Structured parallel programming on COW/NOW

Implementation template based

- P3L, eSkel, Muesli, SkeTo $\rightarrow$ collection of $\langle$Architecture, ProcessNetwork, Model$\rangle$

Macro Data Flow based

- Lithium/Muskel, Skipper, Calcium (Skandium) $\rightarrow$ compile skeletons to MacroDataFlow graphs + Distributed MDF interpreter
Structured parallel programming on COW/NOW

Implementation template based

- P3L, eSkel, Muesli, SkeTo → collection of \langle Architecture, ProcessNetwork, Model \rangle

Macro Data Flow based

- Lithium/Muskel, Skipper, Calcium (Skandium) → compile skeletons to MacroDataFlow graphs + Distributed MDF interpreter

Emphasis

- communication latency hiding
- avoid unnecessary data copies
Multi/many core features

Shared memory hierarchy

- NUMA (C vs. M accesses, non uniform inter-core interconnection structures)
- cache coherence (snoopy vs. directory based)

Control abstractions

- threads (user vs. system space, completion vs. time sharing)
- processes
Multi/many core features

Shared memory hierarchy
- NUMA (C vs. M accesses, non uniform inter-core interconnection structures)
- cache coherence (snoopy vs. directory based)

Control abstractions
- threads (user vs. system space, completion vs. time sharing)
- processes

Focus
- synchronization overheads
- data access patterns
- thread-core binding, affinity scheduling
Advanced programming framework

- targeting multicores
- minimizing synchronization latencies
- streaming support through skeletons
- expandable
- open source

Applications & Problem Solving Environments
- Directly programmed applications and further abstractions targeting specific usage (e.g., accelerator & self-offloading)

FastFlow
- High-level programming
- Composable parametric patterns of streaming networks
  - Skeletons: Pipeline, farm, D&C, ...
- Low-level programming
- Arbitary streaming networks
  - Lock-free SPMC, MPSC, MPMC queues, non-determinism, cyclic networks
- Run-time support
- Linear streaming networks
  - Lock-free SPSC queues and threading model, Producer-Consumer paradigm

Multi-core and many-core
- cc-UMA or cc-NUMA
FastFlow: simple streaming networks

Single Producer Single Consumer (SPSC) queue

- uses results from the ’80s
- lock-free, wait-free
- no memory barriers for Total Store Order processor (e.g. Intel, AMD)
- single memory barrier for weaker memory consistency models (e.g. PowerPC)

→ very low latency in communications
FastFlow: simple streaming networks

Other queues: SPMC MPSC MPCP

- one-to-many, many-to-one and many-to-many synchronization and data flow
- use an explicit arbiter thread
- providing lock-free and wait-free arbitrary data-flow graphs
- cyclic graphs (provably deadlock-free)
FastFlow: high level programming abstractions

Several “streaming” skeletons provided

- farm

\[\ldots x_{i+1}, x_i, x_{i-1} \ldots \rightarrow \text{SPMC} \rightarrow \text{MPSC} \rightarrow \ldots f(x_{i+1}), f(x_i), f(x_{i-1}) \ldots\]

- pipeline

\[\ldots x_{i+1}, x_i, x_{i-1} \ldots \rightarrow f \rightarrow g \rightarrow \ldots g(f(x_{i+1})), g(f(x_i)), g(f(x_{i-1})) \ldots\]

- farm with feedback

\[\ldots x_{j}', x_{j+1}', \ldots \rightarrow \text{SPMC} \rightarrow \text{MPSC} \rightarrow \ldots f(x_{i+1}), f(x_{j-1}'), f(x_i), f(x_{i-1}) \ldots\]
FastFlow: results

![Graph showing Speedup vs. FastFlow's worker threads with different latency times (ideal, 0.5us, 1us, 5us).]
FastFlow: results

The diagram shows the speedup of FastFlow’s worker threads compared to ideal, 0.5us, 1us, and 5us communication latencies. The plot indicates that lock-free communications offer an advantage, with the speedup increasing as the number of FastFlow’s worker threads increases. This suggests that lock-free communications can significantly improve performance in multi-threaded environments.

**Speedup**

Shows the advantage of lock free comms
## FastFlow: results

### Matrix multiplication
- parallel for i only

<table>
<thead>
<tr>
<th>n. workers</th>
<th>matmul_ff_v1</th>
<th></th>
<th>matmul_ff_v2</th>
<th></th>
<th>matmul_OpenMP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
<td>Time</td>
<td>Speedup</td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td>4</td>
<td>2982.24</td>
<td>3.6</td>
<td>2691.74</td>
<td>4.0</td>
<td>2767.48</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>1495.87</td>
<td>7.2</td>
<td>1435.41</td>
<td>7.5</td>
<td>1455.73</td>
<td>7.4</td>
</tr>
<tr>
<td>12</td>
<td>1496.01</td>
<td>7.2</td>
<td>1431.34</td>
<td>7.5</td>
<td>1492.87</td>
<td>7.2</td>
</tr>
<tr>
<td>16</td>
<td>1495.74</td>
<td>7.2</td>
<td>1426.42</td>
<td>7.6</td>
<td>1433.31</td>
<td>7.5</td>
</tr>
</tbody>
</table>
FastFlow: results

Matrix multiplication

- parallel for i only

<table>
<thead>
<tr>
<th>n. workers</th>
<th>matmul_ff_v1</th>
<th></th>
<th>matmul_ff_v2</th>
<th></th>
<th>matmul_OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
<td>Time</td>
<td>Speedup</td>
<td>Time</td>
</tr>
<tr>
<td>4</td>
<td>2982.24</td>
<td>3.6</td>
<td>2691.74</td>
<td>4.0</td>
<td>2767.48</td>
</tr>
<tr>
<td>8</td>
<td>1495.87</td>
<td>7.2</td>
<td>1435.41</td>
<td>7.5</td>
<td>1455.73</td>
</tr>
<tr>
<td>12</td>
<td>1496.01</td>
<td>7.2</td>
<td>1431.34</td>
<td>7.5</td>
<td>1492.87</td>
</tr>
<tr>
<td>16</td>
<td>1495.74</td>
<td>7.2</td>
<td>1426.42</td>
<td>7.6</td>
<td>1433.31</td>
</tr>
</tbody>
</table>

FastFlow vs. OpenMP

Comparable at quite coarse grain
### FastFlow: different apps

#### Microbenchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>Skeleton used</th>
<th>Speedup / #cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Mult.</td>
<td>1024x1024</td>
<td>farm no collector</td>
<td>7.6 / 8</td>
</tr>
<tr>
<td>Quicksort</td>
<td>50M integers</td>
<td>D&amp;C</td>
<td>6.8 / 8</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>Fib(50)</td>
<td>D&amp;C</td>
<td>9.21 / 8</td>
</tr>
</tbody>
</table>
FastFlow: different apps

Microbenchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>Skeleton used</th>
<th>Speedup / #cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Mult.</td>
<td>1024x1024</td>
<td>farm no collector</td>
<td>7.6 / 8</td>
</tr>
<tr>
<td>Quicksort</td>
<td>50M integers</td>
<td>D&amp;C</td>
<td>6.8 / 8</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>Fib(50)</td>
<td>D&amp;C</td>
<td>9.21 / 8</td>
</tr>
</tbody>
</table>

Real use cases

<table>
<thead>
<tr>
<th>Application</th>
<th>Skeleton used</th>
<th>Measure</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>YaDT-FF (C4.5 data mining)</td>
<td>D&amp;C</td>
<td>Speedup</td>
<td>4.5-7.5</td>
</tr>
<tr>
<td>StochKit-FF (Gillespie)</td>
<td>farm</td>
<td>Scalability</td>
<td>10-11</td>
</tr>
<tr>
<td>SWPS3-FF (Gene matching)</td>
<td>farm no collector</td>
<td>GCUPS</td>
<td>12.5-34.5</td>
</tr>
</tbody>
</table>
FastFlow: offloading

General purpose methodology

- use FastFlow as an efficient, fine grain sw accelerator

```c
//<includes; ff.includes>
#define N 1024
int A[N][N],B[N][N],C[N][N];
int main() {
    // < init A,B,C>
    for(int i=0;i<N;++i) {
        for(int j=0;j<N;++j) {
            int C=0;
            for(int k=0;k<N;++k)
                C += A[i][k]*B[k][j];
            C[i][j]=C;
        }
    }
    // < init A,B,C>
    for (int i=0;i<N;i++) {
        for(int j=0;j<N;++j) {
            task_t * task = new task_t(i,j);  
            farm.offload(task);
        }
    }
    farm.offload((void *)ff::FF_EOS);
    farm.wait();  // Here join
}
```

a) Original code (sequential)

```c
#define N 1024
int A[N][N],B[N][N],C[N][N];
int main () {
    // < ... }
}
```

b) FastFlow accelerated code (parallel)
FastFlow pros and cons

Pros:
- very low overhead introduced
- high level abstractions available to application programmers
- streaming fully supported

Cons:
- parallelization of code requires “more code” w.r.t. classical approaches
- “structured” approach to parallelism required
Targeting non functional concerns
The scenario

Several important non functional features to be considered:

- performance → throughput, latency, load balancing
- security → data, code
- fault tolerance → checkpointing, recovery strategies
- power management → power/speed tradeoff

Non functional

- does not contribute to function computed
- policies & strategies most likely in the background of system programmers

Autonomic management

- separation of concerns: system programmers (NF) vs. appl programmers (FUN)
Behavourial skeletons

**Def: Behavioural skeleton**

Co-design of a component including:

- parallelism exploitation pattern
- autonomic management of some non functional concern

Co-design

- improves efficiency
- exploits knowledge relative to structure of the computation
- in the design and implementation of suitable management policies
BS: user view

System programmer

Autonomic manager

Behavioural Skeleton Library

Parallelism exploitation pattern

Application programmer

Application dependent parameters

BS (composition)

Runnable application

Interface
Sample Behavioural skeleton

Functional replication BS
Sample Behavioural skeleton

**Functional replication BS**

**Parallel pattern**

- Master-worker with variable number of workers.
- Master schedules tasks to available workers.
Sample Behavioural skeleton

**Functional replication BS**

**Parallel pattern**
- Master-worker with variable number of workers.
- Master schedules tasks to available workers.

**Performance manager**
- Interarrival time faster than service time $\rightarrow$ increase parallelism degree, unless communication bandwidth is saturated.
- Interarrival time slower than service time $\rightarrow$ decrease the parallelism degree.
- Recent change $\rightarrow$ do not apply any action for a while.
Implementation: behavioural skeleton

Triggers ➔ start manager activities
Sensors ➔ determine pattern perception from the manager
Actuators ➔ determine manager intervention possibilities
Implementation: manager

Monitor
- perceive pattern status

Analyse
- figure out policies

Plan
- devise strategy

Execute
- implement decisions on pattern
Implementation: manager

Monitor
- perceive pattern status

Analyse
- figure out policies

Plan
- devise strategy

Execute
- implement decisions on pattern

MAPE loop implementation
Cyclic execution of a JBoss business rule engine
- RULE :: Priority, Trigger, Condition → Action
- Rule set :: derived from user supplied contract
Hierarchical composition

Program $\rightarrow$ Composition of BS

- pipe(seq, farm(seq), seq)

Hierarchy of managers

- user contract directed to top level manager
- propagated (possibly modified) through manager tree

Management “exceptions”
manager not able to ensure contract

$\rightarrow$ raises violation to upper manager in hierarchy

$\rightarrow$ enters *passive* mode waiting for a new contract
Hierarchical composition

Program → Composition of BS
- pipe(seq, farm(seq), seq)

Hierarchy of managers
- user contract directed to top level manager
- propagated (possibly modified) through manager tree

Management “exceptions”
manager not able to ensure contract
- raises violation to upper manager in hierarchy
- enters *passive* mode waiting for a new contract
**Hierarchical composition**

Program $\rightarrow$ Composition of BS

- $\text{pipe(seq, farm(seq), seq)}$

**Hierarchy of managers**

- user contract directed to top level manager
- propagated (possibly modified) through manager tree

**Management “exceptions”**

manager not able to ensure contract

- raises violation to upper manager in hierarchy
- enters *passive* mode waiting for a new contract
BS: sample run

Medical image processing

- pipe(seq(getImage),
  farm(seq(renderImage)),
  seq(displayImage))
- contract $\rightarrow$ 0.3 to 0.7 images per second
- initial condition
  - enough processing resources
  - image provider stage too slow
BS: sample run

Medical image processing

- \( \text{pipe}(\text{seq}(\text{getImage}), \text{farm}(\text{seq}(\text{renderImage})), \text{seq}(\text{displayImage})) \)
- contract \( \rightarrow \) 0.3 to 0.7 images per second
- initial condition
  - enough processing resources
  - image provider stage too slow

Application programmer *only provides*: 

- code for the seq nodes, properly wrapped in classes
- the parallel program structure (symbolic tree)
- the performance contract (e.g. throughput larger than \( k \))
BS: sample run

[Diagram showing various events and timelines, including:
- Top Manager Logic
- Farm Manager Logic
- Global Behaviour
- Resources]
Independent manager hierarchies

- take care of independent concerns (e.g. Performance and Security)
- must coordinate independently taken decisions
  - to avoid instability
BS: Coordination protocol

Fireable rule (Manager X) \( R_z :: Trig_i, Cond_j, Prio_k \rightarrow a_1, \ldots, a_n \)

**Step 0** broadcast changes in the computation graph eventually induced by \( R_z \)

**Step 1** gather answers from all other manager (hierarchies)

**Step 2** analyze answers:
- all OK \( \rightarrow \) perform \( a_1, \ldots, a_n \)
- at least one NOK \( \rightarrow \) ABORT \( R_z \) & lower \( Prio_k \)
- \( \text{require(Property}_m) \rightarrow \) perform \( a'_1, \ldots, a'_n \)
  such that \( \text{Property}_m \) is ensured
program → farm(seq)

contract :: parDegree=8
BS pros and cons

Pros:
- full decoupling of system and application programmer concerns
- reconfigurable autonomic management through JBoss rule engine
- two prototypes available: GCM BS (ProActive/GCM based) and LIBERO (pure Java/RMI, configurable action/sensor interface)

Cons:
- further investigation needed on multiple concern management
- compiler contract $\rightarrow$ JBoss rules needed
Conclusions

Skeletons from grids to multicores

Grids
- BS suitable to manage typical features of grids: heterogeneity, distribution, security, ...

Multicores
- efficient mechanisms supporting typical grains

Next step
- adopt BS like management to improve (at run time) the performance of FastFlow
- e.g. depending on current system behaviour
  - experiment, evaluate, adopt new skeleton composition
    - e.g. farm(pipe(seq,seq)) → pipe(farm(seq), farm(seq))
References

- Slides at http://www.di.unipi.it/~marcod “Papers” → “Talks”

  http://sourceforge.net/projects/mc-fastflow/

- ProActive/GCM BS home page at
  http://gridcomp.ercim.eu/content/view/26/34/

Acknowledgements

- Main designers & developers of FastFlow →
  M. Torquati (UNIPI) M. Aldinucci (UNITO)

- Main contributors to Behavioural Skeletons →
  M. Aldinucci (UNITO), S. Campa, G. Zoppi (UNIPI), P. Dazzi,
  N. Tonellotto (ISTI/CNR), P. Kilpatrick (QUB)
Any questions?

marcod@di.unipi.it
marco.danelutto@danelutto.org