Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

Skeletons from grids to multicores

Marco Danelutto

Dept. of Computer Science, University of Pisa, Italy

October 2010, Amsterdam

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

Contents

1 Introduction
- Scenario
- Urgencies

2 Multicores
- Scenario
- FastFlow
- Experimental results
- Offloading

3 Non functional concerns
- Scenario
- Behavioural skeletons
- Hierarchical composition
- Multiple concern management

4 Conclusions

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

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

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 applicaion programmes

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 applicaion programmes

Parallel design patterns

- Massingill, Mattson, Sanders 2000 → "Patterns for parallel programming" book (2006) (software engineering community)
- design patterns à la Gamma book
  - name, problem, solution, use cases, etc.
- concurrency, algorithms, implementation, mechanisms

Concept evolution

Parallelism

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

Concept evolution

Parallelism

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

New architectures

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

Concept evolution

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 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

Urgencies

$\rightarrow$ Targeting multi/many cores

- different implementation issued and solutions
- completely different computational grains to be addressed

$\rightarrow$ 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

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

Targeting multicores

Targeting multi/many cores

Structured parallel programming on COW/NOW

Implementation template based

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

Macro Data Flow based

- Lithium/Muskel, Skipper, Calcium (Skandium) $\rightarrow$ compile skeletons to MacroDataFlow graphs + Dstributed MDF interpreter

Targeting multi/many cores

Structured parallel programming on COW/NOW

Implementation template based

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

Macro Data Flow based

- Lithium/Muskel, Skipper, Calcium (Skandium) $\rightarrow$ compile skeletons to MacroDataFlow graphs + Dstributed MDF interpreter

Emphasis

- communication latency hiding
- avoid unnecessary data copies

Multi/many core features

Shared memory hierarchy

- NUMA (C vs. M accesses, non uniform intercore interconnection strctures)
- chache 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 intercore interconnection strctures)
- chache coherence (snoopy vs. directory based)

Control abstractions

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

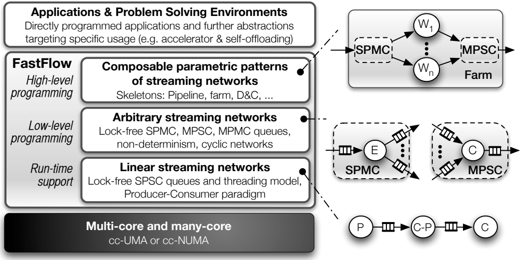
Focus

- synchronization overheads
- data access patterns
- thread-core binding, affinity scheduling

FastFlow

Advanced programming framework

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



FastFlow: simple streaming networks

Single Producer Single Consumer (SPSC) queue

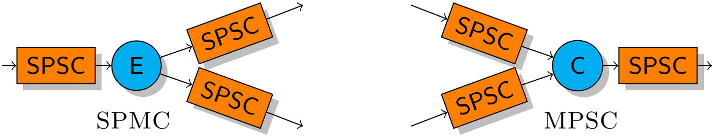
- uses results from the '80
- lock-free, wait-free
- no memory barriers for Total Store Order processor (e.g. Intel, AMD)
- single memory barrier for weaker memory consistency models (e.g. PowerPC)

→ very low latency in communications

FastFlow: simple streaming networks

Other queues: SPMC MPSC MPCP

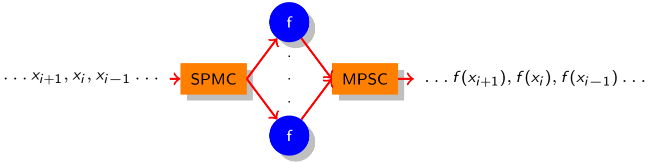
- one-to-many, many-to-one and many-to-many sychronization and data flow
- use a explicit arbiter thread
- providing lock-free and wait-free arbitrary data-flow graphs
- cyclic graphs (provably deadlock-free)



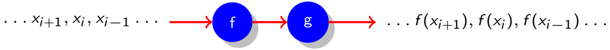
FastFlow: high level programming abstractions

Several "streaming" skeleton provided

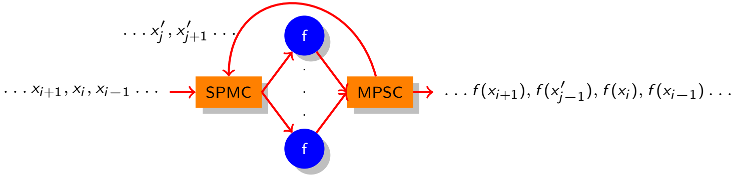
- farm



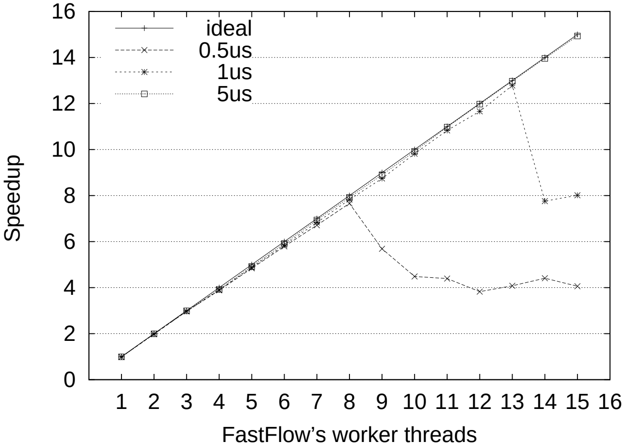
- pipeline



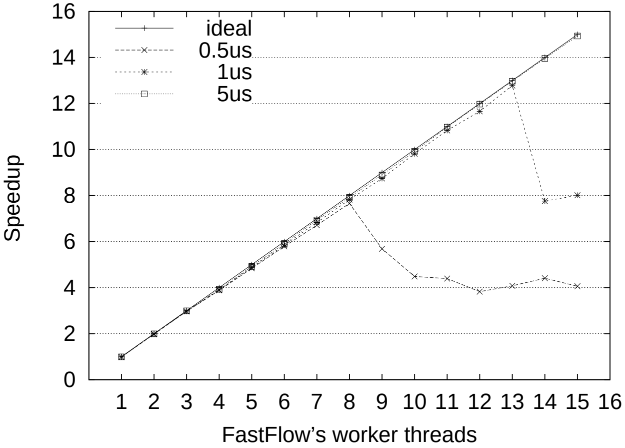
- farm with feedback



FastFlow: results



FastFlow: results



Speedup

Shows the advantage of lock free comms

FastFlow: results

Matrix multiplication

- parallel `for i` only

n. workers	matmul_ff_v1 Time	matmul_ff_v1 Speedup	matmul_ff_v2 Time	matmul_ff_v2 Speedup	matmul_OpenMP Time	matmul_OpenMP Speedup
4	2982.24	3.6	2691.74	4.0	2767.48	3.9
8	1495.87	7.2	1435.41	7.5	1455.73	7.4
12	1496.01	7.2	1431.34	7.5	1492.87	7.2
16	1495.74	7.2	1426.42	7.6	1433.31	7.5

FastFlow: results

Matrix multiplication

- parallel `for i` only

n. workers	matmul_ff_v1 Time	matmul_ff_v1 Speedup	matmul_ff_v2 Time	matmul_ff_v2 Speedup	matmul_OpenMP Time	matmul_OpenMP Speedup
4	2982.24	3.6	2691.74	4.0	2767.48	3.9
8	1495.87	7.2	1435.41	7.5	1455.73	7.4
12	1496.01	7.2	1431.34	7.5	1492.87	7.2
16	1495.74	7.2	1426.42	7.6	1433.31	7.5

FastFlow vs. OpenMP

Comparable at quite coarse grain

FastFlow: different appls

Microbenchmarks

Benchmark	Parameters	Skeleton used	Speedup / #cores
Matrix Mult.	1024x1024	farm no collector	7.6 / 8
Quicksort	50M integers	D&C	6.8 / 8
Fibonacci	Fib(50)	D&C	9.21 / 8

FastFlow: different appls

Microbenchmarks

Benchmark	Parameters	Skeleton used	Speedup / #cores
Matrix Mult.	1024x1024	farm no collector	7.6 / 8
Quicksort	50M integers	D&C	6.8 / 8
Fibonacci	Fib(50)	D&C	9.21 / 8

Real use cases

Application	Skeleton used	Measure	Value
YaDT-FF (C4.5 data mining)	D&C	Speedup	4.5-7.5
StochKit-FF (Gillespie)	farm	Scalabitily	10-11
SWPS3-FF (Gene matching)	farm no collector	GCUPS	12.5-34.5

FastFlow: offloading

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

General purpose methodology

- use FastFlow as an efficient, fine grain *sw accelerator*

```
//<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;

    }
  }

}
```

a) Original code (sequential)

```
//<includes; ff_includes>
#define N 1024
int A[N][N],B[N][N],C[N][N];

int main() {

  // < init A,B,C>

  ff :: ff_farm<> farm(true /* accel */);
  std :: vector<ff :: ff_node *> w;
  for(int i=0;i<NWORKERS;++i)
    w.push_back(new Worker);
  farm.add_workers(w);
  farm.run_then_freeze();

  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

}
```

```
struct task_t {
  task_t(int i,int j):i(i),j(j) {}
  int i;
  int j;
};

class Worker: public ff :: ff_node {
public:
  void * svc(void * task) {
    task_t * t = (task_t *)task;

    int _C=0;
    for(int k=0;k<N;++k)
      _C += A[t->i][k]*B[k][t->j];
    C[t->i][t->j] = _C;

    delete t;
    return GO_ON;
  }
};
```

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)

Behavioural skeletons

Def: Behavioural skeleton

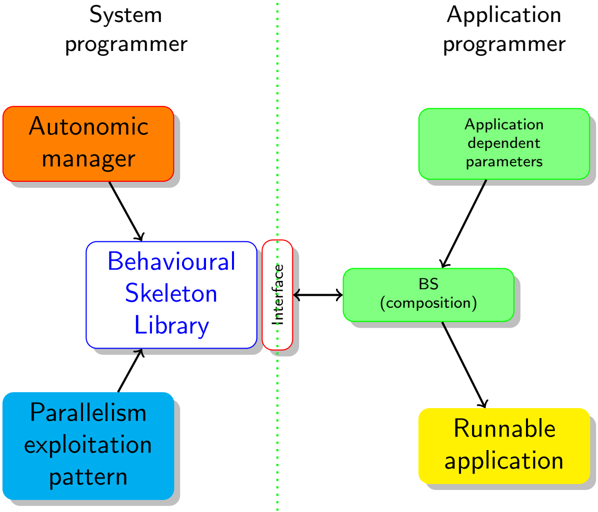
Co-design of a component including:

- 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
Application programmer
Autonomic manager
Application dependent parameters
Behavioural Skeleton Library
Interface
BS (composition)
Parallelism exploitation pattern
Runnable application

Sample Behavioural skeleton

Functional replication BS

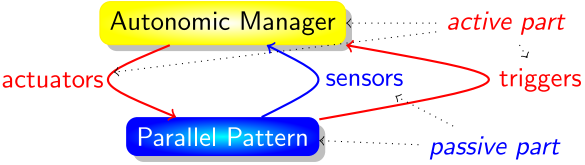
Parallel pattern

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

Performance manager

- Interarrival time faster than service time → increase parallelism degree, unless communication bandwidth is saturated.
- Interarrival time slower that service time → decrease the parallelism degree.
- Recent change → do not apply any action for a while.

Implementation: behavioural skeleton



Behavioural skeleton

- Triggers start manager activities
- Sensors determine pattern perception from the manager
- Actuators determine manager intervention possibilities

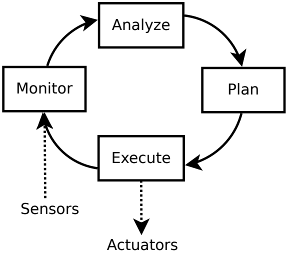
Implementation: manager

Monitor
- perceive pattern status

Analyse
- figure out policies

Plan
- devise strategy

Execute
- implement decisions on pattern



Implementation: manager

Monitor

- perceive pattern status

Analyse

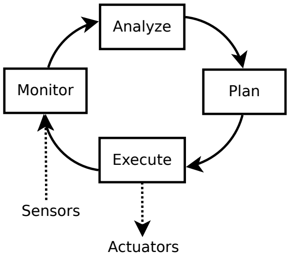
- figure out policies

Plan

- devise strategy

Execute

- implement decisions on pattern



MAPE loop implementation

Cyclic execution of a JBoss business rule engine

- RULE :: Priority, Trigger, Condition $\rightarrow$ 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

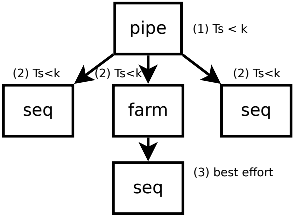
$\rightarrow$ raises violation to upper manager in hierarchy

$\rightarrow$ 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

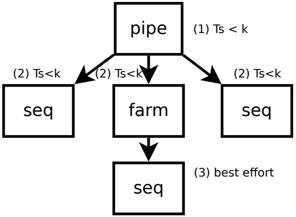
→ enters *passive* mode waiting for a new contract

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

Hierarchical composition

Program → Composition of BS

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



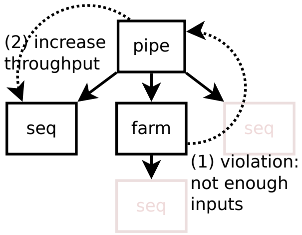
Hierarchy of managers

- 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



BS: sample run

Skeletons from grids to multicores

M. Danelutto

Introduction
Scenario
Urgencies

Multicores
Scenario
FastFlow
Experimental results
Offloading

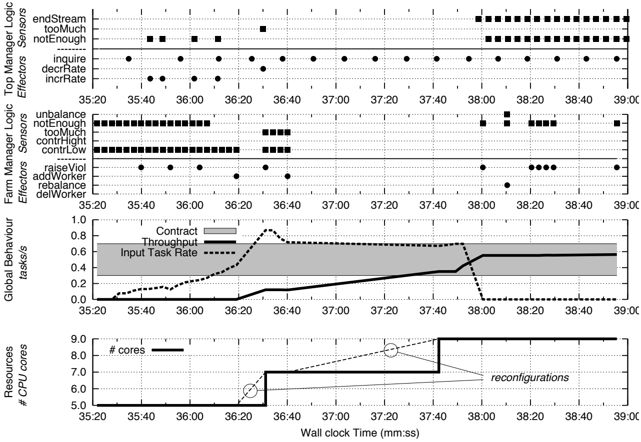
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management

Conclusions

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

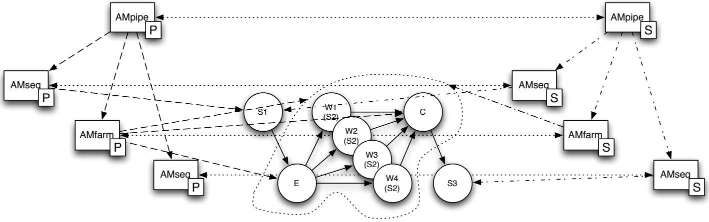
BS: sample run



Multiple concern management

Indipendent manager hierarchies

- take care of indipendent 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

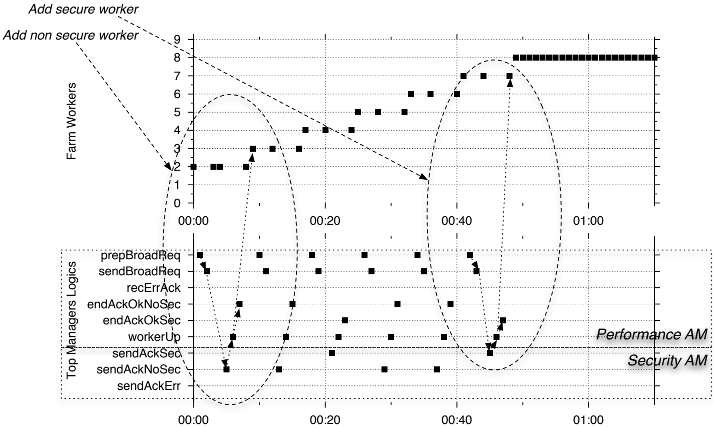
Step 1 gather answers from all other manager (hierarchies)

Step 2 analyze answers:

- all OK $\rightarrow$ perform $a_1, \ldots, a_n$
- at least one NOK $\rightarrow$ ABORT R_z & lower $Prio_k$
- require(Property$_m$) $\rightarrow$ perform $a'_1, \ldots, a'_{n'}$ such that Property$_m$ is ensured

BS: coordination protocol

program $\rightarrow$ 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)) $\rightarrow$ pipe(farm(seq), farm(seq))

References

- These slides available at
  `http://www.di.unipi.it/~marcod`
  follow "Papers" then "Talks" tabs
- FastFlow home page at
  `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`
  source code (GPL) at
  `http://sourceforge.net/projects/mc-fastflow/`
- ProActive/GCM BS home page at
  `http://gridcomp.ercim.eu/content/view/26/34/`
- LIBERO prototype: ask author(s) at
  `marcod@di.unipi.it`

Introduction
Scenario
Urgencies
Multicores
Scenario
FastFlow
Experimental results
Offloading
Non functional concerns
Scenario
Behavioural skeletons
Hierarchical composition
Multiple concern management
Conclusions

Any questions?

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