The multi/many core challenge: a pattern based programming perspective

Marco Danelutto

Dept. Computer Science, Univ. of Pisa
CoreGRID Programming model Institute

XXII Jornadas de Paralelismo
Sept. 7-9, 2011, La Laguna, Tenerife
Contents

Introduction

Structured programming

Targeting HM2C

Managing vs. computing

Exascale & Conclusions

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Introduction

Structured programming

Targeting HM2C

Managing vs. computing

Exascale & Conclusions
Problems driving hw advances

Power wall

► clock frequency increase requires more and more power/heat dissipation

► power cost even bigger than hardware cost

Complexity

► % increase in die area → fraction of % increase in performance

Economy

► replication of “small” hw design easier than design of new, larger hw
User market pressures

Desktop
- user interface more and more demanding (graphics, gesture inputs, voice interaction)
- more and more complex applications

HPC
- more and more complex/large data sets
- finer grain algorithms
- more accurate simulations

Servers
- faster execution of “old” code
- management of more and more performant network interfaces
Multicores

Currently available

▶ simplified core design
▶ shared memory hierarchy
▶ inter core control lines supporting small clusters (4x)
▶ cache coherency protocols

Easy design for

▶ multitasking (desktop)
▶ small scale parallel programs (shared memory)
▶ large scale parallel programs
→ via advanced network interconnection
Multicores (2)

Intel i7
- full x86 microarchitecture (SSE up to 4.2)
- up to 6 cores per socket, 2 way hyperthreading, 4 socket interconnect → 48 thread per board

Oracle SPARC T3
- 16 SPARC cores (with FPU and crypto engine)
- 128 thread per CPU, 4 socket interconnect → 512 thread per board
Multicores (3)

Tilera

- 64 or 100 cores per chip
- 2D mesh interconnect (hw routing unit)
- network interfaces with direct cache injection
- 4 network interfaces to feed all cores

IBM PowerEN

- 16 cores (Power ISA, 4 way SMT, in order 2 way concurrent issue)
- 4 special purpose coprocessors (xml, regex, cypher, compress)
- high speed network interfaces
Hardware advances

GPUs

Started as graphic coprocessors → GP-GPU

- control unit with a number of attached execution units (ALU) *
- highly efficient memory design
  → striped concurrent access, high bandwidth
- only suitably to run data parallel code
  → possibly with no data dependencies
- coprocessors → explicit data management required
  slightly different code may significantly boost performance
- currently up to 960 cores, mem 406.5 Gb/Sec, 515 double precision GFlops
FPGAs

- Experimenting & low scale manufacturing
- Accelerators in mission critical software
- Considered for GP computing
  - on the fly compiling of critical software portions
- usually provided as PCIe cards OR socket mounted (processor replacement)
Software pressures

**Overall**

- available threads/processes
- accelerators
- slower single core

- urgencies
- Parallel programming models
- Parallel programmers

Danelutto

The multi/many core challenge: a pattern based programming perspective

http://www.di.unipi.it

---

**Introduction**

Structured programming

Targeting HM2C

Managing vs. computing

Exascale & Conclusions

---
Assembler vs. HL programming models

**Assembler languages**
- instructions: close to metal
- programmer responsibility:
  - qualitative parallelism exploitation
  - memory allocation
  - communications
  - synchronization
  - scheduling
  - mapping

**High level languages**
- instructions: close to programmer abstractions
- programmer responsibility:
  - qualitative parallelism exploitation
Software pressures

Separation of concerns

**Functional concerns**
- all what’s needed to compute the application result value
- **what** is computed
- algorithm, data types, ...

**Non functional concerns**
- all what’s needed to determine the way the application result is computed
- **how** is computed
- performance, security, power management, fault tolerance, ...
Separation of concerns

**Functional concerns**
- all what’s needed to compute the application result value
- **what** is computed
- algorithm, data types, ...

**Non functional concerns**
- all what’s needed to determine the way the application result is computed
- **how** is computed
- performance, security, power management, fault tolerance, ...

**Application programmer vs. System programmer concerns**
Introduction

Structured programming

Targeting $HM2C$

Managing vs. computing

Exascale & Conclusions
Overview

HPC community

early '90

Algorithmic skeletons

pre-defined parallel patterns, exposed to programmers, as programming constructs/library calls

SW engineering community

early '00

Design patterns

“recipes” to handle parallelism: name, problem, solution, use case, ...
Algorithmic skeletons

- Cole 1988
  algorithmic skeletons → common, parametric, reusable parallelism exploitation pattern
- directly exposed as constructs, library calls, objects, higher order functions, components, ...
- composable
  two tier model → stream parallel skeletons with inner data parallel skeletons
- high level parallel abstractions (HPC community)
  - hiding most of the technicalities related to parallelism exploitation
  - directly exposed to application programmers
Evolution of the concept

Cole PhD, 88

initial concept, no composition, targeting clusters
Evolution of the concept

Cole PhD, 88

initial concept, no composition, targeting clusters

P3L, early '90

first language, targeting COW, two tier composition

Algorithmic skeletons
Evolution of the concept

**Cole PhD, 88**
- Initial concept, no composition, targeting clusters

**P3L, early '90**
- First language, targeting COW, two-tier composition

**ASSIST, early '00**
- Targeting grids, run-time restructuring

**Muskel, Lithium, OcamlP3L, Muesli, Mallba, SkeTo, early '00**
- Libraries, targeting COW, MPI or Java based

- Microsoft TPL, Intel TBB, late '00
- Commercial products with partial support for skeletons
- FastFlow, late '00 (targeting multicores, lock-free, fine-grain, plain C++)
- Muesli, SkePU, early '10 (targeting multi-cores+gpu, C++, MPI, OMP, OpenCL)
Evolution of the concept

**Cole PhD, 88**
- initial concept,
- no composition,
- targeting clusters

**P3L, early '90**
- first language,
- targeting COW, two tier composition

**ASSIST, early '00**
- targeting grids, run time restructuring

**Muskel, Lithium, Ocaml, P3L, Muesli, Mallba, SkeTo, early '00**
- libraries, targeting COW, MPI or Java based

**Microsoft**
- TPL, Intel TBB, late '00
- Commercial products with partial support for skeletons
Evolution of the concept

**Cole PhD, 88**
- initial concept, no composition, targeting clusters

**P3L, early '90**
- first language, targeting COW, two tier composition

**ASSIST, early '00**
- targeting grids, run time restructuring

**Microsoft TPL, Intel TBB, late '00**
- Commercial products with partial support for skeletons

**Muskel, Lithium, OcamP3L, Muesli, Mallba, SkeTo, early '00**
- libraries, targeting COW, MPI or Java based

**FastFlow, late '00**
- target multicores, lock free, fine grain, plain C++

**Muesli, SkePU, early '10**
- targeting multicores+gpu, C++ MPI, OMP, OpenCL
Evolution of the concept

Cole PhD, 88

P3L, early '90

ASSIS†, early '00

Microsoft
TPL, Intel
TBB, late '00

Languages

first language, targeting COW, two composition

targeting grids, run time restructuring

Muskel, Lithium,
OcamP3L, Muesli,
Mallba, SkeTo, early '00

FastFlow, late '00

Muesli, SkePU, early '10

Libraries

libraries, targeting COW, MPI or Java based

Commercial products

get multicores, lock free, fine grain, plain C++

targeting multicores+gpu, C++ MPI, OMP, OpenCL

initial concept, no composition, targeting clusters
Typical algorithmic skeletons

Stream parallel

- pipeline (computation in stages)
- farm (embarrassingly parallel)

Data parallel

- map (embarrassingly parallel)
- stencil (with dependencies)
- reduce (binary, associative and commutative operators)
- scan (parallel prefix)

Control parallel

- loops (determinate, indeterminate)
- if-then-else (speculative parallelism)
- sequential (wrapping of existing code)
- seqcomposition (in place pipelines)
Skeleton applications

- sequential “function” code
  s1, s2, s3 provided by
  application programmer
- along with proper syntax to
  express the tree
- mapping, scheduling,
  communication, synchronization, all in
  charge of the skeleton
  framework
Implementing algorithmic skeletons

Initially

- skeleton tree (nesting) compiled to process network
- one-to-one correspondence in between skeletons and process networks template

P3L, Meusli, ASSIST

Then

- skeleton tree (nesting) compiled to macro data flow graphs
- optimizations of the skeleton tree re rewritings: semantically proven correct transformations, increasing performance

Muskel, Skipper, SkeTo¹

¹Bird Meertens theory, fusion transformations, template based
Key strengths

**Full parallel structure of the application exposed to the skeleton framework**
- optimizations exploit structure knowledge
- support for automatic\(^2\) non functional concern management

**Framework responsibility for architecture targeting**
- write once, executed everywhere code
- with architecture specific compiler back end tools

**Functional debugging (only) in charge to the application programmer**
- possibility to run skeleton programs through sequential back end

\(^2\)autonomic
Algorithmic skeleton (assessments)

**Separation of concerns**
- application programmers $\rightarrow$ **what** has to be computed (algorithm)
- system (skeleton) programmers $\rightarrow$ **how** things are efficiently computed

**Inversion of control**
- programmers suggest a possible implementation
- skeleton framework applies known optimizations

**Performance**
- same as hand written parallel code
- at a fraction of the development time
Parallel design patterns

Software engineering community

- introduce concept in early ’00
  Massingill, Mattson, Sanders *Patterns for parallel programming* 2006
- parallel “branch” of traditional (seq) design patterns
- as defined in the “Gamma book”

Separate communities

- algorithmic skeleton results ignored
- despite
  - skeletons ≡ pre-programmed *incarnations*
    of a parallel design patterns
**Meta-structuring**

Parallel design pattern split in 4 spaces

1. **Finding concurrency space** → modelling concurrent (i.e. potentially parallel) activities
2. **Algorithm space** → modelling implementation of parallel algorithms
3. **Supporting structure space** → modelling suitable ways to implement different parallel algorithms
4. **Implementation mechanism space** → *de facto* targeting different architectures
Design pattern space structure

- Finding concurrency design space
  - Decomposition (task, data), Dependency analysis (group tasks, order tasks, data sharing), Design evaluation
  - Organize by task (task parallelism, divide & conquer), Organize by data decomposition (geometric decomp, recursive data), Organize by flow of data (pipeline, event based coordination)
  - Program structure (SPMD, Master/Worker, Loop parallelism, Fork/Join), Data structures (shared data, shared queue, distrib. array)
  - UE management, Synchronization, Communication

- Algorithm design space

- Supporting structure design space

- Impl. mechanisms design space

Collapsed in the implementation of algorithmic skeletons

- application programmer → concurrency and algorithm spaces
- skeleton implementation (system programmer) → support structures and implementation mechanisms

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Structured parallel programmer: design patterns

Parallel design patterns

Problem

Progr. lang. & libraries

follow, learn, use

Design patterns

Parallel programmer

low level source code

Tools (Standard)

Application code

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Structured parallel programmer: skeletons

Skeleton library

Problem

Parallel programmer

high level source code

Tools (advanced)

Application code

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Structured parallel programmer

Parallel programmer

Design patterns

Skeleton library

use knowledge → instantiate

Problem

Source code

Introduction

Structured programming

Targeting HM2C

Managing vs. computing

Exascale & Conclusions

Parallel design patterns

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Progress ...

Introduction

Structured programming

Targeting $HM2C$

Managing vs. computing

Exascale & Conclusions
Concurrent activity set

- Fine grain, high parallelism
- Coarse grain, low parallelism
- threads, processes, GPU threads, ...

Grain size

Synchronization

Memory access

architecture dependent decisions

Problems

Introduction
Structured programming
Targeting HM2C
Managing vs. computing
Exascale Conclusions

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Memory

- Cache friendliness
- Memory wall
- Cache coherency
- Data alignment
Synchronization

Shared mem

Communications

Collectives

Data dep/data flow

The multi/many core challenge: a pattern based programming perspective
Targeting $HM^2C$

With structured approaches:

→ design patterns
→ algorithmic skeletons
Targeting $HM^2C$

With structured approaches:

→ design patterns
→ algorithmic skeletons

(Quasi) concrete example: embarrassingly parallel pattern

1. design pattern approach
   ▶ with sample concern targeting

2. skeleton approach
   ▶ with more concern targeting

3. usage sample
   ▶ different contexts
Design pattern: embarrassingly parallel
Design pattern: embarrassingly parallel

- In/Out data types, Worker code, ...
- Embarrassingly parallel pattern
- Specialization
Design pattern: embarrassingly parallel

In/Out data types, Worker code, ...

Embarrassingly parallel pattern

Grain, Parallelism degree, ...

Performance model (ideal)

Specialization

http://www.di.unipi.it
**Design pattern: embarrassingly parallel**

- In/Out data types, Worker code, ...
- Grain, Parallelism degree, ...
- Target architecture
- Performance model (ideal)
- Performance model (concrete)
- Specialization
Design pattern: embarrassingly parallel

- In/Out data types, Worker code, ...
- Grain, Parallelism degree, ...
- Target architecture
- Embarrassingly parallel pattern
- Performance model (ideal)
- Performance model (concrete)
- Specialization
- Sample code
- Implementation

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Devising parallelism degree

Ideally → as much parallel/concurrent activities as needed to sustain input task pressure

Need to know:

▶ estimated input pressure, estimated task processing time, communication overhead (network, in memory)

Compiler vs. run time choices:

▶ compile time: devise parallelism degree based on performance model & static estimates

▶ run time: adjust parallelism degree automatically (autonomically) based on performance model & monitored behaviour

3talk parco 2011
NUMA memory exploitation

Auto scheduling:
- workers require tasks from “global” task queue
  → “far” memory → slow execution → less tasks scheduled
- *tolerating* latencies/overheads

Affinity scheduling:
- tasks on cores that produced them
- *reduces* latencies/overheads

Round robin allocation of dynamically allocated memory chunks:
- better support of random / round robinding scheduling of tasks
- *reduces* latencies/overheads
Algorithmic skeleton: overall view

Skeleton library
Algorithmic skeleton: overall view
Algorithmic skeleton: overall view

- Target architecture
- Skeleton library
- Skeleton instance (with perf models)
- Implementation
Algorithmic skeleton: overall view
Algorithmic skeleton: overall view

Target architecture → Skeleton library

I/O types, worker code → Skeleton instance (with perf models) → Implementation

Application programmer

domain specific knowledge
Algorithmic skeleton: overall view

Target architecture \(\rightarrow\) Skeleton library

\[\text{Skeleton instance (with perf models)}\]

\[\text{Implementation}\]

\[\text{I/O types, worker code}\]

optimize

System programmer

Application programmer

domain specific knowledge
Sample usage: FastFlow farm

```cpp
ff_farm<> farm; // create farm

std::vector<ff_node *> w; // create workers
for(int i=0;i<nworkers;++i)
    w.push_back(new Worker);
farm.add_workers(w); // add workers

farm.run_and_wait_end(); // run farm
```
Sample usage: FastFlow farm

ff_farm<> farm; // create farm

std::vector<ff_node *> w; // create workers
for(int i=0;i<nworkers;++i)
    w.push_back(new Worker);
farm.add_workers(w); // add workers

Emitter em; // create a splitting emitter
Collector co; // create a gathering collector
farm.add_emitter(&em); // add them to farm
farm.add_collector(&co);

farm.run_and_wait_end(); // run farm

transforms embarrassingly parallel stream $\rightarrow$ data parallel
Sample usage: FastFlow farm (0.5 $\mu$secs grain)
Sample usage: FastFlow farm (5 μsecs grain)
Sample usage: FastFlow farm (50 μsecs grain)
Domain specific usage

Image stream from camera + denoiser worker(s)
  ▶ image filtering (real time)
Domain specific usage

Image stream from camera + denoiser worker(s)
  ▶ image filtering (real time)

Packets from network device + netprobe analyser worker(s)
  ▶ network monitoring (11 G packets per second on a dual Nehalem (8 core))
Domain specific usage

Image stream from camera + denoiser worker(s)
  ▶ image filtering (real time)

Packets from network device + netprobe analyser worker(s)
  ▶ network monitoring (11 G packets per second on a dual Nehalem (8 core))

Sequences from data base + Smith Waterman worker(s)
  ▶ genome matching (34.5 GCUPS on an 8 core Intel)
Domain specific usage

Image stream from camera + denoiser worker(s)
  ▶ image filtering (real time)

Packets from network device + netprobe analyser worker(s)
  ▶ network monitoring (11 G packets per second on a dual Nehalem (8 core))

Sequences from data base + Smith Waterman worker(s)
  ▶ genome matching (34.5 GCUPS on an 8 core Intel)

Matrices from radar + Cholesky worker(s)
  ▶ factorization as fast as ultra hand optimized code
Heterogeneous architectures (SkePU)

BINARY_FUNC(plus_f, double, a, b,
    return a+b;
)
BINARY_FUNC(mult_f, double, a, b,
    return a*b;
)
int main()
{
    skepu::MapReduce <mult_f, plus_f>
        dotProduct(new mult_f,new plus_f);

double r = dotProduct(v1,v2);
...
Heterogeneous architectures (SkePU)

![Graph showing performance comparison between different methods](http://www.di.unipi.it)
Data flow based implementation

Pattern/skeletons compiled to macro data flow graphs

- macro: instructions $\equiv$ full functions
- graphs: instantiated with input tokens for each input data set
- incremental compilation: instructions $\rightarrow$ subgraphs

Multi threaded macro data flow interpreter:

- fetches fireable instructions from logically centralized task queue
- executes instructions
- stores results in proper (shared mem) locations
- synchronization of the accesses guaranteed by fireability

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Advantages

Data parallelism and stream parallelism handled with the same mechanisms

▸ different macro data flow instruction vs. different macro data flow graph instances

▸ varying grain

Static and dynamic “general purpose” optimizations

▸ graph collapsing for reduced synchronizations

▸ affinity scheduling for improved NUMA memory/cache exploitation

▸ instruction local optimizations still possible (GPU offloading, vectorization, pre-existing libs, etc.)
Expandability

More concrete problem when using patterns/skeletons

- what if provided skeletons do not match your needs?
- answer Cole’s accommodate diversity principle

Application programmers provided with the possibility to name new MDF graphs as skeletons

- provided they satisfy interface constrains
- suitable to model patterns non provided in the system
- rely on MDF run time efficiency
- incremental and domain specific framework extension
Results: data parallel

![Graph showing data parallel results]
Results: stream parallel
Progress ...

Introduction

Structured programming

Targeting $HM2C$

Managing vs. computing

Exascale & Conclusions
More separation of concerns

More and more programming a parallel application is made of

→ programming the algorithm computing the final results out of the input data

→ programming the code needed to make the application performant, secure, fault tolerant, power efficient, ...
More separation of concerns

More and more programming a parallel application is made of

→ programming the algorithm computing the final results out of the input data

→ programming the code needed to make the application performant, secure, fault tolerant, power efficient, ...

Ideally:

- Interoperable components rather than intermingled code
- Non functional code

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Even more: autonomic management of NFC

Structured algorithm code
Even more: autonomic management of NFC

Structured algorithm code exposes

Parallel structure
Even more: autonomic management of NFC

Structured algorithm code

Autonomic Controller

Sensors & actuators

Sensors: determine what can be perceived of the computation
Actuators: determine what can be affected/changed in the computation

Parallel structure
Even more: autonomic management of NFC

**Sensors**: determine what can be perceived of the computation

**Actuators**: determine what can be affected/changed in the computation
Even more: autonomic management of NFC

Autonomic manager: executes a MAPE loop. At each iteration, and ECA (Event Condition Action) rule system is executed using monitored values and possibly operating actions on the structured parallel pattern.
Co-design of parallel pattern and non functional autonomic manager

**a) parallel pattern**
- implements actuators and sensors
- determining manager policies

**b) autonomic management**
- policies coded as ECA rules:

$$\text{event}/\text{trigger}, \text{condition} \rightarrow \text{action}$$
BS user view

Autonomic manager

BS

Behavioural skeleton library

Parallel pattern

System programmer concerns

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
BS user view

Behavioural skeletons

Application programmer view

Problem

Application dependent params

Behavourial skeleton library

BS (composition)

APPL

The multi/many core challenge: a pattern based programming perspective
Sample BS: functional replication

Parallel pattern

- Master-worker with variable number of workers
- Auto or user-defined scheduling of tasks to workers
- Sensors: interarrival time, service time, ...
- Actuators: increase/decrease par degree, ...

Performance manager policies

- 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.
Behavioural skeletons

Functional replication BS (GCM)

P1 :: interarrival faster than service time → increase par degree
P2 :: interarrival slower than service time → decrease par degree
P3 :: recent change → nop
Functional replication BS (GCM)

enact P1

P1 :: interarrival faster than service time $\rightarrow$ increase par degree
P2 :: interarrival slower than service time $\rightarrow$ decrease par degree
P3 :: recent change $\rightarrow$ nop
Functional replication BS (GCM)

- Monitor
- Analyse
- Plan
- Execute

enact P1

P1 :: interarrival faster than service time → increase par degree
P2 :: interarrival slower than service time → decrease par degree
P3 :: recent change → nop
Functional replication BS (GCM)

- **P1**: interarrival faster than service time → increase par degree
- **P2**: interarrival slower than service time → decrease par degree
- **P3**: recent change → nop
Functional replication BS (GCM)

P1 :: interarrival faster than service time → increase par degree
P2 :: interarrival slower than service time → decrease par degree
P3 :: recent change → nop
Functional replication BS (GCM)

enact P2

P1 :: interarrival faster than service time → increase par degree
P2 :: interarrival slower than service time → decrease par degree
P3 :: recent change → nop
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts
**BS: advanced topics**

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts

User contract:

\[ T_S \leq k \]

\[ \text{pipe} \]

\[ \text{seq} \]

\[ \text{farm} \]

\[ \text{seq} \]

\[ \text{seq} \]

\[ \text{seq} \]

\[ \text{seq} \]

\[ \text{seq} \]

\[ \text{seq} \]
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts

\[
\begin{align*}
T_S & \leq k \\
\text{seq} & \text{pipe} \\
T_S & \leq k \\
\text{farm} & \\
T_S & \leq k \\
\text{seq} & \\
\end{align*}
\]
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts

\[
\begin{align*}
T_S & \leq k \\
\text{seq} & \\
T_S & \leq k \\
\text{farm} & \\
T_S & \leq k \\
\text{seq} & \\
\text{pipe} & \\
\text{seq} : & \#Nw
\end{align*}
\]
Hierarchical management of a NF concern

▶ user supplied “contract” propagated top down
▶ local managers ensure subcontracts
▶ in case of failure, report to upper manager
→ change (sub)contracts

fail: $T_S = h$
\[ T_S \leq k \]
\[ T_S \leq k \]
\[ T_S \leq k \]

pipe

seq : \#Nw

User contract:

\[ T_S \leq k \]
\[ T_S \leq k \]
\[ T_S \leq k \]
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts

\[
T_S \leq h
\]

\[
\text{pipe}
\]

\[
\begin{align*}
T_S & \leq k \\
\text{seq} & \\
T_S & \leq k \\
\text{farm} & \\
T_S & \leq k \\
\text{seq} & \\
\text{seq} : & \#Nw
\end{align*}
\]
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  \[ \rightarrow \] change (sub)contracts

\[
\begin{align*}
T_S & \leq h \\
\text{seq} & \text{farm} \\
& \text{seq} : \#Nw
\end{align*}
\]
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts

```
user contract:
T ≤ k

seq : #Nw

farm
↓

dismiss worker(s)

seq : #Nw
```

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
BS: advanced topics

Hierarchical management of a NF concern

- user supplied “contract” propagated top down
- local managers ensure subcontracts
- in case of failure, report to upper manager
  → change (sub)contracts
Danelutto

The multi/many core challenge: a pattern based programming perspective

http://www.di.unipi.it
Progress ...

Introduction

Structured programming

Targeting *HM2C*

Managing vs. computing

**Exascale & Conclusions**
Exascale perspective

Exascale roadmap

Large scale systems
  ▶ unprecedented scale

Heterogeneous systems
  ▶ multicore, accelerators (GPU, FPGA)

Need for new software stack (X-stack)
  ▶ programming models, compilers, run times

The International Exascale Software Project Roadmap

Exascale: programming model strategies

- Hybrid vs. uniform
  several different models hosted within components
  skeleton/patterns represent the coordination level/language (only)

- Evolutionary vs. revolutionary
  adding components is encouraged

- Domain specific vs. general programming models
  pattern/skeleton set expandability → domain specificness

- Widely embraced standards vs. single implementations
  standard frameworks can be used within a component

Danelutto
http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Exascale: programming model agenda

- Research is needed into a variety of promising programming models for exascale computing, including system-wide models that provide a uniform approach to application development across an entire platform, as well as hybrid programming models that combine two or more programming APIs. Such models will need to provide a range of means for the expression of high levels of concurrency and locality and may be capable of supporting application-specific fault tolerance. Enhancements to existing programming interfaces as well as new programming approaches should be explored. For new models, interoperability with existing HPC programming interfaces is highly desirable. Programming models that facilitate productive application development are to be encouraged. Other desirable characteristics are performance transparency and the ability to support incremental application migration.
Exascale: run time agenda

Run time system:

- The objective will be to optimize the application’s utilization of resources for best power/performance by helping the application *adapt to and exploit the level of granularity supported by the underlying hardware*.

- Optimization of resources and infrastructure for implementing the runtime (e.g., memory used by message-passing libraries, overheads for process management and synchronization) and increased usage of prediction techniques to accelerate the runtime, or at least introduction of high levels of asynchrony and communication/computation overlap (i.e., asynchronous MPI collectives, APGAS approaches, *data-flow task based approaches*).
Conclusions

**Structured programming methodology is effective**

- while targeting multi/many core heterogeneous architectures
- reusing and exploiting a number of results from distributed architecture targeting
- exploiting separation of concerns to support more and more efficient implementation of parallel applications

Danelutto

http://www.di.unipi.it

The multi/many core challenge: a pattern based programming perspective
Conclusions

Structured programming methodology is effective

▶ while targeting multi/many core heterogeneous architectures
▶ reusing and exploiting a number of results from distributed architecture targeting
▶ exploiting separation of concerns to support more and more efficient implementation of parallel applications

Management of non functional concerns

▶ orthogonal programming duty
▶ supporting self-awareness, self-healing, ...
Conclusions

Structured programming methodology is effective

▷ while targeting multi/many core heterogeneous architectures
▷ reusing and exploiting a number of results from distributed architecture targeting
▷ exploiting separation of concerns to support more and more efficient implementation of parallel applications

Management of non functional concerns

▷ orthogonal programming duty
▷ supporting self-awareness, self-healing, ...

Alltogether

▷ dramatically decrease design to production time for parallel applications
▷ introduces (performance) portability of parallel applications
Thanks to:
M. Aldinucci, P. Kilpatrick, M. Torquati, P. Dazzi, G. Zoppi, S. Campa

Any questions?

marcod@di.unipi.it

http://www.di.unipi.it/~marcod