Structured parallel programming

Marco Danelutto

Dept. Computer Science, Univ. of Pisa

DSL 2013
July 2013, Cluj, Romania
Contents

Introduction

Structured programming

Targeting \textit{HM2C}

Managing vs. computing
Introduction

Structured programming

Targeting *HM2C*

Managing vs. computing
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 $\rightarrow$ 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

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

Xeon PHI
- 60 cores per chip (with vector unit (512 bit), 4 hw contexts)
- Bi-directional ring interconnection
- PCIe card + tool support (offload pragma + RTS)
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

Danelutto
http://www.di.unipi.it
Structured parallel programming
**FPGAs**

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

Danelutto http://www.di.unipi.it
Structured parallel programming
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
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, OcamP3L, Muesli, Mallba, SkeTo, early '00
- libraries, targeting COW, MPI or Java based
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
- Commercial products with partial support for skeletons

TPL, Intel TBB, late '00
- Targeting multicores, lock free, fine grain, plain C++
### Evolution of the concept

<table>
<thead>
<tr>
<th>Cole PhD, 88</th>
<th>initial concept, no composition, targeting clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3L, early '90</td>
<td>first language, targeting COW, two tier composition</td>
</tr>
<tr>
<td>ASSIST, early '00</td>
<td>targeting grids, run time restructuring</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Commercial products with partial support for skeletons</td>
</tr>
<tr>
<td>TPL, Intel TBB, late '00</td>
<td></td>
</tr>
<tr>
<td>Muskel, Lithium, OcamlP3L, Muesli, Mallba, SkeTo, early '00</td>
<td>libraries, targeting COW, MPI or Java based</td>
</tr>
<tr>
<td>FastFlow, late '00</td>
<td>target multicores, lock free, fine grain, plain C++</td>
</tr>
<tr>
<td>Muesli, SkePU, early '10</td>
<td>targeting multicores+gpu, C++, MPI, OMP, OpenCL</td>
</tr>
</tbody>
</table>
Evolution of the concept

- **Cole PhD, 88**
  - Initial concept, no composition, targeting clusters
  - First language, targeting COW, two composition

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

- **P3L, early '90**
  - Languages
  - Targeting grids, run time restructuring

- **ASSIS, early '00**
  - Languages
  - Targeting grids, run time restructuring

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

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

- **Muesli, SkePU, early '10**
  - Libraries
  - Targeting multicores+gpu, C++, MPI, OMP, OpenCL

---

Danelutto

Structured parallel programming

http://www.di.unipi.it
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
  rewritings: semantically proven correct transformations, increasing performance
  Muskel, Skipper, SkeTo\(^1\)

\(^1\)Bird Meerteens theory, fusion transformations, template based
Template based implementation

Template

- Concurrent activity graph:
  - parametric
    - functional params: function code
    - non functional params: parallelism degree, scheduling details, etc.
  - one input channel, one output channel (composition)

Template library

- entry:
  - skeleton implemented
  - target architecture
  - template
  - performance model
Sample templates
Compilation/execution

Compilation
- visit skeleton tree
- assign templates
  - use models (dimensioning, pick up alternatives)

Execution
- feed data into input channel
- fetch results from output channel
Macro Data Flow

**Compile**
- compile each skeleton composition to a MDF graph
- new instance of the MDF graph for each input task

**Execute**
- fetch fireable instructions
- execute using a set of parallel interpreters
Key strengths

**Full parallel structure of the application exposed to the skeleton framework**

- optimizations exploit structure knowledge
- support for automatic 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

Danelutto

http://www.di.unipi.it
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** $\rightarrow$ modelling concurrent (i.e. potentially parallel) activities
2. **Algorithm space** $\rightarrow$ modelling implementation of parallel algorithms
3. **Supporting structure space** $\rightarrow$ modelling suitable ways to implement different parallel algorithms
4. **Implementation mechanism space** $\rightarrow$ *de facto* targeting different architectures
Design pattern space structure

Finding concurrency design space

Algorithm design space

Supporting structure design space

Impl. mechanisms 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

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

Structured parallel programming
**Structured** parallel programmer: design patterns

- **Progr. lang. & libraries**
- **Problem**
- **Parallel programmer**
- **low level source code**
- **Tools (Standard)**
- **Application code**

Follow, learn, use
**Structured parallel programmer: skeletons**

- **Problem**
- **Skeleton library**
- **Parallel programmer**
  - Instantiate, use
  - **high level source code**
- **Tools (advanced)**
- **Application code**
**Structured parallel programmer**

- **Design patterns**
- **Skeleton library**

Use knowledge → instantiate

- **Parallel programmer**
- **Problem**
- **Source code**

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

Introduction

Structured programming

Targeting *HM2C*

Managing vs. computing
Concurrency space

Grain size
- Fine grain, high parallelism
- Coarse grain, low parallelism

Memory access

Synchronization

Threads, processes, GPU threads, ...

Concurrent activity set

architecture dependent decisions
Memory

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

Memory hierarchy

Problems
Synchronization

Shared mem ─── Communciations ─── Collectives

Data dep/data flow
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

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

Embarrassingly parallel pattern

Danelutto
http://www.di.unipi.it
Devising parallelism degree

Ideally $\rightarrow$ 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

\(^3\)talk 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 robin scheduling of tasks
- *reduces* latencies/overheads
Algorithmic skeleton: overall view

Target architecture → Skeleton library

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

System programmer
Application programmer

optimizations
domain specific knowledge

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

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 → data parallel
Sample usage: FastFlow farm (0.5 µsecs 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)

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)

```c
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)
Macro data flow

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
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 completion time vs. parallelism degree for different methods: mdf^3, PLASMA static, and PLASMA dynamic. The graph illustrates the trends in performance across varying degrees of parallelism.](http://www.di.unipi.it)
Results: stream parallel

![Graph showing speedup vs. parallelism degree for different stream lengths and target environments.]

- **Ideal**
- **farm stream length=512**
- **mdf\(^3\) ++ stream length=512**
- **mdf\(^3\) stream length=512**
- **mdf\(^3\) stream length=64**
- **farm stream length=64**
- **mdf\(^3\) stream length=8**
- **farm stream length=8**

**Danelutto**
http://www.di.unipi.it
Skeleton rewriting rules

Sample rules

- \( \Delta \equiv \text{farm}(\Delta) \)
- \( \text{comp}(\Delta_1, \Delta_2) \equiv \text{pipe}(\Delta_1, \Delta_2) \)
- \( \text{map}(\text{comp}(\Delta_1, \Delta_2)) \equiv \text{comp}(\text{map}(\Delta_1), \text{map}(\Delta_2)) \)

Usage

- improve program expressions
- model driven rewriting
- e.g. map fusion (third rule above)
  - left to right: better load balancing
  - right to left: increase computation grain
Normal form

Definition
- visit seq leaves of the skeleton tree left to right
- \( \text{comp}(s_1, \ldots, s_k) \)
- \( \text{farm}(\text{comp}(s_1, \ldots, s_k)) \)

Statement
- normal form outperforms non normal form skeleton trees in stream parallel computations

Why
- improves granularity
- reduces communication overhead
- proven effective on coarse grain hw (COW/NOW)
- similar results hosts also including data parallel nodes (lower level)
Model driven refactoring

- user identifies possible rewritings
- system evaluates performance models (before and after)
- user commits or aborts rewriting
- possibly reiterating process/backtracking

ParaPhrase perspective

- FP7 STREP project http://paraphrase-ict.eu
  - user level: design patterns
  - system assisted rewriting
  - implementation level: skeletons
- refactoring for:
  - performance optimization
  - heterogeneous resources targeting (via mapping)
  - performance portability
Progress ...

Introduction

Structured programming

Targeting $HM2C$

Managing vs. computing
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:

- Algorithm code
- Interoperable components rather than intermingled code
- Non functional code
Even more: autonomic management of NFC

Structured algorithm code
Even more: autonomic management of NFC

Structured algorithm code

exposes

Parallel structure

Structured parallel programming

Targeting HM2C

Managing vs. computing

Principles

Introduction

Structured programming

Managing vs. computing
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

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

NFC manager

Danelutto
http://www.di.unipi.it
Structured parallel programming
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.

- Sensing: determines what can be perceived of the computation.
- Acting: determines what can be affected/changed in the computation.

Structured parallel programming

Danelutto

http://www.di.unipi.it
Behavioural skeletons

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:
     \[ \text{event/trigger, condition} \rightarrow \text{action} \]
Introduction

Structured programming

Targeting

HM

C

Managing vs. computing

Behavioural skeletons

BS user view

Parallel pattern

BS

Autonomic manager

Behavioural skeleton library

System programmer concerns

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

Structured parallel programming
BS user view

Problem

Application dependent params

Behavioural skeleton library

BS (composition)

APPL

Behavioural skeletons

Introduction
Structured programming
Targeting HM2C
Managing vs. computing

Danelutto
Structured parallel programming

http://www.di.unipi.it
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 → 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.
**Functional replication BS (GCM)**

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)

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)

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

enact P2
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:
\[ T_S \leq k \]

Diagram:
```
  seq
 /    \    /
pipe  farm  seq
   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
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

\[ T_s \leq k \]

\[ T_s \leq k \]

\[
\begin{align*}
\text{pipe} & \quad \text{seq} & T_s & \leq k \\
\text{farm} & \quad \text{seq} & T_s & \leq k \\
\text{seq} & \quad \text{seq} & #Nw & \\
\end{align*}
\]
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:
\[ T_S \leq k \]

\[ T_S \leq h \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]
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
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 \leq k
```

```
seq : #Nw
```

```
dismiss worker(s)
```

```
seq : #Nw
```
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

![Diagram]

User contract:

\[
T_S \leq k
\]

\[
\text{seq} : \#(N_w-d)
\]
Results

<table>
<thead>
<tr>
<th>CoreGRID</th>
<th>GridCOMP</th>
</tr>
</thead>
</table>

Behavioural skeletons

Danelutto http://www.di.unipi.it
Structured parallel programming