Algorithmic Skeletons & Design Patterns: a short introduction

Marco Danelutto

Nov. 2011 - Pisa
Introduction
The concept
Main achievements
Research teams

User view

Implementation

Advanced features

Design patterns

Conclusions
Motivations:

- patterns of parallel computation appear (identical or slightly changed) in different applications
- effort needed to implement patterns should be reused after first implementation
- multiple reuse refines implementation with positive feedbacks on previous instances

Algorithmic skeleton

Common, parametric, reusable parallelism exploitation pattern directly exposed as constructs, library calls, objects, higher order functions, components, ...
Algorithmic skeletons

**Common** appearing in different applications, possibly from different application domains
Algorithmic skeletons

Common appearing in different applications, possibly from different application domains
Parametric parameters to modify the “slightly different” features of the pattern, not the fundamental properties
Algorithmic skeletons

Common appearing in different applications, possibly from different application domains

Parametric parameters to modify the “slightly different” features of the pattern, not the fundamental properties

Reusable provided in such a way it may be easily reused in different places/applications
Algorithmic skeletons

Common appearing in different applications, possibly from different application domains
Parametric parameters to modify the “slightly different” features of the pattern, not the fundamental properties
Reusable provided in such a way it may be easily reused in different places/applications
Parallelism exploitation pattern models a precise parallel pattern and hides/abstracts all the details related to pattern implementation
Algorithmic skeletons

**Common** appearing in different applications, possibly from different application domains

**Parametric** parameters to modify the “slightly different” features of the pattern, not the fundamental properties

**Reusable** provided in such a way it may be easily reused in different places/applications

**Parallelism exploitation pattern** models a precise parallel pattern and hides/abstracts all the details related to pattern implementation

**Directly exposed to programmers** in a way as much as possible similar to the mechanisms used in the “host” programming framework
What skeletons hide

Concurrent activities
decomposition, mapping, scheduling
What skeletons hide

Concurrent activities
decomposition, mapping, scheduling

Synchronization
communications, synchronization, collective operations
What skeletons hide

Concurrent activities
de decomposition, mapping, scheduling

Synchronization
communications, synchronization, collective operations

Hw targeting
management of peculiar features of target architecture
Sample use case

Image processing

A set of images coming from a satellite has to be “filtered” using 3 different filters \( \text{filter}_1 \), \( \text{filter}_2 \) and \( \text{filter}_3 \)

- Single image processing independent of the others
- Processing of \( \text{filter}_i \) on image \( \text{Img}_k \) may be done concurrently with \( \text{filter}_{i-1} \) on image \( \text{Img}_{k+1} \) and \( \text{filter}_{i+1} \) on \( \text{Img}_{k-1} \).
Image processing

A set of images coming from a satellite has to be “filtered” using 3 different filters $\text{filter}_1$, $\text{filter}_2$ and $\text{filter}_3$

- Single image processing independent of the others
- Processing of $\text{filter}_i$ on image $\text{Img}_k$ may be done concurrently with $\text{filter}_{i-1}$ on image $\text{Img}_{k+1}$ and $\text{filter}_{i+1}$ on $\text{Img}_{k-1}$
Sample use case: POSIX

- run process/thread 1
  - connect to process/thread 2
  - loop: read image, compute \( \text{filter}_1 \), deliver result to process/thread 2
- run process/thread 2
  - connect to process/thread 1 and 3
  - loop: read image, compute \( \text{filter}_2 \), deliver result to process/thread 3
- run process/thread 3
  - connect to process/thread 2
  - loop: read image, compute \( \text{filter}_3 \), deliver result
Sample use case: POSIX

Up to application programmer

- Thread or process?
- Mapping?
- Communication mechanisms?
- Synchronization?
- Scheduling?
- Stream (end of) management?

Some $O(100)$ lines of code for processes using TCP/IP communications ... (plus $filter_i$ code and $img_k$ data type)
Sample use case: skeletons

```c
int main(int argc, char * argv[]) {
    ff_pipeline pipe; // declare the pipe
    // add stages
    pipe.add_stage(new Filter1()); // reads from in socket
    pipe.add_stage(new Filter2());
    pipe.add_stage(new Filter3()); // write to file system

    if (pipe.run_and_wait_end()<0) {// run it
        error("running pipeline\n");
        return -1;
    }
    pipe.ffStats(std::cerr);
    return 0;
}
```
Classification of skeletons

Depending on the patterns modelled

Stream parallel skeletons parallelism in the computation of different “tasks” available (on input stream) at different times
Classification of skeletons

Depending on the patterns modelled

Stream parallel skeletons parallelism in the computation of different “tasks” available (on input stream) at different times

Data parallel skeletons parallelism in the computation of a single task: decomposition into parallel sub tasks, computation, recomposition of the results from partial results
Classification of skeletons

Depending on the patterns modelled

Stream parallel skeletons  parallelism in the computation of different “tasks” available (on input stream) at different times

Data parallel skeletons  parallelism in the computation of a single task: decomposition into parallel sub tasks, computation, recomposition of the results from partial results

Control parallel skeletons  parallelism from “control flow” like constructs (loops, conditional, fork/join): usually with limited parallelism degree
Classification of skeletons

Depending on the applications targeted

General purpose skeletons modelling parallel patterns appearing in a wide range of applications, from different application domains
Classification of skeletons

Depending on the applications targeted

**General purpose skeletons** modelling parallel patterns appearing in a wide range of applications, from different application domains

**Domain specific skeletons** modelling parallel patterns appearing in a wide range of applications, from a specific application domain
Sample stream skeletons

Pipeline

- computations in stages
- result of stage $S_i$ is the input of stage $S_{i+1}$
- pure pipeline $\rightarrow$ no feedback nor jumps ahead
Sample stream skeletons

Pipeline

- computations in stages
- result of stage $S_i$ is the input of stage $S_{i+1}$
- pure pipeline → no feedback nor jumps ahead

Farm

- embarrassingly parallel computation (map-like) on stream of tasks
- pure farm → completely independent tasks
Sample data parallel skeletons

Map

- collection of data \((x_1, \ldots, x_n)\)
- each item of the collection independently computed \(y_i = f(x_i)\)
- result from the collection of the independently computed tasks \((y_1, \ldots, y_n)\)
Sample data parallel skeletons

Map
- collection of data \((x_1, \ldots, x_n)\)
- each item of the collection independently computed \(y_i = f(x_i)\)
- result from the collection of the independently computed tasks \((y_1, \ldots, y_n)\)

Reduce
- collection of data \((x_1, \ldots, x_n)\)
- result computed “summing up” all items by means of an associative and commutative operator \((x_1 \oplus x_2 \oplus \ldots \oplus x_n)\)
More sample skeletons

Divide & Conquer (data parallel)

- divide phase: split task to sub tasks recursively until sub tasks may be directly solved
- conquer phase: recursively rebuild result from partial results

Stencil (data parallel)

- map with function computing each as $y_i = f(x_i, N(x_i))$ for some “neighbourhood” function $N$

Comp

- computations in stages, computed “in place”
More sample skeletons

Branch&Bound (domain specific)
- completely encapsulated B&B algorithm

MapReduce (Google) (domain specific)
- map function on \( \langle key, value \rangle \) pairs
- then “reduce” each set of pairs with the same key
Skeleton nesting

Originally
non nestable skeletons: each computation modelled after a single skeleton

Nestability

- parameters of a skeleton may be other skeletons
- arbitrary nesting vs. two tier model → stream parallel skeleton with data parallel skeletons with sequential wrappings
- concept evolution: a number of small, highly reusable skeletons vs. more complex skeletons (RISC vs. CISC approach)
Skeleton nesting

pipe
  seq
  farm
  seq
  seq
Skeleton nesting
Orthogonal aspects

Independent vs. dependent parallel (sub)task

★ theoretic parallelism → actual parallelism
Orthogonal aspects

Independent vs. dependent parallel (sub)task

- theoretic parallelism $\rightarrow$ actual parallelism

State-full vs. stateless

- state introduce synchronization
- different synchronization depending on state usage (read only, accumulator, single owner writes, resource)

1. Propagate the concept with minimal conceptual disruption
2. Integrate ad-hoc parallelism
3. Accommodate diversity
4. Show the pay-back
Feasibility

Demonstrated since ’90

- P3L (1991–1996) stream+data parallel skeletons, targeting COW+MPI scalability, portability, performance, support for different “host languages” (C, C++, Fortran, Java)
- Muesli (early 2000–) C++ based, targeting heterogeneous COW (MPI, OpenMP, Cuda), stream and data parallel patterns, scalability, portability, performance
- Mallba (early 2000–) C based, targeting COW, operational research patterns (branch&bound)
Performance

Skeleton framework vs. OpenMP

![Graph showing performance comparison between m2df-pipe and OpenMP across varying parallelism degrees. The graph plots completion time in seconds on the y-axis against parallelism degree on the x-axis. The m2df-pipe line is solid and the OpenMP line is dashed. At lower parallelism degrees (e.g., 4, 8, 16), m2df-pipe outperforms OpenMP. As parallelism increases, both methods converge, with m2df-pipe consistently taking less time than OpenMP.]
Performance

FastFlow scalability (fine grain)

![Graph showing FastFlow scalability with speedup and ideal times for different worker thread times (0.5us, 1us, 5us).]
Portability

Portability through VM

- OcamlP3L → Ocaml “host language”, uses TCP/IP sockets, targeting any POSIX multiprocessor (with Ocaml)
- Muesli → MPI + OpenMP, targeting multi-core COW/NOW

Portability through “adaptation”

- same API → intermediate high level code → different back ends (implementation templates or distributed macro data flow interpreters (fwd))
Heterogeneity (GPU)

Recent advances

- SkePU: codelet based, map & reduce on GPU, farm implemented through StarPU methodology
- Muesli: added GPU targeting, no modification in the source skeleton code, only for parallel arrays
- Pisa MDF: dynamic scheduling of data parallel skeletons to CPUs and GPUs

<table>
<thead>
<tr>
<th>#cores</th>
<th>Avg task/core</th>
<th>GPU tasks</th>
<th>GPU %</th>
<th>Avg task/core</th>
<th>GPU tasks</th>
<th>GPU %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>459</td>
<td>89%</td>
<td>43</td>
<td>469</td>
<td>91%</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>406</td>
<td>79%</td>
<td>37</td>
<td>438</td>
<td>85%</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>317</td>
<td>61%</td>
<td>34</td>
<td>377</td>
<td>73%</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>193</td>
<td>37%</td>
<td>30</td>
<td>272</td>
<td>53%</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>42</td>
<td>8%</td>
<td>23</td>
<td>141</td>
<td>27%</td>
</tr>
</tbody>
</table>
Adaptivity

Performance

▶ parallelism degree adapted to varying conditions of
  ▶ target architecture (e.g. load)
  ▶ application (e.g. computation burst)

Security

▶ secure communications inserted only when needed
  ▶ e.g. parts of the computation on remote machines, reached by public network

→ without programmer intervention
→ dealt with by the (behavioural) skeleton implementation
Results moved to commercial/general purpose libs

OpenMP
parallel for ≡ map

Intel TBB
partial implementation of pipeline/farm skeletons

Microsoft TPL
stream and data parallel primitives

Google MapReduce
inherit and builds on results from skeleton community dating back to last century
Research results from

Edinburgh Cole → concept introduced (1988)
Research results from

Edimbourgh Cole → concept introduced (1988)

Muenster (early ’00–) C++ embedding, multi-core COW targeting
Research results from

Edimbourgh Cole → concept introduced (1988)

Muenster (early ’00–) C++ embedding, multi-core COW targeting

Tokio (early ’00–) skeletons as pure “sequential” libraries, data parallel optimizations (homomorphisms)
Research results from

**Edimbourgh** Cole → concept introduced (1988)

**Muenster (early '00–)** C++ embedding, multi-core COW targeting

**Tokio (early '00–)** skeletons as pure “sequential” libraries, data parallel optimizations (homomorphisms)

**Pisa (early '90–)** nesting, macro data flow implementation, autonomic managers, languages, fine grain multicore targeting
Research results from

Edimbourgh Cole → concept introduced (1988)

Muenster (early '00–) C++ embedding, multi-core COW targeting

Tokio (early '00–) skeletons as pure “sequential” libraries, data parallel optimizations (homomorphisms)

Pisa (early '90–) nesting, macro data flow implementation, autonomic managers, languages, fine grain multicore targeting

France (mid '90–) Ocaml embedding (Paris, with Pisa), BSP style skeletons (Orleans)
Research results from

Edinburgh Cole $\rightarrow$ concept introduced (1988)

Muenster (early '00–) C++ embedding, multi-core COW targeting

Tokio (early '00–) skeletons as pure “sequential” libraries, data parallel optimizations (homomorphisms)

Pisa (early '90–) nesting, macro data flow implementation, autonomic managers, languages, fine grain multicore targeting

France (mid '90–) Ocaml embedding (Paris, with Pisa), BSP style skeletons (Orleans)

Spain (early '00) domain specific skeletons
Currently “active” frameworks

- Muesli → http://www.wi.uni-muenster.de/PI/forschung/Skeletons/
- SkeTo → http://www.ipl.t.u-tokyo.ac.jp/sketo/
- Osl → http://traclifo.univ-orleans.fr/OSL/
- OcamlP3l → Camlp3L→ http://camlp3l.inria.fr/eng.htm
- SkePu →
  http://www.ida.liu.se/~chrke/skepu/index.html
Introduction

User view
   Language or libraries
   Host language issues

Implementation

Advanced features

Design patterns

Conclusions
Language vs. library based skeleton frameworks

Language approach

New language hosting algorithmic skeleton “constructs”

▶ possibly extension of an existing one
▶ implemented via (possibly complex) compilers
Language vs. library based skeleton frameworks

Language approach
New language hosting algorithmic skeleton “constructs”
- possibly extension of an existing one
- implemented via (possibly complex) compilers

Library approach
Existing language with library calls to:
- declare skeletons and require execution (meta approach)
- or call skeletons (seq library approach)
Languages: P3L

seq filter1 in(img a) out(img b)
$c\{ \ldots \text{C code here}\ldots \}c$
end seq

seq filter2 in(img a) out(img b)
$c\{ \ldots \text{C code here}\ldots \}c$
end seq

pipe main in(img a) out(img b)
  filter1 in(a) out(img c)
  filter2 in(c) out(b)
end pipe
//1. Define the skeleton program structure
Skeleton<Board, Count> subskel =
    new DaC<Board, Count>(
        new ShouldDivide(DEPTH), // Dive until depth is "N-3"
        new DivideBoard(),
        new Solve(),
        new ConquerCount());

Skeleton<Board, Count> nqueens = // Always subdivide the first row.
    new Map<Board, Count>(new DivideBoard(),
        subskel, new ConquerCount());

//2. Create a new Skandium instance with 2 execution threads
Skandium skandium = new Skandium(THREADS);

//3. Open a Stream to input parameters
Stream<Board, Count> stream = skandium.newStream(nqueens);

//4. Input parameters
Future<Count> future = stream.input(new Board(BOARD));

//5. Do something else here.
// ...

//6. Block for the results
Count result = future.get();
int arrayA[] = {10, 11, 12, 13, 14, 15, 16, 17};

// ... more C code here filling up "arrayA" here ...

// convert to SkeTo list
sketo::dist_list<int> as(8, array);

// apply to functions on each element of the array
bs = sketo::list_skeletons::map(f, as);
cs = sketo::list_skeletons::map(g, bs);
Seq code wrapping

Code reuse

- sequential portions of code (functions! no side effects)
- wrapped into “sequential skeleton”
- to support homogeneous nesting parameters

```java
public interface Execute<P, R> extends Muscle<P, R> {
    public R execute(P param)
        throws Exception;
}

public interface Split<P, R> extends Muscle<P, R> {
    public R[] split(P param)
        throws Exception;
}
```

```java
class Worker: public ff_node {
    public:
        // actual processing
        void * svc(void * task) { ... } // init
        int svc_init() { ... } // end up
        void svc_end() { ... }
};
```
Libraries
Usually supported if
▶ available for target architecture nodes (processor, OS)
▶ license available or open source

Howto
▶ special link options, or
▶ simply include libraries in compile/run commands:
  ▶ mpicc meusliSample.cpp ... -lmylib
  ▶ java mySkandiumSample ... -jar mylib.jar
Introduction

User view

Implementation
  Template based implementation
  Macro data flow based implementation
  Optimizations

Advanced features

Design patterns
Implementation: history

Initially

▶ skeletons implemented through “process templates”
  ▶ parametric process network
  ▶ implementing the parallel pattern modelled by the skeleton
  ▶ for a specific target architecture
  ▶ possibly with analytical performance models

More recently

▶ Alternative implementation models
  ▶ skeletons compiled to macro data flow graphs, MDF graphs executed on a parallel MDF interpreter (Muskel, Skipper)
  ▶ skeletons compiled to stack instructions, instruction stacks evaluated in parallel (Calcium, Skandium)
Template concept

Farm template

- emitter schedules tasks from input stream to workers
- collector merges results from workers to output stream
- complete interconnection template (ShMem multicore, TCP/IP COW)
- performance model: \[ \max n_w = \frac{T_f}{T_e}, T_s = \max\{T_e, T_c, \frac{T_f}{n_w}\} \]
Pipeline template

- one concurrent activity per stage
- performance model: $T_s = \max\{T_{S_i}\}$

\[ \cdots x_{i+1}, x_i, x_{i-1} \cdots \xrightarrow{f} \xrightarrow{g} \cdots f(x_{i+1}), f(x_i), f(x_{i-1}) \cdots \]
Template based implementation

Compiler

1. Parses skeleton source
2. Assigns “best” (in terms of perf model) template among those targeting the current architecture to each skeleton
3. Possibly performs optimizations on templates (e.g. pipe(farm,farm) → merge first collector with the second emitter)
4. Produces code as set of concurrent activities for the “processes” in the overall graph of the templates
Template based implementation
P3L resource assignment algorithm

On the skeleton tree:

1. top down: assign “best” template with optimal parallelism degree
   (may exceed currently available resources)

2. loop:
   ▶ take away 1 resource in one skeleton (bottom up chosen)
   ▶ re-balance skeleton tree
   ▶ until assigned resources = available resources
Macro data flow concept

Compiler

- one macro data flow graph defined per skeleton with one input arc and one output arc
- skeleton nesting $\rightarrow$ subgraph composition
- stream parallelism $\rightarrow$ multiple instances of graph rather than special subgraphs
Macro data flow concept

Interpreter

- One MDF interpreter per processing element (Core, Processor)
- MDF interpreter loop:
  1. receive (blocking) a fireable MDF instruction
  2. execute the MDF instruction
  3. dispatch (store) results to proper “next” MDF instructions
  4. notify the scheduling thread (to update fireable instruction list)
  5. loop
Sample MDF subgraphs

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Map</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \mid x \mid y$</td>
<td>$decomp \mid x \mid y_1 \ldots y_n$</td>
<td>$partition \mid x \mid p_1 \ldots p_n$</td>
</tr>
<tr>
<td>$f_2 \mid x \mid y$</td>
<td>$f \mid x \mid y$</td>
<td>$\oplus^* \mid p_1 \mid y$</td>
</tr>
<tr>
<td>$f_3 \mid x \mid y$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td></td>
<td>$recomp \mid x_1 \ldots x_n \mid y$</td>
<td>$\oplus \mid x_1 \ldots x_n \mid y$</td>
</tr>
</tbody>
</table>
Macro data flow implementation
Parallel structure exposed

- Well known optimizations may be automatically applied
  1. at skeleton level (rewriting)
     rewriting equivalent forms with better performance
  2. at implementation level
     refactoring implementations for better performance

- Actions may be taken upon monitored “bad” behavioural
  1. known performance model
  2. known strategies to bring back behaviour to expected one
Rewriting skeletons

Data parallelism → stream parallelism

\[ \text{map}(f) \equiv \text{pipe} \left( \text{decompose}, \text{farm}(f), \text{recompose} \right) \]
- decompose collections to create a stream of sub components
- use stream parallel skeletons to process the stream
- recompose partial results to collection

Pipeline re-structuring

\[ \text{pipe}(f,g,h) \equiv \text{pipe}(f, \text{comp}(g,h)) \text{ iff } T_f \geq T_g + T_h \]
- reduce impact of communications when bottlenecks exist
Rewriting skeletons

Normal form

- $\text{Sk ::= Farm}(\text{Sk}) \mid \text{Pipe}(\text{Sk}, \text{Sk}) \mid \text{seq}(\ldots)$

- $\text{fringe}(\text{seq}(\ldots)) = \text{seq}(\ldots)$
  
  $\text{fringe}(\text{Farm}(\text{Sk})) = \text{fringe}(\text{Sk})$

  $\text{fringe}(\text{Pipe}(\text{Sk}_1, \text{Sk}_2)) = \text{comp}(\text{fringe}(\text{Sk}_1), \text{fringe}(\text{Sk}_2))$
Rewriting skeletons

Normal form

- $\text{Sk ::= Farm(\text{Sk}) | Pipe(\text{Sk,Sk}) | seq(...)}$
- $\text{fringe(seq(...)) = seq(...)}$
- $\text{fringe(Farm(\text{Sk})) = fringe(\text{Sk})}$
- $\text{fringe(Pipe(\text{Sk1,Sk2})) = comp(fringe(\text{Sk1}), fringe(\text{Sk2}))}$
- $\text{NormalForm(\text{Sk}) = farm(fringe(\text{Sk}))}$

computes the same results
with service time $\leq$ service time of original program
Rewriting skeletons

Grain adaptation

\[
\text{pipe(map}(f), \text{map}(g)) \equiv \text{map}(\text{pipe}(f,g)) \equiv \text{map}(\text{comp}(f,g))
\]

▶ may be used to increase computational grain (better efficiency)

Map + Reduce

\[
\text{pipe(map}(f), \text{reduce}(\oplus)) \equiv \\
\text{pipe(comp(map}(f), \text{reduce}(\oplus)), \text{reduce}(\oplus))
\]

▶ compute partial reduce locally → \\
increase parallelism + reduce communications
Progress ...

Introduction

User view

Implementation

Advanced features
   Behavioural skeletons

Design patterns

Conclusions
Behavioural skeletons

Co-design of parallel pattern and non-functional autonomic manager

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

**autonomic management**
- policies coded as ECA rules: \(\text{event/trigger, condition} \rightarrow \text{action}\)
Behavioural skeletons

Co-design of parallel pattern and non functional autonomic manager

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

**autonomic management**
- policies coded as ECA rules: *event/trigger, condition*→*action*

Interoperable components rather than intermingled code
BS: operational view

Structured algorithm code

Autonomic Controller
- sensors & actuators
- implements
- Sensors: determine what can be perceived of the computation
- Actuators: determine what can be affected/changed in the computation

NFC manager reads
- monitors
- operates

ECA rule based program

Autonomic manager: executes MAPE loop, policies described as ECA rules.
Structured algorithm code exposes Parallel structure

Autonomic Controller
sensors & actuators implements

Sensors: determine what can be perceived
Actuators: determine what can be affected/changed

NFC manager reads monitors operates

ECA rule based program

Autonomic manager: executes MAPE loop, policies described as ECA rules.
BS: operational view

Structured algorithm code

implements

Autonomic Controller
sensors & actuators

exposes

Parallel structure

Sensors: determine what can be perceived of the computation

Actuators: determine what can be affected/changed in the computation
BS: operational view

Structured algorithm code

implies

Autonomic Controller
sensors & actuators

exposes

Parallel structure

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

NFC manager

Paraphrase
**BS: operational view**

- **Structured algorithm code** implements **Autonomic Controller sensors & actuators**
- **Parallel structure** exposes sensors & actuators
- **NFC manager** reads sensors
- **Autonomic manager**: executes MAPE loop, policies described as ECA rules.
- **ECA rule based program** operates on the Autonomic Controller.
BS: user view

System programmer concerns

Autonomic manager

BS

Behavioural skeleton library

Parallel pattern

Application programmer view
Autonomic manager

Problem

Application dependent params

Behavioural skeleton library

BS (composition)

APPL

BS: user view

Application programmer view
BS: sample policies

**Performance manager:**
- low throughput in a farm $\rightarrow$ increase parallelism degree
- low throughput in a pipeline $\rightarrow$ “farm out” sequential stages

**Fault manager:**
- fault node $\rightarrow$ recruit new resource, restart appl from checkpoint

**Power manager:**
- overconsuming $\rightarrow$ move computation to “cheaper” nodes
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, ...

![Diagram showing a master-worker parallel pattern with sensors and actuators connected to workers and the master]
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, ...

\[ E \rightarrow w_1 \rightarrow \ldots \rightarrow w_n \rightarrow C \]

\[ \text{sensors} \quad \text{actuators} \]
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 → increase par degree
P2 :: interarrival slower than service time → decrease par degree
P3 :: recent change → nop
Functional replication BS (GCM)

Monitor → Analyse → Plan → Execute

enact P1

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

- Analyse
- Monitor
- Plan
- Execute

- 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

- 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)

- Monitor
- Analyse
- Plan
- Execute

enact P2

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
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
Hierarchical management of a NF concern

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

User contract:
\[ T_S \leq k \]
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
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

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

![Diagram]

User contract:

\[ T_s \leq k \]

fail: \[ T_s = h \]

seq: \[ \#Nw \]
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

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

\[ T_s \leq h \]

\[ \text{seq} \]

\[ \text{farm} \]

\[ \text{seq} : \#Nw \]
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

User contract:

\[ T_S \leq k \]

Pipe:

\[ \text{seq} : \#Nw \]

Dismiss worker(s):

\[ \text{seq} : \#Nw \]
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

```
pipe
  seq
  farm
  seq

seq : #(Nw-d)
```
Pattern rewriting rules

Introduction/Elimination

- $farm(\Delta) \equiv \Delta$
- $pipe(\Delta) \equiv \Delta$

Propagation

- $pipe(map(\Delta_1), map(\Delta_2)) \equiv map(pipe(\Delta_1, \Delta_2))$

Sequenzialization

- $pipe(\Delta_1, \Delta_2) \equiv seqcomp(\Delta_1, \Delta_2)$
Policies may use rewriting rules to transform the global parallel pattern of the application
Policies may use rewriting rules to transform the global parallel pattern of the application

User contract:

\[ T_S \leq k \]
Policies may use rewriting rules to transform the global parallel pattern of the application.
Policies may use rewriting rules to transform the global parallel pattern of the application.

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ T_S \leq k \]

\[ \text{seq} \]

\[ \text{farm} \]

\[ \text{seq} : \#N_1 \]
Policies may use rewriting rules to transform the global parallel pattern of the application.
BS: transformation rules

Policies may use rewriting rules to transform the global parallel pattern of the application
BS: transformation rules

Policies may use rewriting rules to transform the global parallel pattern of the application:

\[
\text{User contract: } T_S \leq k \quad T_S \leq k \quad T_S \leq k
\]

\[
\text{pipe} \quad \text{farm} \quad \text{farm} \quad \text{seq}
\]

\[
\text{seq: } \#N_2 \quad \text{seq: } \#N_1
\]
Progress ...

Introduction

User view

Implementation

Advanced features

Design patterns

Conclusions
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
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
- 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
Structured parallel programmer: design patterns

- Design patterns
  - Problem
    - Progr. lang. & libraries
  - Parallel programmer
    - low level source code
  - Tools (Standard)
    - Application code
  - follow, learn, use

Paraphrase
Structured parallel programmer: skeletons

Skeleton library

- Problem
- Parallel programmer
- Tools (advanced)
  - high level source code
- Application code

instantiate, use
Structured parallel programmer

- Design patterns
- Skeleton library

use knowledge → instantiate

Parallel programmer

Problem → Source code
Introduction

User view

Implementation

Advanced features

Design patterns

Conclusions
ParaPhrase perspectives

- different skeletons implementations to target CPU/GPU
- refactoring and autonomic managers choosing “best” implementation
- rewriting used to improve efficiency
- user interface as friendly as possible
  - default values for “normal” parameters
  - pre-defined classes/objects covering more common parameter cases
Initial/final pattern sets

- includes stream parallel and data parallel skeletons
- classical skeletons only (farm, pipe, map, reduce, stencil, ...)
- more general, domain specific skeletons in second phase
- initial implementation based on FastFlow technology
  - items added to support MDF interpreter skeleton/template
    - alternative implementations possible
  - support for inter processing element communication needed
    - to be discussed