Algorithmic Skeletons & Design Patterns: a short introduction

Marco Danelutto, University of Pisa

Paraphrase International Summer School in Parallel Patterns
Dublin, June 2014
Outline

1. Parallel programming
2. Structured vs. unstructured parallel programming
3. Algorithmic skeletons
4. Design patterns
5. Advanced usage of patterns/skeletons
6. Available Tools
7. Perspectives
Parallel programming

Market pressures

- Low level programming models
- HW advances CPU, GPU, FPGA

Need for:
- Parallel programming models
- Parallel programmers
Parallel programming

Problem

Parallel activities + Coordination

Synch, Sched, Map, Comm, Share
Assembler vs. HL programming models

**High level languages**
- Instructions
- Close to programmer abstractions
- Programmer responsibility
- Qualitative parallelism exploitation

**Assembler languages**
- Instructions
- Close to metal
- Programmer responsibility
- Qualitative parallelism exploitation, Memory allocation, Communications, Synchronization, Scheduling, Mapping
Separation of concerns

**Functional concerns**
- All what’s needed to compute the application result value
- What is computed
- Algorithm(s), data type(s), ...

**Non functional concerns**
- All what’s needed to determine the way the application result is computed
- How it is computed
- Performance, security, power management, fault tolerance, ...

*Application programmers*

*System programmers*
Structured vs unstructured

**Structured**
- Only known, efficient parallel patterns allowed
- Nesting supported
- Arbitrary (possible inefficient patterns) forbidden
- High abstraction level

**Unstructured**
- Mechanisms exposed
- Arbitrary concurrent activities graphs
- Arbitrary interactions
- Low abstraction level
Structured parallel programming

HPC community

early ’90

Algorithmic skeletons

pre-defined parallel patterns, exposed to programmeres, 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 $\rightarrow$ common, parametric, reusable parallelism exploitation pattern
  - Directly exposed as constructs, library calls, objects, higher order functions, components, ...

- Composable
  - Two tier model $\rightarrow$ stream parallel skeletons with nested data parallel skeletons

- High level parallel abstractions
  - Hiding most of the technicalities related to parallelism exploitation
  - Directly exposed to the application programmers
Typical algorithmic skeletons

Stream parallel
- Task farm (embarrassingly parallel computations)
- Pipeline (computations in stages)

Data parallel
- Map (embarrassingly parallel computations)
- Stencil (... with dependencies)
- Reduce (associative and commutative “sum up”)
- Scan (parallel prefix)
Typical algorithmic skeletons (2)

Control parallel
- Loops (determinate, indeterminate)
- If-then-else (speculative parallelism)
- Sequential (wrapping of existing code)
- Seq-composition (*in place* pipelines)
Sample skeleton: task farm

Parameters
- Parallelism degree
- Scheduling policy
- Gathering policy
- Parallel (tree style) implementation of scheduling and gathering
- ...

Input management, scheduling
Computing
Output management, gathering
Implementation is not pattern

Task farm

Input management, scheduling

Unicast

Computing

Gather

Output management, gathering
Implementation is not pattern
Implementation is not pattern

Fault tolerant (diff $W = \text{diff impl}$)
Sample skeleton usage (Meusli)

InitSkeletons();
Initial<float> streamGenerator(...);
Atomic<float,float> worker(...);
Final<float> streamConsumer(...);

Pipe myPipeline(streamGenerator,
                  worker,
                  streamConsumer);
myPipeline.start();
TerminateSkeletons();
Sample skeleton usage (FastFlow)

myTask* stage1(myTask* in,  
                  ff_node* const) {  
    ...  
  }

ff_pipe<myTask>
myPipeline(stage1, stage2, stage3);
myPipeline.run_and_wait_end();
Sample skeleton usage (Erlang skel)

```
myPipeline() ->
    skel:do(
        [{pipe,
          [{seq, fun mypackage:stage1/1},
            {seq, fun mypackage:stage2/1}]}
        ],
        InTasks).
```
Skeleton applications

- Sequential functions
  - s1, s2, s3
  - Provided by the application programmer (business logic code)
- Proper syntax to express the nesting (tree)
- Mapping, scheduling, synch, comms, etc.
  - All in charge of the skeleton framework
Algorithmic skeleton implementation

Initially

- Skeleton tree compiled to process network
- One to one correspondence in between skeletons and process network templates
- P3L, Meusli, ASSIST

Later on

- Skeleton tree compiled to macro data flow graphs
- Optimization of the skeleton tree
  - Rewriting: semantically proven correct transformation increasing performance
- Muskel, Skipper, SkeTo
Template based implementation

Template library
- Entry: <skel, process template, performance model>

Compilation
- Visit the skeleton tree (bottom up)
- Assign templates
- Solve/instantiate parameters (possibly with further, different order visits)
Sample template implementation

pipeline

seq(s1)  farm  seq(s3)

map

seq(s2)

Pipe ➔ Ts=f(…)

Farm ➔ Ts=f(…)
Sample MDF implementation
# Evolution of the concept

<table>
<thead>
<tr>
<th>P3L</th>
<th>BMF</th>
<th>Fortran M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIE</td>
<td>OcamlP3L</td>
<td>CO2P3S</td>
</tr>
<tr>
<td>ASSIST</td>
<td>Skipper</td>
<td>MALLBA</td>
</tr>
<tr>
<td>OSL</td>
<td>SkeTo</td>
<td>Muesli</td>
</tr>
<tr>
<td>MS TPL</td>
<td>INTEL TBB</td>
<td>Google MapReduce</td>
</tr>
<tr>
<td></td>
<td>SkeCL</td>
<td>FastFlow</td>
</tr>
<tr>
<td></td>
<td>skel</td>
<td>SKEPU</td>
</tr>
</tbody>
</table>

- Cole PhD th, 88
- ‘90
- ‘00
- ‘10

**Note:**
- INTEL TBB
- SKEPU
- FastFlow
- SkeCL
- skel
- SkeTo
- OSL
- Google MapReduce
- MS TPL
- INTEL TBB
- SkeCL
- skel
Evolution of the concept

Cole PhD th, 88

P3L BMF Fortran M

SKIE OcamlP3L

Skipper SkeTo MALLBA Muesli

ASSIST OSL

OSL Google MapReduce

OSL

MS TPL INTEL TBB FastFlow SKEPU

SkeCL skel

‘90 ‘00 ‘10
Evolution of the concept

Cole PhD th, 88

P3L BMF Fortran M

SKIE OcamlP3L

Skipper SkeTo MALLBA Muesli

OSL Google MapReduce

MS TPL INTEL TBB FastFlow SKEPU

SkeCL skel

Lithium ’00

CO2P3S Muskel

Calcium Skandium ’10

Skandalium ’00

‘90

INTEL TBB
Key strengths (algorithmic skeletons)

Full parallel structure exposed to the framework

- Optimizations exploit knowledge
- Support for automatic non functional concern management

Framework responsibility for hw targeting

- Write once execute everywhere
- Hw specific back end and optimization tools

Functional debugging (only) in charge to the application programmer

- Possibility to go through functional debugging with a sequential back end
Assessments (algorithmic skeletons)

Separation of concerns
- Application programmers $\rightarrow$ what is computed
- System programmers $\rightarrow$ how the result is computed

Inversion of control
- Programmers suggest possible implementation
- Framework implements optimizations

Performance
- Same as hand written code (or better)
- At a fraction of the development time
Parallel design patterns

Software engineering community

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

Separate communities

- Algorithmic skeleton results ignored
- Despite
  skeleton = pre-programmed incarnations of parallel patterns
Parallel design pattern (2)

Definition

- **Name**
- **Problem** (solved)
- **Context** (where the pattern may be used)
- **Forces** (driving the development)
- **Solution(s)** (outline of possible solution to be adopted to efficiently implement the pattern)

(in the Pattern book, a pattern description takes $O(10)$ full pages ..., no actual implementation code given)
Meta structuring

Finding concurrency space
- Modelling concurrent activities

Algorithm space
- Modelling implementation of parallel algorithms

Supporting structure space
- Modelling suitable, recurrent ways of implementing different parallel algorithms

Implementation mechanisms space
- Modelling mechanisms used to implement support structures (*De facto* targeting different parallel architectures)
Patterns and skeletons

Meta structure collapsed in the algorithmic skeleton approach

- Finding concurrency + Algorithm \(\rightarrow\) application programmer
- Supporting structure + mechanisms \(\rightarrow\) system programmer
Sample design pattern (Intel TBB)

Problem
• Parallelize a divide and conquer algorithm.

Context
• Divide and conquer is widely used in serial algorithms. Common examples are quicksort and mergesort.

Forces
• Problem can be transformed into subproblems that can be solved independently.
• Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

Solution
There are several ways to implement divide and conquer in Intel® Threading Building Blocks (Intel® TBB). The best choice depends upon circumstances.

• If division always yields the same number of subproblems, use recursion and tbb::parallel_invoke.
• If the number of subproblems varies, use recursion and tbb::task_group.
• If ultimate efficiency and scalability is important, use tbb::task and continuation passing style.
Sample usage:

implementation of quicksort

```c
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>=500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                             [=]{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```
Overall programming methodology

- **Design patterns**
  - **Parallel programmer**
    - **Problem**
    - **low level source code**
  - **Tools (Standard)**
    - **Application code**

- **Progr. lang. & libraries**
  - follow, learn, use
Overall methodology (skeletons)

Skeleton library

- Instantiate, use

- Parallel programmer
- Tools (advanced)

- Problem
- High level source code
- Application code
Overall methodology (combined)

- Design patterns
- Skeleton library

Use knowledge → instantiate

Parallel programmer

- Problem
- Source code
Similarities (skeletons <> patterns)

- HL abstractions
- Clear programming model
- Compilers+Run time systems
Advanced struct. progr. techniques (1)

Automatic restructuring of parallel applications

• Stream parallel computations
  (e.g. common graphic, video processing apps)
  • Expressed in terms of
    • Pipeline, farm, seq wrappers

• **Normal form concept**
  • Take sequential leaves in the skeleton tree in the order they are executed
  • Compose them in a seq-comp pattern
  • Use the seq-comp as a farm worker

=> *better performant program using less resources!*
Theoretic background

Rewriting rules

- Skel = farm(Skel)
- Pipe(Skel,Skel) = Comp(Skel, Skel)

therefore

1) Pipe(Seq(S1), Farm(Seq(S2)), Farm(Seq(S3)))
2) Comp(Seq(S1), Farm(Seq(S2)), Farm(Seq(S3)))
3) Comp(Seq(S1),Seq(S2),Seq(S3))
4) Farm(Comp(Seq(S1),Seq(S2),Seq(S3)))

Functional semantics respected, parallel semantics changes!
Performance background

Pipeline service time

\[ \text{Max} \{ \text{stage}_i \text{ service time} \} \]

Farm service time

\[ \text{Max} \{ T_{\text{emitter}}, T_w/N_w, T_{\text{collector}} \} \]

When moving to normal form

Increasing \( T_w \)

Reducing the number of support process/thread (Emitter and Collectors)
Advanced struct. progr. techniques (2)

Dealing with non functional concerns

- Application code:
  - Programming the algorithms computing the final results out of the input data
  - Programming the code needed to make the application performant, secure, fault tolerant, power efficient, ...

Ideally
Separating concerns (the structured way)

Exploit parallel pattern knowledge in *autonomic manager*

- Paired with the skeleton
- Implementing non functional concern management policies
- Clear interface between manager and skeleton
  - Sensors: report status of the computation
  - Actuators: implement manager decisions
- Separate concerns
  - Parallel pattern implementation: skeleton
  - Parallel patterns management: autonomic manager
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.
User view (1)

Autonomic manager

BS

Behavioural skeleton library

Parallel pattern

System programmer concerns
User view

Application programmer view

Problem

Application dependent params

Behavioural skeleton library

BS
(composition)

APPL
Sample behavioural skeleton

Functional replication (aka FARM) skeleton

• Manager rules
  • Service time > Inter arrival time $\rightarrow$ increase par degree
  • Service time $<<$ inter arrival time $\rightarrow$ decrease par degree

• Sensors
  • Service time
  • Inter arrival time

• Actuators
  • Worker ++
  • Worker --
BS at work
Advanced struct. progr. techniques (3)

The ParaPhrase programming methodology

a) Programmer recognizes patterns

- Possible as a collection of well known parallel patterns is available

b) Refactoring tool introduces skeletons

- Semi automatic, user (programmer) hints + abstract syntax tree directed operations

c) Run time support manages optimal skeleton execution on heterogeneous hardware

- Efficient multi back-end skeleton framework + autonomic policies
Heterogeneous map implementation

Provided as a skeleton framework component

- Runs initial fraction of map tasks on both CPU and GPU
  - CPU code and GPU kernel versions of the map $f$ needed
  - Estimates CPU and GPU task execution time
- Applies performance model
  - Taking into account PCIe transfer times, execution times, memory usage, ...
  - Derives $p$ : percentage of tasks to be conveniently executed on the CPU
- Directs $\#\text{task} \times p$ tasks to CPU cores and $\#\text{task} \times (p-1)$ tasks to GPU cores
Sample results

FastFlow + CUDA prototype implementation
Sample results

FastFlow + CUDA prototype implementation

![Graphs showing comparison between CPU/GPU mix and CPU only for two systems: S2 and S3.](image)

- **System S2, map(sq)**
  - CPU/GPU mix
  - GPU only
  - CPU only

- **System S3, matrix multiplication**
  - CPU/GPU mix
  - GPU only
  - CPU only

Flowchart:

1. Scheduling stage
2. CPU
3. GPU
4. Collecting stage
Sample results
Available tools

Skeletal frameworks

- FastFlow (Pisa+Torino), Muesli (Muenster), SKEPU (Linkoping), SkeTO (Tokio), OSL (Orleans), ...
  - Mature, maintained, documented (partially), targeting CPU and GPUs, exploiting optimizations, small *time-to-production*, performance comparable with state of the art non skeletal frameworks

Quasi skeletal frameworks

- Intel TBB (pipeline, parallel for), Microsoft TPL (farm, pipeline, data flow), OpenMP (parallel for (map, reduce)), Google (mapreduce, BSP)
Historical perspective

Past century
• Mostly academic research topics, poor/no market penetration, poor performance (compared to standards (MPI))

Early ‘00s
• Berkeley report, Google Mapreduce, ... => wider acceptance of the concepts

Nowadays
• Structured parallel programming concepts permeating “conventional/commercial” parallel programming tools
Historical perspective (2)

Sequential programming

- Assembler → FORTRAN → Structured languages (Algol, Pascal) → OO languages → Components → ...

Parallel programming

- Assembler (processes + send/receive, threads + locks/semaphores/cond variables) → Algorithmic skeletons → design patterns → parallel components → ...
Radical change needed

- Hardware going parallel everywhere
  - Mobile phone, tablet, laptop, desktop, cluster, ...
- Increasing software development costs
  - Critical time-to-production, maintainance, ...
- Exa-scale problems
  - Fault tolerance, power management
- Pico-scale problems
  - Power management, efficiency, interactivity
THANK YOU!

http://www.paraphrase-ict.eu

@paraphrase_fp7
Any questions?

I’m here up to Thu morning, or write me marco.danelutto@unipi.it
or look at my web site http://www.di.unipi.it/~marcod/
Structured vs. unstructured

1. Unstructured
   - Arbitrary composition of parallel activities
   - Arbitrary coordination patterns
   - Total control of functional and non-functional concerns in charge of the (application) programmer

2. Structured
   - Notable parallel activity schemas exploited
   - Notable coordination mechanisms/policies exploited
   - Only functional concerns in charge of the application programmer
1. MPI program
   - Rank 0 $\rightarrow$ computes function f and delivers results to rank 1
   - Rank 1 $\rightarrow$ computes function g and delivers results to rank 2
   - Rank 2 $\rightarrow$ dispatches data items to 3-11 (round robin)
   - Rank 3-11 $\rightarrow$ compute function h and deliver results to rank 12
   - Rank 12 $\rightarrow$ merges and consolidates results.

2. Setup code (-np 13), coomunication (MPI_Send(...)), termination (MSGTERM), etc.
   - In charge of the application programer
   - The same writing code for f, g, h, ...
Sample “structured”

1. Algorithmic skeleton program
   - let main f g h = pipeline(f, g, farm(h,8));
     main [1,2,3,4,5];

2. Application programmer provides functional code
   - f, g, h

3. System programmer (framework programmer) provides parallelism exploitation related code
   - And solves related problems !!
Algorithmic skeletons

1. Introduced by M. Cole in late ’80

The new system presents the user with a selection of independent “algorithmic skeleton”, each of which describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

2. More recently

An algorithmic skeleton is parametric, reusable and portable programming abstraction modelling a known, common and efficient parallelism exploitation pattern.
Algorithmic skeleton

1. Defines a *common, known* parallel pattern
   - E.g. pipeline, mapreduce, master/worker, ...

2. It is *portable* across different kind of parallel hw
   - Functional + performance portability

3. It is *efficient*
   - Alone or in composition

4. It is *reusable*
   - Parametric in the *business code*
Algorithmic skeletons epochs

1. Introduction (early ‘90s)
   - Non composable, modelling a few patterns, theoretic or library

2. Consolidation (‘90s – ’00s)
   - Skeleton based languages and libraries
   - Different implementations
   - Different host languages

3. Maturity (‘00s, ‘10s)
   - Notable programming frameworks
   - Skeleton concepts migrating to well know programming frameworks
Algorithmic skeletons

Cole’s thesis

Fortran M (Darlington)