Structured parallel programming

M. Danelutto, REPARA meeting Rapperswil, Jan 2014

Contents

- Outline of structured parallel programming
- Algorithmic skeletons
- Design patterns
- Implementation techniques
- Optimizations

Structured parallel programming

- The concept:
  - Parallelism exploited according to (possibly nested) well known patterns
  - Rather than built from scratch on top of low level mechanisms (processes/threads, scheduling, mapping, communication, synchronization, …
- Complete separation of concerns
  - System programmer
    - In charge of the implementation details, including hw targeting
  - Application programmer
    - In charge of the (application/domain specific) qualitative parallelism exploitation

Two main tracks

Algorithmic skeletons

- From HPC community
- Cole, late '80
- Ready to use patterns

Parallel design patterns

- From SW ENG community
- Mattson et al, early '00
- Implementation recipes

In both cases:

- Skeletons / patterns

  - Orchestrate parallel execution of components

  - Components
    - Wrappers (re-use existing sequential code)
    - Skeletons/patterns (semi-arbitrary nesting)

  - Different levels of abstraction provided to the user (the application programmer)

Parallel design patterns

- *A design pattern is a representation of a common programming problem along with a tested, efficient solution for that problem (Gamma book)*

Item	Contents
Problem	the problem to be solved
Context	the context where the pattern may be most suitably applied.
Forces	the different features in influencing the parallel pattern design
Solution	a description of one or more possible solutions to the problem solved by the pattern

(Mattson book)

High level design spaces

Finding concurrency space		
Decomposition	Task decomposition	Data decomposition
Dependency analysis	Group task	Order Task
	Data analysis	
Other	Design evaluation	

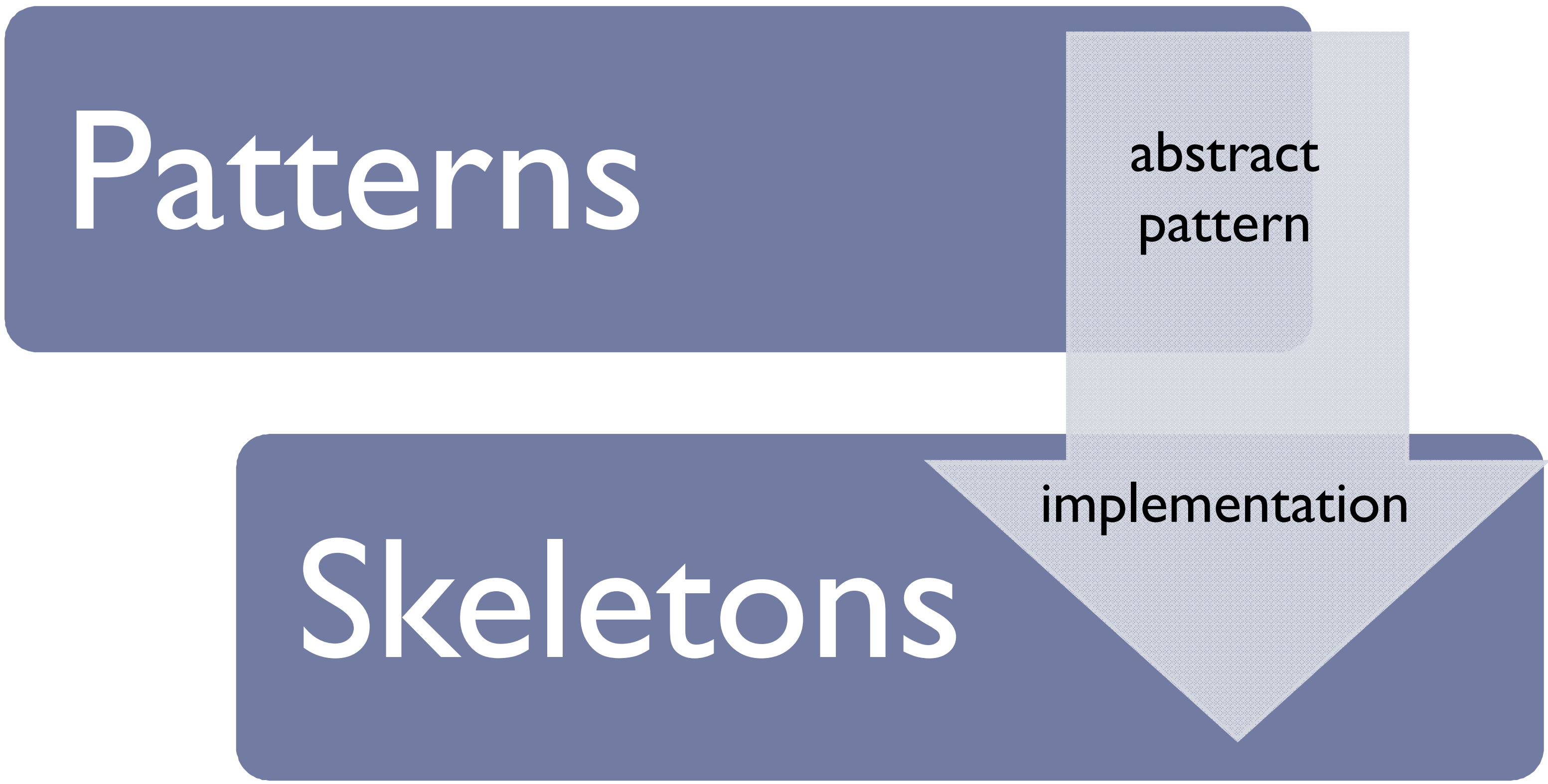
Algorithm structure space		
Organize by task	Task parallelism	Divide and conquer
Organize by data decomp.	Geometric decomp.	Recursive data
Organize by flow of data	Pipeline	Event based coordination

Low level design spaces

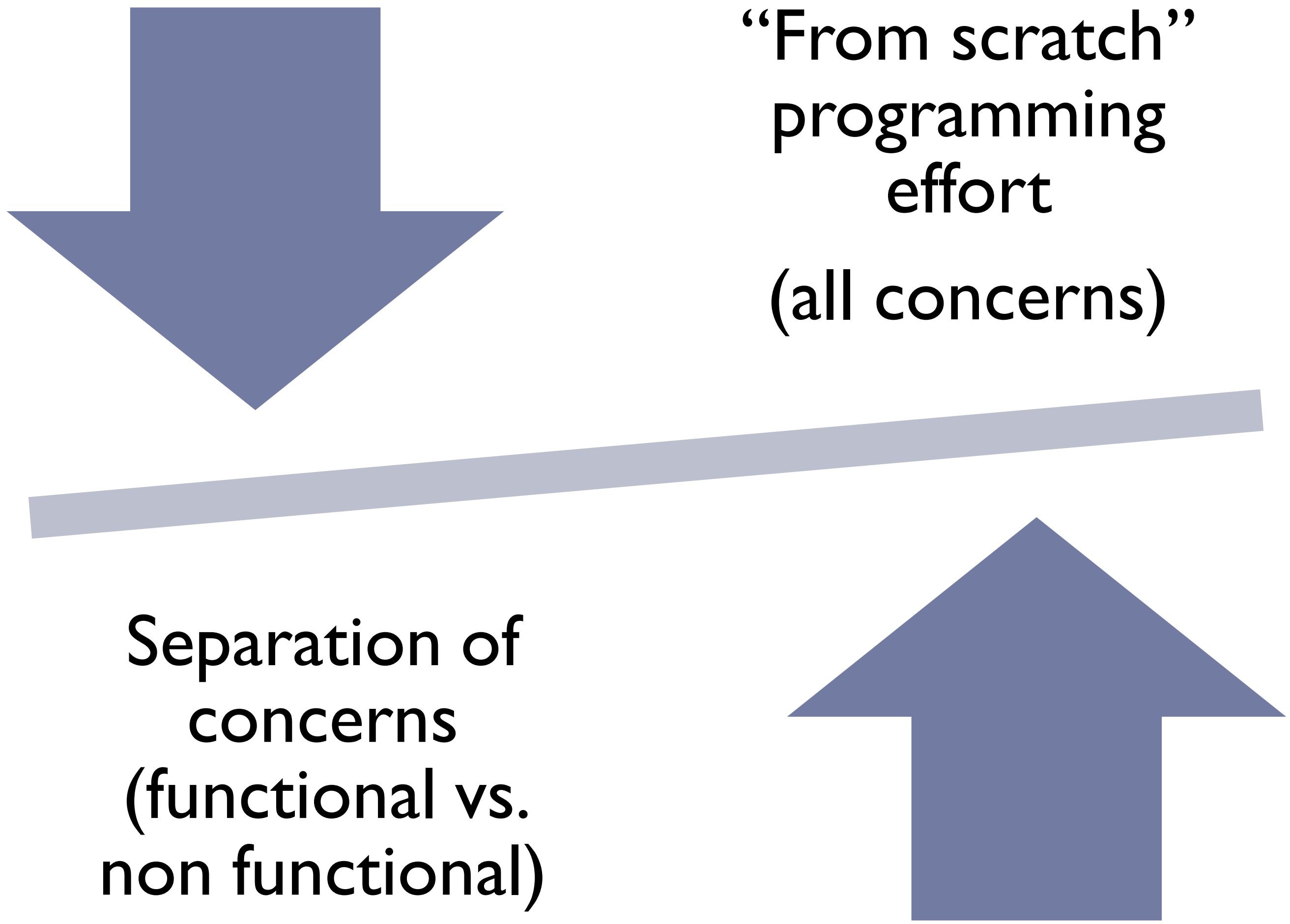
Supporting structures space		
Program structures	SPMD	Master/worker
	Loop parallelism	Fork/join
Data Structures	Shared data	Shared queue
	Distributed array	

Implementation mechanisms space		
UE management	Synchronizazion	Communication

Skeletons vs. Parallel design patterns



Unstructured vs structured



Algorithmic skeletons

- Parallelism exploitation patterns
  - *Notable*
    - Usable in a wide range of applications
  - *Efficient*
    - On a wide range of parallel architectures
  - *Parametric*
    - Functional and non functional parameters
  - *Reusable*
    - Only changing parameters
  - *Provided to application programmers as ready to use programming abstractions*
    - Object, classes, high order functions, library entries, …

Algorithmic skeletons (typical)

- Stream parallel skeletons (computing over stream items)

Skeleton	Parallelism
Task farm	Embarrassingly parallel, same computation over all items
Pipeline	Computation in stages
Feedback	Iterating same computation over results

- Data parallel skeletons (computing over sub-items of a collection)

Skeleton	Parallelism
Map	Embarrassingly parallel, same computation over all sub-items
Reduce	Summing up collection data to scalar (associative, commutative operator)
Stencil	New elements of the collection from old ones + neighbours

Algorithmic skeletons (typical)

- Higher level skeletons

Skeleton	Parallelism
Divide&Conquer	Divide problem into base cases, solve them then rebuild global solution from sub solutions
MapReduce (Google)	Map items to <key, value> and reduce values from same keys usually on big data
Branch&Bound	Explore a tree of solutions looking for optimal ones while pruning "dead" subtrees

Sample algorithmic skeletons

Ocaml

```
Parmap.parmap: (α→β) → α list → β list
```

SKEPU

```
UNARYFUN(inc,float,x, return(x+1);)
skepu::Vector<float> x,y;
skepu::Map<inc> incmap(new inc);
incmap(x,y);
```

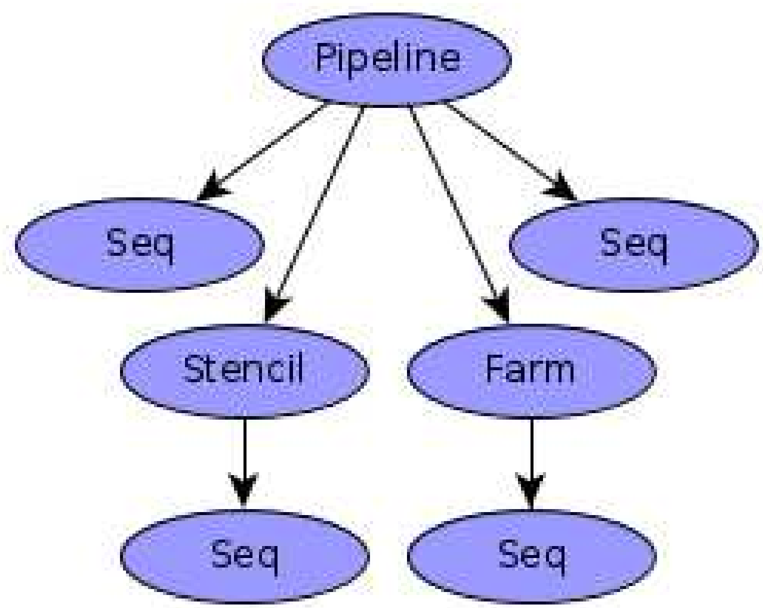
Muesli

```
Atomic<int,int> stage1 (compute, 1);
Pipe myPipe(stage0,stage1);
```

Sample skeleton code

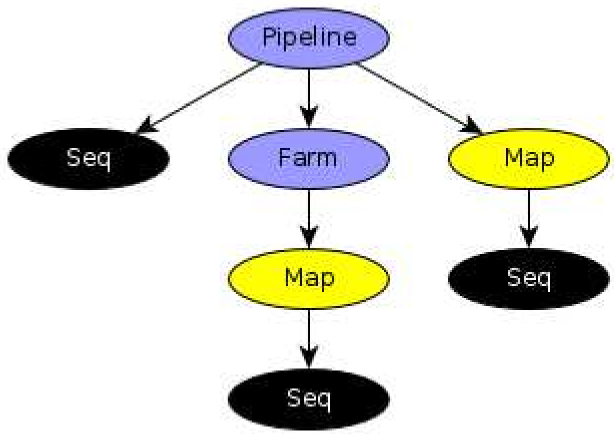
- **Image filtering application**

- ***Pipeline***
  - *Decode video to image*
  - *Stencil (Filter A)*
    - *Local filter function*
  - *Farm (Filter B)*
    - *Filter image (g)*
  - *Encode image to video*



Skeleton composition

- Not in the original Cole work
- Simpler skeleton + composition
  - Enhance the palette of parallel patterns provided
- Two tier composition
  - Top : stream parallel skeletons
  - Middle : data parallel skeletons
  - Bottom : sequential wrappers



Implementation

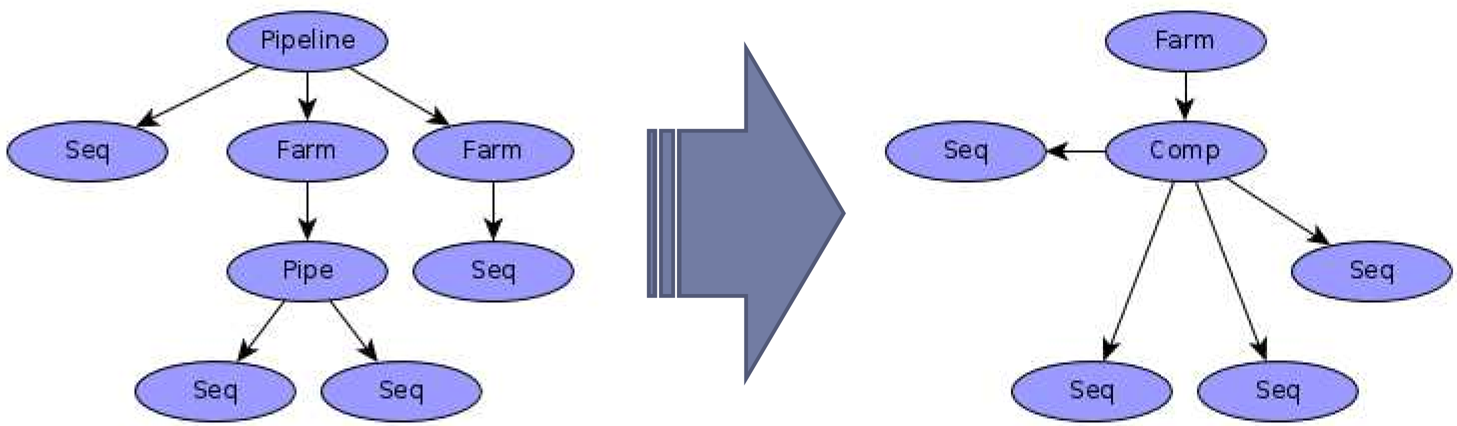
- Template based (FastFlow, SKEPU, Muesli, SkeTo)
  - Multiple visit of the skeleton tree
  - Assignment of existing templates, properly instantiated, to each skeleton
  - Optimising non functional properties (e.g. parallelism degree, mapping, …)

- Macro data flow based (Muskel, Skipper, Skandium)
  - Skeleton tree to MDF graph
  - MDF graph to parallel MDF interpreter

- Static vs. dynamic optimization tradeoff

Optimizations

- Rewriting rules
  - Farm(S) = S
  - Pipe(A, B) = Comp(A, B)
  - Pipe(Map(A), Map(B)) = Map(Pipe(A),Pipe(B))



Optimizations

- Automatically devising non functional parameters
- E.g. Pipeline(Farm(A), Farm(B))
  - Ta and Tb known
  - Pardegree(Farm(A)) : ParDegree(Farm(B)) = Ta : Tb
  - Bounded by
    - Available resources
    - Inter arrival time

Offloading

- Known parallel task structure
  - Decide (schedule) tasks to offload to accelerators
  - Automatically manage data transfers and computation scheduling
  - Profit of different kinds of accelerators in different skeletons

Optimizations

- Optimal allocation of data parallel tasks to CPU and GPU cores
  - Depending on
    - PCIe bandwidth
    - I/O data sizes
    - Kernel execution time
  - Joint exploitation of both CPU and GPU resources
    - Optimizing completion time
    - And, possibly, power consumption

- Similar results for cloud offloading of excess parallel computations

Skeleton workflow

- Program coding
  - Composition of abstract skeletons from a palette

Application programmer concern

- Compiling
  - Assignment of implementations (from library) to skeletons
  - Devising proper non functional parameters (e.g. par degree)
- Optimization
  - Target driven rewriting
- Code generation
  - High level code + library calls

System programmer concern

A little bit of history

1990

- Algorithmic skeletons introduced
- Skeleton languages (P3L, Fortran M) and libraries (OcamlP3L, Muesli)
- Mostly from HPC community, poor success

2000

- OO skeleton programming framework (Muesli, Muskel, SkeTo, ASSIST)
- Design patterns introduced – Parallel design patterns introduced
- More libraries available (C/C++, ML, Java, …)
- Google mapreduce

2010

- Mature skeleton technology (FastFlow, SKEPU, Muesli)
- Parallel design patterns recognized in multiple contexts (TBB, Microsoft TPL)
- Google (Pregel, BSP, mapreduce optimization (Flume), …)
- Co-processors targeted (SKEPU, SkeCL, Muesli, FastFlow)

Perspectives

- Google
  - Map reduce, composition optimizations, BSP on large data
- Intel TBB
  - Common and low level patterns as C++ abstractions
- Microsoft TPL
  - Stream and data parallel patterns in C#
  - Including data flow computing
- HPC community
  - More and more projects
- Sw Engineering community
  - Building new programming models (+ formal tools)

Summary/key concepts

- Parallelism = skeletons (and only skeletons)
- Implementation guarantees
  - Hardware targeting & efficiency
  - Correctness
  - Oher non functional properties (e.g. fault tolerance, power management, etc.)
- Complete separation of concerns
  - Application programmer => qualitative parallelism exploitation
  - System programmer (once and forall) => quantitative & hw dependent aspects, including portability (functional and non functional