



From sequential to parallel programming with patterns

Inverted CERN School of Computing 2018

Plácido Fernández
`placido.fernandez@cern.ch`

07 Mar 2018

Table of Contents

Introduction

Sequential vs Parallel patterns

- Patterns

- Data parallel vs streaming patterns

- Control patterns (sequential and parallel)

- Streaming parallel patterns

- Composing parallel patterns

Using the patterns

- Real world use case

- GrPPI with NUMA

Conclusions

Acknowledgements and references

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

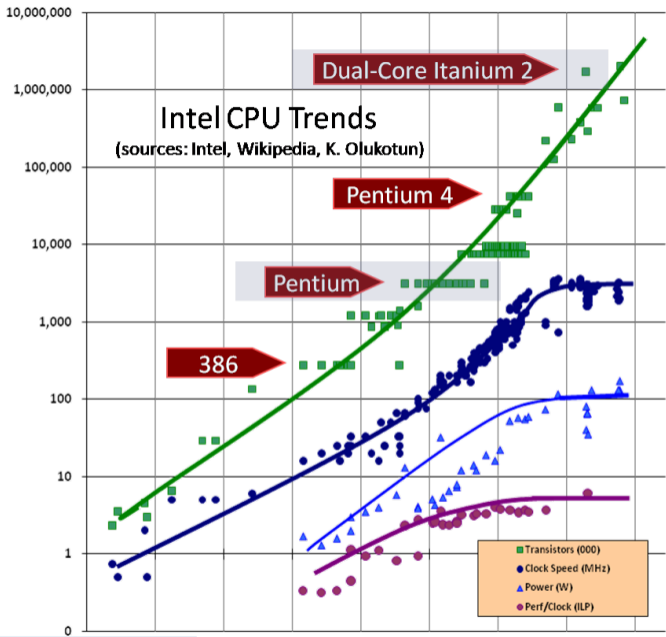
Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

Acknowledgements and references



Parallel hardware history

- ▶ Before ~2005, processor manufacturers increased clock speed.
- ▶ Then we hit the **power and memory wall**, which limits the frequency at which processors can run (without melting).
- ▶ Manufacturers response was to continue improving their chips performance by adding more parallelism.

To fully exploit modern processors capabilities, programmers need to put effort into the source code.

Parallel Hardware

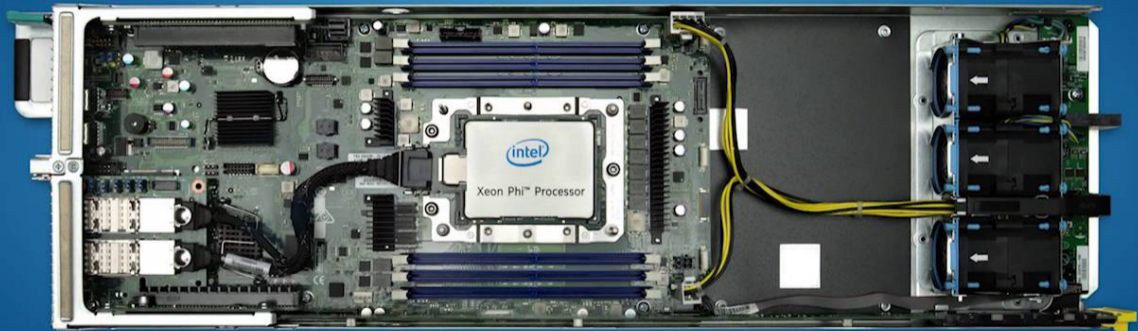
Processors are naturally parallel:

- ▶ Caches
- ▶ Different processing units (floating point, arithmetic logic...)
- ▶ Integrated GPU

Programmers have plenty of options:

- ▶ Multi-threading
- ▶ SIMD / Vectorization (Single Instruction Multiple Data)
- ▶ Heterogeneous hardware







Parallel hardware

	Xeon	Xeon 5100	Xeon 5500	Sandy Bridge	Haswell	Broadwell	Skylake
Year	2005	2006	2009	2012	2015	2016	2017
Cores	1	2	4	8	18	24	28
Threads	2	2	8	16	36	48	56
SIMD Width	128	128	128	256	256	256	512

Figure: Intel Xeon processors evolution

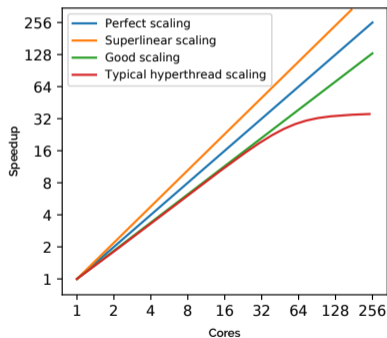
⁰Source: ark.intel.com

Parallel considerations

When designing parallel algorithms where are interested in **speedup**. How much faster (or slower) does my code run with parallel algorithms?

Speedup

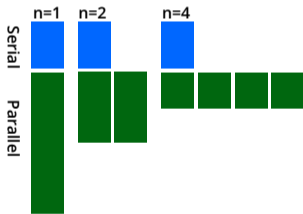
$$S_p = \frac{Time_1}{Time_n}$$



Predicting scalability

Amdahl's Law

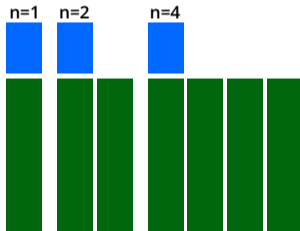
$$S_p = \frac{1}{(1 - p) + p/n}$$



n = number of parallel parts
p = parallel % of the program

Gustafson-Barsis' Law

$$S_p = 1 - p + np$$



Work-Span model

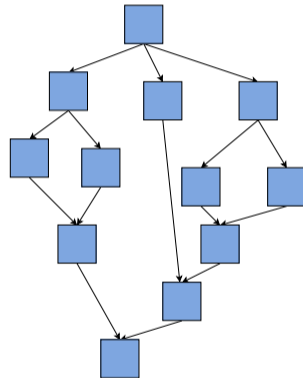


Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

Acknowledgements and references

Pattern

Software design pattern (Wikipedia): general, reusable solution to a commonly occurring problem in a given context in software design.

Parallel pattern: recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.

Here we are mostly interested in **control flow** patterns and distinguish between **data parallel and streaming patterns**.

Sequential and parallel patterns

Structured serial control flow patterns

- ▶ Sequence
- ▶ Selection
- ▶ Iteration
- ▶ Reduce

Parallel control and data management patterns

- ▶ Fork-join
- ▶ Map
- ▶ Stencil
- ▶ Farm
- ▶ Superscalar sequence
- ▶ Parallel reduce
- ▶ Pipeline
- ▶ Geometric decomposition

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

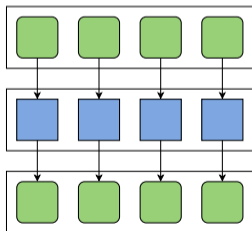
Conclusions

Acknowledgements and references

Data parallel vs streaming patterns

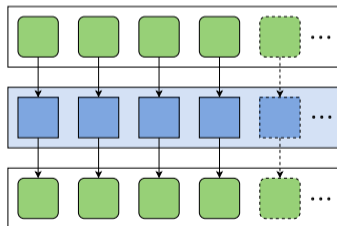
Data parallel patterns

- ▶ Map
- ▶ Farm
- ▶ Reduction
- ▶ Stencil



Streaming patterns

- ▶ Farm
- ▶ Pipeline



Data parallel vs streaming parallel patterns

- ▶ Size of the input + dependencies between items define which pattern to use.
- ▶ Data parallel patterns may not be efficient in streaming scenarios, and the other way around.
- ▶ For streaming patterns, there is usually one (or more) input items that distributes the input elements to working items as they come.

Parallel patterns

- ▶ By using parallel patterns, source code can be "free of threads and vector intrinsics".
- ▶ Which parallel patterns to choose depends the type of problem we are addressing and the dependencies between the items.

Also programming in terms of generic parallel patterns address the typical parallel programming problems:

- ▶ Race conditions
- ▶ Deadlocks
- ▶ Strangled scaling
- ▶ Lack of locality
- ▶ Load imbalance
- ▶ Overhead

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

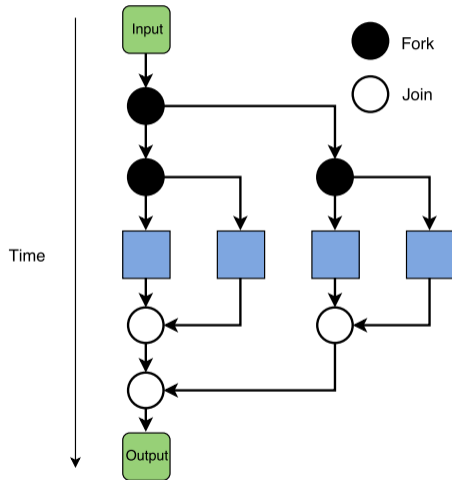
GrPPI with NUMA

Conclusions

Acknowledgements and references

Fork-join (parallel)

- ▶ Generalization of any problem that can be partitioned.
- ▶ The load of each item can be split to get a smaller grain size.
- ▶ Recursive.
- ▶ Base implementation of many parallel patterns.

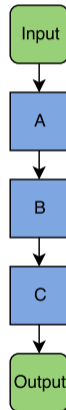


Sequence (sequential)

- ▶ **Ordered** list of items to compute.
- ▶ Dependencies between them don't matter, so side-effects won't affect.
- ▶ Items without dependencies also run one after the other.

Note that

Compilers will try to reorder instructions if they consider that is an optimization.

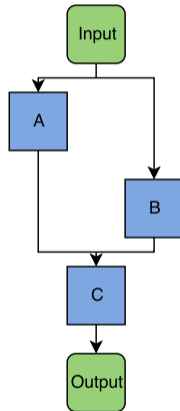


Sequence (sequential)

- ▶ **Ordered** list of items to compute.
- ▶ Dependencies between them don't matter, so side-effects won't affect.
- ▶ Items without dependencies also run one after the other.

Note that

Compilers will try to reorder instructions if they consider that is an optimization.

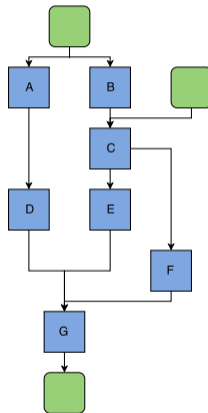


Superscalar sequence (parallel)

- ▶ Is the parallel generalization of the sequence.
- ▶ Items are ordered attending to the dependencies between them.
- ▶ Items without dependencies may run in parallel.

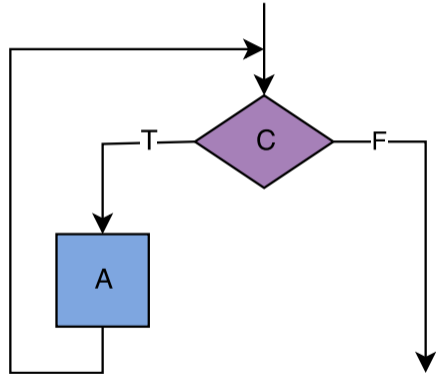
Used in

This is the pattern followed by HEP data processing frameworks!



Iteration (sequential)

- ▶ Used to loop over collections of items.
- ▶ We can fit it as a "while" or "for" loops commonly used in programming languages.
- ▶ Checks if condition is met, then runs an item or continues.
- ▶ May seem trivial to parallelize, but... **Check dependencies!**



Parallel patterns derived from iteration

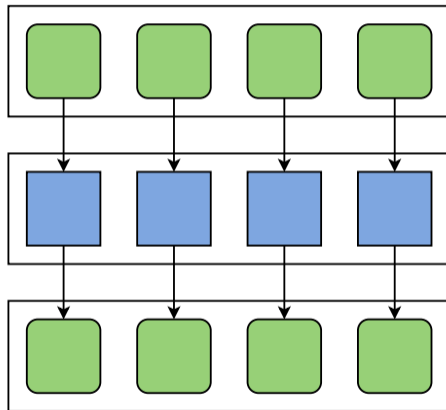
- ▶ There is no one but many parallel pattern equivalents.
- ▶ Many parallel patterns are generalizations of the iteration pattern.
- ▶ Which one to use is given by the dependencies between the items.
- ▶ If the input is not known in advance, usually we have streaming alternatives.

Map (parallel)

- ▶ Used on embarrassingly parallel collections of items.
- ▶ Same function applied to every item, all at the same "time".
- ▶ Applicable if all items are independent.
- ▶ Usually good candidate for SIMD abstractions.

Used in

Ray tracing, Monte Carlo simulations.



Reduction (sequential)

- ▶ Combines a collection of items into one, with a defined operation.
- ▶ Many different partition options.
- ▶ Elements depend on each other, but are associative.

Used in

Matrix operations, computing of statistics on datasets.

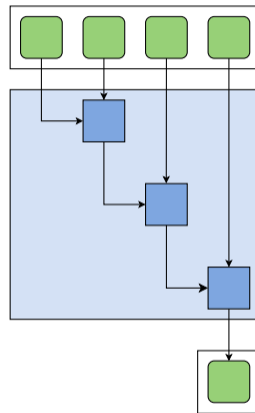


Figure: Sequential reduction

Reduction (parallel)

- ▶ Combines a collection of items into one, with a defined operation.
- ▶ Many different partition options.
- ▶ Elements depend on each other, but are associative.

Used in

Matrix operations, computing of statistics on datasets.

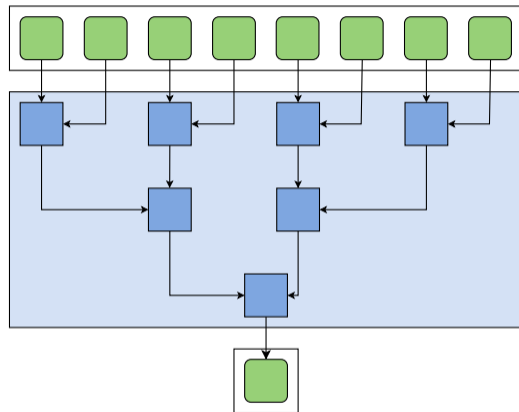


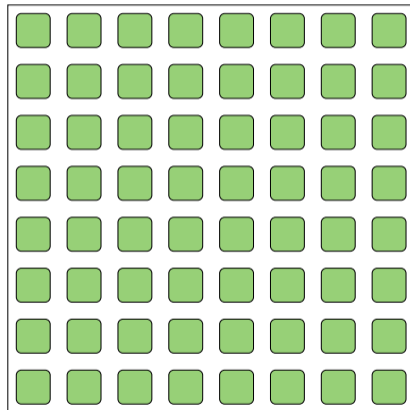
Figure: Parallel reduction

Geometric decomposition (parallel)

- ▶ Divides input into smaller collections of items.
- ▶ It provides a different organization of the data

Used in

Image compression, matrix multiplication, spatial-temporal simulations.

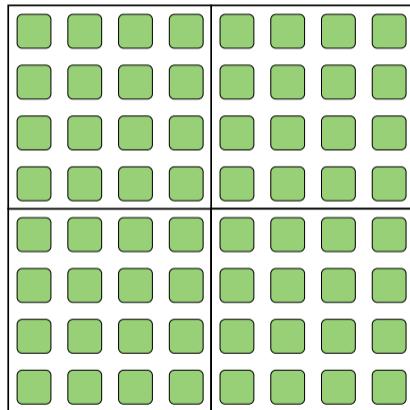


Geometric decomposition (parallel)

- ▶ Divides input into smaller collections of items.
- ▶ It provides a different organization of the data

Used in

Image compression, matrix multiplication, spatial-temporal simulations.

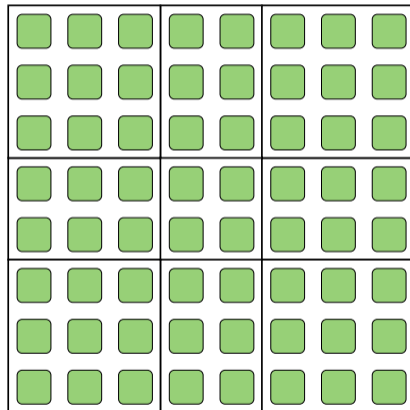


Geometric decomposition (parallel)

- ▶ Divides input into smaller collections of items.
- ▶ It provides a different organization of the data

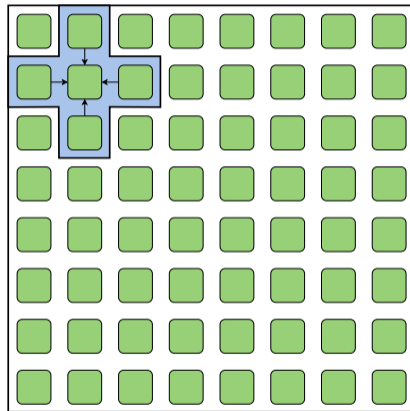
Used in

Image compression, matrix multiplication, spatial-temporal simulations.



Stencil (parallel)

- ▶ When for every item of a collection, we need data from the neighbourhood items.
- ▶ Usually a fixed number of neighbourhood is accessed.
- ▶ Boundary conditions have to be taken into account.
- ▶ Data reuse in the implementation (cache).

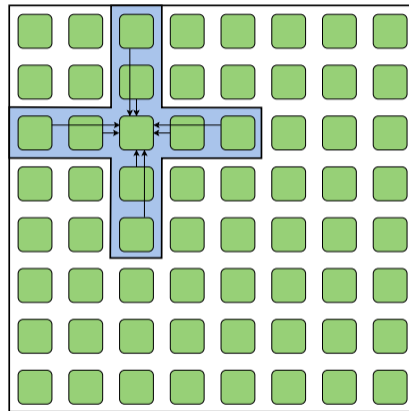


Used in

Signal filtering, image processing, grid methods.

Stencil (parallel)

- ▶ When for every item of a collection, we need data from the neighbourhood items.
- ▶ Usually a fixed number of neighbourhood is accessed.
- ▶ Boundary conditions have to be taken into account.
- ▶ Data reuse in the implementation (cache).



Used in

Signal filtering, image processing, grid methods.

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

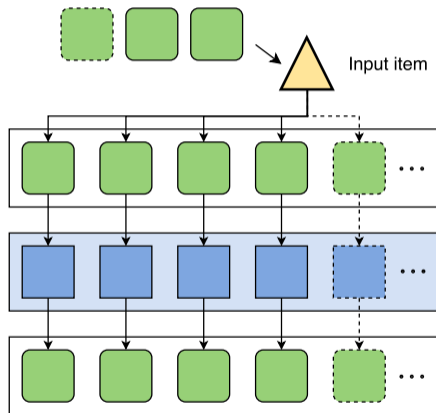
Acknowledgements and references

Farm (parallel streaming)

- ▶ Similar to map, but size of collection is not known in advance.
- ▶ Used for embarrassingly parallel computations in stream computations.
- ▶ There at least one producer item.

Used in

Used in HEP online trigger software.



Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

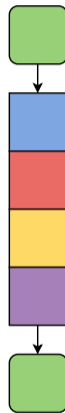


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

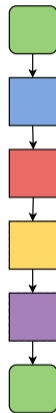


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

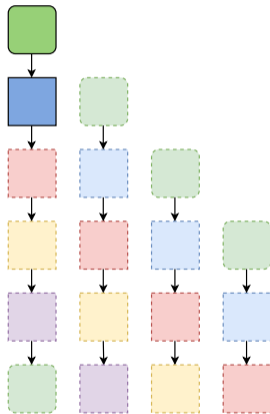


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

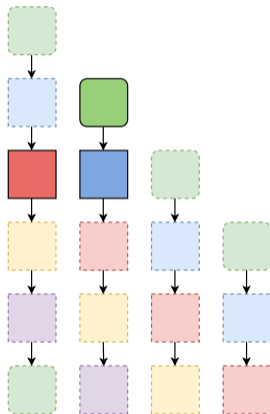


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

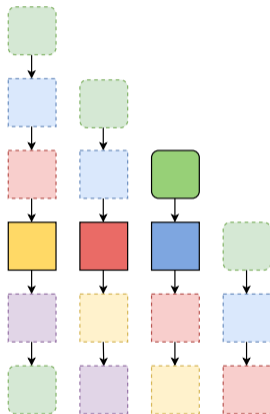


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

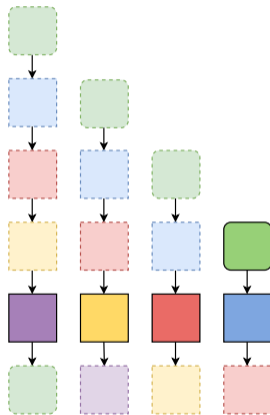


Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.



Pipeline (streaming)

- ▶ Size of collection not needed in advance.
- ▶ Different steps run in parallel, but others may not be able to run in parallel.
- ▶ Different functions are applied in different steps, where the order is important.

Used in

image filtering, signal processing,
game engines.

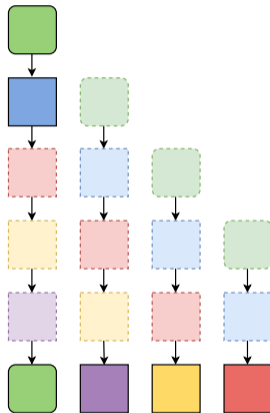


Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

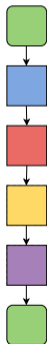
GrPPI with NUMA

Conclusions

Acknowledgements and references

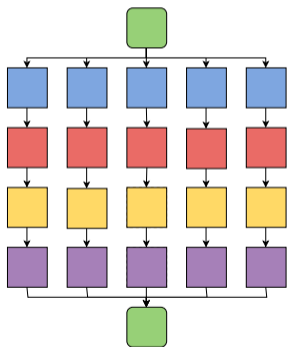
Composing parallel patterns

- ▶ Complex parallelization schemes can be created by composing and nesting different parallel patterns.



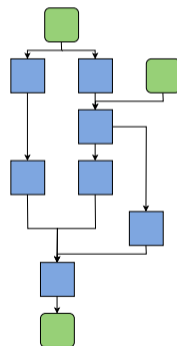
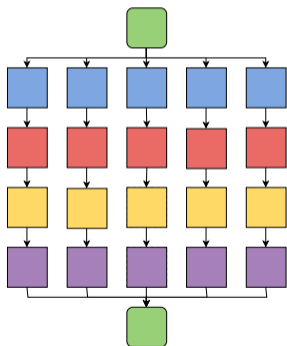
Composing parallel patterns

- ▶ Complex parallelization schemes can be created by composing and nesting different parallel patterns.



Composing parallel patterns

- ▶ Complex parallelization schemes can be created by composing and nesting different parallel patterns.



Composing parallel patterns

- ▶ Complex parallelization schemes can be created by composing and nesting different parallel patterns.

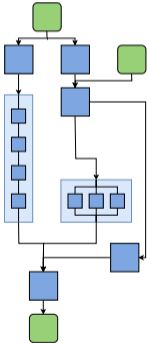
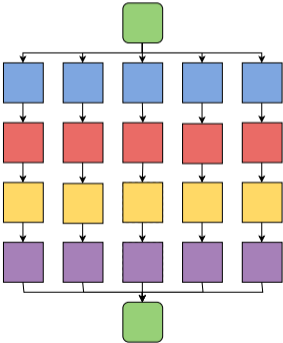


Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

Acknowledgements and references

Using the patterns

- ▶ These patterns are usually found in high performance computing environments, but can be applied to any scenario and any programming language with threading capabilities.
- ▶ Computing in parallel gives an speedup, but also causes overhead.
- ▶ Be careful with overparallelizing or making items to small, watch the grain size.

Many available options

C++

- ▶ Custom thread implementation
- ▶ Intel TBB
- ▶ Open MP
- ▶ Intel Cilk
- ▶ HPX
- ▶ OpenCL

Others

- ▶ MPI
- ▶ Python multiprocessing
- ▶ River-trail Javascript engine
- ▶ Rust Rayon
- ▶ Hadoop, Spark (Java, Scala, etc)
- ▶ Java 8 Streams

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

Acknowledgements and references

- ▶ Generic Reusable Parallel Patterns Interface.
- ▶ Simple interface that hides the complexity of the used concurrency mechanisms.
- ▶ Supports data parallel and streaming computations.
- ▶ No deep understanding of parallel frameworks or third party libraries needed.
- ▶ Various backends supported: C++ threads, Intel TBB, OpenMP, CUDA.
- ▶ Changing the used backend means just changing one line of code.

GrPPI example

A Map example

```
1  #include "grppi.h"
2
3  // execution environment
4  grppi::parallel_execution_thr p {num_threads};
5
6  // collection of items
7  std::vector<int> data(n);
8
9  // Processing lambda
10 const auto processFn = [](InputData data){...}
11
12 grppi::map(p, data, processFn);
```

GrPPI example

A Map example using OpenMP as a backend

```
1  #include "grppi.h"
2
3  // execution environment
4  grppi::parallel_execution_omp p {num_threads};
5
6  // collection of items
7  std::vector<int> data(n);
8
9  // Processing lambda
10 const auto processFn = [](InputData data){...}
11
12 grppi::map(p, data, processFn);
```

GrPPI example

A Map example using TBB as a backend

```
1  #include "grppi.h"
2
3  // execution environment
4  grppi::parallel_execution_tbb p {num_threads};
5
6  // collection of items
7  std::vector<int> data(n);
8
9  // Processing lambda
10 const auto processFn = [](InputData data){...}
11
12 grppi::map(p, data, processFn);
```

GrPPI example

A Farm example

```
1  #include "grppi.h"
2
3  // execution environment
4  grppi::parallel_execution_thr p {num_threads};
5
6  // Input lambda
7  const auto inputFn = [] (InputData data){...}
8
9  // Processing lambda
10 const auto processFn = [] (InputData data){...}
11
12 grppi::farm(p, inputFn, processFn);
```

GrPPI example

A composed pipeline-farm example

```
1  #include "grppi.h"
2
3  grppi::parallel_execution_thr p {num_threads};
4
5  const auto inputFn = [] (InputData data){...}
6  const auto stepOneFn = [] (InputData data){...}
7  const auto stepTwoFn = [] (InputData data){...}
8  const auto stepThreeFn = [] (InputData data){...}
9
10 grppi::pipeline(p,
11     inputFn,
12     grppi::farm(num_threads/3, stepOneFn),
13     grppi::farm(num_threads/3, stepTwoFn),
14     grppi::farm(num_threads/3, stepThreeFn)
15 );
```

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

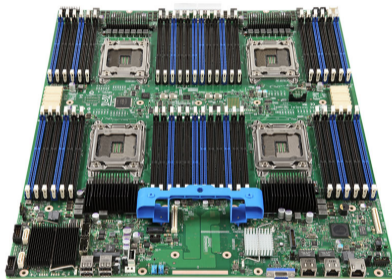
GrPPI with NUMA

Conclusions

Acknowledgements and references

GrPPI with NUMA awareness

- ▶ Topology detection with hwloc library
- ▶ Generate hierarchy and balance the load across the different nodes, cores and hyperthreads.
- ▶ Threads need to be pinned according to the topology.



Source: ark.intel.com

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions

Acknowledgements and references

Conclusions

- ▶ Generic patterns create useful abstraction to express parallelism in a easy and intuitive way.
- ▶ Using them makes it easier to avoid the typical parallelism errors and complexities.
- ▶ Nested parallelism allows to create complex patterns for big problems.
- ▶ Side effects need to ensured by the programmer.
- ▶ Vector and thread parallelism can be covered with these patterns.

Table of Contents

Introduction

Sequential vs Parallel patterns

Patterns

Data parallel vs streaming patterns

Control patterns (sequential and parallel)

Streaming parallel patterns

Composing parallel patterns

Using the patterns

Real world use case

GrPPI with NUMA

Conclusions






Acknowledgements and references

Acknowledgements

Thanks to all these people:

- ▶ My CERN supervisors, Omar and Niko
- ▶ The ARCOS department at University Carlos III of Madrid
- ▶ Daniel Cámpora
- ▶ The HTCC members
- ▶ Sebastien Ponce, Enric Tejedor and Danilo Piparo
- ▶ Sebastian Lopienski and the iCSC team

References

-  D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, J. G. Blas, and J. D. García, “A c++ generic parallel pattern interface for stream processing,” in *Algorithms and Architectures for Parallel Processing*, pp. 74–87, Springer, 2016.
-  “GrPPI: Generic Reusable Parallel Patterns Interface.”
<https://arcosuc3m.github.io/grppi/>.
-  M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, “Fastflow: high-level and efficient streaming on multi-core,” *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2014.
-  E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
-  M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.



www.cern.ch