Parallel Programming Patterns

Francesco De Felice Alessandro Lenzi

Cynny S.p.A

September 7, 2016

Outline

1. Introduction
   - Motivation
   - Parallel Programming Patterns
   - Sources of Parallelism
2. Stream Parallelism
   - Bottlenecks in Stream Computations
   - Pipeline
   - Farm
3. Data Parallelism
   - Map
   - Reduce
   - Stencil
4. Structured Design Methodology

Outline

1. Introduction
   - Motivation
   - Parallel Programming Patterns
   - Sources of Parallelism
2. Stream Parallelism
   - Bottlenecks in Stream Computations
   - Pipeline
   - Farm
3. Data Parallelism
   - Map
   - Reduce
   - Stencil
4. Structured Design Methodology

Motivation

Why parallel design patterns?

- Current hardware is highly parallel (i.e. shared-memory, distributed-memory and heterogeneous architectures).
  Need of efficient ways to exploit parallel architectures.
- Finding the best parallelization is a NP-Hard problem
- Business code and parallel exploitation code require different domains of expertise
- We need metrics to evaluate the goodness of a parallel solution, both *a-priori* and *a-posteriori*
- It is possible to observe that *efficient* parallel implementations often exploit the same basic ideas (*patterns*)

Parallel Programming Patterns

Definition

Schemes of efficient parallel computations that recur in the realization of many real-life applications

With the following features

1. they restrict the parallel computation structure to certain *predefined* and *limited* set of patterns,
2. they are characterized by a specific performance model,
3. they can be composed each other to form complex computations
4. they reduce the complexity of the parallel program design process

Efficient parallel programming patterns implementations are available by means of framework.

Structured Parallel Programming Methodology

Main characteristics of structured parallelization methodology:

1. individuate in a computation graph a module that we want to parallelize
2. explore the problem space finding a suitable parallel pattern
3. use the associated performance model
4. evaluate with empiric results the goodness of the solution
5. eventually repeat

The output of this process is a *functionally equivalent* computation graph with better performances.

The main strength of this approach is the clean separation of the business code and the parallel exploitation code.

Sources of Parallelism

Parallel patterns are divided in two main categories, depending on the feature of the problem space exploited to parallelize.

- *stream parallelism*
- *data parallelism*

Outline

1. Introduction
   - Motivation
   - Parallel Programming Patterns
   - Sources of Parallelism
2. Stream Parallelism
   - Bottlenecks in Stream Computations
   - Pipeline
   - Farm
3. Data Parallelism
   - Map
   - Reduce
   - Stencil
4. Structured Design Methodology

What is Stream Parallelism

Stream parallelism

Parallelism arising from the computation of *distinct* and *independent* input stream elements.

GOAL: *maximize* system throughput.

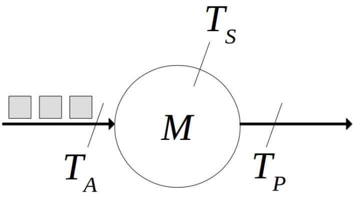
We will deal with two main stream parallel patterns:

- *pipeline*
- *farm*.

Bottlenecks in stream computations

Bottleneck definition

Given a processing module (or a computational graph) M working on stream, with mean inter-arrival time T_A, and performing a computation with service time T_S then it is a **bottleneck** if $T_A < T_S$.



The inter-departure time of M, T_P, is defined as the mean time between two consecutive results onto the output stream, and it is defined as:

$$T_P = max(T_A, T_S)$$

Pipeline

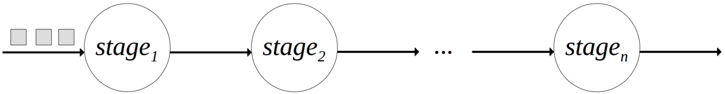
Definition

A **pipeline** is defined as the composition of functions f_1,...,f_n so that the output of f_i is the input of f_{i+1}, $\forall\, i = 1, \ldots, n$.

- The pipeline higher order functions is defined as:

  $$pipeline :: (\alpha_0 \rightarrow \alpha_1) \times (\alpha_1 \rightarrow \alpha_2) \times \cdots \times (\alpha_{n-1} \rightarrow \alpha_n) \times \alpha_0\ stream \rightarrow \alpha_n\ stream$$

- The composed functions, mapped onto different pipeline stages, are computed in parallel onto *different* items of the input stream.



Pipeline Features and Performance Model

- functional partitioning
- parallelism can be exploited only on stream computations
- max parallelism degree is given by the number of composed functions
- stages could be unbalanced according their own computation time

Cost Model

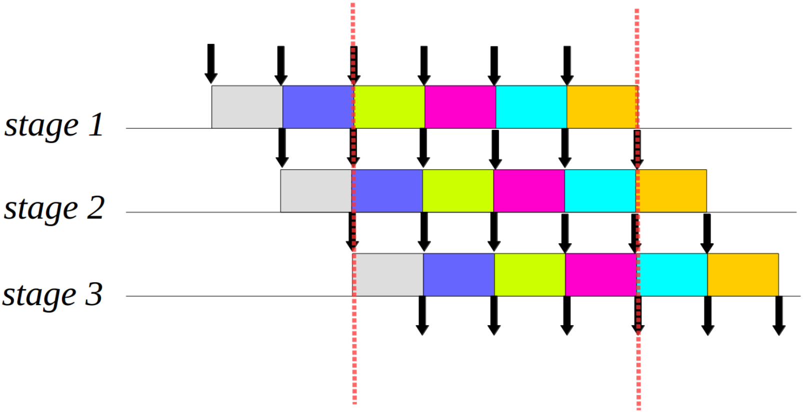
$$T_{pipeline}(f_1, ..., f_n) = max(T_{f_1}, ..., T_{f_i}, ..., T_{f_n})$$

where T_{f_i} is the average time to compute f_i.

Example

A computation that filter images and then recognize characters string appearing in the images could be written as a two stage pipeline with functions: *filter* :: *image* $\rightarrow$ *image* and *recognize* :: *image* $\rightarrow$ *string list*

Pipeline effect graphically



Farm

Definition

A **farm** corresponds to the replication of a given *pure/stateless* function f to be applied, in parallel, to each distinct element of the stream.

- The farm higher order functions is defined as:

$$\textit{farm} :: (\alpha \to \beta) \times \alpha \ \textit{stream} \to \beta \ \textit{stream}$$

Farm

Definition

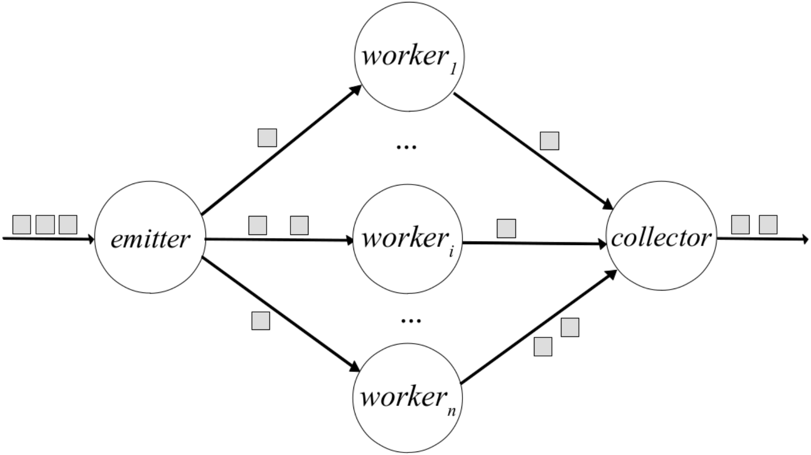
A **farm** corresponds to the replication of a given *pure*/*stateless* function f to be applied, in parallel, to each distinct element of the stream.

- The farm higher order functions is defined as:

$$\mathit{farm} :: (\alpha \rightarrow \beta) \times \alpha\ \mathit{stream} \rightarrow \beta\ \mathit{stream}$$

- The farm pattern can be modelled as a computation graph with n identical functional modules, called *workers*, a scheduling module and a collecting module named *Emitter* and *Collector* respectively.

Farm computation graph



Farm Performance Model

- It works on stateless functions only
- different scheduling strategies can be applied to handle *load-balancing*, i.e. round-robin vs on-demand
- parallelism can be exploited on stream computations only
- data has to be *replicated* in all the functional modules

Cost Model

$$T_{farm}(f, n) = max(T_{emitter}, \frac{T_f}{n}, T_{collector})$$

Outline

1. Introduction
   - Motivation
   - Parallel Programming Patterns
   - Sources of Parallelism
2. Stream Parallelism
   - Bottlenecks in Stream Computations
   - Pipeline
   - Farm
3. Data Parallelism
   - Map
   - Reduce
   - Stencil
4. Structured Design Methodology

What is Data Parallelism

...and when to use it

Data parallelism

Replication of the same functionality and partitioning of data, so that workers carry on the same operations on distinct data partitions in parallel.

- *increase* system bandwidth and *decrease* completion time.
- can be used on single computations as well as on streams

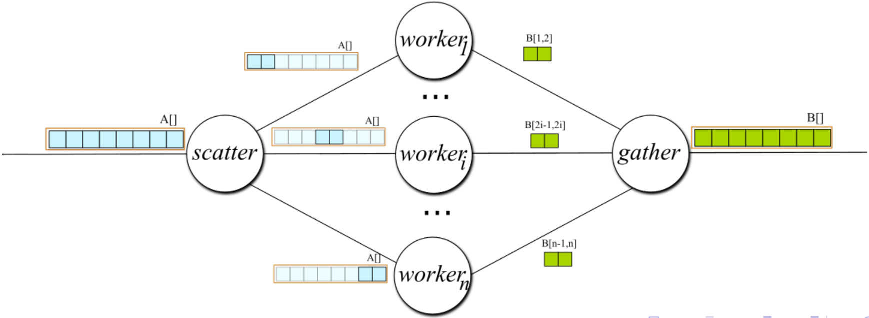
Three parallel patterns here: *map*, *reduce* and *stencil*

Map

Definition

A map applies in parallel a certain function f over independent elements of type $'A$ of a collection A.

- higher order function $map : ('A \rightarrow' B) \times' A\ collection \rightarrow' B\ collection$
- the business code f with type $'A \rightarrow' B$ is applied in parallel on all elements of A.



Features and Cost-Model

- workers are fully independent
- can be used to carry on stateful computations
- can be unbalanced depending on the parallelism degree, on the size of the collection and of the *variance* of T_f

Cost Model

$$T_{map}(f, N, n) = T_{scatter} + T_{gather} + T_f \times \left\lceil \frac{N}{n} \right\rceil$$

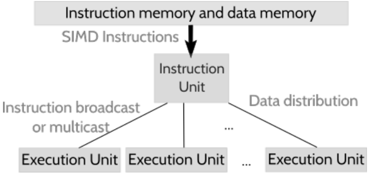
where N is the size of the input collection and n is the number of map parallel workers

Map architectural example

SIMD and SIMT based processing unit are based on the map paradigm

Map architectural example

SIMD and SIMT based processing unit are based on the map paradigm



Map architectural example

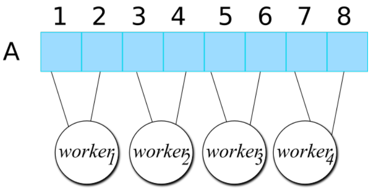
SIMD and SIMT based processing unit are based on the map paradigm

Reduce

Definition

A *reduce* operation applies in parallel a certain associative operator $\otimes$ over a collection of elements, calculating $A[1] \otimes A[2] \otimes \cdots \otimes A[N]$

- higher order function *reduce* : $('A \rightarrow' A) \times' A\ collection \rightarrow' A$
- the business code is contained in a function f with type $'A \rightarrow' A$
- can be implemented either sequentially or in form of a *tree*, depending on the cost of $\otimes$ operator.

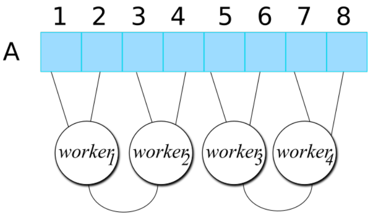


Reduce

Definition

A *reduce* operation applies in parallel a certain associative operator $\otimes$ over a collection of elements, calculating $A[1] \otimes A[2] \otimes \cdots \otimes A[N]$

- higher order function *reduce* : $('A \rightarrow' A) \times' A\ collection \rightarrow' A$
- the business code is contained in a function f with type $'A \rightarrow' A$
- can be implemented either sequentially or in form of a *tree*, depending on the cost of $\otimes$ operator.

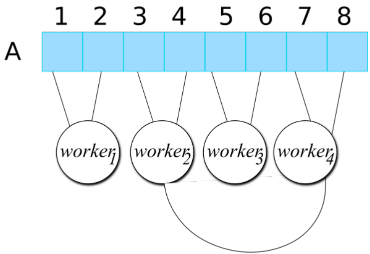


Reduce

Definition

A *reduce* operation applies in parallel a certain associative operator $\otimes$ over a collection of elements, calculating $A[1] \otimes A[2] \otimes \cdots \otimes A[N]$

- higher order function *reduce* : $('A \rightarrow' A) \times' A\ collection \rightarrow' A$
- the business code is contained in a function f with type $'A \rightarrow' A$
- can be implemented either sequentially or in form of a *tree*, depending on the cost of $\otimes$ operator.

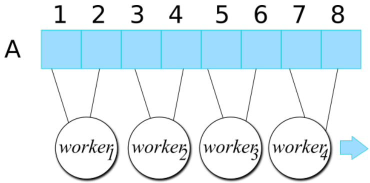


Reduce

Definition

A *reduce* operation applies in parallel a certain associative operator $\otimes$ over a collection of elements, calculating $A[1] \otimes A[2] \otimes \cdots \otimes A[N]$

- higher order function *reduce* : $('A \rightarrow' A) \times' A\ collection \rightarrow' A$
- the business code is contained in a function f with type $'A \rightarrow' A$
- can be implemented either sequentially or in form of a *tree*, depending on the cost of $\otimes$ operator.



Reduce features and cost-model

- if operator $\otimes$ is lightweight, it can be implemented sequentially; in this case associativity is not needed.
- tree-based implementations needed in case of long latency operations; the mapping can be done on a linear string of processors

Cost Model

$$T_{reduce\_tree}(\otimes, N, n) = T_{\otimes} \times \left(\left\lceil \frac{N}{n} \right\rceil + log_2(n) \right)$$

$$T_{reduce\_seq}(\otimes, N, n) = T_{\otimes} \times \left(\left\lceil \frac{N}{n} \right\rceil + n \right)$$

Stencil

Definition

In a *stencil-based* computation the workers cooperate by *exchanging* or *sharing* information, because of data dependencies.

- a stencil represents a data dependence pattern implemented by interworker cooperation
- stencil can be either static (fixed or variable) or dynamic, depending on their predictability
- usually implemented in steps
- *alternative* to data replication or a solution in case the results calculated by other workers are needed.

Stencil (cont.)

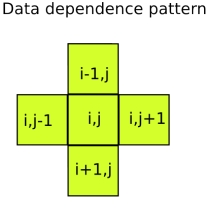
Consider the case in which a function $f(a_{i-1}, a_i, a_{i+1})$ is used to calculate b_i of the output data structure.

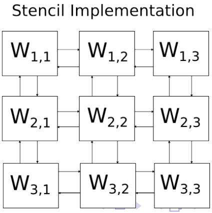
- Trivial solution (map): replicate (a part of) A among all workers
  - growth in memory usage
  - increased cost of communication
- Smarter solution: *partition* data and send it as needed
  - spare memory
  - though capable of executing in parallel, workers are not independent anymore

Stencil example

Consider the case in which every point in a discrete space is updated by a function applied to itself and to some neighbour points.

```
input: A[M][M]
do {
 for every (i,j):
    swap(old_A, A);
    A[i][j] = F(old_A[i][j], old_A[i-1][j], old_A[i+1][j],
                old_A[i][j-1], old_A[i][j+1]);
} until (convergence);
```





Outline

1. Introduction
   - Motivation
   - Parallel Programming Patterns
   - Sources of Parallelism
2. Stream Parallelism
   - Bottlenecks in Stream Computations
   - Pipeline
   - Farm
3. Data Parallelism
   - Map
   - Reduce
   - Stencil
4. Structured Design Methodology

The (in)famous matrix-vector product; imagine to have a stream of matrices and vector coming in input. The sequential algorithm is the following:

```
C[i] = 0
for i = 1 to M:
 for j = 1 to M:
  C[i] += A[i][j] * B[j]
```

Three possible parallelization: *farm*, *map*, *stencil*.

Farm approach

No state, different elements in input

- every worker carries on the whole computation
- receives in input (A[M][M], B[M])

Pros:

- balancing in case of variable sizes of input elements
- embarassingly-parallel computation

Cons:

- huge space occupancy
- latency is *increased*

Map approach

- partition data in terms of rows, replicate B across all processing units
- every worker will own $\frac{N}{n}$ rows of A and elements of C, a copy of B
- partition in terms of single elements of A avoided because of data dependencies

Pros:

- reduced space occupancy w.r.t farm
- advantages both in terms of latency and bandwidth
- in shared memory negligible communication and synchronization costs

Cons:

- load unbalancing
- depending on the size of B and of A the space occupancy is not the best

Stencil approach

- partition A, B and C
- every worker will initially have $\frac{N}{n}$ rows of A, elements of B and of C
- the worker i starts by computing with the current data, while sending them to the following one (in modulo)
- when the next phase begins, the element eventually waits for the data and starts computing with it
  - possible because of commutativity and associativity of addition
  - static, fixed stencil

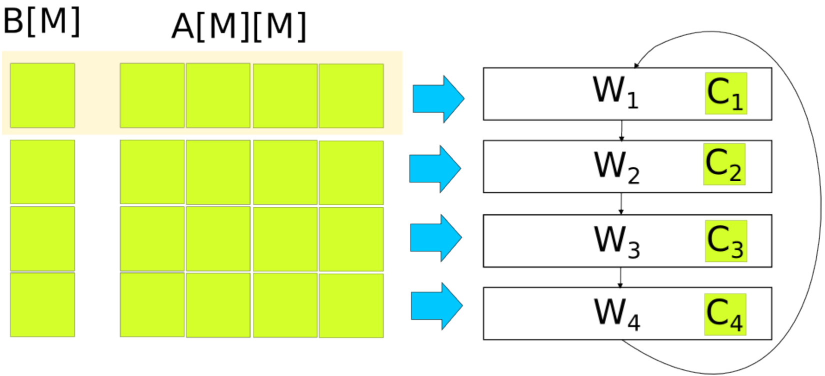
Pros:

- minimal space occupancy and communication
- communication can be overlapped with calculation
- reduced completion time and increased bandwidth

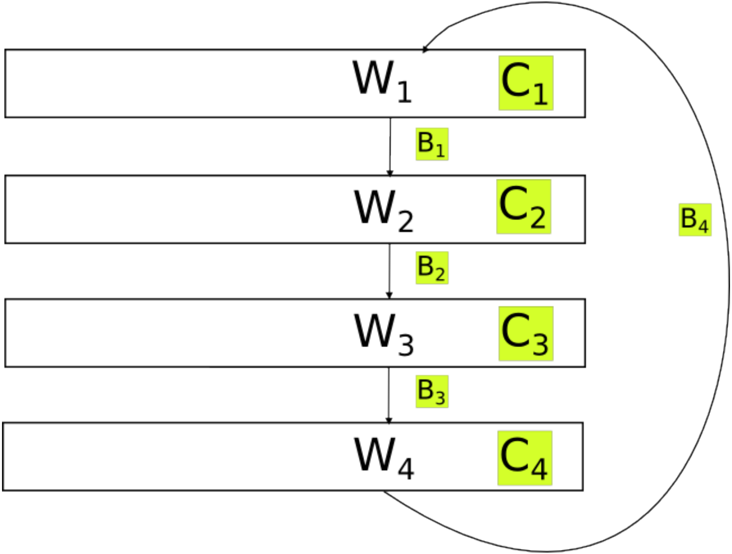
Cons:

- more difficult to implement
- communication costs could potentially decrease efficiency
- issues like *false-sharing* could arise.

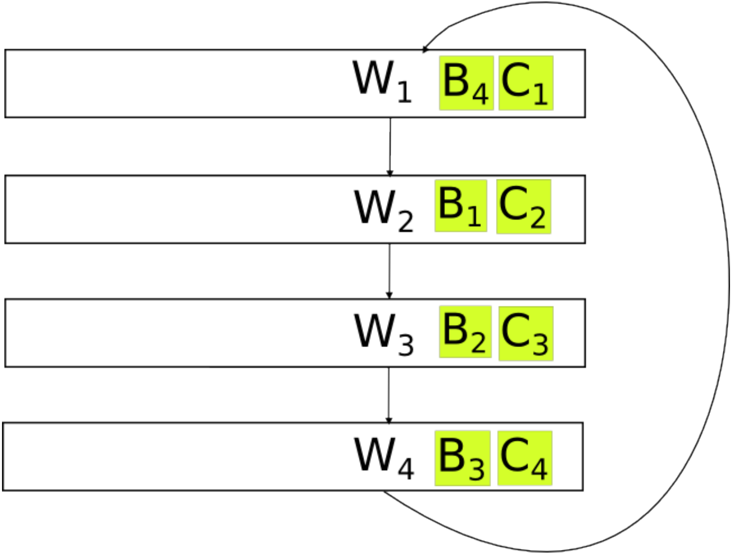
Sample stencil steps



Sample stencil steps



Sample stencil steps



Conclusion

- Parallel programming pattern provide *efficient*, *widely-used* and flexible ways to implement parallel applications
- The availability of cost models allows to determine the goodness of a design (a-priori) and of an implementation (a-posteriori)
- the restriction of the degree of freedom makes easier to find a proper solution
- the clean separation between business code and parallel execution code grants benefits in terms of development times and performances
- a parallel programming framework can exploit the knowledge on the structure of the computation to optimize it

Thank you!

Questions?

For Further Reading I

M. Vanneschi.
High Performance Computing.
Pisa University Press, 2014.

M. McCool, J. Reinders, A. Robinson.
Structured Parallel Programming: Patterns for Efficient Computation.
Morgan Kauffman, 2012.

M. Danelutto.
Distributed System: Paradigms and Models.
Teaching material of the SPM course, September 2014.