Lecture 2-2: Parallel patterns

Advanced Programming for HPC

Professor Lilian Aveneau

1. Introduction

Parallel programming with abstraction of the machine

- Not optimal, but ... easy to read and therefore use!

- Several APIs, for example:
  - Cuda, Nvidia GPU only
  - OpenCL, any GPU ... not functional with Nvidia
    - SyCL, version 2.2
  - C++ extension in STD
  - C# extension, other languages
  - Distributed environments : MPI

1. Introduction

Quite a few patterns:

- MAP (or Transform) : one-to-one transformation (or 2-to-1)
- GATHER and SCATTER : permutation
- REDUCE : sum (in the broadest sense)
- SCAN : prefix sum (in the broadest sense)
- Segmented versions of REDUCE and SCAN
- PARTITION : partitionnement
- COMPACT : or filter
- SORT : sort!

2. Constant time patterns

Here, the simplest patterns of theoretical complexity $O(1)$

In practice:

- Brent's theorem gives $O\left(\frac{n}{p}\right)$
  - Ignoring the cost of memory access ...
- Maximum Efficiency (when p divisor of n)

2.1 MAP

- MAP (or Transform) is the simplest

$$b = \{f(a[1]), f(a[2]), \dots, f(a[n])\}$$

- Input
  - List (a vector) of n values of same type $T1$
  - Function from $T1$ to $T2$, e.g. using functor or lambda function
- Output
  - List (or vector) of n values of same type $T2$
  - The order of entry is preserved

2.1. Quick overview

Example

- Input
  - $a = \{1,2,3,4,5,6,7,8\}$
  - Function: (`x: int -> int = return x * x`)
- Expected output:
  - $b = \{1,4,9,16,25,36,49,64\}$

2.1.2 PRAM implementation

MAP pattern

```
FOR each PE i ∈ [1 … n] in parallel:
    output[i] ← Functor(input[i])
END FOR
```

- Function must not use other input values
- Complexity in $O(1)$

2.1.3 Implantation with p processors

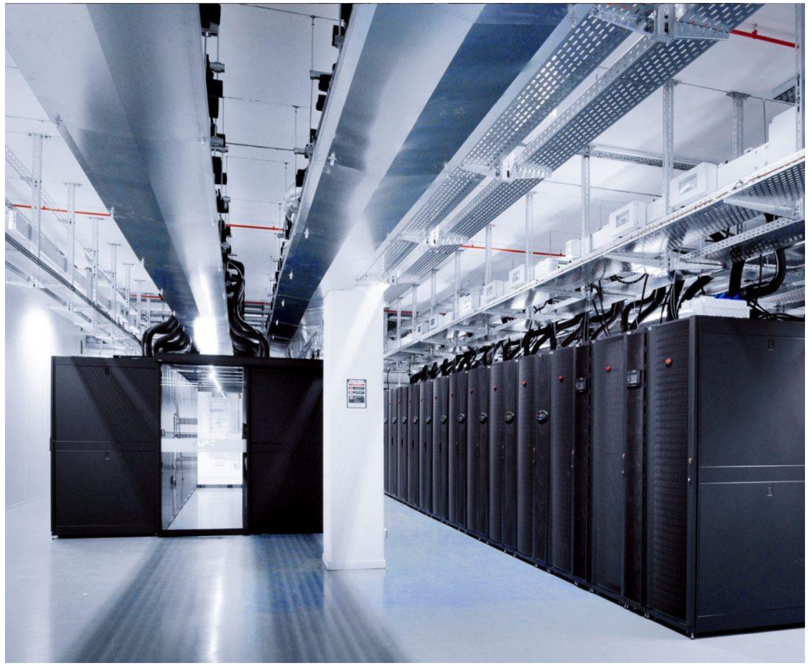
Handling multiple input/output per processors (loop) => First week

- Fixed block slicing
  - Constant time functor
  - Input sliced into continuous subarrays
- Fixed cutting by modulo
  - Increasing/decreasing time functor
  - Process by PE k values $k + p \times i$ starting from $i = 0$
  - Default: cache pollution
- On-demand strategy (dynamic)
  - Dynamic load balancing: FIFO, semaphore ... overhead!
  - Variable time functor and not monotonic

2.1.4 Hybrid machine

Different levels of parallelism

- Instruction vector by wire : MMX, AVX ...
- Intel Core i7 10850 H : 8 cores HT
- Bi-xeon Gold 6238 R : 2x28 cores HT
- Cluster, e.g. Jean Zay
- GPU Nvidia Turing TU 102 :
  - 72 SMP, containing 64 cores: 4608 cores
  - 72 RT cores
  - 576 Tensor cores
  - 288 texture units



2.2 Scatter

- Pattern performing permutation of elements from A to B
  - Not in place ! Sequential in place => $O(n \log n)$
- Permutation defined via array
  - So, a function
  - Example: $A = \{‘fr’, ’en’, ’vn’, ’es’\}$ and B : `string`, then is a function $f = \{‘fr’ \rightarrow « \, bonjour \, », ‘en’ \rightarrow « \, hello \, », ‘vn’ \rightarrow « \, xin \; chào \, », ‘es’ \rightarrow « \, holà \, »\}$
- More precisely, the principle of the SCATTER pattern is to "send" the elements of a source array to a new position in a destination array. This destination position is provided by a function, thus an array
- NB: permutation on A is bijection from A to A

2.2.1 Introduction

Scatter example

- Given input $\{1,2,3,4,5,6,7,8\}$,
- Destination $\{4,0,5,1,6,2,7,3\}$ (index starting at 0)
- Result is $\{2,4,6,8,1,3,5,7\}$

Remarks

- Function is a permutation of indexes
  - For n=8, $\{7,6,5,4,3,2,1,0\}$ is correct,
  - … but not $\{1,2,3,4,0,2,4,6\}$
- At last, it’s EREW !

2.2.2 PRAM implantation

SCATTER pattern

```
FOR each PE i ∈ [1 … n] in parallel:
    output[map[i]] ← input[i]
END FOR
```

Remarks

- Two inputs!
  - Data to permute: `input`
  - And the permutation function: `map`
- Constant complexity

2.2.3 Implantation with p processors

- MAP-like strategy
- However... no spatial coherence of writings ☹
  - Modulo strategy in reading
- On GPU, you have to take advantage of the spatial coherence!
  - Block strategy in reading
- It will be necessary to experiment ☺

2.3 Gather

Very similar to Scatter : permutation of values

- Semantics is reversed!

The principle of the **GATHER** pattern is to "**harvest**" elements from a source array to a new position in a destination array

- Both patterns are reversible ... by changing the permutation!
  - One is the reciprocal of the other

2.3.2 PRAM implantation

GATHER pattern

```
FOR each PE i ∈ [1 … n] in parallel:
    output[i] ← input[map[i]]
END FOR
```

- Note the position of the `map` function!

2.3.3 Implantation with p processors

- MAP-like strategy
- However... no spatial coherence of the readings ☹
  - Modulo strategy in writing
- On GPU, you must take advantage of the spatial coherence!
  - Strategy not block in writing
- It will be necessary to experiment ☺

3. REDUCE pattern

Or β-reduction

- Our first non-constant time parallel pattern
- Reduce set of values to single value
- Determinism: associative binary operation
- Works for all types
  - Integers, real numbers, etc.
  - Matrices
  - Strings of characters
  - Any structure

3.1. Introduction

Why is associativity required?

- Addition $1+2+3+4+5+6+7+8$
  - Produces $8 \times \left(8+1\right)/2$, so $4 \times 9 = 36$
  - Correct, whatever the order of the calculations

- Subtraction: 1-2-3-4-5-6-7-8

$$1-\sum_{i=2}^{8} i = 1-\left(\sum_{i=1}^{8} i - 1\right) = 1-\frac{8\times 9}{2}+1 = 2-36 = -34$$

  - Many associations give a wrong result, e.g.

$$(1-2)-(3-4)-(5-6)-(7-8) = -1\ -3\times(-1) = 2$$

$$(1-2-3-4)-(5-6-7-8) = -8+16 = 8$$

3.2 Implantation on PRAM machine

We have seen a first version in the previous chapter (operation = MAX)

Compute the maximum of the elements of an array in CREW

```
FOR each PE i ∈ [1 … n] in parallel:
    max[i] ← input[i]
END FOR

j ← 1
WHILE j < n:
    FOR each PE i ∈ [1 … n] in parallel:
        IF i+j ≤ n THEN
            max[i] ← MAX(max[i],max[i+j])
        END IF
    END FOR
    j ← j * 2
END WHILE

maximum ← max[1]
```

Poor efficiency!

3.2 Implantation using PRAM machine

- Reducing the number of processors means reducing the work
  - And therefore, increase efficiency!
  - Using exactly $n - 1$ operations and therefore processors?

- Array $\{1,2,3,4,5,6,7,8\}$
  - Compute $(1 + 2)$; $(3 + 4)$; $(5 + 6)$; $(7 + 8)$
  - Then $(3 + 7)$; $(11 + 15)$
  - And finaly $(10 + 26)$

  - Generalization for 2^k values (induction)

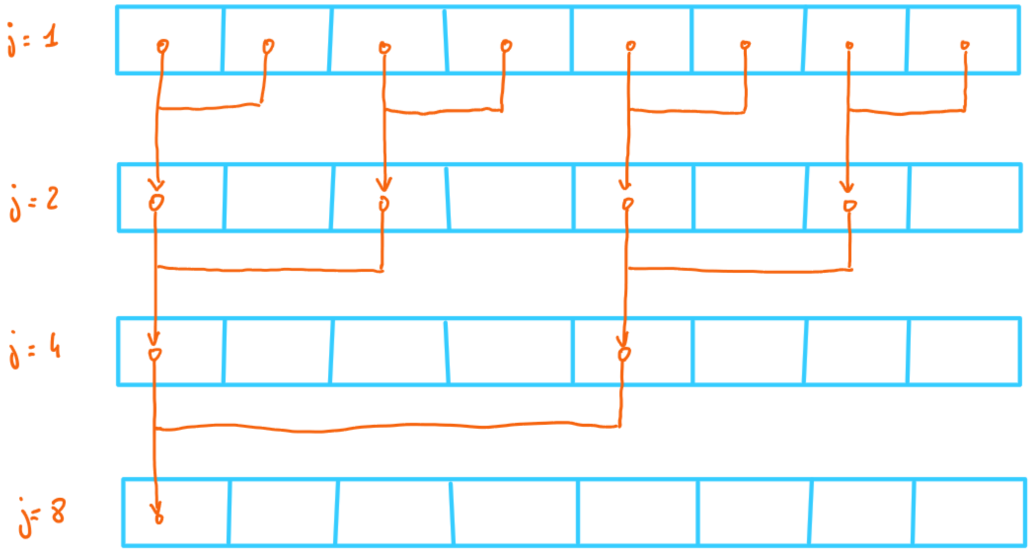
3.2 Implantation using PRAM machine

REDUCE pattern of array elements in CREW

```
FOR each PE i ∈ [1 … n] IN PARALLEL:
    aux[i] ← input[i]
END FOR
j ← 1
WHILE j< n:
    k ← j × 2
    FOR each PE i ∈ [1 … n] STEP k IN PARALLEL:
        IF i+j ≤ n THEN
            aux[i] ← Fun(aux[i],aux[i+j])
        END IF
    END FOR
    j ← k
END WHILE
RETURN aux[1]
```

3.2 Implantation using PRAM machine

- Note the word step
  - PE do not have consecutive numbers
  - But spaced k apart
  - At first $\frac{n}{2}$, then $\frac{n}{4}$, then $\frac{n}{8}$, etc.
  - So $n-1$ applications *in fine*



REDUCE pattern of array elements in CREW

```
FOR each PE i ∈ [1…n] IN PARALLEL:
    aux[i] ← input[i]
END FOR
j ← 1
WHILE j< n:
    k ← j ×2
    FOR each PE i ∈ [1…n] STEP k IN PARALLEL:
        IF i+j ≤ n THEN
            aux[i] ← Fun(aux[i],aux[i+j])
        END IF
    END FOR
    j ← k
END WHILE
RETURN aux[1]
```

3.3 Implantation using p processors

- Possible to use $n - 1$ reduction operations
- Complexity in $O(\frac{n}{p} \log n)$
- Work per processor
  - Sequential reduction of part of the data
  - Write to array of size p
  - Sequential end
  - ... or parallel with p processors!