Lecture 2-1: PRAM model

Advanced Programming for HPC

Professor Lilian Aveneau

1. Introduction

Theoretical model for discussing parallel algorithm complexity on shared memory machines, ideally vector machines

Definition: *In a shared memory parallel machine, all processors have direct read and write access to memory*

Definition: *A vector machine is a particular shared memory parallel machine with a single vector processor. In Flynn's taxonomy, this is the SIMD mode.*

1. Introduction

Definition: A vector processor is a processor with a single instruction decoder and several Processing Units (PUs or ALUs). Thus, the same instruction is carried out in parallel and synchronously by all the ALUs on the various associated registers (or on the same vector register)

Warning: Vector processor performs an instruction synchronously, so synchronization is not necessary. On the other hand, a machine with shared memory equipped with several processors (vector or not) will require synchronization mechanisms.

NB: a GPU usually works like a shared memory machine, equipped with several vector processors...

1.1. Definitions

- Need to introduce some important definitions, concept and keywords
- To learn!

1.1.1 Flynn’s Typology

- Two criteria
  - Number of instructions processed at the same time (in parallel)
  - Number of data processed by a single instruction
- The data?
  - We speak about “vector instructions”
- Examples with Intel processors
  - MMX: 64-bit processing (e.g., 2 integers)
  - SSE: 128 bits processing (e.g., 4 integers)
  - AVX: 256 bits processing (e.g., 8 integers)
  - AVX 512: 512 bits processing (e.g., 16 floats)

Beware of data loading!

1.1.1 Flynn’s typology

- These two criteria give 4 types of parallel processors :
  - SISD : *Single Instruction, Single Data*
  - SIMD : *Single Instructions, Multiple Data*
  - MISD : *Multiple Instruction, Single Data*
  - MIMD : *Multiple Instructions, Multiple Data*

- In practice
  - SISD: one logic processor, classical instructions
  - SIMD: one logic processor, vector instructions (MMX, SSE, AVX...)
  - MISD: several logic processors, classical instructions
  - MIMD: several logic processors, vector instructions

- Example Bi Xeon Gold 6238 R: 2 * 28 cores => 112 EP * 16 (AVX512)

1.1.2 Grain

- Main difficulty: expressing the parallelism
  - Need to consider the material!
  - Machine abstraction: theoretical model

- Define the tasks to be performed on different logical processors
  - Have enough for balancing
    - Example: 2 EP, one 1-ms task, and another 100-ms task…
  - But not too much to avoid critical sections

- Well, when possible!

- Conclusion : the wet finger rule

1.1.3 Parallel algorithm « Efficiency »

Let:

- P be a problem of size n ;
- $T_{seq(n)}$ be computation time of the best sequential algorithm realizing P
- Parallel algorithm realizing P in $T_{par(p,n)}$ with p EP (Elementary Processors)

- Speed-up $$S_q(n) = \frac{T_{seq}(n)}{T_{par}(p,n)}$$
- Work $$W_p(n) = p \cdot T_{par}(p,n)$$
- Efficiency $$e_p(n) = \frac{T_{seq}(n)}{p \cdot T_{par}(p,n)} = \frac{T_{seq}(n)}{W_p(n)}$$

1.1.4 Simulation

- Possible (recommended) to express parallel algorithm without considering the hardware ...
  - Number of tasks (grain) incorrect ?
  - However, it is possible to simulate this theoretical algo. on any machine ...
- That’s the Brent theorem

Assume a parallel computer where each processor can perform an arithmetic operation in unit time. Further, assume that the computer has exactly enough processors to exploit the maximum concurrency in an algorithm with N operations, such that T time steps suffice. Brent’s Theorem says that a similar computer with fewer processors, P, can perform the algorithm in time

$$T_P \leq T + \frac{N - T}{P}$$

Going further: Amdahl’s law, Gustafson’s law…

2. PRAM model

- It is usual to ask the question: "Why a parallel machine model?".
- On sequential machine it solves the problem of computability
- But this is not enough: algorithm can have too high complexity
  - e.g., "vertex cover problem"
- Expressing complexity requires ... an algorithm!
- and thus, a machine model, at least theoretical
- With "big-O" notation, linear constant of complexity is omitted
- So, the precise machine is not useful

2.1 Introduction

- Many sequential models coexist:
  - Turing machine, recursive functions, cellular automata, lambda-calculus ...
- Very few in parallel computing
  - The most known : PRAM model (Parallel RAM)

- Principle based on the following approximations:
  - Memory access costs nothing
  - Memory is infinite (in size)
  - Operations (ALU) cost 1
  - Number of ALUs is infinite

2.1 Introduction

- One single program per vector machine (SIMD)

- Tip: each ALU has a unique identifier
  - As for previous week, using `cuda.grid()`

- Big concurrency problem (shared memory, vector ...)
- Three concurrency models (read/write)
  - EREW (Exclusive Read, Exclusive Write)
  - CREW (Concurrent Read, Exclusive Write)
  - CRCW (Concurrent Read, Concurrent Write)

2.1 Introduction

How to solve concurrent write accesses?

- Arbitrary mode
  - Random decides among values to write
- Consistent mode
  - If processing units write in the same box at the same time, then it is necessarily the same value (for example true, or 0 ...).
- Merge mode
  - Written value results from the application of a function defined by the programmer
  - Typically : addition, min, max, etc.
  - Must be associative

2.2 First example

- Attention: all the PE are synchronized (vector machine)
  - This is a theoretical model ...

- First very simple example

Parity's calculation of the elements of an array

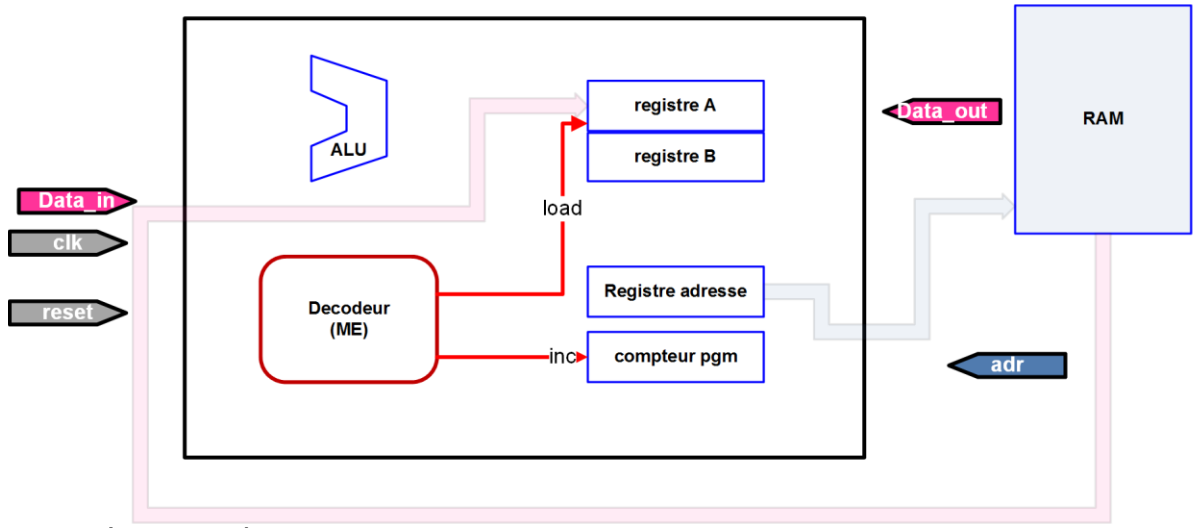
```
FOR each PE i ∈ [1 … n] in parallel:
    IF input[i] IS ODD THEN
        odd[i] ← 1
    ELSE
        odd[i] ← 0
    END IF
END FOR
```

Remove that word, and you have a sequential algorithm!

Complexity ?
O(1) !

2.2 First example

The conditional on a vector machine?



Source : N. Richard

It is very simple:

- All PEs realize both alternatives!
- Use a mask to finalize the writes in register (load)

In the end, each PE performs 3 instructions:

- The test
- First alternative
- Second alternative

2.3 Second example: maximum

Calculation of the maximum of the elements of an array in sequence

```
maximum ← input[1]
FOR each PE i $\in [2 \dots n]$:
    IF maximum < input[i] THEN
        maximum ← input[i]
    END IF
END FOR
```

Maximum of the elements of an array in CRCW, mode fusion

```
FOR each PE i $\in [1 \dots n]$ in parallel:
    maximum ← input[i]
END FOR
```

Fusion using « `max` »

2.3 Second example

- Still CRCW, consistent mode (multiple writing of the same value)

Compute the maximum of the elements of an array in CRCW, consistent mode

```
FOR each PE i ∈ $[1 \dots n]$ in parallel:
    isMax[i] ← true {First, each value is tagged as the maximum}
END FOR

FOR each PE (i,j) ∈ $[1 \dots n]^2$ in parallel:
    IF input[i] < input[j] THEN
        isMax[i] ← false {Found a bigger value!}
    END IF
END FOR

FOR each PE i ∈ $[1 \dots n]$ in parallel:
    IF isMax[i] THEN
        maximum ← input[i]
    END IF
END FOR
```

Complexity ?
O(1) !

2.3 Second example

- How many processors?
  - 1st loop: n
  - 2nd loop: n^2
  - 3rd loop: n

- Hence, n^2 comparisons are made…

- Cost: one single comparison
- Work: n^2
- Efficiency: $\frac{n}{n^2}$ so $\frac{1}{n}$ ☹ (think about Brent’s theorem)

2.4. From CRCW to EREW

Switching the maximum algorithm from CRCW to CREW

Compute the maximum of the elements of an array in CREW

```
FOR each PE i $\in [1 \dots n]$ in parallel:
    max[i] ← input[i]
END FOR

j ← 1
WHILE j $<$ n:
    FOR each PE i $\in [1 \dots n]$ 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]
```

Complexity ?
$O(\log_2 n)$

2.4. From CRCW to EREW

Example with `input={1,2,3,4,5,6,7,8}`

- `max={1,2,3,4,5,6,7,8}`
- `j=1`, `max` **receive** `{2,3,4,5,6,7,8,8}`
- `j=2`, `max` **receive** `{4,5,6,7,8,8,8,8}`
- `j=4` **and** `max` **receive** `{8,…,8}`

Example with `input={8,7,6,5,4,3,2,1}`

- …
- `j=4` **and** `max` **receive** `{8,7,6,5,4,3,2,1}`
- => maximum from beginning ;-)

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]
```

2.4. From CRCW to EREW

- EREW version: avoid concurrent reading (in MAX)

Compute the maximum of the elements of an array in EREW

```
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
            temp ← max[i+j]
            max[i] ← MAX(max[i], temp)
        END IF
    END FOR
    j ← j * 2
END WHILE
maximum ← max[1]
```

2.4. From CRCW to EREW

- Complexity increased to $O(\log n)$, but improved efficiency!
  - Work: $n \log n$
  - Efficiency: $\frac{n}{n \log n}$ so $\frac{1}{\log n}$
- Number of comparisons (maximum) tends to $nlog\ n$ (for n=2^m)
  - $j = 1$ it exists $n - 1$ comparisons
  - $j = 2$ it exists $n - 2$ comparisons
  - …
  - $j = \frac{n}{2}$ it exists $n - \frac{n}{2}$ comparisons

So $n \log n - \sum_0^{m-1} 2^k$

Or $nlog\ n - (2^m - 1)$

This tends to $nlog\ n$ when $n \rightarrow \infty$

2.4. From CRCW to EREW

- This result is generalized with the following theorem

Simulation theorem. *For a given problem, no CRCW algorithm working with p PE can be faster by $O(log p)$ than the fastest EREW algorithm working with p PE.*

- NB
    - This algorithm is generalized with the REDUCE pattern
    - The reverse is a diffusion, or BROADCAST

3. Pointer jumping

Method used to build PRAM algorithms

- Bijection between array of size n and linked list
- Notion used with iterator (C#, C++, ...)

Why linked list ?

- Cell contains one `element` (container), and the pointer `next`
- Easier to visualize algorithm with pointer, than $i+j$ ...

Basically, does not change anything / index

- Still in O(log) ...

3.1 Introduction

EREW calculation of the maximum of the elements of a list

```
FOR each PE i ∈ [1 … n] in parallel:
    value[i] ← cells[i].value
    next[i] ← cells[i].next
END FOR

FOR j ∈ [1 … ⌈log n⌉]: { WHILE ∃ PE j | next[j] ≠ NIL }
    FOR each PE i ∈ [1 … n] in parallel:
        IF next[i] ≠ NIL THEN
            aux[i] ← value[ next[i] ]
            value[i] ← MAX(value[i], aux[i])
            aux[i] ← next[i]
            aux[i] ← next[aux[i]]
            next[i] ← aux[i]
        END IF
    END FOR
END FOR
maximum ← value[1]
```

3.2 Distance of elements to the end of a list

CREW calculation of the distance from the elements of a list to the end of the list

```
FOR each PE i $\in [1 \dots n]$ in parallel:
    next[i] ← cells[i].next
    dist[i] ← 1
END FOR

FOR j $\in [1 \dots \lceil \log n \rceil]$:
    FOR each PE i $\in [1 \dots n]$ in parallel:
        IF next[i] ≠ NIL THEN
            aux[i] ← dist[next[i]]
            dist[i] ← dist[i] + aux[i]
            aux[i] ← next[i]
            aux[i] ← next[aux[i]]
            next[i] ← aux[i]
        END IF
    END FOR
END FOR
```

3.2 Distance of elements to the end of a list

The order of the elements does not matter: let's take the indices of the cells ...

- Initialisation

I	1	2	3	4	5	6	7	8
dist	1	1	1	1	1	1	1	1
next	2	3	4	5	6	7	8	NIL

- $j = 1$

I	1	2	3	4	5	6	7	8
dist	2	2	2	2	2	2	2	1
next	3	4	5	6	7	8	NIL	NIL

3.2 Distance of elements to the end of a list

- $j = 3$

I	1	2	3	4	5	6	7	8
dist	8	7	6	5	4	3	2	1
next	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Let's take another order (permutation)

- Initialy

I	1	2	3	4	5	6	7	8
dist	1	1	1	1	1	1	1	1
next	3	7	8	5	2	NIL	6	4

3.2 Distance of elements to the end of a list

- $j = 1$

l	1	2	3	4	5	6	7	8
dist	2	2	2	2	2	1	2	2
next	8	6	4	2	7	NIL	NIL	5

- $j = 2$

l	1	2	3	4	5	6	7	8
dist	4	3	4	4	4	1	2	4
next	5	NIL	2	6	NIL	NIL	NIL	7

- $j = 3$

l	1	2	3	4	5	6	7	8
dist	8	3	7	5	4	1	2	6
next	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

3.3 Rank of the elements of a list

CREW calculation of the rank of the elements of a list

```
FOR each PE i $\in [1 \ldots n]$ in parallel:
    next[i] ← cells[i].next
    rank[i] ← 1
END FOR

FOR j $\in [1 \ldots \lceil log\ n \rceil]$:
    FOR each PE i $\in [1 \ldots n]$ in parallel:
        IF next[i] ≠ NIL THEN
            aux[i] ← rank[i]
            rank[next[i]] ← rank[next[i]] + aux[i]
            aux[i] ← next[i]
            aux[i] ← next[aux[i]]
            next[i] ← aux[i]
        END IF
    END FOR
END FOR
```

3.3 Rank of the elements of a list

Example with

l	1	2	3	4	5	6	7	8
Rank	1	1	1	1	1	1	1	1
next	2	3	4	5	6	7	8	NIL

- $j = 1$

l	1	2	3	4	5	6	7	8
Rank	1	2	2	2	2	2	2	2
next	3	4	5	6	7	8	NIL	NIL

- $j = 2$

l	1	2	3	4	5	6	7	8
Rank	1	2	3	4	4	4	4	4
next	5	6	7	8	NIL	NIL	NIL	NIL

- $j = 3$

l	1	2	3	4	5	6	7	8
Rank	1	2	3	4	5	6	7	8
next	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

3.3 Rank of the elements of a list

Permutation:

I	1	2	3	4	5	6	7	8
Rank	1	1	1	1	1	1	1	1
next	3	7	8	5	2	NIL	6	4

- $j = 1$

I	1	2	3	4	5	6	7	8
Rank	1	2	2	2	2	2	2	2
next	8	6	4	2	7	NIL	NIL	5

- $j = 2$

I	1	2	3	4	5	6	7	8
Rank	1	4	2	4	4	4	4	3
next	5	NIL	2	6	NIL	NIL	NIL	7

- $j = 3$

I	1	2	3	4	5	6	7	8
Rank	1	6	2	4	5	8	7	3
next	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

4 Conclusion

- PRAM model is simple but powerful
- Complexity on vector machine (SIMD)
- In practice, multiprocessors, even separate memory!
  - Brent's theorem
- Passage from CRCW to EREW
  - Overhead limited to $\log n$