# Algorithmique et programmation parallèle

Frédéric VIVIEN

M1IF
2014-2015

# Contents

# Part I

# Theoretical models

# Chapter 1

# Sorting networks

The *simple comparator* is the basic element of a sorting network. It is modeled as a black box with two inputs and two outputs: $(a, b) \mapsto (\min(a, b), \max(a, b))$.

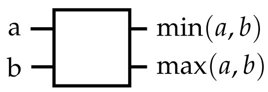



Figure 1.1: A simple comparator

We will build a network with these comparators, in order to sort efficiently a set.

## 1.1 Odd-Even merge sort

Divide and conquer paradigm.

### 1.1.1 Odd-Even merging network

**Definition 1.**

Let $\langle c_1, c_2, \ldots, c_n \rangle$ be some sequence. Then, $Sort(\langle c_1, \ldots, c_n \rangle)$ is the sorted sequence.

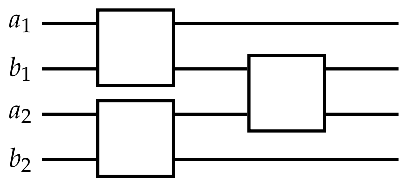
**Definition 2.**

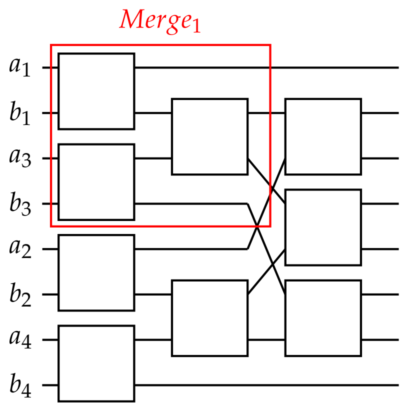
$Sorted(\langle c_1, \ldots, c_n \rangle)$ is true if and only if $c_1 \leqslant \ldots \leqslant c_n$.

**Definition 3** (The merge operation)**.**

If $Sorted(\langle a_1, \ldots, a_n \rangle) \wedge Sorted(\langle b_1, \ldots, b_n \rangle)$,
then $Merge(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_n \rangle) = Sort(\langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle)$.

$Merge_m$ is a sorting network for two sorted sequences of size $2^m$. $Merge_0$ is a simple comparator.

$Merge_1$: 


$Merge_2$: 


$Merge_m$ is a network built with two networks $Merge_{m-1}$ and a comparator column.

**Proposition 1.**

Let the two sorted sequences $A = \langle a_1, \ldots, a_{2n} \rangle$ et $B = \langle b_1, \ldots, b_{2n} \rangle$
$\langle d_1, \ldots, d_{2n} \rangle = Merge\,\langle a_1, a_3, \ldots, a_{2n-1} \rangle, \langle b_1, b_3, \ldots, b_{2n-1} \rangle$
$\langle e_1, \ldots, e_{2n} \rangle = Merge\,\langle a_2, a_4, \ldots, a_{2n} \rangle, \langle b_2, b_4, \ldots, b_{2n} \rangle$
We have $Sorted(d_1, \min(d_2, e_1), \max(d_2, e_1), \ldots, \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n})$

*Proof.* Without loss of generality, let us suppose that all elements are different. $d_1 = \min(a_1, b_1)$ is therefore the global minimum. $e_{2n} = \max(a_{2n}, b_{2n})$ is the global maximum.

General case: for $2 \leqslant i \leqslant 2_n$, $d_i$ and $e_{i-1}$ are at index $2i - 2$ or $2i - 1$ (by the proposition), we will show that this assertion is true by showing that:

(i) $d_i$ is greater than $2i - 3$ elements

(ii) $e_{i-1}$ is greater than $2i - 3$ elements

(iii) $d_i$ is lower than $4n - 2i + 1$ elements.

(iv) $e_{i-1}$ is lower than $4n - 2i + 1$ elements.

**Proof of (i):** Without loss of generality, let us suppose that $d_i$ is an element of $A$. Let $k$ be the number of $A$ elements in the sequence $\langle d_1, \ldots, d_i \rangle$.

This sequence contains $i - k$ elements from $B$: $d_i = a_{2k-1}$.

$A$ is sorted, therefore $d_i$ (i.e. $a_{2k-1}$) is greater than $2k - 2$ elements from $A$. The greatest elements of $B$ in $\langle d_1, \ldots, d_i \rangle$ is $b_{2(i-k)-1}$.

Thus, $d_i$ is greater than $2(i - k) - 1$ elements of $B$.

Therefore, $d_i$ is greater than $(2k - 2) + (2(i - k) - 1) = 2i - 3$ elements.

**Proof of (ii):** identical.

**Proof of (iv):** Without loss of generality, let us suppose that $e_{i-1}$ belongs to $B$. Let $k$ be the number of $B$ elements belonging to $\langle e_1, \ldots, e_{i-1}\rangle$.

$e_{i-1} = b_{2k}$

Then, $e_{i-1}$ is lower than $b_{2k+2}, b_{2k+4} \ldots$ thus by $2n - 2k$ elements from $B$.

$\langle e_1, e_2, \ldots, e_{i-1}\rangle$ contains $(i-1) - k$ elements of $A$.

The greatest one is $a_{2(i-1)-k}$.

Thus $e_{i-1}$ is lower than $2n - [2i - 2k] + 1$ elements of $A$.

Therefore $e_{i-1}$ is lower than $(2n - 2k) + (2n - 2i + 2k + 1) = 4n - 2i + 1$ elements.

**Proof of (iii):** identical. $\square$

**Lemma 2.**

Computation time $t_m$ of $Merge_m$ (merging of two sequences of $2^m$ elements).
Number of comparators: $p_m$
$t_m$ = maximal number of comparators taken by a single value.
$t_m = t_{m-1} + 1$ with $t_0 = 1$. Thus $t_m = m + 1$. Therefore, as $n = 2^m$ we have $t_n = \mathcal{O}(\log n)$.
$p_m = 2p_{m-1} + (m-1)$ with $P_0 = 1$. Thus $p_m = 2^m m + 1$. Therefore $p_m = \mathcal{O}(n \log n)$.

**Definition 4.**

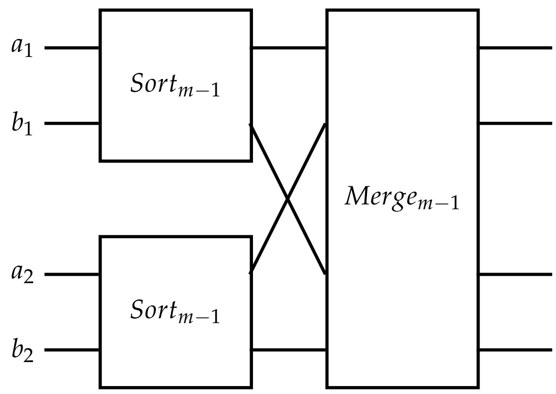
The *work* characterize the efficiency. It is the product of the number of actors by the computation time.

**Lemma 3.**

Parallel network: $w_n = t_n p_n = \mathcal{O}\left(n \log^2 n\right)$.
Sequential network: $w_n = \mathcal{O}(n)$.

### 1.1.2 Sorting network

We want to build a sorting network $Sort_m$ for $2^m$ elements. We build it using two networks $Sort_{m-1}$ and one network $Merge_{m-1}$.



**Lemma 4.**

Computation time $t'_m$:
$t'_1 = 1$, $t'_m = t'_{m-1} + t_{m-1}$.
$t'_m = \mathcal{O}\left(m^2\right)$
Number of comparators $p'_m$:
$p_1 = 1$, $p'_m = 2p'_{m-1} + p_{m-1} + 1$
$p'_m = \mathcal{O}\left(2^m m^2\right)$
$n = 2^m$.
Computation time: $\mathcal{O}\left(\log^2 n\right)$, number of comparators: $\mathcal{O}\left(n \log^2 n\right)$.
Work: $\mathcal{O}\left(n \log^4 n\right)$.
Sequentially:
Time: $\mathcal{O}\left(n \log n\right)$.
Work $\mathcal{O}\left(n \log n\right)$.

## 1.2 0-1 principle

**Proposition 5.**

A network is a sorting network for an arbitrary sequence if and only if it is a sorting network for 0-1 sequences (sequences containing only 0's and 1's).

*Proof.* Direct implication is trivial.

Other implication, by contraposition:

Let us suppose that there exists some sequence $x = \langle x_1, \ldots, x_n \rangle$ which is not sorted by the network $R$. Then, there exists an index $k$ such that $R(x)_k > R(x)_{k+1}$

Let $f$ be a non-decreasing function.

A comparator behaves in the same way on the input $\langle y_1, y_2 \rangle$ and on the input $\langle f(y_1), f(y_2) \rangle$.

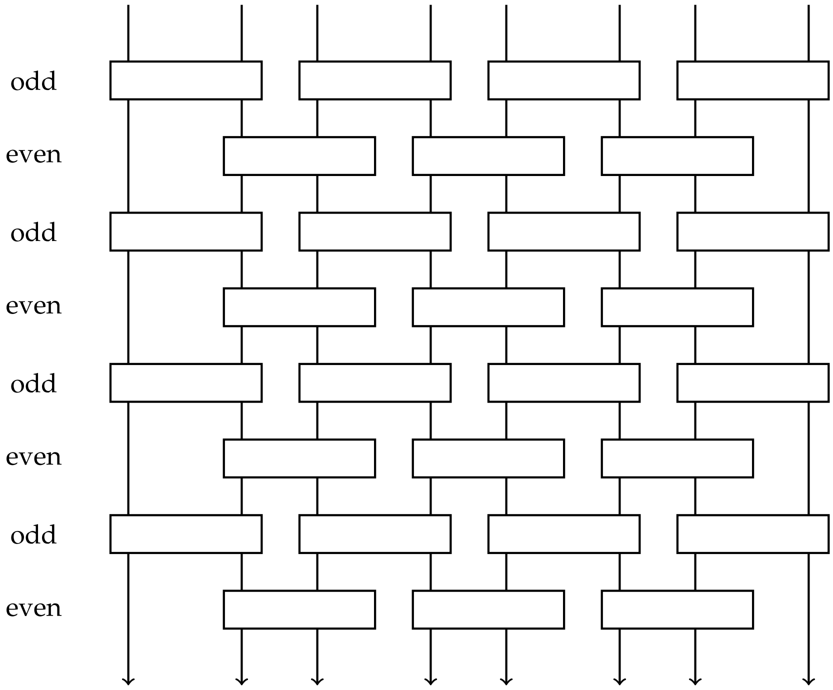
Let us define a non-decreasing function $f$: $f(y) = \begin{cases} 0 & \text{if } y < R(x)_k \\ 1 & \text{otherwise} \end{cases}$

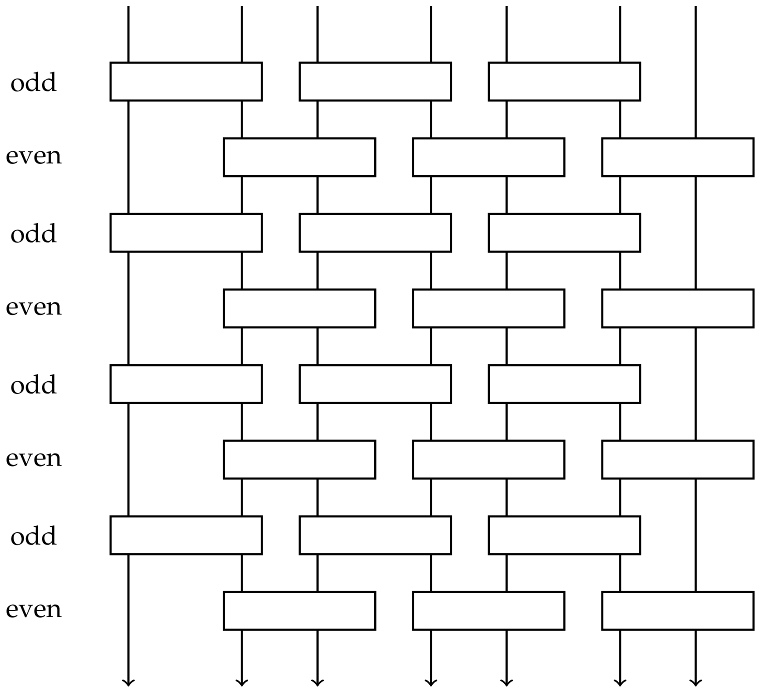
Since $f(R(x)_k) = 1$ and $f(R(x)_{k+1}) = 0$, then $\langle f(x_1), \ldots, f(x_n) \rangle$ is a 0-1 sequence which is not sorted by $R$. □

**Vocabulary: English vs French** Positive $\equiv$ *strictement* positif ($> 0$)
Non-negative $\equiv$ positif ($\geqslant 0$)
Negative $\equiv$ *strictement* negatif ($< 0$).
Non-positive $\equiv$ negatif ($\leqslant 0$)
Pour une fonction $f$ et $x < y$
Increasing $\equiv$ *strictement* croissant ($f(x) < f(y)$)
Non-decreasing $\equiv$ croissant ($f(x) \leqslant f(y)$)

## 1.3 Odd-even *transposition* sort

We still use the comparator.




$n$ inputs, $n$ comparator lines and $\frac{n(n-1)}{2}$ comparators.

**Proposition 6.**

The transposition network odd-even is a sorting network

*Proof.* We use the 0-1 principle. Let $\langle a_1, .., a_n \rangle$ be a $0-1$ sequence. Let $j_0$ be the index of the rightmost 1 in $\langle a_1, .., a_n \rangle$.

We have: $a_{j_0} = 1$ and $a_j = 0$ for $j > j_0$.

If $j_0$ is even, then this 1 does not move at the first step.
If $j_0$ is odd, then it moves by one position on the right at the first step.

Thus, after the first step, this 1 is at least at position 2.

Therefore, it moves gradually from left to right, ending at position $n$, since there is $n-2+1 = n-1$ steps to reach the final destination.

Let $j_1$ be the index of the second rightmost 1 of the input. From step 3, this 1 will move by one position on the right at each step (since $a_{j_0}$ moves by one position on the right at each step, from step 2). This 1 will reach the $(n-1)$th position in at most $n-1$ steps.

The $m$th 1 take $n-m$ steps to reach its position $(n-m+1)$. □

**Fact 7.**

The computation time of this network is $t_n = n$, its work is $\mathcal{O}\left(n^3\right)$.

## 1.4 Odd-even sorting on one dimension input

**Idea** We will use a line made of generic processors, to mimic the behaviour of the transposition networks.

We have $n$ inputs, $p$ processors, with $n \gg p$. For the sake of simplicity, we will suppose that $p|n$.

**Algorithm 1:** One line sort

Assign to each processor $\frac{n}{p}$ values
Each processor sorts its values
**for** *p times* **do**
- Each processor share its values with one of its two neighbours (one time the left one, one time the right one...we alternate).
- Both of the two processors merge the two sorted lists.
- The left processor keep the $\frac{n}{p}$ lowest values, the right processor keep the $\frac{n}{p}$ highest values.

**Fact 8.**

Time complexity: $\mathcal{O}\left(\frac{n}{p}\log\frac{n}{p} + p \times 2 \times \frac{n}{p}\right) = \mathcal{O}\left(n + \frac{n}{p}\log\frac{n}{p}\right)$
Work: $\mathcal{O}\left(p(n + \frac{n}{p}\log n\right) = \mathcal{O}\left(np + n\log n\right)$
If $p \leqslant \log n$ then work = $\mathcal{O}\left(n\log n\right) =$ is optimal (because a sorting algorithm by comparison requires $\mathcal{O}\left(n\log n\right)$ comparisons).

# Chapter 2

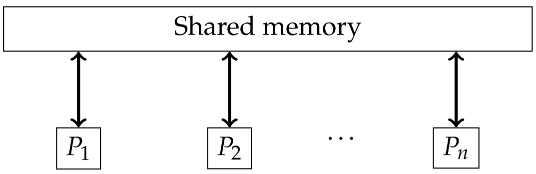
# PRAM : Parallel Random Access Machine

**Definition 5** (PRAM)**.**

Parallel Random Access Machine

**Remark 1.**

The $M$ of PRAM does not mean *memory* (mistake made in the original course).




PRAM is a central shared memory, that a set of processors (PUs, for processing units) can use.

All PUs run synchronously the same algorithm. They can work on different parts of the memory.

- The memory is infinite.
- The number of PUs is infinite.

**Eventual problem** several processors can try to use a same memory unit at the same time. There exists three different models for this problem.

**CREW** concurrent read, exclusive write. No limit for the number of PUs reading simultaneously a same memory unit. At most one PU can write on a single memory unit in one time unit.
Nothing is specified concerning a simultaneous read and write on a same memory unit. The behaviour of PRAM is undefined. We will try to avoid these situations.

At first, each processor will evaluate the condition. Then, all processors for which the condition holds will run the `then` branch. Finally, all processors for which the condition does not hold will run the `else` branch.

**Algorithm 2:** Synchronous run of the same algorithm

Array A
Processors $P_i$ working on $A[i]$
**forall the** $i \in parallel$ **do**
  **if** $A[i] > 0$ **then**
    $A[i] \leftarrow A[i] \times 2$
  **else**
    $A[i] \leftarrow -A[i]$

**EREW** Exclusive Read, Exclusive Write. At most one processor per memory unit and time unit can read and write.

**CRCW** Concurrent Read, Concurrent Write. Several processors can read silmutaneously a same data unit. Several processors can write silmutaneously a same data unit.

**Consistent mode** All PUs wishing to write in a given data unit **must** write the same value.

**Arbitrary mode** One value among all available ones is choosen randomly.

**Priority mode** The choosen value is those of the highest priority PU (e.g. the lowest index PU).

**Fusion Mode** An associative and commutative operation (eg. max, sum, and, etc.) is applied on the fly to all values. The result is written in memory.

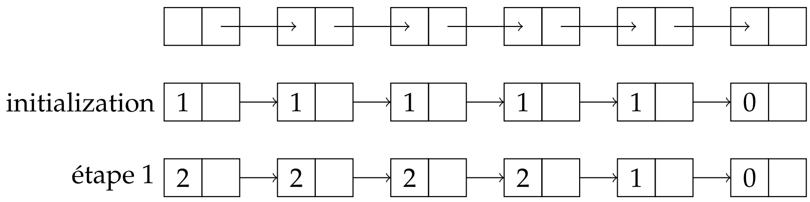
## 2.1 Pointer jumping

### 2.1.1 Linked list

We consider a linked list $L$ containing $n$ elements. For each element $i$, we would like to compute $d[i]$, its distance to the queue of the list.

$$d[i] = \begin{cases} 0 & \text{if } next[i] = \textsc{Nil} \\ 1 + d[next[i]] & \text{otherwise} \end{cases}$$

The sequential cost is linear (for simply linked lists, a first go through the list to count the number of elements, a second go to compute all distances... even easier for doubly linked lists).




**Algorithm 3:** Rank computation

**forall the** *i in parallel* **do**
  **if** *next[i] = Nil* **then**
    d[i]=0
  **else**
    d[i]=1

**while** *there exists a node i s.t. next[i]* $\neq$ *Nil* **do**
  **forall the** *i in parallel* **do**
    **if** *next[i]* $\neq$ *Nil* **then**
      d[i] $\leftarrow$ d[i] + d[next[i]];
      next[i] $\leftarrow$ next[next[i]];

We use $n$ PUs. Each processor work on a single memory unit. Processors are indexed from 1 to $n$.

#### Condition evaluation

**First idea** We keep the number of elements such that $next[i] = \text{NIL}$. It can work for PRAM CRCW model, with fusion mode (addition).

Other idea to transform the **While** loop:
**forall the** *i* **do**
  **while** *next[i]* $\neq$ *NIL* **do**
    ...

The loop is executed $\lceil \log_2 n \rceil$ times.

**Algorithm 4:** Loop conversion

**for** *s=1 to* $\lceil \log_2 n \rceil$ **do**
  **for** *i in parallel* **do**
    **if** *next[i]* $\neq$ *Nil* **then**
      ...

The list head is $h$.

**while** *next[h]* $\neq$ *Nil* **do**
  `/* Works on a EREW PRAM model */`
  **forall the** *i in parallel* **do**
    head[i] $\leftarrow \top$
  **forall the** *i in parallel* **do**
    **if** *next[i]*$\neq$ *Nil* **then**
      head[next[i]] $\leftarrow \perp$
  **forall the** *i in parallel* **do**
    **if** *head[i]* = $\top$ **then**
      h $\leftarrow$ i

**Transformation** $d[i] \leftarrow d[i] + d[next[i]]$
by
$temp[i] \leftarrow d[next[i]]$
$d[i] \leftarrow d[i] + temp[i]$
In order to run this algorithm on a PRAM EREW model.

**Complexity** List of size $n$, $n$ processors. Computation time: $\mathcal{O}(\log_2 n)$

### 2.1.2 Prefix computation

We consider a sequence:$x_1, x_2, \ldots, x_n$, stored as a simply linked list.
A binary associative operator $\otimes$

We wish to compute the sequence $y_1, \ldots, y_n$:

$$\begin{cases} y_1 & = 1 \\ y_k & = y_{k-1} \otimes x_k \\ & = x_1 \otimes x_2 \otimes \cdots \otimes x_k \end{cases}$$

---

**Algorithm 5:** Prefix computation

---

**Data**: list l
**forall the** *i in parallel* **do**
  y[i] ← x[i]
**while** ∃*i such that next[i]* ≠ *Nil* **do**
  **forall the** *i in parallel* **do**
    **if** *next[i]* ≠ *Nil* **then**
      y[next[i]] ← y[i] ⊗ y[next[i]]
      next[i] ← next[next[i]]

---

This algorithm runs in $\lceil \log_2 n \rceil$ steps, and with the very same transformations as for the rank computation, we can obtain a EREW algorithm.

## 2.2 Performances evaluation for PRAM algorithms

### 2.2.1 Cost, work, speedup and efficiency

Let $P$ be some problem of size $n$.

Let $T_S(n)$ be the computation time of the best (known) sequential algorithm.

Let $T_P(p, n)$ be the PRAM computation time, using $p$ processors.

**Cost** $C_p(n) = p.T_P(p, n)$

**Work** Sum of all processors computation time.

**Remark 2.**

$W_p(n) \leqslant C_p(n)$

**Speedup** $S_p(n) = \frac{T_S(n)}{T_P(p,n)}$

**Efficiency** $e_p(n) = \frac{S_p(n)}{p} = \frac{T_S(n)}{p \cdot T_P(p,n)} = \frac{T_S(n)}{C_p(n)}$

### 2.2.2 A simple result of simulation

**Proposition 9.**

Let $A$ be an algorithm which has an execution time of $t$ on a PRAM using $p$ processors. $A$ can be simulated on a PRAM of the same kind with $p' \leqslant p$ PUs in time $\mathcal{O}\left(\frac{tp}{p'}\right)$. The cost of the algorithm on the smallest PRAM is, at most, two times the cost on the biggest PRAM.

*Proof.* Each step of $A$ is simulated in $\left\lceil \frac{p}{p'} \right\rceil$ steps with $p'$ PUs.

$$\begin{aligned}
C_{p'}(n) &= p' T_P(p', m) \\
&\leqslant p' \left\lceil \frac{p}{p'} \right\rceil T_P(p,n) \\
&\leqslant p' \left( \frac{p}{p'} + 1 \right) T_P(p,n) \\
&\leqslant (p + p') T_P(p,n) \\
&\leqslant 2p T_P(p,n) = 2C_p(n)
\end{aligned}$$

□

**Prefix calculation** $n$ elements, $n$ PUs and a complexity of $\mathcal{O}(\log(n))$
According to the simulation theorem, with $p$ PUs ($p \leqslant n$), we can simulate the computation in time $\mathcal{O}\left(\frac{n}{p}\log n\right)$.

**A smarter solution** We have $\frac{n}{p}$ elements for each PU.

1st phase prefix calculus on the set of $\frac{p}{n}$ elements of the PU. Done in $\mathcal{O}\left(\frac{n}{p}\right)$

2nd phase pointer-jumping on the prefix calculus on $p$ processors with $p$ values which have the entire prefix of the $\frac{n}{p}$ value of process.
(1th processor $x_1, \ldots, x_{n/p} \to x_1 \otimes \cdots \otimes x_{n/p}$)
Computation in $\mathcal{O}(\log p)$

3rd phase Combination of local prefixes and of the new ones computed with the pointer-jumping in $\mathcal{O}\left(\frac{n}{p}\right)$

Total complexity: $\mathcal{O}\left(\frac{n}{p} + \log p\right)$

### 2.2.3 BRENT's theorem

**Theorem 10.**

Let $A$ be an algorithm which computes a total number of $m$ operations and which halts in time $t$ on a PRAM (with a fixed number of PUs).
$A$ can be simulated in time $\mathcal{O}\left(\frac{m}{p}+t\right)$ on a PRAM of the same kind with $p$ PUs.

*Proof.* $m[i]$: the number of operations computed by $A$ at the step $i$.

The step $i$ on $p$ processors can be computed in time $\left\lceil\frac{m[i]}{p}\right\rceil$.

The running time of $A$ for $p$ processors:

$$\begin{aligned}\sum_{i=1}^{t}\left\lceil\frac{m[i]}{p}\right\rceil &= \sum_{i=1}^{t}\left(\frac{m[i]}{p}+1\right)\\ &= \sum_{i=1}^{t}\frac{m[i]}{p}+\sum_{i=1}^{t}1\\ &= \frac{m}{p}+t\end{aligned}$$

□

**Example 1.**

EREW PRAM: maximum of a list of $n$ integers. Assume we have $n$ PUs, the time complexity is $\mathcal{O}(\log n)$ (prefix computation or reduction time).
$p$ processors, complexity ?
Number of operations: $m=n-1$, $t=\log n$.
According to the BRENT's theorem: $\mathcal{O}\left(\frac{n}{p}+\log n\right)$
If $p=\frac{n}{\log n}$, running time: $\mathcal{O}(\log n)$, which is the same complexity, but with less PUs!

**Remark 3.**

$S_p(n)=\frac{T_S(n)}{T_P(p,n)}\geqslant 1$
$e_{p(n)}\in[0,1]$
$p\geqslant S_p(n)\geqslant 1$

## 2.3 Comparisons of PRAM models

### 2.3.1 Model separation

#### 2.3.1.1 CRCW vs. CREW

We're looking for a problem separating CRCW and CREW, ie that can be solved more efficiently when allowing concurrent write. The following problem has this property:

**Maximum of $n$ integers**

**CRCW**

**Algorithm 6:** Maximum of $n$ integers in a constant time using $n(n-1)$ PUs

**forall the** *i in parallel* **do**
  isMaximum[i] $\leftarrow \top$
**forall the** *i,j, $i \neq j$, in parallel* **do**
  **if** *A[i] < A[j]* **then**
    isMaximum[i] $\leftarrow \perp$
**forall the** *i in parallel* **do**
  **if** *isMaximum[i]* **then**
    maximum $\leftarrow$ i

This algorithm runs in constant time.

**CREW** Complexity: $\Omega(\log n)$.

This complexity cannot be improved, since at each timestep, a CREW algorithm can merge at most a constant amount of variables into one.

#### 2.3.1.2 CREW vs. EREW

A separating problem between CREW and EREW is as follow:

Given a set $S$ of values and an object $e$: does $e$ belong to $S$? (We assume that each element of $S$ is unique).

**CREW** Found $\leftarrow \perp$
**forall the** *i in parallel* **do**
  **if** *S[i]=e* **then**
    Found $= \top$

There are parallel reads because each processor accesses $e$ simultaneously. Since each element is unique, we ensure that only one processor will write in the variable Found, thus the algorithm is Exclusive Write.

This algorithm runs in constant time.

**EREW** At most, we can duplicate the numbers of copy of $e$ at each step to allow processors to use it. The number of processors aware of the value of $e$ double each time, so we need $\mathcal{O}(\log n)$ step to ensure that each and every processor has received it.

In the worst case, $e$ is not in $S$, and in this case we have to wait that each processor knows the value of $e$ to conclude $\rightarrow \Omega(\log n)$.

### 2.3.2 Simulation theorem

**Theorem 11.**

Any algorithm on a CRCW with $p$ PUs has an running time of, at most, $\mathcal{O}(\log p)$ times lower than the running time of an algorithm for EREW with $p$ PUs to solve the same problem.

*Proof.* We assume that the CRCW PRAM use the consistent mode (only the concurrent writing of the same value are allowed).

The method to simulate a concurrent write with exclusive read (symmetrically for reading):

We consider a step of the CRCW algorithm.

We use a temporary vector $A$ of size $p$.

When the PU $P_i$ write $x_i$ in memory at the position $l_i$ in CREW, $P_i$ write $(l_i, x_i)$ in $A[i]$.

We sort $A$ in $\log p$ (we assume that we can sort $p$ values in time $\log p$ using $p$ PUs).

Each $PU$: $P_i$ reads the two cells:
$A[i] = (l_j, x_j)$ and $A[i-1] = (l_k, x_k)$
**if** $l_j = l_k$ **then**
  $p_i$ does nothing ($x_i = x_k$)

**else**
  $p_i$ writes $x_j$ at the position $l_j$

□

## 2.4 COLE's sort machine

**1986** Algorithm CREW PRAM to sort $n$ values in time $\mathcal{O}(\log n)$ with $\mathcal{O}(n)$ processors. The cost (and the work) of this algorithm is in $\mathcal{O}(n \log n)$, which is optimal.
(A EREW version with the same constraint was proposed by COLE in 1988.)

A merge sort on a binary tree with $n$ values and of height $\log n$ is done in time $\log n$, the time to handle a tree level is $\mathcal{O}(n)$. We have to be able to merge two sorted lists of arbitrary size ($n/2$ at best) in constant time.

The underlying characteristics:

- computation pipelining
- pre-computation

### 2.4.1 The merge operation

If $J$ and $K$ are two sorted lists, $J|K$ is the result of the fusion.

The rank of the element $x$ in the sequence $J$ is the number of elements in $J$ that are smaller than $x$.

$rank(x, J) = card\{y \in J | y < x\}$

The cross-rank of $A$ in $B$:

$$\begin{aligned} R[A,B] : A &\to \mathbb{N} \\ e &\mapsto rank(e,B) \end{aligned}$$

**Definition 6** (Good sampler (GS)).

A sequence $L$ is called a good sampler of a sequence $J$ if, for any $k \geqslant 1$, there are at most $2k+1$ elements of $J$ between $k+1$ (arbitrary) consecutive elements of $\{-\infty\} \cup L \cup \{+\infty\}$

**Example 2.**

For $k = 1$, between 2 values of the GS, there are at most 3 values of $J$.

**Example 3.**

Using a good sampler, two sorted lists can be merged quickly.
Let

$$J = [2,3,7,8,10,14,17,18,21]$$
$$K = [1,4,6,9,11,12,13,16,19,20]$$
$$L = [5,10,12,17]$$

$L$ is a GS of $J$ and $K$. We can check it exhaustively:

$$k = 1 : (-\infty, 5); (5, 10); (10, 12); (12, 17); (17, \infty)$$
$$k = 2 : (-\infty, 10); (4, 12); (10, 17); (12, \infty)$$

The sets $odd(J)$ and $even(J)$ are good samplers of $J$ ($odd(J)$ set of the elements of odd rank in $J$).

Deux listes triées $J$ et $K$ et un GS $L$ de $J$ et de $K$.

$\forall i, |J(i)| \leqslant J$ et $|K(j)| < 3$
On peut fusionner $J(i)$ et $K(i)$ en temps constant.

---

**Algorithm 7:** MergeWithHelp(J,K,L)

---

J et K sont partitionnés en $|L| + 1$ sous ensembles $J(i) = \{j \in J | l_{i-1} < j \leqslant l_i\}$ et $K(i) = \{k \in K, l_{i-1} < k \leqslant l_i\}$
**forall the** *i in parallel* **do**
 $res_i \leftarrow$ Merge(J(i),K(i))
$J|K \leftarrow res_1 res_2 \ldots res_{|i|+1}$

---

**Lemma 12.**

If $L$ is a GS of the sorted list $J$ et $K$ and if the ranks $R[L, J], R[L, K], R[J, L] et R[K, L]$ are known, then MergeWithHelp is running in a constant time with $|J| + |K|$ PUs on a CREW PRAM.

*Proof.* 1) We need $J$ PUs. Each $P_j$ read $j \in J$, $rank(j, L)$ and insert $j$ in $J(rank(j, l)$ (there are at most 3 PUs which write at the same time in the same array $j$, writing can be sequential).

2) We merge sequentially at most 2 sorted list containing 3 elements $\Rightarrow \mathcal{O}(1)$ and $|L|$ PUs.

3) We use $|J| + |K|$ PUs:
$rank(l, J|K) = rank(l, J) + rank(l, K)$

The value $l$ which is the $m^{\text{th}}$ element of $J(i)|K(i)$ must be stocked at the place $(rank(l_{i-1}|J|K) + m$ □

**Algorithm 8:** COLEMERGE

We receive $X(t+1)$ et $Y(t+1)$ respectively from the left and right child
**merge** de $val(t+1) \leftarrow$ MERGEWITHHELP$(X(t+1), Y(t+1), Val(t))$
**reduction** we send $Z(t+i) \leftarrow Reduce(Val(t+1)auparent)$
$(Reduce(z_1, z_2, \dots, z_p) = \{z_4, z_8, z_{12}, \dots\}$ (every fourth value)).

### 2.4.2 Sorting trees

We have $n = 2^m$ elements.

A node is complete when it has received all inputs.

A node at level $k$ has $2^k$ inputs (a leaf is at level 0).

As soon as a node begin to receive data, its data quantity is doubled at each step (and for $2^k$ step).

**if** *if a node is complete as step t* **then**
  At step $t+1$, every fourth element of $val(t+1)$ is sent to its father.
  At step $t+2$, every other element of $val(t+1)$ is sent to its father.
  At step $t+3$ every element is sent to its father.
  At step $t+4$ and later no elements are sent.

Proof of correctness of COLE algorithm. is based on 3 invariants.

- The size of the inputof a node at level $k$ doubles at each step from 1 to $2^k$ (from the time it receives its first input).
- If $X(t)$ is a good sampler of $X(t+1)$ and $Y(t)$ is a good sampler of $Y(t+1)$ then $Z(t)$ is a good sampler of $Z(t+1)$ $(Z(t) = Reduce(X(t), Y(t)))$.
- We can compute $R[S(t+1), S(t)]$ for any sequence of input or output of a node.

### 2.4.3 Complexity and correctness

**Lemma 13.**

The pipelined sorting tee algorithm runs in time $\mathcal{O}(\log n)$ with $\mathcal{O}(n)$ PUs.

*Proof.* **Execution time**: A level $k$ node is complete at step $t = 3k$ (by induction) that leads to a complexity in $O(\log n)$.

**Number of processing units**: At level $k$ there are $\frac{n}{2^k}$ nodes. They all produce sorted list of size $\leqslant 2^k$ for which they need at most $2^k$ PUs.

So, $\frac{n}{2^k}$ nodes executing at most $2^k$ PUs. There is at most a total $O(n)$ PUs.

At time $t = 3k$ level $k$ nodes handle input of size $2^{k-1}$ and use each $2^k$ PUs. Hence a total of $n$ PUs.

Nodes at level $< k$ have stopped everything and use no PUs.

Nodes at level $k+1$: there are $\frac{n}{2^{k+1}}$ nodes, they handle input of size $2^{k-2}$ and use each $2^{k-1}$ PUs. So at most $\frac{n}{2^{k+1}} 2^{k-1} = \frac{n}{4}$ PUs.

Nodes at level $k+2$ use $\frac{n}{16}$ PUs.

Hence, the total number of PUs used at time $3k$ is $n + \frac{n}{4} + \frac{n}{16} + \cdots = n\frac{1}{1-\frac{1}{4}} = \frac{4}{3}n$. □

**Lemma 14.**

Lets$X$, $X'$, $Y$, $Y'$ 4 sorted sequences.
Si $X$ is a GS of $X'$, $Y$ a GS of $Y'$ then $Reduce(X|Y)$ is a GS of $Reduce(X'|Y')$.
$X|Y$ is not (necessary) a GS of $X'|Y'$.

**Example 4.**

$X = [2,7]$ $X' = [2,5,6,7]$
$Y = [1,8]$ $Y' = [1,3,4,8]$
$X|Y = [1,2,7,8]$ $X'|Y' = [1,2,3,4,5,6,7,8]$
$[2,7]$ is a set of 2 consecutive elements of $X|Y$.
There are 5 consecutive elements of $X'|Y'$ between two consecutive elements of $X|Y$ when the maximum number authorized is 3.

*Proof.* We have counter-example for $X|Y$ as GS of $X'|Y'$.

**Lemma 15** (Intermediate result)**.**

The are at most $2r + 2$ elements of $X'|Y'$ between $r$ consecutive elements of $X|Y$ (we assume $-\infty$ and $+\infty$ belong to $X|Y$).

*Proof.* Let us consider a sequence of $r$ consecutive elements of $X|Y$: $e_1, \ldots, e_r$.

Let $h_X$ (resp. $h_Y$) the number of elements of $X$ (resp. $Y$) in $e1, \ldots, e_r$. We have $h_X + h_Y = r$. Without lost of generality, we assume that $e_1$ belongs to $X$.

2 cases:

$e_r \in X$: As $X$ is a good sample of $X'$ there are at most $2(h_X - 1) + 1$ consecutive elements of $X'$ between $(h_X - 1) + 1$consecutive elements of $X$.

As $Y$ is a GS of $Y'$.

There are at most $2(h_Y + 1) + 1$ consecutive elements of $Y'$ between $(h_Y + 1) + 1$ consecutive elements of $Y$.

At most there are $2h_X - 1 + 2h_y + 3 = 2r + 2$ elements of $X'|Y'$ between $r$ consecutive elements of $X|Y$.

$e_r \in Y$: Let us add $e_0 \in Y$ preceding $e_1$ and $e_{r+1}$ succeeding $e_r$.

Elements of X′ and Y′ lying between $e_1, \ldots, e_r$ come from elements of $X'$ (resp. $Y'$) that lie between $e_1, \ldots, e_{r+1}$ (resp. $e_0, \ldots, e_r$) that is between $h_X + 1$ (resp. $h_Y + 1$) elements of $X$ (resp. $Y$).

Overall, there are at most $2r + 2$ elements of $X|Y$ between $r$ elements of $X'|Y'$. □

$Z = Reduce(X|Y)$ and $Z' = Reduce(X'|Y')$

We consider $k + 1$ consecutive elements of $Z$: $z_1, \ldots, z_{k+1}$.

By definition of the reduction operation, we have$z_1 = e_{4k}, z_2 = z_{4k+4}, \ldots$ where$X|Y = e_1, \ldots, e_n$.

There are $4k + 1$ elements of $X|Y$ between $z1, \ldots, z_{k+1}$.

Between these $r = 4k + 1$ elements of $X|Y$, there are (because of the intermediate result) at most $2r + 2 = 8k + 4$ elements of $X'|Y'$.

Because the reduction operation takes every fourth element of its input in between $k + 1$ consecutive elements of $Z$, there are, at most, $\frac{8k+4}{4} = 2k + 1$ elements of $Z'$. $\square$

<u>In steady-state at node</u>: receive $X(t+1)$ (resp. $Y(t+1)$) from its left (resp. right) child, computes $val(t+1) = \textsc{MergeWithHelp}(X(t+1), Y(t+1), Val(t))$, sends $Z(t+1) = Reduce(val(t+1))$ to its father.

The invariants are:

- $val(t+1 = X(t+1)|Y(t+1)$
- $X(t)$ is a GS of $X(t+1)$
- $Y(t)$ is a GS of $Y(t+1)$

<u>Cross-ranks</u>

At a given step, $X$ (resp. $Y$) is a GS of $X'$ (resp. $Y'$).

$U = X|Y$

$Z = Reduce(u)$

We assume that the cross-rank $R[X', X]$ and $R[Y', Y]$ are known (induction hypothesis). To compute $U' = X'|Y'$ with $\textsc{MergeWithHelp}$ we need to know the cross-ranks $R[X', U]$, $R[Y', U]$, $R[U, X']$, $R[U, Y']$ (we will then be able to compute $Z' = Reduce(U')$ and $R[Z', Z]$).

We assume that for any sorted sequence S, we know $R[S, S']$

**Lemma 16.**

if $S = [b_0, \ldots, b_k]$ is a sorted sequence the the rank of any element $a$ in $S$ can be computed in $\mathcal{O}(1)$ with $\mathcal{O}(k)$ PUs on a CREW PRAM.

*Proof.* $b_0 = -\infty$, $b_{k+1} = +\infty$

**for** $0 \leqslant i \leqslant k$ *in parallel* **do**
  **if** $b_i < a \leqslant b_{i+1}$ **then**
    $rank \leftarrow i$

$\square$

**Lemma 17.**

Let $S_1, S_2$ and $S$ be 3 sorted sequences with $S = S_1|S_2$ and $S_1 \cap S_2 = \emptyset$. Then, we can compute the cross-ranks $R[S_1, S_2]$, $R[S_2, S_1]$ in $\mathcal{O}(1)$ with $\mathcal{O}(|S|)$ PUs.

*Proof.* We assume we know $R[S_2, S_1]$, $R[S_1, S_2]$, $R[S, S]$.

$\forall a \in S_1 \subseteq S, rank(a, S_2) = rank(a, S) - rank(a, S_1)$ $\square$

**Lemma 18.**

Let $X$, $Y$, $X'$ and $Y'$ be sorted sequences.
$U = X|Y$ and such that $X$ is a GS of $X'$ and $Y$ is a GS of $Y'$. We assume we know the $R[X', X]$ and $R[Y', Y]$. Then we can compute in time $\mathcal{O}(1)$ using $\mathcal{O}(|X| + |Y|)$ PUs. The cross-rank $R[X', U]$, $R[Y, u]$, $R[U, X']$ and $R[U, Y']$.

**Lemma 19.**

If $Z = \mathit{Reduce}(u)$ and $Z' = \mathit{Reduce}(u')$ then one can compute $R[Z', Z]$ in $\mathcal{O}(1)$ using $\mathcal{O}(|X| + |Y|)$ PUs.

**Theorem 20.**

The COLE's merge sort algorithm can sort a list of size $m$ in time $\mathcal{O}(\log m)$ using $\mathcal{O}(m)$ processors on a CREW PRAM.

# Part II

# Algorithms for rings and grids of processors

# Chapter 3

# Algorithms for rings of processors

## 3.1 Model

We have a model with distributed memory. Each processor have a piece of the memory and this is the only part of the memory that the processor can reach.

Data transfers are explicitly done by communication channels.

$p$ processors in an unidirectional ring: processor $P_i$ can only send messages to processor $P_{i+1}$ and receive messages from processor $P_{i-1}$.

- No need to precise source or destination of a message.
- $P_{i+1}$ should be understood modulo $p$.

Two functions:

- Number of processors in the ring: `NbProcs()`.
- Rank of a processor in the ring: `MyNum()`

Primitives of communication:

- `Send(addr, m)`: send a message stored at the address `addr` and which is of size `m`.
- `Receive(addr, m)`: receive a message of size `m` et write it at the address specified by `addr`.

Each `Send` (resp. `Receive`) must be matched by a `Receive` (resp. `Send`).

Communications can be blocking or non blocking. Most of the time, we will assume blocking `Receive` and non blocking `Send`.

**Definition 7** (Blocking communication).

The algorithm (or program) does not "return" from the call to a communication primitive (i.e. does not start to execute the next instruction) as long as the communication has not completed.

**Definition 8** (Non-blocking communication)**.**

The call of the communication primitive returns instantaneously. The communication itself takes place later on.

Most of the time, we will consider blocking `Receive` and non-blocking `Send` .

Usually, one needs to check that the communication has taken place before using the data "received" or reusing the memory location where the data was stored.

**Definition 9** (Communication time)**.**

Time needed to send or receive a message of length $m$ is $L + mb$, where $L$ is the latency (= communication time) and $b$ is the inverse of the bandwidth (= maximum bit rate).

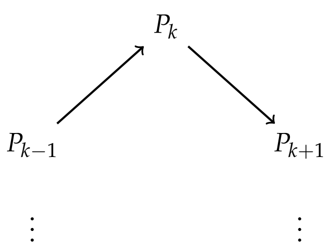
## 3.2 Macro communication

### 3.2.1 Broadcast

Consider a processor $P_k$.

$P_k$ sends the same message of length $m$ to every other processor.

We have a function `Broadcast(k,addr,m)` where `k` is the origin of the broadcast, `addr` is the location of the data, and `m` is the size of the message.




**Algorithm 9:** Broadcast

**Data**: $k$, `addr`, `m`
$q \leftarrow$ `myNum()`
$p \leftarrow$ `nbProcs()`
**if** $q = k$ **then**
  `Send(addr,m)`
**else if** $q = k - 1 \mod p$ **then**
  `Receive(addr,m)`
**else**
  `Receive(addr,m)/* Blocking. */`
  `Send(addr,m)/* Blocking or non blocking. */`

**Execution time** $(p - 1)(L + mb)$

### 3.2.2 Scatter

Processor $P_k$ sends a different message to each other processor. Initially, processor $P_k$ holds the message for processor $P_q$ in `addr`$[q]$.

We assume that `addr`$[k]$ holds a message for $P_k$ (in order to ease the writing).

At the end of the algorithm, each processor should have its own message stored at address `msg`.

---

**Algorithm 10:** Scatter

---

**Data**: $k$, `msg`, `addr`, `m`
$q \leftarrow$ `myNum()`
$p \leftarrow$ `nbProcs()`
**if** $q = k$ **then**
  **for** $i = 1$ ***to*** $p - 1$ **do**
    `Send(addr[k-i mod p], m)`
**else**
  **for** *i=1* ***to*** *p-q+k-1 mod p* **do**
    `Receive(tempR, m) /* Blocking */`
    `Send(tempR, m) /* Blocking */`
  `Receive(msg,m)`

---

**Remark 4.**

We can not perform non-blocking send in the above algorithm, because of the `tempR` variable (we might lose messages).

**Assuming** the messages are received in the same order ther are sent.

**Execution time** $2(p-2)+1$ communications $\Rightarrow$ $(2p-3)(L+mb)$

---

**Algorithm 11:** Parallel (blocking) communication version

---

`Receive` (tempR,m);
**for** *i=1* ***to*** *p-q+k-1 mod p* **do**
  `Send` (tempR, m) || `Receive` (tempS, m) `/* Parallel execution. */`
  `swap`(tempS,tempR)

---

**Execution time** $(p-1)(L+mB)$

**Version where each processor sends a single message** $P_k$ sends a message of size $(p-1)m$ in time $L+(p-1)mb$

$P_{k+1}$ sends a message of size $(p-2)m$ in time $L+(p-2)mb$

$\vdots$

$P_{k-2}$ sends a message of size $m$ in time $L+mb$

Hence, a total execution time of $L(p-1)+\frac{p-1}{2}pmb$.

### 3.2.3 All to all

Each processor in the ring sends the same message to every other processor. In other words, each processor is the source of a broadcast.

Naive solution: use $p$ times the broadcast algorithm, for an execution time of $p(p-1)(L+mb)$.

---

**Algorithm 12:** allToAll

---

```
Data: myMsg, addr, m
q ← myNum()
p ← nbProcs()
addr[q] ← myMsg
for i = 1 to p-1 do
    Send (addr[q-i+1 mod p], m) || Receive (addr[q-i mod p], m)
```

---

Receive instruction has to be blocking.

**Execution time** $(p-1)(L+mb)$

### 3.2.4 Pipelined broadcast

- Shorter communications can enable processors to start forwarding data earlier.
- Splitting messages enables some parallelism.
- Splitting messages decreases the size of messages, and thus increases the impact of latencies.

Consider a message of size $m$. Split it in $r$ same size pieces (assuming that $r$ divides $m$).

The $r$ pieces are stored in $addr[0], addr[1], \ldots, addr[r-1]$.

At the end, we want all pieces to be stored on each processor.

---

**Algorithm 13:** Broadcast

---

```
Data: k,addr,m
q ← myNum()
p ← nbProcs()
if q=k then
    for i=0 to r-1 do
        Send(addr[i], m/r)
else if q=k-1 mod p then
    for i=0 to r-1 do
        Receive(addr[i], m/r)
else
    Receive(addr[0], m/r)
    for i=1 to r-1 do
        Send(addr[i-1], m/r) || Receive(addr[i], m/r)
    Send(addr[r-1], m/r)
```

---

**Execution time** There are two ways to compute the execution time.

1. - Time at which the first message arrives at $P_{k-1}$: $(p-1)\left(L+\frac{m}{r}b\right)$

- Time needed for the other messages to arrive at $P_{k-1}$: $(r-1)\left(L+\frac{m}{r}b\right)$

Hence a total execution time of $(p+r-2)(L+\frac{m}{r}b)$.

2. - Time at which the communication of the last message starts on $P_k$: $(r-1)\left(L+\frac{m}{r}b\right)$
   - Time needed for that message to arrive at $P_{k-1}$: $(p-1)\left(L+\frac{m}{r}b\right)$

   Hence a total execution time of $(p+r-2)(L+\frac{m}{r}b)$.

This is minimized for $r_{opt} = \sqrt{\frac{m(p-2)b}{L}}$, given an execution time of $\left(\sqrt{(p-2)L}+\sqrt{mb}\right)^2 \underset{m\to+\infty}{\sim} mb$.

## 3.3 Matrix-vector multiplication

Consider a $n \times n$ matrix $A$ and a vector $x$ of size $n$. We want to compute $y \leftarrow y + Ax$.

**Algorithm 14:** Sequential algorithm

**for** $i = 0$ **to** $n-1$ **do**
  $y_i \leftarrow 0$
  **for** $j = 0$ **to** $n-1$ **do**
    $y_i \leftarrow y_i + A_{ij} \times y_j$

Each iteration of the loop executes a scalar product. The products are independant. If $p$ divides $n$, give to each processor $r = \frac{n}{p}$ scalar products to compute. The complexity depends (in part) of the way data is distributed among the processors.

The matrices are large, one can not have a copy of $A$ on each processor. We assume that $A$ is distributed by blocks of rows. $P_0$ is holding the first rows of $A$, $P_i$ is holding rows $ri$ to $(r+1)i-1$ of $A$.

Assume that $x$ is distributed the way $A$ is distributed (processor $P_i$ holds the components $x_{ir}$ to $x_{(r+1)i-1}$ of $x$). We want output data to have the same distribution than input data, because after having computed $y = Ax$, we want to be able to directly compute $z = By$ (assuming that $A$ and $B$ are distributed the same way).

**Obvious solution** We give $r$ values of $y$ to be computed by each processor.
$P_i$ compute $y_{(i-1)r}, \ldots, y_{ir-1}$
$P_i$ need the lines of index from $(i-1)r$ to $ir-1$ of the array $A$.
$P_i$ need all values of $x$.
Each processor work independently: no communication is needed.

We won't use this way.

**Usual assumptions** All variables are distributed in the same direction. In the previous solution, that wasn't true. Only $r$ components of $y$ are given to each processor but all components of $x$ are needed.

**Initially** $P_i$ knows the line $(i-1)r$ to $ir-1$ of $A$ and the components $(i-1)r$ to $ir-1$ of $x$ and $y$.

**Execution time** $p \times \max\left(L + br, r^2 \cdot w\right) \sim \frac{n^2 w}{p}$

**Efficiency** $\xrightarrow{\infty} 1$

**Algorithm 15:** matrixVectorProduct

```
Data: A, x, y, n, s
q ← myNum();
p ← nbProcs();
r ← n/p;
tmpS ← x // initialize y ← 0
for step = 0 to p − 1 do
    Send (tmpS, r);
    ||Receive (tmpR, r);
    || for i = 0 to r − 1 do
        y[i] ← y[j] + A [q − (step mod p) × r + j] × tmpS [j];
    tmpR ↔ tmpS // swap pointers
```

## 3.4 Matrix-matrix product

We have 3 square matrices $A, B, C$ of size $n \times n$

We want to compute $C \leftarrow C + A \times B$.

**Algorithm 16:** Sequential algorithm

```
for i=0 to n-1 do
    for j=0 to n-1 do
        for k=0 to n-1 do
            C_{i,j} ← C_{i,j} + A_{ik} B_{kj}
```

$n$ is divisible by $p$: $r = \frac{n}{p}$

The three matrices are distributed in the same way, as blocks of rows.

Processor $P_q$ holds the rows of index $(q-1)r$ to $qr - 1$ of matrices $A$, $B$ and $C$.

With the block notation, processor $P_q$ holds the blocks $\hat{A}_{q,l}$, $\hat{B}_{q,l}$, $\hat{C}_{q,l}$ with $l \in [\![0, p-1]\!]$.

Processor $P_q$ will compute completely the blocks $\hat{C}_{q,l}$.

Step 0 $P_q$ holds $\hat{A}_{q,q}$ and all the $\hat{B}_{q,l}$

$\forall l \in [\![0, p-1]\!], \hat{C}_{q,l} \leftarrow \hat{C}_{q,l} + \hat{A}_{q,q}\hat{B}_{q,l}$

Processor $P_q$ send all its blocks of $B$ to processor $P_{q+1}$

Step 1 $P_q$ holds $\hat{A}_{q,l}$, $\hat{C}_{q,l}$, $\hat{B}_{q-1,l}$

It can compute $\hat{C}_{q,l} \leftarrow \hat{C}_{q,l} + \hat{A}_{q,q-1}\hat{B}_{q,l}$

Sending $\hat{B}_{q-1,l}$ to $P_{q+1}$ (receives $\hat{B}_{q-r,l}$ from $P_{q-1}$)

**Execution time** $p \times \max\left(L + nrb, pr^3 w\right) = \max\left(pL + n^2 b, \frac{n^3}{p} w\right) \underset{n\to\infty}{\sim} \frac{n^3}{p} w$

**Efficiency** $\xrightarrow{\infty} 1$

**Execution time for $n$ matrix-vector multiplication**

$np \max\left(L + br, \frac{n^2}{p^2} w\right) = \max\left(npL + n^2 b, \frac{n^3}{p} w\right)$

**Algorithm 17:** Matrix-matrix product

```
Data: A, B, C, n
q ← myNum()
p ← myProcs()
r ← n/p
tmpS ← B
for step = 0 to p − 1 do
    Send (tmpS, n × r)
    ||Receive (tmpR, n × r)
    ||for l = 0 to p − 1 do
        for i = 0 to r − 1 do
            for j = 0 to r − 1 do
                for k = 0 to r − 1 do
                    C[i, lr + j] ← C[i, lr + j] + A[i, r × ((q − step) mod q) + k] × tmpS[k, lr + j]
    tmpR ↔ tmpS
```

**Asymptotic behaviour** Same volume of communication but different latency.

## 3.5 Stencil computation

A discrete domain $A$ made of cells. Iteratively, repeatedly, we update the content of these cells.

The update is made using a rule that defines the new value of the cell as a function of the previous values of the neighbourhood.

### 3.5.1 A sequential stencil

2-dimensional array (a grid) of size $n \times n$. Each cell has 8 neighbours (MOORE's neighbourhood).

| | | |
|---|---|---|
| NW | N | NE |
| W | C | E |
| SW | S | SE |

But we don't use this neighbourhood. Only the new value of two neighbours are useful for the computation.

Update rule $C_{new} \leftarrow \text{UPDATE}\left(C_{old}, W_{new}, N_{new}\right)$

If no North neighbour, replace by NIL.

For the first line, the update rule is $C_{new} \leftarrow \text{UPDATE}\left(C_{old}, W_{new}, NIL\right)$

The rule of this model use the new values of the neighbours. This is the main difference with a cellular automaton which use only old values of the neighbours to compute the new ones.

### 3.5.2 Algorithms

We have a ring of $p$ processors. For the moment, we assume $p = n$. Each processor holds one row of $A$.

Greedy algorithm: Principle: a processor sends the cell it has computed as soon as it has completed them.

**Algorithm 18:** Greedy stencil

```
Data: A
q ←MyNum()
p ←NumProcs()
if q = 0 then
    for i = 0 to n − 1 do
        if i = 0 then
            A[0] ← Update(A[0], Nil, Nil)
        else
            A[i] ← Update(A[0], A[i − 1], Nil)
        Send (A[i], 1)
else if q = p − 1 then
    for i = 0 to n − 1 do
        Receive (tmpR, 1)
        if i = 0 then
            A[0] ← Update(A[0], Nil, tmpR)
        else
            A[i] ← Update(A[0], A[i − 1], tmpR)
        Send (A[i], 1)
else
    ...
```

But bof... because `Send` and `Receive` force the other parts to be too sequential.

So... new version!

**Algorithm 19:** Greedy update

```
Data: A
q ←MyNum()
p ←NumProcs()
if q = 0 then
    A[0] ← Update(A[0], Nil, Nil)
    Send (A[0], 1)
else
    Receive (v, 1)
    A[0] ← Update(A[0], Nil, v)
for j = 1 to p − 1 do
    if q = 0 then
        A[j] ← Update(A[j], A[j − 1], Nil)
        Send (A[j], 1)
    else if q = p − 1 then
        Receive (v, 1)
        A[j] ← Update(A[j], A[j − 1], v)
    else
        Receive (v, 1)||Send (A[j − 1], 1)
        A[j] ← Update(A[j], A[j − 1], v)
if q ≠ 0 ∧ q ≠ p − 1 then
    Send (A[p − 1], 1)
```

**Remark 5.**

The first and last processors are working together for one step.
$p \ll n : r = \frac{n}{p}$

**Overall execution time** : Focus on the last processor it starts working after $(p-1)$ steps. It works for $p$ steps.

Length of a step: $L + b + w$.

Overall execution time: $(2p-1)(L+b+w)$.

**General case:** $p < n$ (we assume that $p$ divides $n$). Distribution of data: cyclic distribution of rows. Row $j$ is allocated to processor $P_{j \mod p}$.

---

**Algorithm 20:** CyclicUpdate

---

**Data**: $A, n$
**for** $i = 0$ **to** $r-1$ **do**
  **if** $q = 0 \wedge i = 0$ **then**
    $A[0][0] \leftarrow \textsc{Update}(A[0][0], Nil, Nil)$
    `Send` $(A[0,0], 1)$
  **else**
    `Receive` $(v, 1)$
    $A[i][0] \leftarrow \textsc{Update}(A[i][0], Nil, v)$
**for** $j = 1$ **to** $n-1$ **do**
  **if** $q = 0 \wedge i = 0$ **then**
    $A[i][j] \leftarrow \textsc{Update}(A[i][j], A[i][j-1], Nil)$
    `Send` $(A[i][j], 1)$
  **else if** $q = p-1 \wedge i = \frac{n}{p} - 1$ **then**
    `Receive` $(v, 1)$ $A[i][j] \leftarrow \textsc{Update}(A[i][j], A[i][j-1], v)$
  **else**
    `Receive` $(v, 1) \|$ `Send` $(A[i][j-1])$
    $A[i][j] \leftarrow \textsc{Update}(A[i][j], A[i][j-1], v)$
**if** $\neg(p = 0 \wedge i = 0) \wedge \neg(q = p-1 \wedge i = \frac{n}{p} - 1)$ **then**
  `Send` $(A[i][n-1])$

---

**Overall execution time** The last processor begin to work after $p-1$ steps. There are $n^2$ elements to update: $\frac{n^2}{p}$ elements assigned to each processor. $\rightarrow P_{p-1}$ not after $p - 1 + \frac{n^2}{p}$ step.

Overall execution time: $\left(p - 1 + \frac{n^2}{p}\right)(\underbrace{w}_{\text{update time}} + L + b)$

**Efficiency** $\frac{n^2 w}{p\left(p-1+\frac{n^2}{p}\right)(w+L+b)} \sim \frac{w}{w+L+b} < 1$ (bad efficiency)

$P_0 = \frac{n}{p}$ at the first line, $P_1 : \frac{n}{p}$ follows $\ldots \rightarrow$ decrease the volume and the number of communications.

**But** The last processor have to wait a lot before be able to work.

1 element per communication: too small $\ldots \rightarrow$ send $k$ results per communication.

### Other idea

- We send a message every $k$ update ($k|n$).
- We assign $r$ consecutive lines to a processor (a cyclic distribution for line bloc) $\rightarrow$ that decrease at the same time the volume and the number of communications.

### Disadvantages

- $K > 1$: we add latency at the start of each processor(except for $P_0$).
- $r > 1$: the second processor have to wait that $kr$ data are completed before being able to start.
- The time when $P_0$ can start to work on the second bloc of $r$ columns:
  - If it had finished the computation of the $r$ first columns: $T_{r,k} :=$ the time needed to compute a bloc of $r \times k$ data and send the result $\geqslant \frac{n}{k} T_{r,k}$.
  - $P_0$ have to have received its first messages from $P_{p-1} \rightarrow$ at time $pT_{r,k}$.
  - No waiting time if $\frac{n}{k} \geqslant p$

We assume that $\frac{n}{k} \geqslant p$ and $pr|n$.

**Execution time** $\left((p-1) + \frac{n}{k} \times \frac{n}{rp})\right) T_{r,k}$
$(p-1)$: starting time of $P_{p-1}$
$\frac{n}{k}$: number of blocs in a column
$\frac{n}{rp}$: number of blocs in a column per processor.

**Overall execution time** $\left(p - 1\frac{n^2}{prl}\right)(rkw + L + kb)$

**Asymptotic efficiency** $\frac{n^2 w}{p(p-1+\frac{n^2}{prk})(rkw+L+bk)}$
The cyclic distribution algorithm has a asymptotic efficiency of $\frac{w}{w+b+L}$
The execution of functions is lowest for $k = k'(r) = n\sqrt{\frac{L}{p(p-1)+(rw+b)}}$
Then the optimal value (no waiting time) is $K^{opt}(r) = \min\left(\frac{n}{p}, K'(r)\right)$

# Chapter 4

# Matrix multiplication for a grid of processor

## 4.1 Topology

We have $p = q \times q$ processors organized in a grid: each processor has 4 neighbours (torus structure).

We assume that each link is bidirectional and full duplex:

- 1 port model: a processor can simultaneously send and receive messages, but at most one message is sent and receive at the same time.
- 4 ports model: the same except that 4 messages can be shared at the same time: one for each neighbour.

Number of connexions: $p$

## 4.2 Communication Primitives

The processors are indexed by $P_{i,j}$, with $0 \leqslant i, j \leqslant q - 1$. The index are given by `myProcRow` and `myProcCol`.

`Send` (dest, addr, L), `Receive` (source, addr, L)

`BroadcastRow`(i, j, srcAddr, destAddr, L)

Every processor $P_{i,k}$ for $k \in [\![0, q-1]\!]$ is involved in this broadcast.

$P_{k,l}$ with $k \neq i$: nothing happens, return immediately.

`BroadcastCol`(i, j, srcAddr, destAddr, L)

Same idea.

## 4.3 Outer product algorithm

$(A, B, C) \in (\mathcal{M}_n)^3$

The algorithm 14 describe the sequential algorithm of matrix multiplication.

It's a inner multiplication algorithm.

The outer product algorithm does loops in this order:

**for** $k = 0$ **to** $n-1$ **do**
  **for** $i = 0$ **to** $n-1$ **do**
    **for** $j = 0$ **to** $n-1$ **do**
      $c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}$

Using the block notation:

**for** $k = 0$ **to** $q-1$ **do**
  **for** $i = 0$ **to** $q-1$ **do**
    **for** $j = 0$ **to** $q-1$ **do**
      $\hat{C}_{i,j} = \hat{C}_{i,j} + \hat{A}_{i,k} \cdot \hat{B}_{k,j}$

For a given $k$: $P_{i,j}$ needs $A_{i,k}$ form $P_{i,k}$ and $B_{k,j}$ from $P_{k,j}$
$\rightarrow P_{k,j}$ broadcasts $B_{k,j}$ on the column $j$, and $P_{i,k}$ broadcasts $A_{i,k}$ on the row $i$.

**Algorithm 21:** outerMatrixProduct

**Data**: A,B,C,n
$q \leftarrow \sqrt{nbProcs}$
myRow, myCol $\leftarrow \dots$
**for** $k = 0$ **to** $q-1$ **do**
  **for** $i = 0$ **to** $q-1$ **do**
    `BcastRow`(i, k, A, bufferA, $m \times m$)
  **for** $j = 0$ **to** $q-1$ **do**
    `BcastCol`(k, j, B, bufferB, $m \times m$)
  **if** $myRow == k \wedge myCol == k$ **then**
    `MatrixMultiplyAdd`(C, A, B, m)
  **else if** $myRow == k$ **then**
    `MatricMultiplyAdd`(C ,bufferA, B, m)
  **else if** $myCol == k$ **then**
    `MatrixMultiplyAdd`(C, A, bufferB, m)
  **else**
    `MatrixMultiplyAdd`(C, bufferA, bufferB, m)

**Time for a pipelined broadcast** $T_{Bcast} = \left(\sqrt{(q-2)L} + \sqrt{m^2 b}\right)^2$

**Overall execution time** For a 1 port-model: $T_{n,p} = q \times (2T_{Bcast} + m^3 w)$
For a 4-part model: $T_{n,p} = q \times (T_{Bcast} + m^3 w)$

**With less time?** We could parallelize broadcasts and `MultAdd`:

**Algorithm 22:** Broadcast for $k = 0$

**for** $k = 0$ **to** $q-2$ **do**
  Broadcast for $k+1$
  Computation for $k$
Computation for $q-1$

**Execution time** $2T_{Bcast} + (q-1)\max(2T_{Bcast}, m^3w) + m^3w$

$T_{Bcast} \sim qm^2b = \frac{n^2b}{q} = \frac{n^2b}{\sqrt{p}}$

$T(n,p)\sum qm^3w = \frac{n^3w}{p}$

## 4.4 The Common algorithms

**Advantage** all communication are done with ones neighbours.

Initially, each line (resp. column) block of A (resp. B) is shifted such that each processor in the first column (resp. row) holds a diagonal element of A (resp. B) (pre-skewing).

$$\begin{pmatrix} \hat{A}_{0,0} & \hat{A}_{0,1} & \hat{A}_{0,2} & \hat{A}_{0,3} \\ \hat{A}_{1,1} & \hat{A}_{1,2} & \hat{A}_{1,3} & \hat{A}_{1,0} \\ \hat{A}_{2,2} & \hat{A}_{2,3} & \hat{A}_{2,0} & \hat{A}_{2,1} \\ \hat{A}_{3,3} & \hat{A}_{3,0} & \hat{A}_{3,1} & \hat{A}_{3,2} \end{pmatrix}$$

$$\begin{pmatrix} \hat{B}_{0,0} & \hat{B}_{1,1} & \hat{B}_{2,2} & \hat{B}_{3,3} \\ \hat{B}_{1,0} & \hat{B}_{2,1} & \hat{B}_{3,2} & \hat{B}_{0,3} \\ \hat{B}_{2,0} & \hat{B}_{3,1} & \hat{B}_{0,2} & \hat{B}_{1,3} \\ \hat{B}_{3,0} & \hat{B}_{0,1} & \hat{B}_{1,2} & \hat{B}_{3,3} \end{pmatrix}$$

After the first step:

$$\begin{pmatrix} \hat{A}_{0,1} & \hat{A}_{0,2} & \hat{A}_{0,3} & \hat{A}_{0,0} \\ \hat{A}_{1,2} & \hat{A}_{1,3} & \hat{A}_{1,0} & \hat{A}_{1,1} \\ \hat{A}_{2,3} & \hat{A}_{2,0} & \hat{A}_{2,1} & \hat{A}_{2,2} \\ \hat{A}_{3,0} & \hat{A}_{3,1} & \hat{A}_{3,2} & \hat{A}_{3,3} \end{pmatrix}$$

$$\begin{pmatrix} \hat{B}_{1,0} & \hat{B}_{2,1} & \hat{B}_{3,2} & \hat{B}_{0,3} \\ \hat{B}_{2,0} & \hat{B}_{3,1} & \hat{B}_{0,2} & \hat{B}_{1,3} \\ \hat{B}_{3,0} & \hat{B}_{0,1} & \hat{B}_{1,2} & \hat{B}_{3,3} \\ \hat{B}_{0,0} & \hat{B}_{1,1} & \hat{B}_{2,2} & \hat{B}_{3,3} \end{pmatrix}$$

**Algorithm 23:** Common algorithm

```
horizontal pre-skewing of A
vertical pre-skewing of B
for k = 0 to q − 1 do
    MatrixMultiplyAdd(C, A, B, m)
    Horizontal shift of A
    Vertical shift of B
horizontal post-skewing de A
vertical post-skewing de B
```

$$T_{skew}^{4\ ports} = \overbrace{2}^{\text{pre + post}} \times \left\lfloor \frac{q}{2} \right\rfloor (L + m^2b)$$

$$T_{skew}^{1p} = 2 \times T_{skew}^{4p}$$

$$T_{comp}^{4p} = q \times \max(m^3w, L + m^2b)$$

$$T_{comp}^{1p} = q \times \max(m^3w, 2L + 2m^2b)$$

**Algorithm 24:** `Fox` algorithm

**for** $k = 0$ **to** $q - 1$ **do**
- horizontal broadcast of the $k^{\text{th}}$ diagonal of $A$
- `MatrixMultiplyAdd`(C, A, B, m)
- vertical shift of $B$

$$T_{BCast} = (\sqrt{(q-2)L} + \sqrt{m^2 b})^2$$

$$T^{4p} = T_{Bcast} + (q-1)\max(m^3 w, T_{BCast}) + \max(m^3 w, L + m^2 b)$$

$$T^{1p} = T_{Bcast} + (q-1)\max(m^3 w, T_{BCast} + 4m^2 b) + \max(m^3 w, L + m^2 b)$$

Part III

# Tasks scheduling graphs

# Chapter 5

# Introduction to tasks graphs

$A \cdot x = b$ where

- $A \in GL_n(\mathbb{R})$ known lower triangular matrix
- $b \in \mathbb{R}^n$ known

**Algorithm 25:** Classical algorithm

**for** $i = 0$ **to** $n-1$ **do**
  Task $T_{i,i} : x_i \leftarrow \frac{b_i}{a_{i,i}}$
  **for** $j = i+1$ **to** $n-1$ **do**
    Task $T_{i,j} : B_j \leftarrow b_j - a_{i,j}x_i$

The program is sequential, the entire computation is done by the program and a total order on tasks.

$<_{seq}$: the total order in the original program
$T <_{seq} T'$: $T$ is computed before $T'$ in the original program.

$T_{1,1} <_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \cdots <_{seq} T_{1,n-1} <_{seq} T_{2,1} <_{seq} \cdots <_{seq} T_{n-1,n-1}$

$T_{i,j}$: $T_{1,1}$ is the first task and computes $x_1$, $T_{1,2}$ and $T_{2,3}$ reads $x_1$, they need to be computed after $x_1$.

$T_{1,2}$ updates $b_2$ and $T_{1;,3}$ updates $b_3$. $T_{1,2}$ and $T_{1,3}$ are independent and can be executed in any order.

Task $T$:
$in(T)$: set of variables read by task $T$.
$out(T)$: set of variables written by task $T$.

$T \bot T'$ if $T$ and $T'$ are not independent, iff they share some variable that is written by at least one of them.

$T \bot T' \iff (Out(T) \cap Out(T') \neq \emptyset) \bigvee (Out(T) \cap In(T') \neq \emptyset) \bigvee (Out(T') \cap In(T) \neq \emptyset)$

(BERNSTEIN conditions)

**Definition 10** (Precedence relation $\prec$)**.**

If $T \bot T'$ and if $T \leqslant_{seq} T'$ then we have $T \prec T'$

$$\prec := (<_{seq} \cap \bot)^+$$

$\prec$: is a partial order (consistent with $<_{seq}$):
$<= (<_{seq} \cap \perp)^+$

$T_{2,4}$ and $T_{4,4}$:
$T_{2,4}$ updates $b_4$ and $T_{4,4}$ read $b_4 \;\Rightarrow\; T_{2,4} \perp T_{4,4}; T_{2,3} < T_{4,4}$

$T_{4,4}$ and $T_{4,5}$:
$T_{4,4}$ write $x_4$ and $T_{4,(}$ write $x_4 \;\Rightarrow\; T_{4,4} \perp T_{4,5}; T_{4,4} < T_{4,5}$

$T_{2,4}$ and $T_{4,5}$: $T_{2,4}$ and $T_{4,5}$ are independent $\Rightarrow$ we need to use the transitive closure of the dependency relation.

**Representation in a directed graph** $G = (V, E)$
$V$ = set of tasks.
$e = (T, T') \in E \iff T < T'$

We do not include in this graph the transitive relationships: if $(T, T') \in E$ and $(T', T'') \in E$ there is no need to include $(T, T'')$

# Chapter 6

# Scheduling task graph

**Definition 11.**

A task system on a task graph is a directed graph which vertices have weight.
$G = (V, E, w)$

- $V$: the list of tasks
- The set of edges represent the constraints (or dependencies) of the procedure: $e = (u, v) \in E \iff u \prec v$
- $w : V \to \mathbb{N}^*$ gives the execution time.

**Definition 12.**

A *Scheduling* of a task system $G = (V, E, w)$ is a function $\sigma : V \to \mathbb{N}$ such that $\sigma(u) + w(v) \leqslant \sigma(x)$, moreover $u \prec v \iff (u, v) \in E$

If the number of available processors, $p$, is bounded ($p < \infty$), then $alloc(T)$ specifies on each processor the task $T$ which is executed.

At a given time, only one task can be compute by each processor.

$alloc(T)$: the processor run the task $T$

$\forall T, T'$ such that $alloc(T) = alloc(T')$, then $\sigma(T) + w(T) \leqslant \sigma(T') \bigvee \sigma(T') + w(T') \leqslant \sigma(T)$

**Theorem 21.**

Let $G = (V, E, w)$ A task system. There exists a scheduling of $G$ if and only if $G$ does not contain cycles.

*Proof.* We assume that $G$ contains a cycle $v_1 \to v_2 \to \cdots \to v_1$, then $v_1 \prec v_1$ and $\sigma(v_1) + w(v_1) \leqslant \sigma(v_1)$. Impossible because $w \in \mathbb{N}^*$

We assume that $G$ does not contain cycles. We compute a topological sort of vertices (and build a scheduling). $\square$

Task system $G = (V, E, w)$ , $\sigma$ a scheduling for $G$ using $p$ processors.

$MS(\sigma, p)$: lifetime of $\sigma$

$MS(\sigma, p) = \max_{v \in V} \{\sigma(v) + w(v)\} = \min_{u \in V} \sigma(v)$

**Definition 13.**

We name $Pb(p)$ the problem of finding a scheduling of minimal lifetime using at most $p$ processors (we write $Pb(\infty)$ if the number of processor is not limited).

**Definition 14.**

$MS_{opt}(p)$ lifetime of a scheduling using at most $p$ processors.
$MS_{opt}(p) = \min_{\sigma} MS(\sigma, p)$

**Proposition 22.**

$G(V, E, w)$ and a scheduling $\sigma$ using $p$ processors.
$MS(\sigma, p) = w(\phi)$ where $\phi$ is a path $G$
$\phi : v_1 \to v_2 \to \cdots \to v_k$
$w(\phi) = \sum_{i=1}^{k} w(v_i)$
$\forall i \in [1, k-1], \sigma(v_i) + w(v_i) \leqslant \sigma(v_{i+1})$
Per sum, $\sigma(v_1) + w(\phi) \leqslant \sigma(v_k), w(\phi) \leqslant \sigma(v_k) - sigma(v_1) \leqslant MS(\sigma, p)$

The speedup get by scheduling $\sigma$ using $p$ processors.

$S(\sigma, p) = \frac{Seq}{MS(\sigma,p)}, Seq = \sum_{v \in V} w(v)$

Efficiency: $e(\sigma, p) : \frac{Seq}{pMS(\sigma,p)}$

**Theorem 23.**

$0 \leqslant e(\sigma, p) \leqslant 1$

**Theorem 24.**

$Seq = MS_{opt}(1) \Rightarrow MS_{opt}(2) \geqslant \ldots \geqslant MS_{opt}(|V|) = \cdots = MS_{opt}(\infty)$

*Proof.* Every scheduling with $p$ processors can be scheduled on $p + 1$ processors, the last one remains unemployed.

$MS'_{opt}(p)$: minimal lifetime get by scheduling using $p$ processors (at worst, there is one task by processor).

If $p \leqslant |V|$, $MS'_{opt}(p) = MS_{opt}(p)$ (because there is no communication time). □

# Chapter 7

# Solve $Pb(\infty)$

$G = (V, E, w)$

$\forall v \in V$, $Pred(v)$ is the set of immediate predecessors of $v$. $Succ(v)$ is the set of immediate successors of $v$.

On input, a top vertex (or empty vertex) is a vertex $v$ such that $Pred(v) = \emptyset$.

On output, a bottom vertex is a vertex $v$ such that $Succ(v) = \emptyset$.

**Definition 15.**

$\forall \sigma \in V$, the top level of $v$, $tl(v)$, is the path of maximum weight from any entry vertex to vertex $v$ without taking the weight of $v$ into account.

**Definition 16.**

$\forall \sigma \in V$, the bottom level of $v$, $bl(v)$, is the path of maximum weight from $v$ to an exit vertex with taking the weight of $v$ into account.

$$tl(v) = \max_{u \in Pred(v)} (tl(u) + w(u)) \tag{7.1}$$

if $v$ is not an entry point, else $tl(v) = 0$.

$$bl(v) = \max_{u \in Succ(v)} (bl(u) + w(v)) \tag{7.2}$$

if $v$ is not an exit point, else $bl(v) = w(v)$.

**Theorem 25.**

Let $G = (V, E, w)$ be a task system. Let $\sigma_{free}$ be defined as $\forall v \in V, \sigma_{free}(v) = tl(v)$
Then $\sigma_{free}$ is a *optimal scheduling*.
$\sigma_{free}$:= as soon as possible (ASAP)

*Proof.* The scheduling is proved by the equation 7.1.

Optimality: each task is scheduled at the lower bound on its beginning time (induction on the vertices beginning with $k$ nodes). □

$MS_{opt}(\infty) = MS(\sigma_{free}, \infty) = \max_{v \in V} \{tl(v) + w(v)\}$

**Corollary 26.**

For $G = (V, E, w)$, $Pb(\infty)$ can be solved in time $\mathcal{O}(|V| + |E|)$.

*Proof.* The scheduling as late as possible (ALAP, or procrastinator): $\sigma_{late}(v) = MS_{opt}(\infty) - bl(v)$

$\sigma_{late}$ is another optimal scheduling. □

# Chapter 8

# Solve $Pb(p)$

## 8.1 NP-completeness of $Pb(p)$

**Definition 17.**

Knowing the tasks given by $G = (V, E, w)$, the number of processors $p \geqslant 1$ and a bound $k \in \mathbb{N}^*$, the problem $Dec(p)$ is: does it exist a scheduling $\sigma$ of $G$ using at most $p$ processors such that $MS(\sigma, p) \leqslant k$.

When the tasks are independent $E = \emptyset$, the problem is named $Indep(p)$.

When $p$ is fixed a priori, for instance $p = 2$, we note $Dec(2)$ or $Indep(2)$

**Theorem 27.**

$Indep(2)$ is NP-complete but can be solved by a pseudo polynomial algorithm.
$Indep(p)$ is strongly NP-complete.
$Dec(2)$ is strongly NP-complete.

*Proof.* $Indep(2)$ is exactly 2-partition with $k = \dfrac{\sum\limits_{v \in V} w(v)}{2}$

$Indep(p)$: reduction from 3-partition.

Given $3n$ integers, $a_1, \ldots, a_{3n}$, and a bound $B$, assuming $\frac{B}{4} < a_i \leqslant \frac{B}{2}$ and such that $\sum\limits_{i=1}^{3n} a_i \leqslant nB$.

Is there a partition of $a_i$ in sets $I_1, \ldots, I_n$ such that $\sum\limits_{a_j \leqslant I_i} w(a_j) = 3$ (each containers contains exactly 3 of the $a_i$).

$\forall (i, j), I_i \cap I_j = \emptyset$

The reduction: $p = n, k = 3$. $3n$ tasks and the tasks $T_i$ have a weight $a_i$

$Dec(2)$: reduction from 3-partition.

We have 2 processors. We have:

- $3n$ independent tasks $T_1, \ldots, T_{3n}$ with $w(T_i) = a_i$
- $3n$ other tasks, all of weight $B$

$K = nB$

3-partition has a solution $I_1, \ldots, I_n$. $\sigma$ is defined as follows: Processor $P_1$:

- executes task $X_i$ at time $(i-1)2B$
- executes task $Y_i$ at time $B + (i-1)2B = (2i-1)B$

Processor $P_2$:

- executes the tasks of $I_i$ at time $(i-1)2B$
- executes the task $Z_i$ at time $(2i-1)B$

We assume that ther exists a schedule $X_1 \to Y_1 \to X_2 \to Y_2 \to \cdots \to X_n \to Y_n$. Weight of this path is $2nB = K$. No freedom! Task $X_i$ must be executed at time $2(i-1)B$. Task $Y_i$ must be executed at time $(2i-1)B$. $X_1 \to Z_1 \to X_2 \to Z_2 \to \cdots \to X_n \to Z_n$. task $Z_i$ must be executed at time $(2i-1)B$.

Without loss of generality, we assume that $P_1$ executes all the $X_i$s and $Y_i$s and processor $P_2$ executes all the $Z_i$s and $T_i$s.

Let $I_i$ be the next of tasks $T_j$ that are executed during the time interval $[2(i-1)B, (2i-1)B]$

$$\sum_{T_j \in I} w(T_j) \leqslant B$$

Because $\sigma$ is a schedule, $\bigcup_{i=1}^{n} I_i = \{T_1, \ldots, T_{3n}\} \Rightarrow$ solution to 3-partition.

□

## 8.2 List scheduling heuristics

Principle : do not deliberately let a processor idle (if there is some tasks that can be scheduled).

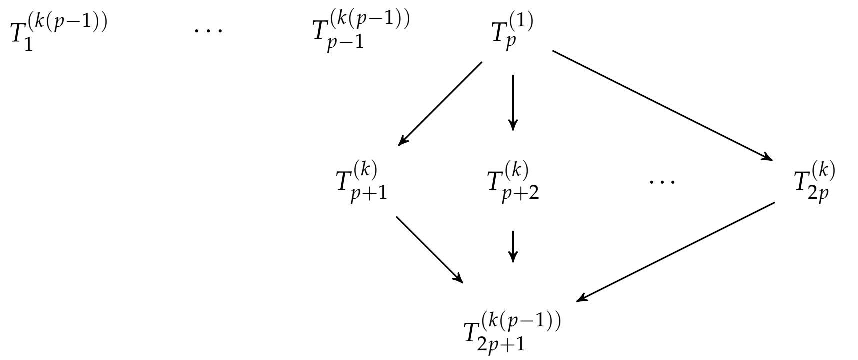
Vocabulary : a task is free at a time $t$ if all its predecessors have completed at time $t$.

**Theorem 28.**

Let $G = (V, E, w)$ be a DAG and assume that there are $p$ processors available. Let $MS_{opt}(p)$ be the optimal makespan when using at most $p$ processors. Then, if $\sigma$ is a list schedule, then $MS(\sigma, p) \leqslant (2 - \frac{1}{p})MS_{opt}(p)$

**Lemma 29.**

There exists a path $\Phi$ in $G$, such that $Idle \leqslant (p-1)w(\Phi)$




Let $T_{i_1}$ be one task where completion time is equal to the makespan. Let $t_1$ be the largest time, with $t_1 < \sigma(T_{i_1})$, such that at least one processor is idle at time $t_1$. Task $T_{i_1}$ was not started at time $t_1$ besause one of its predecessors (may be $d$ transitive predecessors) is executed at time $t_1$. Let $T_{i_2}$ be such a predecessor. Let $t_2$ be the last time prior to $\sigma(T_{i_2})$ at which the procssor was idle. By induction we build a dependence path $\Phi : T_{ik} \rightarrow T_{i_{k-1}} \rightarrow \cdots \rightarrow T_{i_2} \rightarrow T_{i_1}$

Processors can only be idle while one task of $\Phi$ is executed : when one task of $\Phi$ is executed, at most $(p-1)$ processors are idle. $Idle \leqslant (p-1)w(\Phi)$

$$
\begin{aligned}
p \cdot MS(\sigma, p) &= Idle + Seq \\
&\leqslant (p-1)w(\Phi) + Seq \\
w(\Phi) &\leqslant MS_{opt}(p) \\
\frac{Seq}{p} &\leqslant MS_{opt}(p) \\
p \cdot MS(\sigma, p) &\leqslant (p-1)MS_{opt}(p) + pMS_{opt}(p) \\
&= (2p-1)MS_{opt}(p)
\end{aligned}
$$

**Proposition 30.**

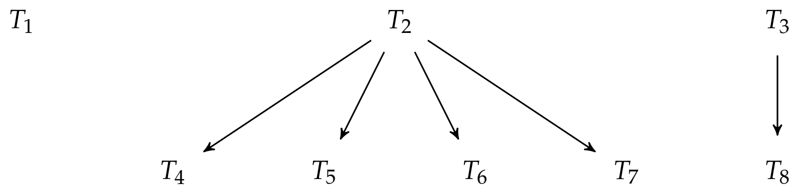
Let $MS_{list}(p)$ be the shortest possible makespan produced by a list scheduling algorithm. Then, the bound $MS_{list}(p) \leqslant (2 - \frac{1}{p})MS_{opt}(p)$ is tight.

$T_1 \ldots T_{p-1}$ are tasks of size $K(p-1)$

Let $\sigma$ be any list schedule. $\sigma$ schedules tasks $T_1 \ldots T_{p-1}$ and $T_p$ at time 0. At time 1, $T_p$ is completed and one processor, say $P_1$, si free. Then $P_1$ processes one task among $T_{p+1} \ldots T_{2p}$ between times 1 and $k+1$, then a second of these tasks between time $k+1$ and $2k+1$. $P_1$ processes $p-1$ of the tasks $T_{p+1}$ to $T_{2p}$ between time 1 and $1 + (p-1)k$ (if we assume $k \geqslant 2$). At time $k(p-1)$ all processors except $P_1$ are available, one of them computes the last of the $T_{p+1} \ldots T_{2p}$ tasks. All these tasks are completed at time $kp$. Task $T_{2p+1}$ is processed between times $kp$ and $k(2p-1)$.

#### Optimal schedule

- At time 0 : process task $T_p$
- At time 1 : process task $T_{p+1}$ to $T_{2p}$




- At time $k+1$ : process task $T_1$ to $T_{p-1}$ plus $T_{2p+1}$

The makespan is $k+1+k(p-1) = kp+1$ $MS(\sigma, p) = \frac{(2p-1)k}{kp+} MS_{opt}(p)$ on this instance. The quotient $\overset{k\to\infty}{\to} 2 - \frac{1}{p}$

## 8.3 Critical path scheduling

Intuition: the largest is the bottom-level of a task, the most urgent is the task.

Critical path scheduling: schedule tasks by non-increasing bottom-levels (break ties arbitrarily).

# Chapter 9

# Taking communications into account

**Classical model : macro-dataflow** Two tasks T and T'. Cos(T, T') = 0 if alloc(T) = alloc(T'), $c(T, T')$ otherwise (independent of the choice of the two processors)

Assumptions:

- communication can start as early as task are completed
- no contention for network links*

A schedule must satisfy dependence constraints: if $(T, T') \in E$ if $alloc(T) = alloc(T')$, $\sigma(T') \geqslant \sigma(T) + w(t)$ and, otherwise, $\sigma(T') \geqslant \sigma(T) + w(T) + c(T, T')$

*`https://en.wikipedia.org/wiki/Contention_(telecommunications)`

# Chapter 10

# $Pb(\infty)$ with communications

**Sequential case** $MS_{opt}(1) = 13$. Assume we schedule each task on a different processor.

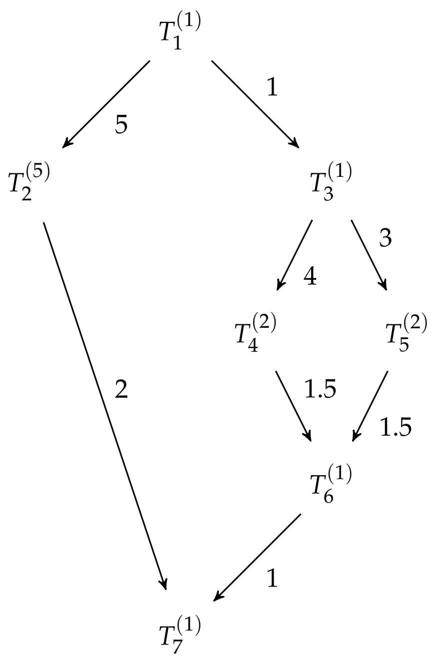



Figure 10.1:

**Theorem 31.**

$Pb(\infty)$ with communications is NP-complete.

Reduction from 2-partition. Let $a_1, \ldots, a_n$ be an instance of 2-partition with $\sum_{i=1}^{n} a_i = \alpha$ We build the following instance of $Pb(\infty)$ 10.2. All communications have a length of C : $C(T_0, T_i) = C(T_i, T_{i+1}) = C$ for $1 \leqslant i \leqslant n$ $w(T_0) = w(T_{n+1}) = A$ For $1 \leqslant i \leqslant n$, $w(T_i) = 2a_i$ $C$ is an integer in the interval $]\alpha - \min_{1 \leqslant i \leqslant n} 2a_i, \alpha[$ -> interval of length at least 2; it contains at least one integer. $k = 2A + \alpha + C$ Is there a schedule whose makespan is $\leqslant \alpha$ ?

$$
\begin{tikzcd}
 & T_0 \arrow[dl] \arrow[dr] & \\
T_1 \arrow[dr] & \cdots & T_n \arrow[dl] \\
 & T_{n+1} &
\end{tikzcd}
$$

Figure 10.2:

Let us assume that there is a solution to the partition problem : $\exists I \sum\limits_{i \in I} a_i = \sum\limits_{i \min I} a_i$

Assume that we have a solution to the scheduling problem. Let $\sigma$ be a schedule whose makespan is at most $k = 2A + \alpha + C$

Tasks $T_0$ and $T_{n+1}$ are not executed on the same processor. Let us assume that $T_0$ and $T_{n+1}$ are executed on the same processor. Let $T_{i_0}$ be a task not executed on the same processor than $T_0$ and (or ?) $T_{n+1}$.

$$
\begin{aligned}
Makespan &\geqslant w(T_0) + c(T_0, T_{i_0}) + w(T_{i_0}) + c(T_{i_0}, T_{n+1}) + w(T_{n+1}) \\
&= A + C + w(T_{i_0}) + C + A \\
&= 2A + C + C + w(T_{i_0}) \\
&= k - \alpha + C + w(T_{i_0}) \\
&\geqslant k - \alpha + C + \min_{1 \leqslant i \leqslant n} 2a_i
\end{aligned}
$$

By definition, $C > \alpha - \min\limits_{1 \leqslant i \leqslant n} 2a_i$ $k - \alpha + C + \min\limits_{1 \leqslant i \leqslant n} 2a_i > k$

Thus, if $T_0$ and $T_{n+1}$ are executed on the same processor, all tasks are executed on the same processor.

$$
\begin{aligned}
Makespan &= \sum_{i=1}^{n+1} w(T_i) \\
&= 2A + 2\alpha \\
&= k + \alpha - C
\end{aligned}
$$

But, by definition, $C < \alpha$. Contradiction. Therefore, $T_0$ and $T_{n+1}$ are executed on two different procs.

Any task is executed either on the same processor that $T_0$ on that $T_{n+1}$. Otherwise, let $T_{i_1}$ be such a task.

$$
Makespan \leqslant w(T_0) + c(T_0, T_{i_1}) + w(T_{i_1}) + c(T_{i_1}, T_{n+1}) + w(T_{n+1})
$$
$$
\geqslant 2A + 2C + \min_{1 \leqslant i \leqslant n} 2a_i > k
$$

Let I be the set of tasks among $T_1 \dots T_n$ that are executed on the same processor than $T_0$ $\sigma(T_{n+1}) = \max(A + w(I) + C, A + C + 2d - w(I))$ The schedule satisfies the bound : $k = 2A + \alpha + C \geqslant A + \sigma(T_{n+1}) \geqslant 2A + C + w(I), 2A + C + 2\alpha - w(I)$ $\alpha \geqslant w(I)$ $w(I) \geqslant \alpha \Rightarrow w(I) = \alpha$

# Chapter 11

# List of heuristics for $Pb(\infty)$ with communications

How to take the notion of communications into account when defining critical paths. Conservative assumption considers that all communications are going to take place when computing the critical paths.

## 11.1 Naive critical path

$p = 3$ processors

## 11.2 Modified critical path scheduling

Map a task on the processor that enables to start it at the earliest.

**Definition 18.**

Let $G = (V, E, c, w)$ be a communication DAG. The granularity of $G$ is the computation to communication ratio of $G$

$$g(G) = \frac{\min\limits_{T \in V} w(T)}{\max\limits_{T,T'} c(T, T')}$$

$G$ is said to be coarse-grained if $g(G) \geqslant 1$.

**Theorem 32.**

Let $G = (V, E, c, w)$ be a communication DAG. Let $MS_{opt}(p)$be the optimal makespan. Then we can derive a schedule $\sigma$ whose makespan satisfies

$$MS(\sigma, p) \leqslant \left(2 - \frac{1}{p}\right)\left(1 + \frac{1}{g(G)}\right) MS_{opt}(p)$$

if $g(G) > 0$.

Let $\sigma$ be any list schedule for $G$ when communications are not taken into account.

$$\overbrace{MS^{wc}(\sigma,p)}^{\text{without communication time}} \qquad \leqslant \left(2-\frac{1}{p}\right) MS^{wc}_{opt}(p)$$

## 11.3 Two Step clustering heuristics

1. Cluster the tasks (or partition them) as if an infinite number of processors was available.
2. Map the clusters on the processors (number of clusters at the end of step 1 can be dirrent than the number of processors).

Motivation : "if tasks are scheduled on the same processor, on the best possible architecture with unbounded number of processors, then that should be scheduled on the same processor in any other architecture"

## 11.4 KIM and BROWNE's linear clustering

1. Initially all edges are marked unexamined. Initial clustering $\mathcal{C}_0$ where each tasks is in a different cluster.
2. For each tasks $v$, compute, $bl(b,\mathcal{C}_i)$ and $tl(b,\mathcal{C}_i)$. Select a longest dependence path in the graph. Build $\mathcal{C}_{i+1}$ by grouping all tasks of the chosen dependence path. Mark as examined all the edges incident to these tasks.
3. While there remain unexamined edges in the graph, go to step 2.

Estimated parallel time for a clustering

$$EPT(\mathcal{C}) = \max\{tl(v) + bl(v) \mid v \in V\}$$

A new clustering is accepted only if it does not (strictly) increase the $EPT$.

SARKAR's greedy clustering.

1. Sort the edges by non-increasing communication costs
2. For each edge in this order, zero out the edge (merge the cluster containing the incident tasks) if the $EPT$ does not increase.

## 11.5 Dominant sequence clustering

At each step, try to zero out one edge of the longest path (ie. dominant sequence).

1. Initially all the edges are marked unexamined. Initial clustering $\mathcal{C}_0$ compute the bottom and top levels of each task $EPT(\mathcal{C}_0)$ and a dominant sequence $DS$.
2. While (there remain unexamined edges)
   - Pick an unexamined edge of $DS_i$.
   - zero out the edge if the $EPT$ does not increase.
   - Mark this edge as examined.
   - Compute the top and the bottom levels, $EPT(\mathcal{C}_{i+1})$, $DS_{i+1}$.

### 11.5.1 From clustering to scheduling

1. Cluster the tasks
2. (a) Assign clusters to processors
   (b) Order the execution of the tasks on the processors.

### 11.5.2 Cluster assignment

1. Compute the load ($\sum$ of weight of tasks) of each cluster.
2. Assign clusters by non increasing weight, to the best leaded processor/in a round robin way.

### 11.5.3 Final task scheduling

- Use in some way the critical paths (knowing which communications are going to take place).
- – At each step assign the free tasks of highest priority.
  – At each step start the ready tasks of highest priority (a task is ready when all its incomming communications have completed.)

# Part IV

# To go further

# Chapter 12

# Automatic parallelization: LAMPORT's hyperplane method

Ideal goal:

- Take an existing sequential program,
- Automatic analysis and transformation to obtain a parallel program that executes effectively on the target parallel platform.

## 12.1 Uniform loop nests

$$
\begin{aligned}
S_1 &: a \leftarrow b + 1 \\
S_2 &: b \leftarrow a - 1 \\
S_3 &: a \leftarrow c - 2 \\
S_4 &: d \leftarrow c
\end{aligned}
$$

We assume we deal with unaliased variables

$<_{text}$: textual order.

$<_{seq}$: sequential order.

Dependence analysis: find out which statements are dependent and which are independent, in order to determine which statements can be executed in parallel.

The output dependence from $S_1$ to $S_3$ is already obtained through transitive closure of $S_1 \xrightarrow{flow} S_2$ and $S_2 \xrightarrow{anti} S_3$

No dependences related to $S_4$ which is independent of the other statements.

**for** $i = 0$ **to** $N$ **do**
  **for** $j = 0$ **to** $N$ **do**
    $S_1(i,j) : a(i,j) = b(i,j-6) + d(i-1,j+3)$
    $S_2(i,j) : b(i+1,j-1) = c(i+2,j+5) + 1$
    $S_3(i,j) : c(i+3,j-1) = a(i,j+2)$
    $S_4(i,j) : d(i,j-1) = a(i,j-1) - 1$

A set of for loops and some statements. The loops are perfectly nested: each statement is surrounded by all the loop.

The nested loops surround a set of statements. An operation is the execution of a statement for a given values of the loop indices, that is for an iteration of the loops. The iteration vector is the vector of loop indices, here $\begin{pmatrix} i \\ j \end{pmatrix}$.

Iteration domain, set of values of the iteration vector.

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| 0 \leqslant i, j \leqslant N \right\}$$

Let $I$ be the iteration vector.

$$I = \begin{pmatrix} i \\ j \end{pmatrix}$$

The operations $S_k(I)$ are ordered in the loop nest by the $<_{seq}$ order.

$$S(I) <_{seq} T(J) \Leftrightarrow I <_{lex} J \vee (I = J \wedge S <_{text} T)$$

There is a dependence from $S(I)$ to $T(J)$ (ie. $T(J)$ depends on $S(I)$) if

- $S(I) <_{seq} T(J)$,
- Both $S(I)$ and $T(J)$ access the same memory location $M$, and at least one of the accesses is a write,
- The memory location $M$ was not written between $S(I)$ and $T(J)$.

$$S(I) \longrightarrow T(J)$$

Dependence vector:

$$d_{S,I,T,J} = J - I$$

The loop nest is said to be uniform if $d_{S,I,T,J}$ is independent of $I$ and of $J$. We then denote it $d_{S,T}$. All dependence vectors are lexicographically non negative (if all loop increments are positive).

$S_1(i, j)$ writes $a(i, j)$, $S_4(i, j)$ reads $a(i, j-1)$, $S_4(i, j+1)$ reads $a(i, j)$.

There is a flow dependence from $S_1(i, j)$ to $S_4(i, j+1)$. Dependence vector is $\begin{pmatrix} i \\ j+1 \end{pmatrix} - \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$: constant independent of $i$ and $j$: uniform dependence

Formally: Consider statements $S_1$ and $S_3$ that both access array $a$.

The question is: "Is there a dependence from $S_1$ to $F_3$ (because of array $a$)?". Is there an iteration vector $I$ and an iteration vector $J$, such that $I \leqslant_{lex} J$ and such that

$$a(i, j) = a(i', j' + 2)$$

where $I = \begin{pmatrix} i \\ j \end{pmatrix}$ and $J = \begin{pmatrix} i' \\ j' \end{pmatrix}$

$$\begin{pmatrix} i \\ j \end{pmatrix} \leqslant_{lex} \begin{pmatrix} i' \\ j' \end{pmatrix}$$

with $i = i'$ and $j = j' + 2$

$$\begin{pmatrix} i \\ j \end{pmatrix} \leqslant_{lex} \begin{pmatrix} i \\ j-2 \end{pmatrix}$$

which is impossible.

There does not exist any flows dependence for $S_1$ to $S_3$.

Is there a dependence from $S_3$ to $S_1$ because of a

$$I = \binom{i}{j} \leqslant_{lex} J = \binom{i'}{j'}$$

such that $S_3(i,j)$ access the same memory location than $S_1(i',j') \Leftrightarrow$

$$\binom{i}{j+2} \leqslant_{lex} \binom{i'}{j'}$$

$$\left\{ \begin{array}{lcl} i' & = & i \\ j' & = & j+2 \end{array} \right. \implies \binom{i}{j} \leqslant_{lex} \binom{i'}{j'}$$

There is an anti-dependence. from $S_3$ to $S_1$ because of array $a$, of dependence vector

$$\binom{i'}{j'} - \binom{i}{j} = \binom{i}{j+2} - \binom{i}{j} = \binom{0}{2}$$

uniform dependence

Dependence form $S_2$ to $S_1$ (flow dependence on $b$).

$$I = \binom{i}{j} \leqslant_{lex} J = \binom{i'}{j'}$$

and $S_2(I)$ and $S_1(J)$ acessing the same location

$$\binom{i+1}{j-1} = \binom{i'}{j'-6}$$

with

$$\begin{cases} i' = i+1 \\ j' = j+5 \end{cases}$$

$S_2 \overset{flow}{\longrightarrow} S_1$ of dependence vector $\frac{1}{5}$.

Dependence from $S_4$ to $S_1$(flow on $d$)

$$I = \binom{i}{j} \leqslant_{lex} J = \binom{i'}{j'}$$

$$\binom{i}{j-1} = \binom{i'-1}{j'+3} \Leftrightarrow \begin{cases} i' = i+1 \\ j' = j-4 \end{cases}$$

$S_4 \overset{flow}{\longrightarrow} S_1$ of dependence vector $\binom{1}{-4}$.

Dependence from $S_3$ to $S_2$ (flow on $c$)

$$\binom{i+3}{j-1} = \binom{i'+2}{j'+5} \Leftrightarrow \begin{cases} i' = i+1 \\ j' = j-6 \end{cases}$$

$S_4 \overset{flow}{\longrightarrow} S_1$ of dependence vector $\binom{1}{-6}$.

## 12.2 LAMPORT's hyperplane method

Let $Dom$ = the iteration domain.

Let $D$ be the matrix of dependence vector in non decreasing lexicographic order.

Let $p_1 \in Dom$ and $p_2 \in Dom$. We write $p_1 \prec p_2$ if $\exists \alpha \in D$ such that $p_2 = p_1 + \alpha$.

Let $\sigma$ be a schedule $\sigma : Dom \to \mathbb{Z}$. The schedule must satisfy the dependences.

$$\forall (p_1, p_2) \in Dom^2, p_1 \prec p_2 \Rightarrow \sigma(p_1) < \sigma(p_2)$$

Each iteration takes a unitary time.

The makespan of the schedule

$$T_\sigma = 1 + \max_{p \in Dom} \sigma(p) - \min_{p \in Dom} \sigma(p)$$

We assume we have an unlimited number of processors.

We could compute the free schedule by a traversal of the extended dependence graph, which is of size $\Omega\left(N^2\right)$. This could be as expensive in order of magnitude as executing the original sequential loop nest. We cant a compact schedule, computed once and for all values of N.

We restrict ourselves to linear schedules.

A linear schedule $\sigma_\pi$ is defined by a vector $\pi$

$$\begin{aligned} \sigma_\pi : Dom &\to \mathbb{Z} \\ p &\mapsto \sigma_\pi(p) = \lfloor \pi \cdot p \rfloor \end{aligned}$$

---

**for** $time = time_{min}$ **to** $time_{max}$ **do**
  **for** *all* $p \in E(time)$ *in parallel* **do**
    $S_1(p)$
    $\vdots$
    $S_k(p)$

---

where

$$E(time) = \{p \in Dom | \lfloor \pi \cdot p \rfloor = time\}$$

$$p_1 \prec p_2 \Rightarrow \sigma_\pi(p_1) < \sigma_\pi(p_2)$$

$p_2 = p_1 + d$

$$\begin{aligned} \sigma_\pi(p_1) < \sigma_\pi(p_2) &\Leftrightarrow \lfloor \pi \cdot p_1 \rfloor < \lfloor \pi \cdot p_2 \rfloor \\ &\Leftrightarrow \lfloor \pi \cdot p_1 \rfloor + 1 \leqslant \lfloor \pi \cdot p_2 \rfloor \\ &\Leftrightarrow \lfloor \pi \cdot p_1 \rfloor + 1 \leqslant \lfloor \pi \cdot (p_1 + d) \rfloor \\ &\Leftarrow 1 < \pi \cdot d \end{aligned}$$

$$\begin{aligned} \pi \cdot d \geqslant 1 &\Rightarrow \pi d + \pi p_1 \geqslant 1 + \pi p_1 \\ &\Rightarrow \pi d + \pi p_1 \geqslant \lfloor 1 + \pi p_1 \rfloor \\ &\Rightarrow \lfloor \pi (p_1 + d) \rfloor \geqslant \lfloor \pi p_1 \rfloor + 1 \end{aligned}$$

LAMPORT's condition for the validity of a schedule.

$$\pi \cdot D \geqslant 1 \Leftrightarrow \forall d \in D, \pi \cdot d \geqslant 1$$

Let $k_1$ be the index of the first non null component of $d_1$. We let $\pi_{k_1+1} = \ldots = \pi_n = 0/$ We let $\pi_{k_1} = 1 \rightarrow \pi d_1 \geqslant 1$ Let $d_a$ be the first vector of $D$ whose first non null component is of rang $k_2 < k_1$. Let $\pi_{k+2} + 1 = \ldots = \pi_{k_1-1} = 0$.

We let $D_2 = \{d \in D | \text{the first non null component of } d \text{ is } k_2\}$. We want to have $\forall d \in D_2, \pi \cdot d \geqslant 1$.

$$\forall d \in D_2, \pi_{k_2} \geqslant 1 - \pi_{\substack{k_1+1\\ \vdots \\ n}} d_{\substack{k_2+1\\ \vdots \\ n}}$$

$$D = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 2 & -6 & -4 & 5 \end{pmatrix}$$

$$\pi = \begin{pmatrix} a \\ b \end{pmatrix}$$

$$\left\{ \begin{array}{rcl} b & \geqslant & 1 \\ 2b & \geqslant & 1 \end{array} \right. \Rightarrow b \geqslant 1$$

$$\left\{ \begin{array}{rcl} a-6 & \geqslant & 1 \\ a-4 & \geqslant & 1 \\ a+5 & \leqslant & 1 \end{array} \right. \Leftrightarrow \left\{ \begin{array}{rcl} a & \geqslant & 7 \\ a & \geqslant & 5 \\ a & \geqslant & -4 \end{array} \right. \Leftrightarrow a \geqslant 7$$

$$\pi = \begin{pmatrix} 7 \\ 1 \end{pmatrix}$$

is a solution

How to find a good solution?

$$T_{\sigma_\pi} = 1 + \max_{p \in Dom} \lfloor \pi p \rfloor - \min_{p \in Dom} \lfloor \pi p \rfloor$$

$$\left\{ \begin{array}{rcl} a-6b & \geqslant & 1 \\ a-4b & \geqslant & 1 \\ a+5b & \geqslant & 1 \end{array} \right. \Leftrightarrow \left\{ \begin{array}{rcl} a & \geqslant & 1+6b \\ a & \geqslant & 1+4b \\ a & \geqslant & 1-5b \end{array} \right. \Leftrightarrow \left\{ \begin{array}{rcl} b & \geqslant & 1 \\ a & \geqslant & 1+6b \end{array} \right.$$

$\Leftarrow a, b$ are both non negative

$$\min_{p \in Dom} \lfloor \pi p \rfloor = 0$$

$$\max_{p \in Dom} \lfloor \pi p \rfloor = \lfloor aN + bN \rfloor$$

This is minimized for $a = 7$ and $b = 1$.

$$\pi = \begin{pmatrix} 7 \\ 1 \end{pmatrix}$$

time = $7i + j$

$$\begin{pmatrix} time \\ proc \end{pmatrix} = \begin{pmatrix} 7 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

unimodular matrix = integral matrix of determinant 1 or $-1$.

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -7 \end{pmatrix} \begin{pmatrix} time \\ proc \end{pmatrix}$$

$i = proc$

$0 \leqslant proc \leqslant N$

$j = time - 7proc$ $(0 \leqslant j \leqslant N)$

$$\left\lceil \frac{time - N}{7} \right\rceil \leqslant proc \leqslant \left\lfloor \frac{time}{7} \right\rfloor$$

$time = 7i + j$

$0 \leqslant time \leqslant 8N$

---

**for** $time = 0$ **to** $8N$ **do**
    **for** $proc = \max(0, \left\lceil \frac{time-N}{7} \right\rceil)$ **to** $\min(N, \left\lfloor \frac{time}{7} \right\rfloor)$ **do**
        $a(proc, time - 7proc) = b(proc, time - 7proc - 6) + d(proc - 1, time - 7proc + 3)$
        $\vdots$

---

# Chapter 13

# Algorithms for GPUs – An introduction

cf. ASR-1

GPU: graphical processing unit

GPGPU: graphical purpose GPU.

Motivation for a change in architecture

- memory wall: what is dictating the performance of an application is the cost of data movement and not the CPU processing power,
- energy wall: most powerful supercomputer on top 500 is using 26MW power for a performance of $\simeq$30 pétaflops. Objective 2018-2020:
  - exaflop ($\times$30),
  - energy budget $\leqslant$ 80MW.

Evolution:

- More parallelism. Single processor $\rightarrow$ multicore $\rightarrow$ manycore
- specialized hardware: GPU.

First underlying principle of GPUs simplification:

- Suppress big data cache,
- Suppress optimizing hardware: branch prediction etc..

Consequence: use many simple computing units in parallel.

The same operation is applied to different data at once.

There also exists in scientific computing: addition of vectors, matrix multiplication etc..: data parallelism.

Second principle: SIMD processing. SIMD: simple instruction many data.

SIMD processing $\neq$ SIMD instructions.

Data parallelism is not necessarily made explicitly in the program.

Chapter 14

# Processeurs hétérogènes

# Appendix A

# Introduction à MPI

## A.1 Parallel architectures

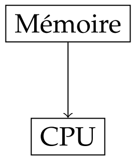



Figure A.1: Architecture séquentielle

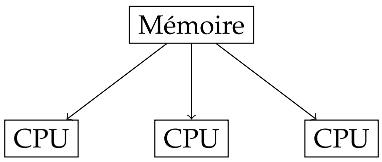



Figure A.2: Mémoire partagée

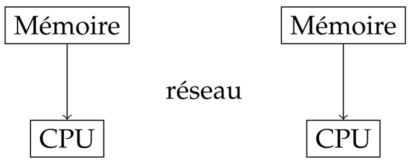



Figure A.3: Mémoire distribuée

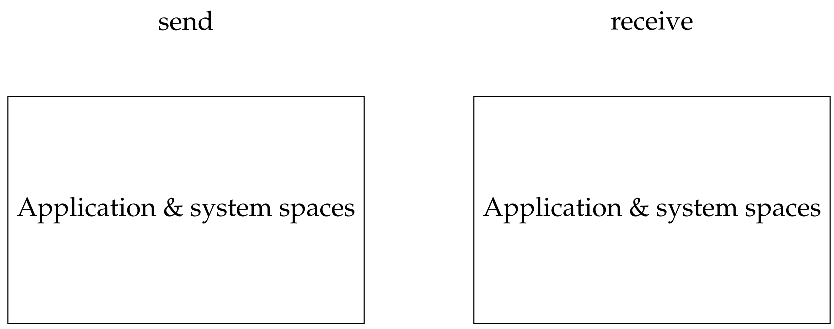
We also can have a hybrid version between shared and distributed memory.

**MPI** Message passing interface. It is not a program neither a library, It is a specification which describe to the programmer how to communicate with other process.

MPI is used by including the library MPI, writing sequential code, and, between the two functions `MPI_init` and `MPI_finalize`, the code is executed by each process.

## A.2 Blocking and non blocking operations

Par exemple, imaginons qu'on ait 2 processus:




Dans un monde idéal, quand une information est envoyée, elle est reçue (communication synchrone). Or on peut avoir des pertes ou des canaux bloquants. Comme on veut éviter de mettre le processeur en attente, on met en place une communication asynchrone. On a donc des opérations bloquantes et non bloquantes : elles indiquent si le contenu du buffer peut être altéré par de nouvelles informations (bloquantes) ou non (non bloquantes).

On a donc 4 versions d'une même fonction (bloquante ou non, synchrone ou non): Send, Ssend, ISend, ISsend.

On peut de plus dire à MPI de placer le buffer système dans l'espace d'application pour pouvoir y toucher avec plus de précision. Ce qui rajoute entre 4 versions supplémentaires d'une fonction: Bsend, IBsend, Rsend, IRsend.

Dans la pratique, le nom des fonctions MPI commence par `MPI_`.

Avant de parler de la forme générale, nous allons nous examiner trois fonctions : wait, test (any, all, some) et prob. Qui traitent des opérations non bloquantes. Elles servent à vérifier si un message est arrivé ou non: par exemple Iprob pour effectuer d'autres calculs si le message n'est pas arrivé. Sinon prob permet d'attendre que le message est arrivé.

Si on veut vérifier si la mémoire dans laquelle a été stockée le message est sure pour pouvoir y écrire de nouvelles informations, on appelle test. Finalement wait attends tant que les opérations non bloquantes n'ont pas été effectuées.

## A.3 MPI Functions Architecture

La fonction Isend : `MPI_Isend(&msg, nb_elt, size, dest, tag, comm, &req)`

tag : un identifiant du message.
comm : le communicateur, il désigne un groupe de processus MPI, où chaque process a un identifiant. Quand un programme démarre, il y a un communicateur : `MPI_Comm_World` qui désigne tous les processus. Si on veut subdiviser l'ensemble des processus pour qu'ils travaillent sur des calculs différents, on crée deux communicateurs, où les processus ont leur identifants propres. Cette notion est importante dans le cadre de communication/opérations collectives.
req : permet d'identifier l'opération pour vérifier par la suite si elle a terminée ou non.

Pour les communications collectives, nous avons :

- broadcast
- scatter
- gather
- reduce

`MPI_Comm_size` donne le nombre de processus existants dans le réseau MPI.
`MPI_Comm_rank` permet de récupérer son rang/identifiant.
`MPI_WTime` permet de synchroniser les horloges

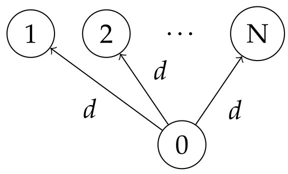



Figure A.4: MPI_Bcast : broadcast

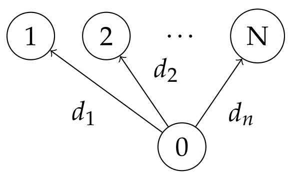



Figure A.5: Scatter

# Bibliography


[CLR08] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. CRC Press, 2008.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.