# Algo et prog parallèles

par Frédéric Vivien

# Contents

# Theoretical models

Partie 1.1

**Sorting networks**

**Question :** How to organize sorting cells to build a sorting networks ?

1.1.1

Odd-Even merge sort

*Merging network*

Arbitrary sequence $< c_1, \ldots, c_n >$, SORT($< c_1, \ldots, c_n >$) sorted sequence.
If $c_1 \leq c_2 \leq \ldots \leq c_1 \leq \ldots \leq c_n$ then SORTED($< c_1, \ldots, c_n >$) is $True$
If SORTED($< a_1, \ldots, a_n >$) and SORTED($< b_1, \ldots, b_n >$) then MERGE($< a_1, \ldots, a_n >, < b_1, \ldots, b_n >$) = $SORT(< a_1, \ldots, a_n >, < b_1, \ldots, b_n >)$

**Proposition :** $A =< a_1, \ldots, a_{2n} >$, $B =< b_1, \ldots, b_{2n} >$, SORTED($A$) = SORTED($B$) = $True$
$< d_1, \ldots, d_{2n} >$= MERGE($< a_1, a_3, \ldots, a_{2n-1} >, < b_1, b_3, \ldots, b_{2n-1} >$)
$< e_1, \ldots, e_{2n} >$= MERGE($< a_2, a_4, \ldots, a_{2n} >, < b_2, b_4, \ldots, b_{2n} >$)
SORTED($d_1, \min(d_2, e_1), max(d_2, e_1), \ldots, \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n}) = True$

**Without loss of generality, all values are distinct (two by two)**

- $d_1$ is the minimum of all value as $d_1 = \min(a_1, b_1)$
- $e_{2n}$ same argument

In the column of comparators we compare $d_i$ and $e_{i-1}$ for $2 \leq i \leq 2n$
We only need to prove that $d_i$ and $e_{i-1}$ should be in the sorted sequence at ranks $2i-2$ and $2i-1$

**Four things to show**

1. $d_i$ dominates $2i-3$ values (at least)
2. $e_i$ dominates $2i-3$ values
3. $d_i$ is dominates by $4n-2i+1$ values
4. $e_i$ is dominates by $4n-2i+1$ values

**Proving 1:** Without loss of generality, assume $d_i$ is an element of $A$. There is some index $k$ such that

$$d_i = a_{2k+1}$$

so $d_i$ dominates $2k-2$ elements of $A$.
In the sequence $d_i, d_2, \ldots d_i$ there are $k$ elements of $A$ and $i-k$ elements of $B$. The last element of $B$ in $d_1, \ldots, d_i$ is $b_{2(i-k)-1}$ so $d_i$ dominates $2(i-k)-1$ elements of $B$
Overall, $d_i$ dominates $(2k-2)+2(i-k)-1 = 2i-3$

Figure 1.1: Merge for two sorted sequences of two values

Figure 1.2: Sequences of length 4

Figure 1.3: Sorting network

**Proving 4:** Let us assume $e_{i-1}$ is an element of $B$. There is a value $k$ such that

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

so $e_{i-1}$ is dominated by $2n - (2k+1) + 1 = 2n - 2k$ values
$e_1, \ldots, e_{i-1}$ : The values of $B$ are $b_2, b_4, \ldots, b_{2k}$. There are $k$ such values. There are $(i-1) - k$ values of $A$.
The rank of the greatest value of $A$ there: $2i - k$
The rank of the first value of $A$ known to dominate $e_{i-1}$: $2_{i-k}$
$e_{i-1}$ is dominated by $a_{2i-k}, a_{2i-k+1}, \ldots, a_{2n}$: $2n - (2i-k) + 1$ values.
Overall, $e_i$ is dominated by $(2n - 2k) + (2n - 2i + 2k + 1) = 4n - 2i + 1$

**Lemma** Let $t_m$ be the processing time and $p_m$ be the numbers of sorting cells of an odd-even merge operator for sequences of size $2^m$

$$\begin{array}{cccccc} t_0 = 1 & t_1 = 2 & t_m = t_{m-1} + 1 & \rightarrow & m+1 \\ p_0 = 1 & p_1 = 3 & p_m = 2p_{m-1} + 2^m - 1 & \rightarrow & 2^m m + 1 \end{array}$$

$n = 2^m$
Time $= \mathcal{O}(\log n)$
Number of comparators $= \mathcal{O}(n \log n)$
Work = Time $\times$ Number of operators $= \mathcal{O}(n \log^2 n)$

---

*Sorting network*

Processing time $t'_m$
Number of comparations $p'_m$

$$\begin{array}{ccc} t'_0 = 1 & t'_m = 4t'_{m-1} + t_m & t'_m = \mathcal{O}(m^2) \\ p'_0 = 1 & p'_m = 2p'_{m-1} + p_m & p'_m = \mathcal{O}(2^m m^2) \end{array}$$

Time $= \mathcal{O}(\log^2 m)$
Number of comparations $= \mathcal{O}(n \log^2 n)$
Work $= \mathcal{O}(n \log^4 n)$

### 1.1.2 Sorting on a one-dimensional network

---

*Odd-even transposition sort*

If $\alpha$ is a sorting network, it can be modeled by a sequence of comparaitors

$$\alpha = [x_k, y_k] \circ [x_{k-1}, y_{k-2}] \circ \ldots \circ [x_1, y_1]$$

where $[a, b]$ outputs $(\min(a, b), \max(a, b))$ and $k$ is the number of comparators. It is a *primitive* sequence of comparators if $\forall 1 \leq i \leq k, y_i = x_{i+1}$

**Lemma :** If $\alpha$ is a primitive sequence of comparators, $\alpha$ is a sorting networks if a sorting network if and only if $\alpha$ sorts $< n, n-1, \ldots, 2, 1 >$

**Proof**

$\Rightarrow$ Trivial

$\Leftarrow$ Proof by contradiction

Figure 1.4: $n = 8$

Figure 1.5: $n = 7$

**Assumption :** there exist an input sequence $x$, two indices $i$ and $j$, $\alpha(x)_i > \alpha(x)_j$
Let $y =< n, n-1, \ldots, 2, 1 >$
Proof by induction on the number $k$ of comparators
*Induction hypothesis*: $H_{i,j,x}(q)$ For any sorting network $\beta$ including $q$ comparators, $\beta(x)_i > \beta(x)_j \Rightarrow \beta(y)_i > \beta(y)_j$

- $k = 0$ : no comparaison, doesn't sort anything.
- We assume the result holds up to a value $k$. Let $\alpha$ be a sequence of $k+1$ comparators

$$\alpha = [p, p+1] \circ \beta$$

  - If $\{i, j\} \cup \{p, p+1\} = \emptyset$
  - If $p = i$
    $\alpha(x)_i = \alpha(x)_p = \min(\beta(x)_p, \beta(x)_{p+1})$. By Hyp $\alpha(x)_i > \alpha(x)_j$
    $\alpha(x)_p \leq \alpha(x)_{p+1} = max(\beta(x)_p, \beta(x)_{p+1})$
    We cannot have $j = p+1 \Rightarrow j > p+1 \Rightarrow \alpha(x)_j = \beta(x)_j$
    1. $\beta(x)_i = \beta(x)_p > \alpha(x)_j = \beta(x)_j$
       We use $H_{i,j,(q-1)} : \beta(y)_i > \beta(y)_j$
    2. $\beta(x)_{i+1} = \beta(x)_{p+1} > \alpha(x)_j = \beta(x)_j$
       We use $H_{i+1,j,q-1} : \beta(y)_{i+1} > \beta(y)_j$

    $\alpha(y)_i = \alpha(y)_p = min(\beta(y)_p, \beta(y)_{p+1}) = min(\beta(y)_p, \beta(y)_{i+1} > \beta(y)_j = \alpha(y)_j$

$$\alpha(y)_i > \alpha(y)_j$$

---

*Odd.even sorting on a one dimensional network*

Input of size $n$

- Set of $p$ processors ($p < n$)
- Each processor receive $\frac{n}{p}$ inputs
- Each processor sorts its inputs $\mathcal{O}(\frac{n}{p} \log \frac{n}{p}) = \mathcal{O}(\frac{n}{p} \log n)$
- We procced with odd-even steps
  At one step, a processor sends its $\frac{n}{p}$ data to its neighbor
  - even step, processor $P_{2i}$ sends it's data to processor $P_{2i+1}$ (and reciprocally)
  - odd step, processor $P_{2i}$ sends it's data to processor $P_{2i-1}$ (and reciprocally)
- all processors merge the sorted lists. Processor $P_{2i}$ keeps the $\frac{n}{p}$ smallest value at an even step, and so on ...

**Overall** $p$ steps of odd-even sorts

$$\mathcal{O}(\frac{n}{p} \log n + p \times \frac{n}{p}) = \mathcal{O}(\frac{n}{p} \log n + n)$$

Figure 1.6: PRAM : Shared memory

Partie 1.2

## PRAM

Parrallel Random Access Machine
Any processor can access any location, all accesses take the same time.

**CRCW :** concurrent read, concurrent write

An unbounded number of processors an read the same memory location

An unbounded number of processors an write at the same memory location

**consistent mode :** processors writing at the same loation must write the same value

**arbitrary mode :** processors can write different values, one of them, chosen arbitrarily (by the machine) is written

**priority mode :** the processor of smallest index is atually writing the value

**fusion mode :** an associative operation ($\max, \times, +, \ldots$) is applied to all values that processor try to write at a given location, and its result is stored

**CREW :** concurrent read, exclusive write

Only one processor can write at a given time at a given location

**EREW :** exclusive read, exclusive write

1.2.1

### Pointer jumping

*List ranking*

**Question :** compute the distance of an element to the end of the list

$$\text{element } i : d[i] = \begin{cases} 0 & \text{si } next[i] = NIL \\ 1 + d[next[i]] & \text{sinon} \end{cases}$$

*Underlying hupothesis : we have an unbaunded number of processors available.*
We assume that there is one element of the linked list by processor

```
Rank_computation(L)
    forall i in parallel do
        if next[i] = NIL
            then d[i]<-0
            else d[i]<-1
    while there exists at least one element i such as next[i] != NIL do
        forall i in parallel do
            if next[i] != NIL do
                d[i] <- d[i] + d[next[i]]
                next[i] <- next[next[i]]
```

1.2.2

### Performance evaluation of PRAM algorithms

*Cost, Work, Speed-up, Efficiency*

Let $P$ be a problem of size $n$
Let $T_{seq}(n)$ be the execution time of the best sequential algorithm
We have a PRAM algorithm that runs in $T_{par}(n)$

**Cost of a PRAM :**

$$C_p(n) = p \times T_{par}(p, n)$$

**Work :** sum over all processing units of the number of performed operations $W_p(n)$

$$W_p(n) \leq C_p(n)$$

$$C_p(n) - W_p(n) = \sum \text{idle times of processors}$$

**Speed-up :**

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(n)}$$

**Efficiency :**

$$e_p = \frac{S_p(n)}{p} = \frac{T_{seq}(n)}{p \times T_{par}(n)}$$

---

*A simple simulation theorem*

**Question:** We have an algorithm designed to run on $p$ processors, what can we say if we have $p' < p$ processors ?

**Theorem:** If $A$ is an algorithm whose execution time is $t$ on $p$ processors for a certain PRAM model, then $A$ can be simulated on a number $p' < p$ of processors, on the same model od PRAM in time $\mathcal{O}(\frac{p}{p'}t)$. The cost of the algorithm on $p'$ processors is at most twice the cost on $p$ processors.

**Proof:** We can simulate each step. One step can be executed in time

$$\lceil \frac{p}{p'} \rceil$$

**Cost:**

$$C(p') = \lceil \frac{p}{p'} \rceil t \times p' \leq (\frac{p}{p'} + 1)p't = pt + p't \leq 2pt = 2C(p)$$

**Computation of prefixes:** Set of values $x_1, x_2, \ldots, x_n$

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

Can be realized in time $\mathcal{O}(\log n)$ on $n$ processors using pointer jumping

**Simulation theorem:** $\frac{n}{p} \log n$

We give $\frac{n}{p}$ input to each processor. First processor has first $\frac{n}{p}$ values. Each processor computs the prefix of its $\frac{n}{p}$ values.

$$p_1 : x_1 \otimes x_2 \otimes \ldots \otimes x_{\frac{n}{p}}$$

Executetime: $\frac{n}{p}$

We apply the pointer-jumping scheme. Execution time : $\log(p)$.

Local propagation step in time $\frac{n}{p}$

$$p_2 : x_{\frac{n+1}{p}} \otimes x_{\frac{n+2}{p}} \otimes \ldots \otimes x_{\frac{2n}{p}}$$

First step: $x_{\frac{n+1}{p}}, x_{\frac{n+1}{p}} \otimes x_{\frac{n+2}{p}}, \ldots, x_{\frac{n+1}{p}} \otimes \ldots \otimes x_{\frac{2n}{p}}$

Pointer jumping: $x_1 \otimes \ldots \otimes x_{\frac{n}{p}}$

Last step: $x_1 \otimes \ldots \otimes x_{\frac{n+1}{p}} \ldots x_1 \ldots x_{\frac{2n}{p}}$

$$\mathcal{O}(\frac{n}{p} + \log p)$$

---

*Brent's theorem*

**Theorem:** Let $A$ be an algorithm that executes a total $m$ of operations and that runs in time $t$ on a certain PRAM model. $A$ can be simulated on a same PRAM with $p$ processors in time $\mathcal{O}(\frac{m}{p} + t)$

**Proof:** $A$ was working in $t$ steps.
At step $i$, $A$ was performing $m(i)$ operations

$$\sum_{i=1}^{t} m(i) = m$$

The $m(i)$ operations can be simulated in time $\lceil \frac{m(i)}{p} \rceil$ on $p$ processors.
Overall running time :

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

**Maximum over $n$ values:** Organize values under a binary tree : time is $\mathcal{O}(\log n)$
First step: $\frac{n}{2}$ comparaisons done in parallel $\Rightarrow \mathcal{O}(n)$ processors.
Number of operations: $n - 1(\mathcal{O}(n))$
$p$ processors, execution time is $\mathcal{O}(\frac{n}{p} + \log n)$ where $p = \frac{n}{\log n}$
Execution time :

$$\mathcal{O}(\log n)$$

### 1.2.3 Comparaison of PRAM models

*Model separation*

Is a CRCW PRAM more powerful then a CREW PRAM ?
Is there a problem $P$ and an algorithme $A$ designed for a CRCW PRAM such that any algorithm on a CREW PRAM is slower ?
We take a CRCW PRAM under fusion model with the maximum as the fusion operator.
We have $n$ inputs, $n$ processing units, all of them writing to the same memory location: in $\mathcal{O}(1)$ we have the maximum over $n$ numbers.
Fusion of $n$ values cost $\Omega(\log n)$ on a CREW.
CRECW PRAM in constant mode : $n \times (n-1)$ processors

```
Initialisation max[i] <- TRUE
Forall i,i <- [1,n]², i != j in parallel do
    if A[i] < A[j] then
        max[i] <- FALSE
done
Forall i in [1,n] in parallel do
    if max[i] = TRUE then
        return A[i]
done
```

**What about CREW versus EREW PRAMS ?** A set of $n$ distinct values $e_1, \ldots, e_n$
Another value $x$. Does $x$ belong to the set $e_1, \ldots, e_n$ ?

**CREW:** All processors are reading $x$ in parallel

```
Found <- FALSE
Forall i in parallel do
    if x = e[i] then
        Found <- TRUE
done
```

**EREW:** Only one processor can read from a memory cell at a time step, at each step we at most double the number of processors to whom $x$ is available $\Rightarrow \Omega(\log n)$

*Simulation theorem*

**Theorem:** Any CRCW algorithm with $p$ processing units has an execution time at most $\mathcal{O}(\log p)$ lower then the times the best algorithm for an EREW PRAM using $p$ processors.

**Proof:** We assume a CREW PRAM under the consistant mode.
We look at one step of the algorithm. We transform the step to remove any concurrent writes.

**Initially:** (under the CRCW PRAM)
processor $P_i$ was writing value $v_i$ at adress $a_i$
we take a temporary array $T$
Processor $P_i$ writes $(ai, v_i)$ in $T[i]$
We sort the entries of $T[i]$ according to the first value in the couples.

**Assumption:** There exists a sorting algorithm able to sort $p$ values in time $\mathcal{O}(\log p)$ using $\mathcal{O}(p)$ processors on a EREW PRAM.

**After the sorting:** $T[1]$ contains $(x, y)$ where $x$ is the adress and $y$ the value.
$A'$ writes $y$ at adress $x$

```
Forall i in [2,n] in parallel do
    if first(T[i]) != first(T[i-1]) then
        A' writes second(T[i]) at adress first(T[i])
done
```

1.2.4

## Cole's sorting machine

Based on the merge sort.

**Target:** $\mathcal{O}(p)$ processors, $\mathcal{O}(\log p)$ time.
$p$ values $\Rightarrow \mathcal{O}(\log p)$ steps.
We have $\mathcal{O}(1)$ time to perform one step of the algorithm, that is merging two sorted lists (int he worst case of size $\frac{p}{2}$ each).

*Merge*

**Rank:** The rank of $x$ in the set $J$

$$rank(x, J) = card(j \in J | j < x)$$

Cross-rank of $A$ in $B$, $R[A, B]$.

$$A \to \mathcal{N}$$
$$x \mapsto rank(x, B)$$

**Good sampler:** A sequence $L$ is said to be a good sampler (GS) of a sequence $J$ if, for any $k \geq 1$, there are at most $2k + 1$ elements of $J$ between $k + 1$ (arbitrary) consecutive elements of $L \cup \{-\infty\} \cup \{+\infty\}$. We say that $a$ is between $x$ and $y$ (with $x < y$) if $x < a \leq y$.
$J = [2, 3, 7, 8, 10, 14, 15, 17, 18, 21]$
$K = [1, 4, 6, 9, 11, 12, 13, 16, 19, 20]$
$L = [5, 10, 12, 17]$ is a good sampler of $J$ and $K$

| | | |
|---|---|---|
| $J(1) : [2, 3]$ | $K(1) : [1, 4] \to$ | $[1, 2, 3, 4]$ |
| $J(2) : [7, 8, 10]$ | $K(2) : [6, 9] \to$ | $[6, 7, 8, 9, 10]$ |
| $J(3) : []$ | $K(3) : [11, 12] \to$ | $[11, 12]$ |
| $J(4) : [11, 15, 17]$ | $K(4) : [13, 16] \to$ | $[13, 14, 15, 16, 17]$ |
| $J(5) : [18, 21]$ | $K(5) : [19, 20] \to$ | $[18, 19, 20, 21]$ |

```
MergeWithHelp(J,K,L)

We partition J and K in |L|+1 subsets
    J[i] = {j in J, L[i-1] < j < L[i]}
    K[i] = {k in K, L[i-1] < l < L[i]}
Forall i in [1, |L|+1] in parallel do
    result[i] <- Merge(J[i], K[i])
done
return result[1].result[2]...result[|L|+1]
```

**Lemma :** if $L$ is a good sampler of the sorted lists $J$ and $K$, and if the crossranks $R[L,J], R[L,K], R[J,L]$ and $R[K,L]$ are known, then merge with help runs in $\mathcal{O}(1)$ on $\mathcal{O}(|J|+|K|)$ processors.

**Step 1 :** Processor $P_j$ takes care of $J_j$.
$P_j$ reads $R[J,L]$ (more precisely reads rank $[J_j, L]$). The value is stores in $J(rank[J_j, L])$. Done in constant time except that we have up to three processors trying to add values to a same subset.
Attached to subsut $J(i)$ we have its number of elements $n_j(i)$

```
Forall i in parallel do
    r <- rank[Ji,L]
    s <- nj(i)++
    resr[s] <- Ji
```

**Step 3 :** `Inputs :` $res_1, \ldots, res_{|L|+1}$ with $|res_i| \leq 3$
`Ouput :` $M[A, \ldots, |L|+1]$

```
Forall i = 1 to |L|+1 in parallel do
    start <- rank(L[i-1],J|K)
    Forall j=1 to |res i| do
        M[start+j] <- res i[j]
```

---

*Sorting trees*

We assume $n = 2^m$

**Cole_Merge()** Receive $X(t+1)$ from the left child
Receive $Y(t+1)$ from the right child
Merge : $val(t+1) \leftarrow MergeWithHelp(X(t+1), Y(t+1), val(t))$
Reduce : $sendZ(t+1) \leftarrow REDUCE(val(t+1))$
Reduce only sends the last element in four, $Reduce(\{z_1, z_2, \ldots, z_n\}) = \{z_4, z_8, \ldots\}$
Form the time a node has started receiving data, the ammount of data it receives doubles at every step.
Suppose step $t$ is the first step at wich a node has received all its data

**At step $t$ :** the node sends one data every four $(z_4, z_8, z_{12}, \ldots)$

**At step $t+1$ :** the node sends every other date $(z_2, z_4, z_6, z_8, \ldots)$

**At step $t+2$ :** the node sends all data

**At step $t+3$ and later :** the node does not do anything

Partie 1.3

## Algorithm design for processor rings

- Distributed memory: each processor owns a piece of memory, and has it s exclusive usage.
- Processors exchange data through communications

  Two primitives :

  `SEND(addr, size)` , where `addr` is the address where the data to be send starts and `size` is the size of the data.

  `RECEIVE(addr, size)`, where `addr` is the address where to store the date and `size` is the size of the data.

Processor $P_i, 0 \leq i \leq n-1$ can only send messages to processor $P_{i+1 \mod n}$

**Different possible behaviours:**

**Blocking behaviour:** the primitive returns only after the communication completed

**Non Blocking behaviour:** the primitive returns instantly and the communication takes place "sometime"

We assume overlapp of computations and communications (several communications can happen simultaneously in the same processor)

**Cost of communications:** Sending a message of size $m$ costs

$$L + m \times b$$

where $b$ is the inverse of the bandwidth.

**Basic functions:**

**MyNum():** returns the identifier of a node

**NbProc():** returns the number of processors in the ring

1.3.1

### Global communications

*Broadcast*

Processor $P_0$ wants to send a message of size $m$ and stored at address addr to all the other processors.

```
BROADCAST(addr,m)
    n <- NbProc()
    id <- MyNum()
    IF (id=0)
    THEN
        SEND (addr,m)
    ELSE
        IF (id=n-1)
        THEN
            RECEIVE (addr,m)
        ELSE
            RECEIVE (addr,m)
            SEND(addr,m)
```

The algorithm works under the assumption that the receive operations are blocking. Same program running on each of the processors. The difference of behaviour between the processors depends on the data

**SPMD:** Simple Program Multiple Data

**Cost:** $(n-1)$ communications happening sequentially

$$(n-1)(L + mb)$$

*Scatter*

Processor $P_k$ sends a different message to each processor.
$P_k$ holds the message for $P_q$ at `addr[q]`
At the end, each processor holds its value at the address msg.
We can assume that there is a message for processor $P_k$ at `addr[k]`

```
SCATTER(k,msg,addr,m)
n <- NbProc()
id <- MyNum()
IF (id = k)
THEN
    FOR i=1 TO n-1 do
        SEND(addr[k-1 mod n],m)
ELSE
    FOR i=1 TO id+k_id-1 do
        RECEIVE(msg,m)              // Blocking
        SEND(msg,m)                 // Blocking
    RECEIVE(msg,m)
```

**Cost:** Cost on $P_{k+1}$ is $(2n-3)(L+mb)$ because of blocking communications

```
RECEIVE(tmp,m)
FOR i=1 to n+k-id-1 do
    SEND(tmp,m) || RECEIVE(msg,m)   // Blocking
    tmp <- msg
```