Frederic Vivien*
ENS de Lyon

Contents

*frederic.vivien@ens-lyon.fr

Bibliography:

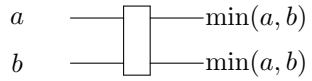
- *Parallel Algorithms*, H. Casanova, A. Legrand, Y. Robert
- *Introduction to distributed algorithms*, G. Tel
- *Scheduling and automatic parallelization*, A. Darte, Y. Robert, F. Vivien

Final grade: 50% Finale exam + 25% mid-term exam + 25% programming project

Part I
Theoretical models

1 Sorting networks

They aim at sorting values.
Comparators:



Question: How to arrange comparators to quickly sort a large number of values?

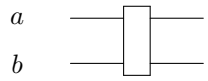
1.1 Odd-even merging network

Odd-even merge sort: Algorithm due to Batcher
Arbitrary sequence $c_1, c_2, ..., c_n$ $SORT(c_1, ..., c_n$ denotes the sorted sequence: $c_1 \leq c_2 \leq ... \leq c_n$

Warning: Non-decreasing = "croissant" and increasing = "strictement croissant" (same for positive and non-negative).

- If $c_1 \leq c_2 \leq ... \leq c_n$
  $SORTED(c_1, ..., c_n)$
- Merged operator:
  If $SORTED(a_1, ..., a_n)$ and $SORTED(b_1, ..., b_n)$,
  then $MERGE((a_1, ..., a_n), (b_1, ..., b_n)) = SORT(a_1, ..., a_n, b_1, ..., b_n)$

Two list of size 1:



Two lists of size 2:
$MERGE_2$

Merge 2 lists of size 4:
$MERGE_3$

Merging networks $MERGE_m$ to merge two sorted lists of size 2.
1st merging sub-network takes as input the odd elements of the two input lists.

Propriety 1. *Let $A = (a_1, a_2, ..., a_{2n})$ and $B = (b_1, b_2, ..., b_{2n})$ be two sorted sequences ($SORTED(A)$ and $SORTED(B)$).*

$$(d_1, ..., d_{2n}) = MERGE((a_1, a_3, a_5, ..., a_{2n-1}), (b_1, b_3, b_5, ..., b_{2n-1})$$
$$(e_1, ..., e_{2n}) = MERGE((a_2, a_4, a_6, ..., a_{2n}), (b_2, b_4, b_6, ..., b_{2n})$$

Then, we have

$$SORTED((d_1, \min(d_2, e_1), \max(d_2, e_1), ..., \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n}))$$

Proof. Without loss of generality, we assume all values to be distinct.

- d_1 is the overall minimum, being the minimum of the minimum elements of the two lists.
- e_{2n} is the overall maximum.

General case: We look at d_i and e_{i-1} (for $2 \leq i \leq 2n$) which are, each, either in position $2i-2$ or $2i-1$.

The result is going to be correct if d_i and e_{i-1} dominate $2i-3$ values and are dominated by $4n-2n+1$ values.

We have to prove for $2 \leq i \leq 2n$ that

1. d_i dominates $2i-3$ values
2. e_{i-1} dominates $2i-3$ values
3. d_i is dominated by $4n+2i+1$ values
4. e_{i-1} is dominated by $4n-2i+1$ values

1. d_i dominates $2i-3$ values $(d_1, d_2, ..., d_i)$

   We assume (wlog) that d_i belongs to A. There are k elements of A in $(d_1, ..., d_i)$. $d_i = a_{2k-1}$. d_i dominates $a_1, a_2, ..., a_{2k-2}$, that is $2k-2$ elements of A.

   $d_1, ..., d_i$ contains $i-k$ elements of B. The largest of these element is $b_{2(2-k)-1}$. d_i dominates $2(2-k)-1$ elements of B. Therefore d_i dominates (at least) $(2k-2)+(2(i-k)-i) = 2i-3$ elements.

2. Same principal as above
3. Same principal as above
4. e_{i-1} is dominated by $4n-2i+1$ elements $(e_1, e_2, ..., e_{i-1})$. We assume e_{i-1} belongs to B.

   Let k be the number of B in $e_1, ..., e_i$, $e_{i-1} = b_{2k}$.

   Hence, e_{i-1} is dominated by $2n-(2k+1)+1$, that is, $2n-2k$ elements of B.

   $(e_1, ..., e_{i-1})$ includes $(i-1-k)$ elements of A. The largest of these elements is $a_{2(i-1-k}$.

   Therefore e_{i-1} is dominated by $a_{2(i-1-k)+2} = a_{2(i-k}$.

   e_{i-1} is dominated by $a_{2(i-k}$ through a_{2n} and so by $2n-2(i-k+1)$ elements.

   Overall e_{i-1} is dominated by $(2n-2k)+(2n-2i+2k+1) = 4n-2i+1$ elements

□

Performance evaluation:

- Unit time to traverse a comparator

- Execution time of the network: largest number of comparators on a path from an input to an output

Lemma 1. *The processing time t_m, and the number of comparators, p_m, of $MERGE_m$, satisfy the recursion:*

$$t_1 = 1 \qquad t_m = t_{m-1} + 1 \qquad\qquad t_m = m$$
$$p_1 = 1 \qquad p_m = 2p_{m-1} + 2^{m-1} - 1 \qquad\qquad p_m = 2^{m-1}(m-1) + 1$$

Let n be the size of the input lists. So, $n = 2^{m-1}$, $t'_n = o(\log(n))$ $p'_n = o(n \log n)$
Work $= t'_n \times p'_n = O(n \log^2 n)$

1.2 Sorting network

Recursive construction:

Lemma 2. *The processing time t''_m and the number of comparators P''_m of $SORT_m$ satisfy the recursions:*

$$t''_1 = 1 \qquad t''_m = t_{m-1} + t_m \qquad\qquad t''_m = O(m^2)$$
$$p''_1 = 1 \qquad p''_m = 2p''_{m-1} + p_m \qquad\qquad p''_m = O(2^m m^2)$$

For inputs of size n, the time is $O(\log^2 n)$ and the number of comparators in $O(n \log^2 n)$.
Work is in $O(n \log^4 n)$.

1.3 The 0-1 principle

Propriety 2. *A network is a sorting network if and only if it is a sorting network for 0-1 sequences.*

Proof. If a network sorts arbitrary sequences, it sorts 0-1 sequences.

We now show that a network that does not sort arbitrary sequences does not sort 0-1 sequences. we assume that there exists a sequence $(x_1, x_2, ..., x_n)$ that is not sorted correctly.

Let R be the network. The output is $R(x)$. There exists some index k such that $R(x)_k > R(x)_{k+1}$. Let is consider a non decreasing function f. Applying the input of the network does not change the paths followed by the different values inside the network (because what matters is the relative position of values in the overall sequence).

$$f(y) = \begin{cases} 0 & \text{if } y < R(x)_k \\ 1 & \text{otherwise} \end{cases}$$

I fed the sequence $f(x)$ to R.

$$R(f(x))_k = f(R(x)_k) = 1 \qquad R(f(x))_{k+1} = 0$$

R does not sort all 0-1 sequences. □

2 Sorting on a one dimensional network

2.1 Odd-even transposition sort

Version to sort 8 inputs:

In total: n rows of comparators for n inputs.

Performance:

- 1st row: $\frac{n}{2}$ comparators
- 2nd row: $\frac{n}{2} - 1$ comparators

A pattern contains (n-1) comparators, and we have $\frac{n}{2}$ patterns.
Overall $\frac{n(n-1)}{2}$ comparators.

Execution time: n

Cost: $o(n^3)$ *(#comparators $\times$ execution time)*

Correction. 0-1 principle.
Correct output if all zeroes on the left.
Let $a_1, ..., a_n$ a 0-1 sequence. Let k be the number of 1's in this sequence. et k_0 be the initial position of the right-most of these 1's.

- During the first step, this 1 moves one position to the right if and only if k_0 was odd. After step 1, this 1 is at least at position 2.
- Then, for each step, starting at step 2, it moves one position to the right, for the $(n-1)$ remaining steps.

Eventually, it reaches the n-th position.
We now look at the second right-most 1. It moves one position to the right at each step, as soon as the right-most 1 move one position to the right at each step. Ti has one fewer move possible, but has to move to position $(n-1)$. □

2.2 Odd-even sorting on a one dimensional network

- A one dimensional network of processors
- Neighbour processors can exchange values.
- Mimic the sorting network on the row of processors.
- At step $2i-1$, processors P_{2k-1} and P_{2k} exchange their data P_{2k_1} will keep the smallest of their data.
- At step $2i$, processor $2k$ and $2k+1$ exchanges data.

We have n values, p processors. $n > p$, and each processor initially holds $\frac{n}{p}$ data.

Complexity:
Step 0 (local sort): $O(\frac{n}{p}.\log(\frac{n}{p}))$
Step 1: $O(\frac{n}{p})$
There are twice as many steps as there are processors.

Overall complexity:

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

If $p \leq \log n$ we have $\frac{\log n}{p} \geq 1$. The complexity become $O(\frac{n}{p}\log n)$. $\Rightarrow$ if $p \leq \log n$ the algorithm has an optimal running time.

Cost:

$$O(n \log n + np)$$

Part II
PRAMs

Parallel Random Access Machines

- Theoretical model
- No communications:
  ⇒ Shared memory model

  1 large memory
  n processing elements/units directly connected to it

All processing units execute the same algorithm simultaneously: each time step all PU[1] execute the same instruction (but some PUs may be inactive, like in an if ... then ... else ...).

All PU can access memory location in a unit of time. Different processing units may access the very same memory location simultaneously.

3 variants:

- CREW: Concurrent Read Exclusive Write
  - Any number of PU can simultaneously read any given memory location
  - At any time, at most one PU can write into any given memory location
- EREW: Exclusive Read, Exclusive Write
  - At any given time, at most on PU can access any given memory cell
- CRCW: Concurrent Read, Concurrent Write
  - Consistent mode: all PU writing in the same memory location **must** write the same value
  - Arbitrary mode: among the different values that PUs attempt to simultaneously write in a given location, one of them is arbitrarily (i.e. randomly) written.
  - Priority mode: the value of the PU of highest priority/index is written
  - Fusion mode: a commutative and associative operation is applied to the values that the PUs write in a same memory location

1 Pointer jumping

1.1 List ranking

Linked list L that contains n objects. For any element i in the list, we want to compute

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

Sequential complexity: $O(n)$ (go backward from the tail with a doubly-linked list)

- Associate one processor to each list element (P_i is associated to element i)
- At each step, we split the list in two sub-lists: one of the odd elements and one of the even ones.
  ⇒ at each step the size of the list is halved
  ⇒ $O(\log n)$ steps

[1] Processing Unit

Rank computation (L):

```
1  for all i in parallel do
2  |  if next[i] = nil then
3  |  |  d[i]=0
4  |  else
5  |  |  d[i]=1
6  |  while there exists a node i such that next[i] != nil do
7  |  |  for all i in parallel do do
8  |  |  |  if next[i] != nil then
9  |  |  |  |  d[i] ← d[i] + d[next[i]]
10 |  |  |  |  next[i] ← next[next[i]]
```

Evaluation of *while*:
We know the value of n.
Counter initialized to 0.
Each time a next field is set to nil, the counter is incremented, when we reach n, we are done.

```
1  for all i in parallel do
2  |  if next[i] = nil then
3  |  |  d[i]=0
4  |  else
5  |  |  d[i]=1
6  |  while counter < n do
7  |  |  for all i in parallel do do
8  |  |  |  if next[i] != nil then
9  |  |  |  |  d[i] ← d[i] + d[next[i]]
10 |  |  |  |  next[i] ← next[next[i]]
11 |  |  |  if next[i] = nil then
12 |  |  |  |  counter ← counter + 1
```

Only work with CRCW PRAM in fusion mode (with addition).

```
1  for all i in parallel do
2  |  if next[i] = nil then
3  |  |  d[i]=0
4  |  else
5  |  |  d[i]=1
6  |  Finished ← true
7  |  while ¬ Finished do
8  |  |  for all i in parallel do do
9  |  |  |  if next[i] != nil then
10 |  |  |  |  d[i] ← d[i] + d[next[i]]
11 |  |  |  |  next[i] ← next[next[i]]
12 |  |  |  if next[i] != nil then
13 |  |  |  |  Finished ← false
```

Work for any variant of CRCW PRAM.
→ We know that we will have at most $\lceil \log_n \rceil$ steps. *For i=1 to* $\lceil \log_n \rceil$ works also on EREW PRAM.

- About "race condition" in updating next[i]

```
1 temp ← next[next[i]]
2 (Only works if concurrent reads)
3 next[i] ← temp
4 temp← d[i]+d[next[i]]
5 d[i]←temp
6 temp1← d[i]
7 temp2← d[next[i]]
8 d[i]← temp1 + temp2
```

Execution time: $O(\log n)$

1.2 Prefix computation

Sequence $x_1, x_2, ..., x_n$, we want to compute the sequence

$$y_1, y_2, ..., y_n \text{ where } \begin{cases} y_1 = x_1 \\ y_k = y_{k-1} \otimes x_{k-1} = x_1 \otimes x_2 \otimes ... \otimes x_n \end{cases}$$

Where $\otimes$ binary associative operation.
We assume that x is given as a linked bit.

```
1 for all i in parallel do
2 |  y[i]=x[i]
3 while there exists a node i such that next[i]!=nil do
4 |  for all i in parallel do
5 |  |  if next[i]!=nil then
6 |  |  |  y[next[i]]← y[i] ⊗ y[next[i]]
7 |  |  |  next[i] ← next[next[i]]]
```

Algorithm 1: Prefix Computation (L)

2 Performance evaluation of PRAMs Algorithms

2.1 Cost, work, speed-up, efficiency

Let P be a problem of size n.
$T_{seq}(m)$: execution time of the best known sequential algorithm to solve P.
Let $T_{par}(p, n)$: execution time of our solution using p processors.

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

Work: Sum on all processing units of the number of operations performed by each units (or, the time each PU is effectively used).

$$C_p(n) = W_p(n) + \text{some idle time}$$

Speed-up:

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

Efficiency:

$$\begin{aligned} e_p(n) &= \frac{S_p(n)}{p} \\ &= \frac{T_{seq}(n)}{pT_{par}(p,n)} \\ &= \frac{T_{seq}(n)}{C_p(n)} \end{aligned}$$

Another definition of speed-up: *(We will not use it)*

$$S'_p(n) = \frac{T_{par}(1,n}{T_{par}(p,n)}$$

Ideal: $S_p(n)$ close to p, efficiency close to 1.

2.2 A simple simulation result

Propriety 3. *Let A be an algorithm, whose execution time is t when using p PUs, then A can be run on $p' \leq p$ PUs of the same type in time $O(\frac{p}{p'}t)$. The cost on p' PUs is at most twice the cost on p PUs.*

Proof. The algorithm runs in t steps. At each step we allocate (at most) $\frac{p}{p'}$ operations to each PU for the PUs to process them sequenctially in time $O(\frac{p}{p'})$.

$$\begin{aligned} C_{p'} &= p'T_{par}(p',n) \\ &= p't' \\ C_{p'} &\leq p' \left(\left\lceil \frac{p}{p'} \right\rceil t \right) \\ &\leq p' \left(\frac{p}{p'} + 1 \right) t \\ &= pt + p't \\ &\leq 2pt = 2C_p \end{aligned}$$

□

Remark: This proposition give an upper bound of what s possible, but it is sometimes possible to do better.
Let us consider prefix computation: n values, n PUs, $O(\log_2(n))$.
Using the proposition:

$$\text{With } p \leq n\text{PUs, } O\left(\frac{n}{p}\log(n)\right)$$

One other solution: Each processor has $\frac{n}{p}$ values (consecutive values)

- Each PU computes the prefixes for its $\frac{n}{p}$ values in time $\frac{n}{p}$.
- Pointer-jumping propagation over the p PUs of p prefixes in time $O(\log p)$.
- Each PU applies the prefixes computed above to its $\frac{n}{p}$ values in $O(\frac{n}{p})$.

The overall execution time is $O\left(\frac{n}{p} + \log p\right)$

"Coarsening of the computation."

Efficiency: Usually, when p increases the efficiency decreases

$$0 \leq e_p(n) \leq 1$$
$$0 \leq S_p(n) \leq p$$

Super linear speed-up:

$$s_p(n) > p$$

It can happen in practice if, for instance, by using more processors, all data fit in memory and the program no longer swaps.

2.3 Brent's theorem

Theorem 1 (Brent's theorem)**.** *Let A be an algorithm that executes a total of m operations and that runs in time t on a PRAM (on an unspecified number of PUs).*

Then A can be simulated of time $O\left(\frac{m}{p} + t\right)$ on P PUs of a PRAM of the same time.

Proof. A runs in t steps, performing m_i operations at step i.

$$\sum_{i=1}^{t} m_i = m$$

We simulate step i in time $\left\lceil \frac{m_i}{p} \right\rceil$ using p processors.

The overall execution time is thus

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

□

Example: Computing the maximum of n values on a EREW PRAM. We do that tournament-like with a binary tree of comparisons.

Time: $t = O(\log n)$ (n PUs), $m = O(n)$

On p PUs, using Brent's theorem, we know we can compute the maximum in time $O\left(\frac{n}{p} + \log n\right)$.

$p = \frac{n}{\log n} \Rightarrow$ time $O(\log n)$

3 Comparison of PRAM models

3.1 Model separation

Is there a problem that can be solved significantly faster on a CRCW than on a CREW PRAM?

Choosing the fusion mode and computing a sum of n values.

CRCW: all PUs write to the same memory location: $O(1)$

CREW: at each time step each PU can cimbine at most 2 values: $O(\log n)$ time steps.

Using CRCW PRAM in coherent mode?

Computing the maximum of n values:

```
1 for all i from 1 to n, in parallel do
2 |  ismax[i] ← true (initialization)
3 for all i, j, 1 ≤ i ≤ n, 1 ≤ j ≤ n, i ≠ j in parallel do
4 |  if value[i]<value[j] then
5 |  |  ismax[i]←false
6 for all i, 1 ≤ i ≤ n in parallel do
7 |  if ismax[i] = true then
8 |  |  result← value[i]
```

Time: $O(1)$ on n processors

On a CREW PRAM, at best a $\log n$ execution time

Separation between CREW and EREW models ?

Does the element e belongs to a set X of n distinct values. On a CREW PRAM, each of n PUs compares its values of X to e. If equality, a boolean is sett to true $\Rightarrow$ constant time.

On a EREW PRAM, it takes at least a time $\log n$ for al processors to have a copy of e (at best we double at each step the number of copies of e).

3.2 Simulation theorem

Theorem 2 (Simulation theorem)**.** *Any CRCW algorithm with p PUs has an execution time at most* $O(\log n)$ *lower than the* best *EREW algorithm to solve the same problem using p processors.*

Proof. We assume the CRCW PRAM to be i coherent mode (different PUs writing simultaneously in the same memory cell must write the same value).

We show how to execute one step of concurrent write in at most $\log p$ steps on a EREW PRAM.

1. We take a temporary array A. Each processor P_I writes in $A[i]$ the couple (location, value).
2. The array A is sorted by non-decreasing value of the 1st element of the couples
3. Each processor P_i compares the address in $A[i]$ to the address held by the previous processor. If the addresses differ (or if $i = 1$), it performs a write, otherwise, it does nothing.

Complexity: (1) and (3) are executed in constant time on $O(p)$ PUs. (2) We assume that there exists am EREW algorithm[2] sorting p values on p PUs in time $O(\log p) \Rightarrow$ time $O(\log p)$ □

4 Cole's sorting machine

Proposed in 1986.

CREW algorithm to sort n values in time $\log n$ using n PUs, based on a merge sort algorithm: We have $\log n$ levels in the tree, and $\log n$ steps in Cole's algorithm. We want an algorithm running in time $O(\log n)$. The algorithm should thus be able to merge two sorted lists on any size in constant time.

[2] Cf part 4

4.1 Merge

Definition 1 (Good sampler)**.** *A sorted sequence L is said to be a good sampler (GS) of a sequence L if, for any $k \leq 1$, there are at most $2k+1$ elements of J* between *$k+1$ (arbitrary) consecutive elements of $L \cup \{-\infty\} \cup \{+\infty\}$.*

- *If J and K are two sorted sequences, $J|K$ is the merge of J and K.*
- *If a and b are two values with $a < b$, we say that x is between a and b if and only if $a < x \leq b$*
- $rank(x, J) = card\{j \in J \mid j < x\}$
  The cross-ranked of sorted sequence A in the sorted sequence B is:

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

In practice, there are at most 3 elements of the sequence J in between two consecutive elements of one of its good sampler (extended with $-\infty$ and $+\infty$).

Example: The subset of elements of odd ranks (or even ranks) is a good sampler.

Two sorted sequences, J and K,

$$\begin{aligned} J &= [2,3,7,8,10,14,15,17,18,21] \\ K &= [1,4,6,9,11,12,13,16,19,20] \\ L &= [5,10,12,17] \text{ is a good sampler of } \textit{both } J \text{ and } K \end{aligned}$$

In theory we should check that L is a good sampler. $k = 1$, we consider the intervals $]-\infty, 5],]5, 10[,]10, 12]$, $]12, 17]$ and $]17, +\infty[$ and we show that there are at most $2k+1 = 3$ values of J, and $2k+1 = 3$ values of K in each of these intervals. We do know for $k = 2, k = 3, ...$

We assume we know the crossrank of J and K in L ($R[J, L]$ and $R[K, L]$).

$$\begin{array}{rll} J(1) = (2,3) & K(1) = (1,4) & \Rightarrow (1,2,3,4) \\ J(2) = (7,8,10) & K(2) = (6,9) & \Rightarrow (6,7,8,9,10) \\ J(3) = () & K(3) = (11,12) & \Rightarrow (11,12) \\ J(4) = (14,15,17) & K(4) = (13,16) & \Rightarrow (13,14,15,16,17) \\ J(5) = (18,21) & K(5) = (19,20) & \Rightarrow (18,19,20,21) \end{array}$$

```
1 Merge with Help(J, K, L):
2 J and K are partitioned using L in L + 1 subsets.
3 J(i) = {j ∈ J | l_{i-1} < j < l_i}   (l_0 = -∞)
4 for all i in parallel (1 ≤ i ≤ |L| + 1) do
5 |   res ← MERGE(J(i), K(i))
6 J|K ← res_1.res_2. ... .res_{|L|+1}
```

Lemma 3. *If L has a good sampler of both J and K, and if crossranked $R[L, J], R[L, K], R[J, L]$ and $R[K, L]$ are known, then Merge with Help runs in $O(1)$ time with $|J| + |K|$ PUs on a CREW PRAM.*

Proof. (b) By definition of a good sampler $J(i)$ and $K(i)$ contains at most 3 values each and can be merged sequentially in $O(1)$.

(a) Each P_m reads the rank $r = rank(j_m, L)$.

```
1 if m = 1 or (rank(j_m, L) ≠ rank(j_{m-1}, L)) then
2 |  Add j_m to J(r) //J_{m-1} put r − 1 and r − 2
3 else
4 |  if rank(j_m, L) ≠ rank(j_{m-2}, L)) then
5 |  |  Add j_m to J(r) //J_m and J_{m-1} should be put r − 1 and r − 2
6 |  else
7 |  |  Add j_m to J(r) //J_{m-1} put r − 1 and r − 2
```

We use $|J| + |K|$ PUs.

(c) We need to know where to write the smallest element of $res_i = J(i)|K(i)$
$rank(l_i, J|K) = rank(l_i, J) + rank(l_i, K)$. we write sequentially (at most using 6 steps) res_i starting at position $rank(L_{i-1}, J|K)$. We use $|L| + 1$ PUs. □

4.2 Sorting trees

We assume $n = 2^m$.
→ binary tree used in a pipelined way
→ the value of a node at step t will be a good sampler of the value at step $t + 1$.

```
1 Cole-Merge():
2 Receive X(t + 1) from its left child
3 Receive Y(t + 1) from its right child
4 Merge: val(t + 1) ← Merge with Help (X(t + 1), Y(t + 1), val(t))
5 Send: Z(t + 1) = REDUCE(val(t + 1)) to the parent
6 REDUCE: keeps only every forth value.
```

A node of the tree is said to be complete when it has received all its inputs, i.e., a sorted sequence of size 2^k for a node at level k. As soon as the input sequence is no longer empty it doubles at each step, from 1 to 2^{k-1}

If a node is complete at step t, then the $REDUCE$ operation changes:

- at step $t + 1$: once again sends every forth element
- at step $t + 2$: sends every other elements (even ranked elements)
- at step $t + 3$: sends every element
- From step $t + 4$ on, stops working, stop reading sequences.

Example:

- $t = 0$: leaves are complete
- $t = 1$: every fourth element out the leaves
- $t = 2$: every other elements out of the leaves
- $t = 3$: each leaf sends its value.
  all the nodes at level 1 complete
- $t = 4$
- $t = 5$: level 1 nodes send every other value

- $t = 6$: all level 2 nodes are complete
- $t = 7$: the root merges [8] and [4]
- $t = 8$: the root merges $[6, 8]$ and $[2, 4]$ using $[4, 8]$ as a good sampler.
- $t = 9$: the root merges $[5, 6, 7, 8]$ and $[1, 2, 3, 4]$ using $[2, 4, 6, 8]$ as a good sampler.

The root is complete, we are done.

Execution time: Nodes at level k are complete at time $3k$. Execution time is $O(3 \log n) = O(\log n)$

Number of processors: To merge 2 lists of size k, in constant time, we need $2k$ PUs. (We have $\log n$ levels, and each requires at most n PUs, so we know how to do it in $O(n \log n)$).

Level k:
There are $\frac{n}{2^k}$ nodes at level k.

- They are complete at time $3k$. They each have a value $val(3k)$ of size 2^k. Overall, they use $\frac{n}{2^k} \times 2^k = n$ PUs.
- $3k - 1$, total size of input 2^{k-1}, need $\frac{n}{2}$
- $3k - 2$, $\frac{n}{4}$
- ...

Level k needs $\frac{n}{8}$ PUs at step $3k - 3$.
Level $k + 1$ needs $\frac{n}{8}$ PUs at step $3k$

At step 3k:

- level $k + 2 = \frac{n}{64}$ PUs
- level $k + 1 = \frac{n}{8}$ PUs
- level $k = n$ PUs
- level $k - 1 = 0$ PUs
- ...
- level $0 = 0$ PUs

Overall: Cole's algorithm needs less than $2n$ PUs.

4.3 Corrections

Lemma 4. *Les X, X', Y and Y' be four sorted sequences. If X is a GS of X' and if Y is a GS of Y', then $REDUCE(X|Y)$ is a GS of $REDUCED(X'|Y')$*

Warning: $X|Y$ is not *always* a GS of $X'|Y'$. If we take $X = [2, 7]$ $X' = [2, 5, 6, 7]$, and $Y = [1, 8]$ $Y' = [1, 3, 4, 8]$; then between the consecutive elements 2 and 7 of $X|Y$ we have $\{3, 4, 5, 6, 7\}$ in $X'|Y'$, that is 5 values (> 3).

Proof.

Propriety 4. *There are at most $2r + 2$ elements of $X'|Y'$ in between r consecutive elements of the $X|Y$.*

Proof. Let $e_1, e_2, ..., e_r$ be a sequence of r consecutive elements of $X|Y$. There are h_X elements of X in it and h_Y elements of Y; $h_X + h_Y = r$. Without loss of generality, we assume $e_1 \in X$.

Case 1: $e_r \in X$. X is a GS of X'. In between e_1 and e_r we have at most $2(h_X - 1) + 1 = 2h_X - 1$ elements of X'. Because Y is a GS of Y', there are at most $2(h_y + 1) + 1$ elements of Y' between $h_y + 2$ elements of Y. Overall there are at most $(2h_X - 1) + (2(h_Y + 1) + 1) = 2h_X + 2h_Y + 2 = 2r + 2$ elements of $X'|Y'$ in between the r consecutive elements $e_1, ..., e_r$.

Case 2: $e_r \in Y$. We add an element $e_0 \in Y$ preceding e_1, and an element $e_{r+1} \in X$ greater than e_r. The elements of X' that are between e_1 and e_r are between e_1 and e_{r+1}, so between $h_X + 1$ elements of X. Because X is a GS of X', there are at most $2h_x + 1$ such elements. Symmetrically, there are at most $2h_y + 1$ elements of Y' between $e_1, ..., e_r$. Hence, overall at most $(2h_X + 1) + (2h_Y + 1) = 2r + 2$ elements. □

Let $Z = REDUCE(X|Y)$, $Z' = REDUCE(X'|Y')$. Let us consider $k + 1$ consecutive values of Z: $z_h, z_{h+1}, ..., z_{h+k}$.

By definition of $REDUCE$, $z_h = e_{4h}, z_{h+1} = e_{4h+4}, ..., z_{n+k} = e_{4n+4k}$ where $X|Y = e_1, e_2, ...$ There are $4k + 1$ elements of $X|Y$ between z_h and z_{h+k}. We take $r = 4k + 1$. we know that there are at most $2r + 2 = 8k + 4$ values of $X'|Y'$ in between z_h and z_{h+k}. The $REDUCE$ operator keeps every forth value therefore there are at most $\frac{8k+4}{4} = 2k + 1$ values of $REDUCE(X'|Y')$ in between $z_h, ..., z_{h+k}$, that is, $k + 1$ (arbitrary) consecutive values of $REDUCE(X|Y)$ □

Lemma 5. *If we have the sorted sequences $X, Y, U = X|Y, X'$ and Y', with X a GS of X', Y a GS of Y', and we know the crossranks $R[X', X]$ and $R[Y', Y]$. Then, we can compute the cross $R[X', U], R[Y', U], R[U, X']$, $R[U, Y']$ in $O(1)$ time and using $O(|X| + |Y|)$ PUs.*

Part III
Algorithm for rings and grid of processors

1 Algorithm for rings of processors

Distributed memory model

Each processor owns some private memory and is the only processor allowed to access it (both for reading and writing).

Considered topology: Unidirectional ring of processors. We have p processors.

One direct communication link from P_i to $P_{i+1(modp)}$ for $0 \leq i \leq p-1$ (most of the time the "mod p" expression will be implicit.

$NOM\_PROCS()$: gives the number of processors in the ring.

$MY\_NUM()$: identifier/rank of the calling processors.

Communication links are unidirectional: to send a message to P_0, P_1 must go through $P_2, P_3, ..., P_{p-1}$.

Primitives:

- For sending

$$SEND(\underbrace{addr}_{\substack{\text{address where}\\ \text{to read the data}\\ \text{to send}}}, \underbrace{m}_{\substack{\text{the size of}\\ \text{the message}}})$$

  Point-to-point communications: we do not need to specify the destination because processor P_i can only send a message to processor P_{i+1}.

- For receiving message:

$$RECIEVE(\underbrace{addr}_{\substack{\text{address where to}\\ \text{store what is}\\ \text{received}}}, \underbrace{m}_{\substack{\text{the size of}\\ \text{the message}}})$$

- $SEND$ and $RECIEVED$ works in pairs
- *Two kinds of primitives :*
  - blocking primitives: when reaching a communication primitive, the algorithm stops and only resume its execution when the communication is completed.
  - asynchronous or non-bocking communications: the algorithm instantaneously returns from the communication itself will take place at some later time (we have no idea when). One may *test* whether the communication has completed. Enabled an overlap of communication and computation.

Most of the time we will use asynchronous sends and blocking receives.

Program Multiple Data (SPMD) model: (no longer the synchronization we had with PRAM) Different processors can execute different instructions (of the same program) at the same instant.

Cost (time) of communications: Sending a message of size m from a processor to its neighbour takes a time

$$\underbrace{L}_{\text{latency}} + m. \underbrace{b}_{\substack{\text{the inverse} \\ \text{of the bandwidth}}}$$

Each processor can simultaneously:

- send a message
- received a message
- perform some computation

1.1 Macro-communication

1.1.1 Broadcast

A given processor k wants to send a message of size m to all other processors. Addresses :

- For P_k: where the message is initially stored
- For $P_i (i \neq k)$: where to store the message

```
1  BROADCAST(k, addr, m)
2  p←NUM_PROCS()
3  q←MY_NUM()
4  if q = k then
5  |  SEND(addr, m)
6  else
7  |  if q = k − 1  mod p then
8  |  |  RECEIVE(addr, m)
9  |  else
10 |  |  RECEIVE(addr, m)
11 |  |  SEND(addr, m)
```

The algorithm is automatically correct, and has same running time with blocking receives and asynchronous sends.

1.1.2 Scatter

Processor P_k sends a different message to every other processor. Initially, processor P_k holds a message for processor P_q at address $addr[q]$ (we may assume that $addr[k]$ is holding a message for P_k)

```
1  SCATTER(k, msg, addr, m)
2  p←NUM_PROCS()
3  q←MY_NUM()
4  if q = k then
5  |  for i=1 to p-1 do
6  |  |  SEND( addr[k − 1 − i + 1  mod p], m)
7  else
8  |  for j=1 to p-(q-k)-1 do
9  |  |  RECEIVE(tempR, m)
10 |  |  SEND(tempR, m)
11 |  RECEIVE(msg, m)
```

Execution time: $(p-1)(L + m.b)$

```
1 RECEIVE(tempR, m)
2 for i=1 to p-q+k-1 do
3 |   SEND(tempR, m)
4 |   RECEIVE(tempS, m)
5 |   tempR ↔ tempS
```

If messages are in order $Addr[k+1], Addr[k+2], ...$
$[(p-2)+(p-1)](L+mb)$
$(2p-3)(L+mb)$

1.1.3 All to all

p simultaneous broadcast

```
1 ALL-TO-ALL(my_message, addr, m)
2 p←NUM_PROSC()
3 q←MY_NUM()
4 addr[q] ←my_message
5 for i=1 to p-1 do
6 |   SEND(addr[q − i + 1  mod p], m)
7 |   RECEIVE(addr[q − i  mod p], m)
```

Execution time:$(p-1)(L+m.b)$

1.1.4 Pipelined broadcast

Split the message of size m into r pieces (we assume that r divides m).

```
1 PIPELINED_BROADCAST(k,addr,m)
2 p←NUM_PROSC()
3 q←MY_NUM()
4 if q=k then
5 |   for i=0 to r-1 do
6 |   |   SEND(addr[i],m/r)
7 else
8 |   if q=k+1 then
9 |   |   for i=0 to r-1 do
10|   |   |   RECEIVE(addr[i],m/r)
11|   else
12|   |   RECEIVE(addr[0],m/r)
13|   |   for i=0 to r-2 do
14|   |   |   SEND(addr[i],m/r)
15|   |   |   RECEIVE(addr[i + 1],m/r)
16|   |   SEND(addr[r − 1],m/r)
```

Execution time:

$$(p-1)(l+\frac{m}{r}.b)+(r-1)(L+\frac{m}{r}b)=(p+r-2)(L+\frac{m}{r}b)$$

The value of r minimizing the execution time is:

$$r=\sqrt{\frac{m(p-2)b}{L}}$$

The execution time becomes:

$$(\sqrt{(p-2)L}+\sqrt{mb})^2 \underset{m\to+\infty}{\sim} mb$$

1.2 Matrix vector multiplication

Let A be a matrix of size $n \times n$, x a vector of size n.

The aim is to calculate $y = A.x$ (all indices starting at 0)

Sequential version:

```
1 for i=0 to n-1 do
2 |  y_i ← 0 for j=0 to n-1 do
3 |  |  y_i ← y_i + A_ij × x_j
```

The computation of the n values of vector can be computed in parallel. We assume $p < n$, and p divides n ; let $r = \frac{n}{p}$.

We charge each processor to compute r entries of vector y. We assume there is not enough memory to replicate matrix A on each processor. A is distributed among the p processors.

Classical solution: each processor is given a set of r consecutive rows.
Processor P_0 holds row 0 to $r-1$,
Processor P_1 holds row r to $2r-1$,
...

This is called a *block* distribution of rows.

We could assume that we have enough memory available to have one copy on each processor.

In such a case, the whole computation could be performed without communication. Processor P_i would hold rows $i \times r$ through $(i+1)r-1$ of A and x and thus compute y_{ir} through $y_{(i+1)r-1}$. At the end of the algorithm, y would be distributed. Therefore, if we wanted to apply to apply a matrix-vector multiplication to y, we could not reuse our algorithm. Because, most of the time, matrix operations happen not alone but in a series of operations, we always assume that the inputs and outputs of algorithms are distributed the same way.

We assume that x (and later y) is distributed by blocks of r values. Processor P_i holds the component x_{ir} to $x_{(i+1)r-1}$ and will compute the component y_{ir} to $y_{(i+1)r-1}$.

P_0	A_{00}	A_{01}	A_{02}	A_{03}	A_{04}	A_{05}	A_{06}	A_{07}	x_0
	A_{10}	A_{11}	A_{12}	...				$\vdots$	x_1
P_1	A_{20}	A_{21}	A_{22}	...				$\vdots$	x_2
	A_{30}	A_{31}	A_{32}	...				$\vdots$	x_3
P_2	A_{40}	A_{41}	A_{42}	...				$\vdots$	x_4
	A_{50}	A_{51}	A_{52}	...				$\vdots$	x_5
P_3	A_{60}	A_{61}	A_{62}	...				$\vdots$	x_6
	A_{70}	A_{71}	A_{72}	...	...	...	...	A_{77}	x_7

```
1  Matrix-vector Multiplication(A, x , y)
2  p ← NUM_PROCS()
3  q ← MY_NUM()
4  tempS ← x
5  for step=0 to p-1 do
6  |  SEND(tempS, r)
7  |  RECEIVE(tempR, r)
8  |  for i=0 to r-1 do
9  |  |  for j=0 to r-1 do
10 |  |  |  y[i] ← y[i] + A[i, (q − step  mod p)r + j]tempS[j]
11 |  |  tempR ↔ tempS
```

Execution time:

$$
\begin{aligned}
& p \times \max(L + rb, L + rb, \underbrace{r^2 w}_{\substack{\text{cost of a} \\ \text{multiply-add}}}) \\
&= p \max(L + rb, r^2 w) \\
r &= \frac{n}{p} \\
\text{Execution time} &= \max pL + nb, \frac{n^2}{p} w \\
&\underset{n \to \infty}{\sim} \frac{n^2}{p} w
\end{aligned}
$$

Asymptotically, our algorithm is efficient.

1.3 Matrix-Matrix Multiplication

3 matrix A, B, C square $n \times n$ matrices.

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

This is nothing but

- n^q scalar products
- n matrix-vector multiplication

We assume all matrix to be distributed the same way. Processor P_i holds rows $i \times r$ to $(i+1)r - 1$ of matrices A, B and C.

We logically divide the r rows of A assigned to processor P_i in p blocks of size r, this set of row is seen as p $r \times r$ matrices.

P_i holds the block $\hat{A}_{i,l}, \hat{B}_{i,l}$ and $\hat{C}_{i,l}$ of element of A, B and C.

$$P_i \quad \boxed{\begin{matrix} A_{ir,0} & \dots & A_{ir,r-1} \\ A_{ir+1,0} & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ A_{(i+1)r-1,0} & \dots & \vdots \end{matrix}} \quad \boxed{\begin{matrix} \cdot & \dots & A_{ir,(p-1)r} \\ \vdots & & A_{ir+1,(p-1)r} \\ \vdots & & \vdots \\ \vdots & \dots & A_{(i+1)r-1,(p-1)r} \end{matrix}}$$

Step 0:

$$\begin{aligned} P_q : \quad & \hat{A}_{q,l}, \hat{B}_{q,l}, \hat{C}_{q,l} \\ & \hat{C}_{q,l} \leftarrow \hat{A}_{q,q}.\hat{B}_{q,l} \\ & \text{Sending } \hat{B}_{q,l} \text{ to } P_{q+1} \\ & \text{Recieving } \hat{B}_{q-1,l} \text{ from } P_{q-1} \end{aligned}$$

Step 1:

$$P_q : \quad \hat{C}_{q,l} \underset{\text{for } 0 \le l \le p-1}{\leftarrow} \hat{C}_{q,l} + \hat{A}_{q,q-1} \times \hat{B}_{q-1,l}$$

```
1  Matrix Matrix Multiplication (A, B, C)
2  p ← NUM_PROCS()
3  q ← MY_NUM()
4  tempS ← B
5  for step=0 to p-1 do
6  |  SEND(tempS, r × n)
7  |  RECEIVE(tempR, r × n)
8  |  for l=0 to p-1 do
9  |  |  for i=0 to r-1 do
10 |  |  |  for j=0 to r-1 do
11 |  |  |  |  for k=0 to r-1 do
12 |  |  |  |  |  C[i, lr + j] ← C[i, lr + j] + A[i, ((q − step) mod p) × r + k] × tempS[k, lr + j]
13 |  |  tempR ↔ tempS
```

Execution time: $(r = \frac{n}{p})$

$$p \times \max(L + rnb, pr^3 w) = \max(pL + n^2 b, \frac{n^3 w}{p})$$

$$\underset{n \to \infty}{\sim} \frac{n^3 w}{p}$$

Asymptotically an efficient algorithm
Complexity of using n times the matrix-vector product:

$$n \times \max(pL + nb, \frac{n^2 w}{p}) = \max(npL + n^2 b, \frac{n^3 w}{p})$$

Benefit of using matrix-matrix multiplication: the number of communications, and thus the cost of latencies, is divided by n.

1.4 Stencil computation

A a 2 dimensional array of data. The data is repeatedly updated.

A cell is updated using a function that takes as input the value (past and/or new) of some of its neighbouring cell.

NW	N	NE
W	C	E
SN	S	SE

neighbourhood

We take an array of size $n \times n$.

Our stencil:

$$C_{new} \leftarrow UPDATE(C_{old}, W_{new}, N_{new})$$

If the cell has no west neighbour, W_{new} is replaced by NIL; if the cell has no north neighbour, N_{new} is replaced by NIL.

```
1  for i=0 to n-1 do
2  |  for j=0 to n-1 do
3  |  |  a_ij ← UPDATE(a_{i,j}, a_{i,j-1}, a_{i-1,j})
```

Greedy Parallelization: We assume that $n = p$. Each processor holds one row.

Idea: do things as soon as possible.

```
 1  p ← NUM_PROCS()
 2  q ← MY_PROC()
 3  if q=0 then
 4  |  A[0] ← UPDATE(A[0], NIL, NIL)
 5  |  SEND(A[0], 1)
 6  else
 7  |  RECEIVE(v, 1)
 8  |  A[0] ← UPDATE(A[0], NIL, v)
    // ----- To correct number of SEND/RECEIVED
 9  if 0 ≠ p − 1 then
10  |  SEND(A[0], 1)
11  if q ≠ p − 1 then
12  |  SEND(A[0], 1)
    // -----
13  for j=1 to n-1 do
14  |  if q=0 then
15  |  |  A[j] ← UPDATE(A[j], A[j − 1], NIL)
16  |  |  SEND(A[j], 1)
17  |  else
18  |  |  if q=p-1 then
19  |  |  |  RECEIVE(v, 1)
20  |  |  |  A[j] ← UPDATE(A[j], A[j − 1], v)
21  |  |  else
22  |  |  |  SEND(A[j − 1], 1)
23  |  |  |  RECEIVE(v, 1)
24  |  |  |  A[j] ← UPDATE(A[j], A[j − 1], v)
```

General case $p < n$: p divides n. How to distribute rows to processors?

First solution: cycle distribution of rows to processors. $p = 3, n = 9$ rows.

p_0	0	
p_1	1	
p_2	2	
p_0	3	
p_1	4	
p_2	5	
p_0	6	
p_1	7	
p_2	8	

Row i is allocated to processor $P_{i \mod p}$.

```
1  StencilWithCyclicDistribution(A)
2  p ← NUM_PROCS()
3  q ← MY_PROC()
4  for i=0 to n/p − 1 do
5      if q=0 and i=0 then
6          A[0,0] ← UPDATE(A[0,0], NIL, NIL)
7          SEND(A[0,0], 1)
8      else
9          RECEIVE(v, 1)
10         A[i,0] ← UPDATE(A[i,0], NIL, v)
11         if q ≠ p − 1 or i ≠ n/p − 1 then
12             SEND(A[i,0], 1)
13     for j=1 to n-1 do
14         if q=0 and i=0 then
15             A[i,j] ← UPDATE(A[i,j], A[i − 1,j], NIL)
16             SEND(A[i,j], 1)
17         else
18             if q=p-1 and i=n/p − 1 then
19                 RECEIVE(v, 1)
20                 A[i,j] ← (A[i,j], A[i − 1,j], v)
21             else
22                 RECEIVE(v, 1)
23                 A[i,j] ← UPDATE(A[i,j], A[i − 1,j], v)
24                 SEND(A[i,j], 1)
```

Where A is set of $\frac{n}{p}$ rows of size n

Alternate version: Replace line 21 by $SEND(A[i,j],1)||RECV(v,1)$ and add at the end of the program $if\ q \neq -1\ then\ SEND(A[i,n],1)$

Performance: For the alternate version:

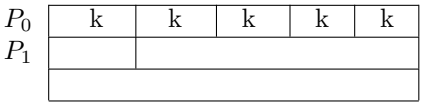
$$T = (\underbrace{p-1}_{\substack{\#\text{ step before} \\ P_{p-1}\text{ starts to} \\ \text{work}}} + \underbrace{\frac{n^2}{p}}_{\substack{\#\text{ of elements/} \\ \text{steps for } P_{q-1}}}) \times (w + L + b)$$

Asymptotical efficiency:

$$e = \frac{p \times T_{seq}(n)}{pT_p(n)} = \frac{pn^2w}{p(p-1+\frac{n^2}{p})(w+b+L)}$$
$$\underset{n\to\infty}{\sim} \frac{w}{w+b+L}$$

L can be large and the efficiency small.

Coarsen the communications: Instead of sending data one at a time, we send them by bulk of k (k divides n).



$$T_p(n,k) = \left(p-1+\frac{n^2}{pk}\right)(wk+L+kb)$$
$$e_p(n,k) = \frac{n^2w}{p[(p-1)(wk+L+kb)+(\frac{n^2w}{p}+\frac{n^2L}{pk}+\frac{n^2b}{p})]}$$
$$e_p(n,k) \underset{n\to\infty}{\sim} \frac{w}{w+\frac{L}{k}+b}$$

1st solution: Block distribution P_k holds the row from $\frac{kn}{p}$ to $\frac{n}{p}-1$.

2nd solution: (generalization) Block cyclic distribution.

Formally: Processor P_i holds row j such that $i = \lfloor \frac{j}{r} \rfloor$. *Now in one step* a processor:

- Updates: $k \times n$
- Receiving: k
- Sending: k

$$T_p(n,k,r) = \left(\frac{n^2}{pkr}\right)(krw+L+kb)$$

Let $t = \max(p, \frac{n}{k})(kw+L+kb)$. P_0 does not stop if and only if $p \leq \frac{n}{k}$. We assume that $n \geq pk$ to write $T_p(n,k,r)$.

$$e_p(n,k,r) \underset{n\to+\infty}{\sim} \frac{w}{w+\frac{L}{kr}+br}$$

Start form $T_p(n,k,r)$. Fix n,p. We differentiate T_p with respect to k. T_p is minimized for $k'(r) = n\sqrt{\frac{L}{p(p-1)r(rw+b)}}$.

Best solution: take the two functions $\lceil k'(r) \rceil$ and $\lfloor k'(r) \rfloor$, inject in T_p and look numerically for the best value.

2 Algorithm for grids of processors

2D square grid of processors, $p = q^2$.

Each processor link to its four neighbour. We only consider torus: so $P_{i,j}$ has connection to processors $P_{i-1,j}, P_{i+1,j}, P_{i,j-1 \mod p}, P_{i,j+1 \mod p}$ So every processor belongs to two rings (processor of sane row, processor of same column)

Links are bidirectional: full-duplex (they could have been unidirectional) (no performance degradation if two simultaneous communications (of different directions)).

Simultaneously, a processor can:

- do some computation
- receive a message
- send a message

Multiport (4-port):

- can send/receive 4 messages simultaneously (one per link)

1-port:

- at most one receive (or send) at any time

$NUM\_PROCS()$ return the number of processors in the system
$MY\_PROC\_ROW()$ and $MY\_PROC\_COLUMN()$ return the coordinates of the calling processor.

$$SEND(dest, \underbrace{addr}_{\substack{\text{address of} \\ \text{data to} \\ \text{send}}}, \underbrace{L}_{size})$$

$$RECV(src, addr, L)$$

$$BROADCASTROW(\underbrace{i, j}_{\substack{\text{coordinates of the} \\ \text{processor source of} \\ \text{the broadcast}}}, srcaddr, destaddr)$$

$$BROADCASTCOLUMN(i, j, srcaddr, destaddr, L)$$

If broadcast is called by a processor whose row is not i, we assume that it does nothing and returns immediately.

2.1 Outer matrix product

3 square matrix A, B, C of size n.

$$C \leftarrow A \times B$$

The three matrices are distributed in the same manner on the processors.

We use a 2D distribution of data. $q = 4, p = 16, n = 16$.

$\hat{A}_{00}$	$\hat{A}_{01}$	$\hat{A}_{02}$	$\hat{A}_{03}$
$\hat{A}_{10}$	$\hat{A}_{11}$	$\hat{A}_{12}$	$\hat{A}_{13}$
$\hat{A}_{20}$	$\hat{A}_{21}$	$\hat{A}_{22}$	$\hat{A}_{23}$
$\hat{A}_{30}$	$\hat{A}_{31}$	$\hat{A}_{32}$	$\hat{A}_{33}$

Processor $p_{i,j}$ $(0 \leq i, j \leq q-1)$ holds the block matrices $\hat{A}_{i,j}, \hat{B}_{i,j}$ and $\hat{C}_{i,j}$ that includes the elements $A_{k,l}, B_{k,l}$ and $C_{k,l}$ (respectively) where $i.m \leq k \leq (r-1)m - 1$ where $m = \frac{n}{q}$ and $j.m \leq l \leq (j+1)m - 1$.

Sequential product:

```
for k=0 to n-1 do
    for i=0 to n-1 do
        for j=0 to n-1 do
            C_ij ← C_ij + A_ik × B_kj
```

(idem with blocks)

Step k: processor holding $\hat{C}_{i,j}$ will completely compute it. $P_{i,j}$ needs block matrices $\hat{A}_{i,k}$ and $\hat{B}_{i,k}$ at step k. $\hat{A}_{i,k}$ is owned by $\hat{P}_{i,k}$. So $P_{i,k}$ must send $\hat{A}_{i,k}$ to $P_{i,j}$ at step k (for all value of j). At step k, $P_{i,k}$ must broadcast $A_{i,k}$ to its row of processors.

$\hat{B}_{k,j}$ is held by processor $P_{k,j}$. Processor $P_{k,j}$ must send $\hat{B}_{i,k}$ to $P_{i,j}$ (whatever the value of i). At step k, $P_{k,j}$ must broadcast $\hat{B}_{i,k}$ to its column of processors.

```
OUTER MatrixMultiplication(A,B,C)
q ← SQRT(NUM_PROSC())
myrow ← MY_PROC_ROW()
mycol ← MY_PROC_COL()
for k=0 to q-1 do
    for i=0 to q-1 do
        BROADCASTROW(i,k,A,bufferA,m×n)
    for i=0 to q-1 do
        BROADCASTCOL(i,k,B,bufferB,m×n)
    if (myrow=k)and(mycol=k) then
        MATRIXMULTIPLYADD(C,A,B,m)
    else
        if (myrow=k) then
            MATRIXMULTIPLYADD(C,bufferA,B,m)
        else
            if (mycol=k) then
                MATRIXMULTIPLYADD(C,A,bufferB,m)
            else
                MATRIXMULTIPLYADD(C,bufferA,bufferB,m)
```

$$T(m,q) = q(2T_{bcast} + m^3 w)$$

Better solution:

- Communications for $k = 0$
- Loop for $k = 0$ to $q - 2$ do:
  - Communication for step $k - 1$
  - Computation for step k
- Computation for step $q - 1$

$\Rightarrow$ communication and computation overlap 1-port

$$T(m,q) = 2T_{bcast} + (q-1)\max(2T_{bcast}, m^3 w) + m^3 w$$
$$T_{bcast} = (\sqrt{(q-2)L} + \sqrt{m^2 b})^2$$

- rings of size q
- matrices size of size m^2 are sent

$$T_{bcast} \underset{n\to+\infty}{\sim} \frac{n^2 b}{p}$$
$$T(m,q) \underset{n\to+\infty}{\sim} \frac{n^3}{p}$$
$$m = \frac{n}{p} \qquad p = q^2$$
$$qm^3 w = q\frac{n^3}{q^3} w$$
$$= \frac{n^3}{p} w$$

$\rightarrow$ there is a factor $\frac{\sqrt{p}}{2}$ between the two cost of communication.

Part IV
Introduction to distributed systems: the model

1 Transition system and algorithms

1.1 Transition systems

Discrete algorithms, algorithm being described by events. The processes communicate through messages (message passing), messages are asynchronous (no shared memory).

Definition 2 (Transition system)**.** *A transition system S is a triplet $(C, \rightarrow, I)$, with*

- *C: a set of configurations*
- *$\rightarrow$: a binary transition relation on C*
- *I: ($I \subset C$) the subset of the initial configurations*

Each process has a *state*. The set of all the states of the processes is called a *configuration*.

Definition 3 (Execution)**.** *Let $S = (C, \rightarrow, I)$. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$ where $\gamma_0 \in I$, and for all $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$*

A terminal configuration is a configuration γ such that there does not exist $\delta \in C$ such that $\gamma \rightarrow \delta$.

Definition 4. *A configuration γ is reachable if there exists $\delta_0 \in I$, and an execution $(\gamma_0, \gamma_1, \gamma_2, ..., \gamma_k)$ such that $\gamma_k = \gamma$.*

1.2 Systems with asynchronous message passing

A distributed system: A set of processes and a communication subsystem.

Each of the processes is a transition system by itself.

We use the words : *transition* and *configuration* for the whole system, and *event* and *state* for a single process.

Three types of events:

- Internal event
- Send event
- Receive event

Definition 5. *The local algorithm of a process is a quintuple* $(Z, I, \vdash^i, \vdash^s, \vdash^r)$ *where:*

- Z *is the set of states*
- $I \subset Z$ *is the set of initial states*
- $\vdash^i$ *is a relation on* $Z \times Z$
- $\vdash^s$ *and* $\vdash^r$ *are relations on* $Z \times M \times Z$ *where* M *is the set of possible messages.*

The binary relation $\vdash$ *on* Z *is defined*

$$c \vdash d \Leftrightarrow c \vdash^i d \vee \exists m \in M \ ((c, m, d) \in \vdash^s \cup \ \vdash^r)$$

A configuration of the system is defined by the state of each of the processes and the message present in the system.

Definition 6. *The transition system induced, under asynchronous communications, by a distributed algorithm for processes* $p_1, ..., p_n$ *where the local algorithm for process* p_i *is* $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$ *is* $S = (C, \to, I)$ *where*

1. $C = \{(c_{p_1}, ..., c_{p_n}, M)\}$ *with* $\forall p \in \underbrace{\{p_1, ..., p_n\}}_{=\mathbb{P}}, c_p \in Z_p$, *and* $M \in \underbrace{\mathbb{M}}_{multiset}\ (\ \underbrace{\mathcal{M}}_{set\ of\ messages}\ )$
2. $I = \{c_{p_1}, ..., c_{p_n}, M), \forall p \in \{p_1, ..., p_n\}, c_p \in I_p$ *and* $M = \emptyset\}$
3. $\to = \bigcup_{p \in \mathbb{P}} \to_p$
   $\to_{p_i}$ *the set of pairs* $(c_{p_1}, ..., c_{p_i}, ..., c_{p_n}, M_1), (c_{p_1}, ..., c_{p_{i-1}}, c'_{p_i} c_{P_{i+1}}, ..., c_{p_n}, M_2)$ *for which one of these conditions hold:*
   - $(c_{p_i}, c'_{p_i}) \in \vdash^i_{p_i}$ *and* $M_1 = M_2$
   - *for some* $m \in \mathcal{M}, (c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i}$ *and* $M_2 = M_1 \cup \{m\}$
   - *for some* $m \in \mathcal{M}, (c_{p_i}, m, c'_{p_i}) \in \vdash^r_{p_i}$ *and* $M_1 = M_2 \cup \{m\}$

2 Proving properties

2.1 Safety property

A safety property P is a property that is true in each configuration of each execution of the algorithm (P is always true). A safety property is called an invariant.

If γ is a configuration and P is a propriety, $P(\gamma)$ is the boolean value of P on the configuration γ.

We write $\{P\} \to \{Q\}$ to denote that for each transition $\gamma \to \delta$, if $P(\gamma)$ then $Q(\delta)$. Hence, $\{P\} \to \{Q\}$ means that if P holds before the transition, then Q holds afterwards.

Definition 7. *An assertion* P *is an invariant if*

1. $\forall \gamma \in I, P(\gamma)$
2. $\{P\} \to \{P\}$

Theorem 3. *If* P *is an invariant of* S*, then* P *holds for each configuration of each execution.*

2.2 Liveness property

Let P be an assertion true in some configuration of each execution of the algorithm (property P will eventually be true).

Let S be a system, and P a property. Let *term* be a predicate true in all terminal configuration and false otherwise.

Definition 8. *The system S terminates properly (or is deadlock-free) if the predicate $(term \Rightarrow P)$ is always true.*

Definition 9. *A partial order $(W, <)$ is well founded if there is no infinite decreasing sequence.*

Definition 10. *Let S be a transition system, and P an assertion. A function f from C to a well-founded set W is called a norm function (with respect to P) if for each transition $\gamma \to \delta$, then $f(\gamma) > f(\delta)$ or $P(\gamma)$*

Theorem 4. *Let S be a transition system, and P be an assertion. If S terminates properly and a norm function f (with respect to P) exists, then P is true in some configuration of each execution of S.*

3 Causal order of events and logical clocks

The view of execution as sequences of transitions naturally induces a notion of time. Events of an execution (of a distributed system) can *sometimes* be interchanged.

3.1 Independence and dependence of events

Two consecutive events that influence disjoint parts of the system are independent and can occur in reversal order (you can interchange them).

Theorem 5. *Let γ be a configuration, and let e_p be events of different processes p and q, both applicable in γ. Then e_p is applicable in $e_q(\gamma)$ and e_q is applicable in $e_p(\gamma)$ and $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.*

The premise of this theorem applies to any pair of events e_p and e_q except if *(i)* $p = q$ and except if *(ii)* e_p is a send event and e_q is the corresponding receive event.

Definition 11. *Let E be an execution. The relation $\prec$, called the casual order, on the events of the execution is the smallest relation that satisfies:*

1. *if e and f are events of the same process and e occurs before f, then $e \prec f$*
2. *if e is a send and f is the corresponding receive, then $e \prec f$*
3. *$\prec$ is transitive*

$a \preceq b$ if $a \prec b$ or $a = b$. $\preceq$ is a partial order. There may be event a and b such that neither $a \preceq b$ nor $b \preceq a$. Such events are called concurrent.

3.2 Equivalence of executions

Let $f = (f_0, f_1, f_2, ...)$ be a sequence of events. It is related to an execution $F = (\delta_0, \delta_1, \delta_2, ...)$ if for each i, f_i is applicable to δ_i and $f(\delta_i) = \delta_{i+1}$. F is called the implicit execution of f. Let us consider a permutation σ of the events. The permutation $(f_{\sigma(0)}, f_{\sigma(1)}, f_{\sigma(2)}, ...)$ of the events is consistent with the casual order if

$$f_{\sigma(i)} \prec f_{\sigma(j)} \Rightarrow i \leq j$$

Theorem 6. *Let $f' = (f_{\sigma(0)}, f_{\sigma(1)}, f_{\sigma(2)}, ...)$ be a permutation of the events of f that is consistent with the casual order. Then f'defines a unique execution F' starting in the execution of F. F' has as many events as F, and if F is finite, then F' has the same last configuration.*

Executions F and F' have the same collection of events, and the causal order of these events are the same. We say that execution F and F' are equivalent, and we denote $F \sim F'$.

Definition 12. *A computation if a distributed algorithm is an equivalence class under $\sim$ of executions of the algorithms.*

3.3 Logical clocks

Definition 13. *A clock is a function Θ from the events to an order set such that*

$$a \prec b \Rightarrow \Theta(a) < \Theta(b)$$

3.3.1 Order in sequence

Execution E defined by a sequence of events $(e_0, e_1, e_2, ...)$. Set $\Theta_g(e_i) = i$. This *cannot* be computed within the system, this cannot be computed by a distributed algorithm.

3.3.2 Lamport's logical clock

Event a. Let k be the length of the longest sequence such that $e_1 \prec e_e \prec ... \prec e_k = a$ then $\Theta_L(a) = k$. Obviously, $a \prec b \Rightarrow \Theta_L(a) < \Theta_L(b)$

a. if a is an internal event or a send event and a' is the previous event of the same process then $\Theta_L(a) = \Theta_L(a') + 1$

b. If a is a receive event, a' the previous event in the same process and b the corresponding send event, $\Theta_L(a) = 1 + \max(\Theta_L(a'), \Theta_L(b))$. In order to compute Θ_L, we add to each message the clock of the sending event.

Part V
Wave and traversal algorithms

Wave algorithms cover broadcasting, synchronisation, compute some global function (e.g., maximum of values stored among the processes).

1 Definition and use of the wave algorithm

- Topology is fixed (no communication link will appear during the execution of the algorithm)
- Communication links/channels are undirected: for two processes p and q, if p can send a message to q, then q can send a message to p.
- the set of processes is connected: $\forall p, q \in \mathbb{P}$, there is a path from p to q, where $\mathbb{P}$ is the set of processes and E a set of communication channels/links
- Asynchronous messages
- No global link

1.1 Definition of wave algorithms

- A special type of internal event: *decide*.
- If C is a computation, $|C|$ is the number of event in C.

Definition 14. *A wave algorithm is a distributed algorithm that satisfies 3 properties:*

1. *Termination: each computation C is finite, i.e. $|C| < +\infty$*
2. *Decision: each computation contains at least one decide event, i.e. $\forall C, \exists e \in C$, e is a decide event*
3. *Dependence: in each computation,* each *decide event is causally preceded by an event of* each *process, i.e. $\forall C, \forall e \in C : e$ is a decide event $\Rightarrow \forall p \in \mathbb{P}, \exists f \in C_p, f \preceq e$ where C_p is the set of events of computation C happening on process p)*

We call *wave* one computation of a wave algorithm; *indicators (or starters)* the processes that spontaneously start the algorithm and *non-initiate (or followers)* the processes that start their algorithm on reception of a message.

Differences among wave algorithms

1. *Centralized*: algorithm that always has exactly one initiator (the opposite is *decentralised*). Centralized = single-source
2. *Topology*: algorithm may be design for a special topology for any topology
3. *Initialisation*:
   (a) Process identity = each process knows its unique name
   (b) Neighbour's identity
4. *Number of decisions*: In all cases we are going to consider: at most one decision per process. However, one processor or all processors can take a decision (or in one case two decisions)
5. *Complexity*:

- The number of messages exchanged
- The number of bit exchanged
- (The time of a computation)

Often messages will be empty (signals)

1.2 Elementary results on wave algorithms

1.2.1 Structural properties

Lemma 6. *$\forall e \in C$, there exists an initiator p and an event $f \in C_p$ such that $f \preceq e$*

Proof. Let f be any minimal predecessor of e $f \preceq e$ and $\neg(\exists g, g \prec f)$. Let p be the process where f happens. If p is not an initiator, the first event g on p was a received event. Let h be the corresponding send event

$$\underbrace{h}_{\text{send}} \prec \underbrace{g}_{\text{receive}} \preceq f$$

which contradicts the definition of f. □

Lemma 7. *Let C be a wave with one initiator p, then for each non-initiator q, let $father_q$ be the neighbour of q which q received a message in its first event. Then the graph defined by $(\mathbb{P}, E_T)$ where $E_T = \{(q, father_q) \mid q \in \mathbb{P} \setminus \{p\}\}$ is a spanning tree.*

Proof. wave ⇒ at least one decide event ⇒ at least one event in each process ⇒ all processes (except maybe p) have received at least one message ⇒ $father_q$ was defined for all $q \in \mathbb{P} \setminus \{p\}$ we have exactly $N-1$ edges ($N = |P| - 1$). We must show that there is no cycle in the graph. By definition, if $(q, father_q) \in E_T$, then $father_q \prec q$ ⇒ no cycle possible. □

Lemma 8. *Let C be a wave and $d_p \in C$, d_p being a decide event. Then $\forall q \neq p$: $\exists f \in C_q$ ($f \preceq d_p \wedge f$ is a send event)*

Proof. Let $q \neq p$ be any process. By definition of wave algorithms, there exists an event $g \in C_q$ such that $q \preceq d_p$. We consider on (arbitrary chosen) causality path(?) between g and d_p. $g = g_0 \prec g_1 \prec g_2 \prec ... \prec g_n = d_p$ $g_0 \in C_q$. We consider the last event in $g_0, ..., g_{n-1}$ that is an event of q (say g_i). $g_i \in C_q, g_{i+1} \notin C_q$ ⇒ g_i is a send event. □

1.2.2 Lower bounds on the complexity of waves

(Lower bound on the number of messages)

Corollary of previous lemma: at least $N-1$ messages (sends) in any execution.

Theorem 7. *Let C be a wave with only one initiator p, such that a decide event occurs in p. Then at least N messages are exchanged in C.*

Theorem 8. *Let A be a wave algorithm, for arbitrary networks without any initial knowledge of the neighbour identities. Then A exchanges at least $|E|$ messages.*

Proof. By contradiction, there is a wave algorithm A and a graph $G = (\mathbb{P}, E)$ such that there exists a computation C for which A take a decision using at most $|E| - 1$ messages. ⇒ There is at least one edge $e = (p, q)$ such that no message is exchanged through that link.

Let $G' = G$ except I remove edge (p, q) and add a process z and two edges (p, z) and (z, q). I execute on G' all the events on C. This is possible because nothing on C dealt with the edge (p, q) I removed. As in G, A reaches a decision on G'. However, no events had place on process z, which contradicts the *dependence* property of wave algorithms. □

2 Some wave algorithm

2.1 Ring algorithm

The topology is a ring. Each process p has a dedicated neighbour $next_p$. We note $< tok >$ a token.

Algorithm

- For the initiator:
  send $< tok >$ to $next_p$
  receive $< tok >$
  decide
- For the non initiators:
  receive $< tok >$
  send $< tok >$ to $next_p$

Theorem 9. *The ring algorithm is a wave algorithm.*

Proof.
- Each execution has a finite number of events ($|C| \leq 2N + 1$)
- We must show that there is at least one decide event and that the decide event is preceded (causally) by an event in each other process. We look at a terminal configuration γ. Let p_0 be the initiator.
  - In γ, p_0 has sent a token (otherwise it can do it and γ is not terminal)
  - In γ there is no $< tok >$ message in the system (because it could be received and γ would not be terminal).
  - Each non initiators that received the token has sent it (otherwise γ non terminal) $\Rightarrow$ All non initiator processes has received the token, then p_0 has received the token, and p_0 has taken a decision, and this decision depends on a send in each other process.

□

2.2 The tree algorithm

- The topology of the network is a spanning tree (can be applied on arbitrary network with a spanning tree).
- All the leaves of the tree are initiators.

```
1  rec_p[number of neighbour of p] initialised to false
2  while |{q | rec_p[q] = false}| > 1 do
3  |   Receive < tok > from any q
4  |   rec_p[q] = true
5  send < tok > to q_0 such that rec_p[q_0]=false
6  receive < tok > from q_0
7  rec_p[q_0] = true
8  decide
9  (for all q ∈ Neighbour(p) ≠ q_0 do
10 |   send < tok > to q
11 ) // Propagation on decision
```

2.3 The echo algorithm

Centralized wave algorithm for any arbitrary network. The algorithm floods the network with $< tok >$ messages.

```
1  rec_p ← 0 // # of received message
2  father_p ← undefined
3  for The initiator do
4  |  for all q ∈ Neighbourhood(p) do
5  |  |  send < tok > to q
6  |  while rec_p < #Neighbourhood(p) do
7  |  |  receive < tok >
8  |  |  rec_p ← rec_p + 1
9  |  decide
10 for non initiators do
11 |  receive < tok > from some neighbour q
12 |  father_p ← q
13 |  rec_p ← rec_p + 1
14 |  for all q ∈ Neighbourhood(p) ∖ {father_p} do
15 |  |  send < tok > to q
16 |  while rec_p < #Neighbourhood(p) do
17 |  |  receive < tok >
18 |  |  rec_p ← rec_p + 1
19 |  send < tok > to father_p
```

The set of values of $faher_p$ define a spanning tree. One can prove that the decide event is causally preceded be an event in each process: We do that by induction on the spanning tree, starting by the leaves.

Each process holds a value (integer). How can I compute the minimum ?

2.4 Usage of wave algorithms

2.4.1 Computation of infinimum functions

If $(X, \leq)$ is a partial order. c is the infinimum of a and b if $c \leq a$ and $c \leq b$ and $\forall a(d \leq a$ and $d \leq b,$ then $d \leq c)$ infinimum of a and b $a \wedge b$

Infinimum computation: Each process q holds an input j_q, in a partially ordered set. Some processes compute the value of infinimum of $\{j_1\}_q$ and the processes knows when the computation has completed. They write the outcome of the computation as a variable out and are not allowed to changed it afterwards.

Theorem 10. *Every Infinimum computation algorithm is a wave algorithm.*

Theorem 11. *Every wave algorithm can be used to compute an infinimum.*

Theorem 12. *If $\star$ is a binary operator on a set X such that*

1. $\star$ is commutative

2. $\star$ is associative

3. $\star$ is idempotent ($a \star a = a$)

Then there is a partial order $\leq$ on X such that $\star$ is an infinimum function.

Corollary 1. *We can compute $\wedge, \vee, gcd, lcm, \min, \max, \cup, \cap$ of local values using wave algorithms*

2.4.2 Synchronisation

Synchronization algorithm: In each process q an event a_q must be executed and, in some process, an event b_p must be executed such that the execution of all a_q events must take place temporarily before any event b_p.

Theorem 13. *Every synchronization algorithm is a wave algorithm.*

Theorem 14. *Every wave algorithm can be employed as a synchronization algorithm.*

2.4.3 Propagation of information with feedback (PIF)

A subset of processes have a same message M which must be broadcast (all processes must receive it and acknowledge it). Certain processes must be notified of termination of the broadcast (they must execute a notify statement), but only after al processes have received the message. There is a finite number of messages exchanged.

Theorem 15. *Any PIF algorithm is a wave algorithm.*

Theorem 16. *Every wave algorithm can be used as a PIF algorithm.*

3 Traversal algorithm

Definition 15. *A traversal algorithm is a wave algorithm with the following time proprieties*

i. In each computation, there is a single initiator. The first time the initiator sends some messages, it sends a signle message and does not send any more message before receiving a message.

ii. A process upon reception of a message, either sends out one message or decides.

iii. The algorithm terminates in the initiator and when this happens each process has sent a message at least once

At any time there is at most one message in the system: there is a single token that is passed around processes. Traversal algorithms are wave algorithms in which all events are totally ordered by the causality relation.

3.1 Traversing cliques

Clique = complete graph

```
1  rec_p ← 0
2  for the initiator do
3  |  while rec_p < #Neighbourhood(p) do
4  |  |  send < tok > to q_{rec_p}
5  |  |  receive < tok >
6  |  |  rec_p ← rec_p + 1
7  |  decide
8  for the non-initiator do
9  |  receive < tok > form some process q
10 |  send < tok > to q
11 |  (the sequential polling algorithm)
```

3.2 Traversing connected network

```
1  used_p[q] ← false (∀q ∈ Neighbourhood(p))
2  father_p ← undefined
3  for the initiator do execute one
4  |  father_p ← p
5  |  choose q ∈ Neighbourhood(p)
6  |  used_p[q] ← true
7  |  send< tok > to q
8  for each process, upon reception of < tok > from q_0 do
9  |  if father_q is undefined then
10 |  |  father_p ← q_0
11 |  if ∀q ∈ Neighbourhood(p)such thatq ≠ father_p and used_p[q]isfalse then
12 |  |  choose q ∈ Neighbourhood(p) ∖ {father_p} such that used_p[q] is false
13 |  |  used_p[q] ←true
14 |  |  send < tok > to q
15 |  else
16 |  |  used_p[father_p] = true
17 |  |  send < tok > to father_p
```

4 Election algorithms

Problem: Start form a configuration where all processes are in the same state, and arrive in a configuration where exactly one process is in the state leader and all the other are in state lost.

Definition 16. *An election algorithm is an algorithm that satisfies the following proprieties:*

i. Each process has the same local algorithm

ii. The algorithm is decentralized (an arbitrary number of initiators)

iii. The algorithm reaches a terminal configuration in each computation, and in each computation, and each reachable terminal configuration, there is exactly one process in the state leader and all the other ones in the state lost.

Assumptions:

1. System is fully asynchronous.
2. Each process is identified by a unique name which it initially knows.

- We design election algorithms such that the process with the smallest name (identifier) is elected.
- We can use a wave algorithm to compute the minimum of the names.
- Our previous solution was the echo algorithm, a centralized algorithm. We can add to it a preliminary wake-up phase.

Election algorithm based on the tree algorithm.

- Boolean ws_p: whether process p has woken up.
- Integer $wr_p \leftarrow 0$: number of wake up message received.
- boolean $rec_p[q]$: whether process p receive a message (after the wake-up phase) from process q.

- v_p: temporary variable to store the id of the current minimum
- $state_p$: (sleep, leader, lost)

```
1  if p is the initiator then
2  |  ws_p ← true
3  |  for all q ∈ Neighbourhood(p) do
4  |  |  send < wake up > to q
5  while wr_p < #Neighbouhood(p) do
6  |  receive < wake up >
7  |  wr_p ← wr_p + 1
8  |  if not ws_p then
9  |  |  for all q ∈ Neighbourhood(p) do
10 |  |  |  send < wake up > to q
   // Start of the tree algorithm
11 while #{q | ¬rec_p[q]} > 1 do
12 |  receive < tok, r > from q
13 |  rec_p[q] ← true
14 |  v_p ← min(v_p, r)
15 send < tok, v_p > to q_0 such that rec_p[q_0] = false
16 receive < tok, r > from q_0
17 v_p ← min(v_p, i) // decide
18 if v_p = p then
19 |  state_p ←leader
20 else
21 |  state_p ←lost
22 for all q ∈ Neighbourhood(p) \ {q_0} do
23 |  send < tok, v_p > to q
```

5 Extinction principle and the echo algorithm

- We extend a wave algorithm
- All initiators are starting a wave
- We tag each wave by its initiator
- We distinguish messages and only forward those corresponding to the smallest initiator

caw : currently active wave.

```
   // Variables:
 1 caw_p ← undefined (minimum of the identifiers of encountered waves)
 2 rec_p ← 0 # of messages received for the wave caw_p
 3 father_p ←undefined (father in caw_p)
 4 lrec_p ← 0 # of leader messages received
 5 win_p ←undefined (identity of the leader)
 6 if p is initiator then
 7 |  caw_p ← p
 8 |  for all q ∈ Neighbourhood(p) do
 9 |  |  send < tok, caw_p > to q
10 while lrec_p < #Neighbourhood(p) do
11 |  receive msg from q
12 |  if msg =< leader, r > then
13 |  |  if lrec_p = 0 then
14 |  |  |  for all q' ∈ Neigbhourhood(p) do
15 |  |  |  |  send < leader, r > to q
16 |  |  lrec_p ← lrec_p + 1
17 |  |  win_p ← r
18 |  else
   |  |  // msg =< tok, r >
19 |  |  if r < caw_p then
20 |  |  |  caw_p ← r
21 |  |  |  rec_p ← 1
22 |  |  |  father_p ← q
23 |  |  |  for all s ∈ Neighbourhood(p) ≠ q do
24 |  |  |  |  send < tok, r > to s
25 |  |  else if r = caw_p then
26 |  |  |  rec_p ← rec_p + 1
27 |  |  |  if rec_p = #Neighbourhood(p) then
28 |  |  |  |  if caw_p = p then
29 |  |  |  |  |  for all s ∈ Neighbourhood(p) do
30 |  |  |  |  |  |  send < leader, p > to s
31 |  |  |  |  else
32 |  |  |  |  |  send < tok, caw_p > to father_p
33 if win_p = p then
34 |  state_p = leader
35 else
36 |  state_p = lost
```

Part VI
Task-graph scheduling

1 Introduction

Where do task graphs come from?

Linear system $Ax = B$. A is a lower triangular matrix (b and x are two vectors), b is known. A is a $n \times n$ matrix.

```
1 for i = 1 to n do
2 |  T_ii : x_i ← b_i / a_ii
3 |  for j = i + 1 to n do
4 |  |  T_ij : b_j ← b_j − a_{j,i}.x_i
```

Original algorithm is sequential.
Total ordering of the tasks:

$$T_{1,1} <_{seq} T_{1_2} <_{seq} T_{1,3} <_{seq} ... <_{seq} T_{1,n} <_{seq} T_{2,2} <_{seq} 2,3 <_{seq} ... <_{seq} T_{n,n}$$

Some parallelism: $T_{1,2}$ and $T_{1,3}$ are independent.
Condition for two tasks T and T' to be a dependence relation:

- both access the same variable and at least one access is a *write* access.
- $In(T)$: the set of variables read by task T
- $Out(T)$: the set of variable written by task T

T and T' are independent ($T \perp T'$)

$$T \perp T' \Leftrightarrow \begin{cases} In(T) \cap Out(T') \neq \emptyset & \text{or} \\ Out(T) \cap In(T') \neq \emptyset & \text{or} \\ Out(T) \cap OutT' \neq 0 \end{cases}$$

We define a partial order $\prec$ by:
if $T \perp T'$ and $T <_{seq} T'$ then $T \prec T'$
$\prec$ is $(<_{seq} \cap \perp)^+$ (transitive closure)
Representation of dependencies with a graph:

$$G = (\underbrace{V}_{\text{set of tasks}}, \underbrace{E}_{orientededges})$$

With

$$e = (T, T') \in E \Leftrightarrow T \prec T'$$
$$\big(\text{and } \nexists V \text{ s.t. } T \prec T' \text{ and } V \prec T'\big)$$

(we do not include transitive edges for the sake of readability)

2 Scheduling task graph

Definition 17. *A task graph is a directed weighted graph $G = (V, E, \omega)$*
Where:

- *The set V represent the tasks*
- *The set E of edges represents the dependences: $e = (u, v) \in E$ if and only if $u \prec v$.*
- *A weight function $\omega : V \to \mathbb{N}$ gives the weight or execution time of a test.*

Definition 18. *A schedule of a task graph $G(V, E, \omega)$ is a function $\sigma : V \to \mathbb{N}$ such that if $e = (u, v) \in E$ then $\sigma(u) + \omega(v) \leq \sigma(v)$. σ:starting time of the tasks.*

If we have a limited number p of processor (if $p < |v|$), *alloc*: $v \to \{1, ..., p\}$ states on which processor a task is executed.

$$alloc(u) = alloc(v) \Leftrightarrow \begin{cases} \sigma(u) + \omega v \leq \sigma(v) & \text{or} \\ \sigma(v) + \omega(v) \leq \sigma(u) \end{cases}$$

Theorem 17. *Let G be a task graph. There exists a schedule for G if and only if G does not contain any cycle.*

Proof.
- There is a cycle $T_1 \to T_2 \to ... \to T_k \to T_1$, so $T_1 \prec T_2 \prec ... \prec T_k \prec T_1$, and $T_1 \prec T_1$, $\sigma(T_1) + \omega(T_1) \leq \sigma(T_1)$ impossible.
- There is no cycle
- Topological order

```
1 pick a task T without predecessor
2 σ(T) ← t
3 t ← t + ω(T)
4 remove T from G
5 repeat
```

□

Definition 19 (Makespan)**.** *Task graph $G = (V, E, \omega)$*

1. *Let σ be a schedule for G using p processors. The makespan of G is the total execution time.*

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

   Usually we assume that $\min_{v \in V}\{\sigma(v)\} = 0$

2. *Pb(p): Problem of finding a best scheduling (i.e. a schedule of minimum makespan) using p processors. Let $MS_{opt}(p)$ be the makespan of an optimal scheduling using p processors.*

   $\omega : V \to \mathbb{N}^*$. We extend it on paths $\Phi : T_1, T_2, ..., T_k$ where $T_1 \prec T_2 \prec ... \prec T_k$ by $\omega(\Phi) = \sum_{T_i \in \Phi} \omega(T_I)$

Propriety 5. *Let $G = (V, E, \omega)$ be a DAG (directed acyclic graph) and σ a schedule for G with p processors. Then $MS(\sigma, p) \geq \omega(\Phi) \geq \omega(\Phi)$ for all paths Φ in G.*

Proof. Let Φ be any path.

$$\Phi = T_1 \prec T_2 \prec ... \prec T_k$$

$$T_i \prec T_{i+1} \Rightarrow \sigma(T_i) + \omega(T_{i+1} \leq \sigma(T_{i+1})$$

Sum for i to $d - 1$:

$$\sigma(T_1) + \sum_{i=1}^{k-1} \omega(T_i) \leq \sigma(T_k)$$

$$MS(\sigma, p) \geq \sigma(T_k) + \omega(T_1) \geq \omega(\Phi)$$

□

Definition 20. *$G = (V, E, \omega)$, σ on p procc.*

1. *Speedup ratio:*

$$s(\sigma, p) = \frac{seq}{MS(\sigma, p)} = \frac{\sum_{v \in V} \omega(v)}{MS(\sigma, p)}$$

2. Efficiency:

$$e(\sigma, p) = \frac{s(\sigma, p)}{p} = \frac{\sum_{v \in V} \omega(v)}{p.MS(\sigma, p)}$$

Theorem 18. *Let $G = (V, E, \omega)$ for any schedule G on p process $0 \leq e(\sigma, p) \leq 1$*

Proof. The area of a rectangle on the scheduling timetable is $p.MS(\sigma, p) = Idle + work = Idle + seq$ and $l = Idle + \frac{Seq}{p.MS(\sigma,p)} = Idle + e(\sigma, p)$ □

Theorem 19. *Let $G = (V, E, \omega)$ a task graph.*

$$seq = MS_{opt}(1) \geq ... \geq MS_{opt}(n)$$

Let $MS'(p)$ be the minimum makespan of all schedule that use exactly p processors. Then for all $1 \leq p \leq |V|$ $MS'(p) = MS_{opt}(p)$, because there is no communication (implicit hypothesis).

3 Solving problems (∞)

We have an unbounded number of resources ($p \geq |V|$). Let $G = (V, E, \omega)$ be a task system.

Definition 21. *1. $\forall v \in V$:*

- *$Pred(v)$ = set of immediate predecessors of a task v.*
- *$Succ(v)$ = set of immediate successors.*

2.
- *v is an entry task if $Pred(v) = \emptyset$*
- *v is an exit task if $Succ(v) = \emptyset$*

3. The top level $tl(v)$ is the maximum weight of a path from an entry task to the task v, excluding the weight of v.
$tl(v)$: lower bound on the time elapsed in any schedule between the start of the execution of G and the start of v.

$$tl(v) = \max_{v' \in Pred(v)} \{tl(v')\}$$

4. The bottom level $bt(v)$ is the maximum weight of a path from the task v (included) to an exit task (included).
$bt(v)$: lower bound on the execution time on any schedule once the execution of v has started.

$$bl(v) = \max_{v' \in Succ(v)} \{bl(v')\} + \omega(v')$$

Both can be computed through a traversal of G ($O(|V| + |E|)$)

Theorem 20. *Let $G = (V, E, \omega)$ be a task system. We define σ_{free} as follow:*

- *$\sigma_{free} = tl(v) \quad \forall v \in V$*
- *σ_{free} is an optimal schedule for G*

Proof. This is a schedule (dependences are satisfied by definition).
We show by induction (through a topological sort) that each task starts as soon, as possible.
ASAP schedule. □

Another optimal schedule: as late as possible schedule: $\sigma_{ALAP}(v) = MS_{OPT}(\infty) - bt(v)$

4 Solving Pb(p)

4.1 NP-completeness of Pb(p)

(2-)partition: given a set of n (strictly) positive integers $a_1, ..., a_n$, is there a subset I of $\{1, ..., n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$

NP-hard (weak sense, there is a pseudo polynomial algorithm to solve it)

3-partition: we have a set of $3n$ (strictly) positive integers $b_1, ..., b_n$. Let B be such that $\sum_{i=1}^{3n} b_i = nB$. We assume that for each $i \leftarrow \{1, ..., 3n\}$, $\frac{b}{4} \leq b_i \leq \frac{B}{2}$. Can we partition the $3n$ b_i's in n subsets $I_1, ..., I_n$ such that $\sum_{i \in I} b_i = B$?

Definition 22. *Let $Dec(p)$ the problem; let $G = (V, E, \omega)$ be a tsk system, let p be the number of processors, and let $k \in \mathbb{N}^*$, is there a schedule for G using at most p processors, whose makespan is no greater than k? If $E = \emptyset$ (no dependencies) we denote the problem by $Indep(p)$.*

Theorem 21.
- *$Indep(2)$ is NP-complete but can be solved in pseudo-polynomial time*
- *$Indep(p)$ is NP-complete in the strong sense*
- *$Dec(2)$ is NP-complete in the strong sense*

Proof.
- $Indep(2)$:
  - Problem belongs to NP, for each proc, check that the sum of the weights of the tasks associated to it does not exceed k.
  - Reduction from 2-partition:
    $k = \frac{1}{2} \sum_{i=1}^{n} a_i$, we have an identical problem.
- $Indep(p)$:
  Reduction from 3-partition: $b_1, ..., b_{3n}$ of 3-partition, we take $p = n$ and $k = B$.
- $Dec(2)$:
  Reduction from 3-partition
  - I_1 is an instance of 3-partition $b_1, ..., b_{3n}$ with $B = \frac{1}{n} \sum_{i=1}^{n} b_i$
  - I_2 an instance of $Dec(2)$
    We have $3n$ tasks: $T_1, ..., T_{3n}$, independent with $\omega(T_i) = b_i$
    We take $3n$ tasks $X_1, ..., X_n$, $Y_1, ..., Y_n$ and $Z_1, ..., Z_n$, with $\forall i \in \{1, ..., n\}$ $\omega(X_1) = \omega(Y_i) = \omega(Z_i) = B_i$.
    X_i depends on Y_{i-1} and Z_{i-1}, both depending from X_{i-1}.
  - $|I_2|$ is polynomial in $|I_1|$
  - Let us assume that I_1 has a solution $I_1, ..., I_n$ such that $\sum_{j \in I_j} b_j = B$. On processor P_1, I execute $X_1, Y_1, ..., X_n, Y_n$; on processor P_2, I execute Z_i when P_1 executes X_i.
    It works because $\omega(X_i) = \omega(Y_i) = \omega(Z_i) = B$ and because $\sum_{T \in I_i} \omega(T) = B$
  - Assume we have a solution to the scheduling problem $\Phi_1 = X_1 \to Y_1 \to X_2 \to Y_2 \to ... \to X_n \to Y_n$. Then $\omega(\Phi_1) = 2nB$, so there is no freedom on the execution time of these tasks. So $\sigma(X_i) = 2(i-1)B$, and $\sigma(Y_i) = (2i-1)B$. Besides, $\Phi_2 = X_1 \to Z_1 \to X_2 \to Z_2 \to ... \to X_n \to Z_n$, with $\omega(\Phi_2) = 2nB$ and $\sigma(z_i) = (2i-1)B$. Without loss of generality, we can assume that P_1 executes all the X_i's and Y_i's and P_2 executes all the T_i's and the Z_i's.
    We have exactly n disjoint intervals of size B to execute the T_i's on P_2.
    Let I_j = the subset of the T_i's that are executed by P_2 while P_1 executes X_j.
    This schedule is satisfying the bound $k = 2nB \Rightarrow \bigcup_{i=1}^{n} I_i = \{1, ..., 3n\}$, $\sum_{j \in I_i} \omega(T_j) \leq 3$, which gives a solution to the partition.

□

4.2 List scheduling heuristics

Historically: set priorities to tasks, put the tasks in a list ordered by these priorities and greedily schedule the tasks.

Principle: do not voluntarily let a process idle.

Definition 23. *Let $G = (V, E, \omega)$ be a task system, let σ be a schedule and let v be a task ($v \in V$). The task v is free at time t if and only if the processing of v has started but all its predecessors have completed.*

Theorem 22. *Let $G = (V, E, \omega)$ be a task system, and let σ be any list schedule for G, then*

$$MS(\sigma, p) \leq \left(2 - \frac{1}{p}\right) MS_{OPT}(p) \qquad \text{(Graham's bound)}$$

Proof.

Lemma 9. *There exists a dependence path Φ in G such that $Idle \leq (p-1)\omega(\Phi)$ with $Idle = \sum$ Idle times.*

Proof. Let T_1 be a task that completes last: $\sigma(T_1) + \omega(T_1) = MS(\sigma, p)$. Let t_1 be the last time before $\sigma(T_1)$ such that at least one processor was idle. T_1 could not be started at time t_1 because at least one task T_1 depends upon was still proceed at that time.

Let T_2 be one such task. Let t_2 be the last time before the start $\sigma(T_2)$ of T_2 at which one processor was idle.

I end up with a path $\Phi : T_k \to T_{k-1} \to ... \to T_2 \to T_1$. I can only have idle times while a task of Φ is executed: $Idle \leq \underbrace{(p-1)}_{\substack{\text{execution of the} \\ \text{tasks of } \Phi}} \omega(\Phi)$. □

$$pM(\sigma, p) = Idle + \sum_{i=1}^{n} \omega(T_i) \leq (p-1)\omega(\Phi) + \sum_{i=1}^{n} \omega(T_i)$$

$$MS(\sigma, p) \leq \left(1 - \frac{1}{p}\right) \omega(\Phi) + \sum_{i=1}^{n} (\omega(T_i)) \frac{1}{p} \leq \left(1 - \frac{1}{p}\right) MS_{OPT}(p) + MS_{OPT}(p) = \left(2 - \frac{1}{p}\right) MS_{OPT}(p)$$

□

Propriety 6. *Let $MS_{list}(p)$ be the shortest possible makespan produced by a list scheduling algorithm. Then the following bound is tight:*

$$MS_{list}(p) \leq \left(2 - \frac{1}{p}\right) MS_{OPT}(p)$$

Proof. Let $2p + 1$ task such that $T_1, ..., T_p$ are independent, and $T_{p+1}, ..., T_{2p}$ depending from T_p; and T_{2p+1} depending from the latter.

Consider a list schedule $\Rightarrow$ all entry task start at time 0. Let us say that task T_i for $1 \leq i \leq p-1$ is executed by P_i during $[0, k(p-1)]$. Then T_p is executed by P_p in $[0, 1]$. Wlog, P_p executes T_{p+1} during $[1, k+1]$, and wlog, P_p executes T_{p+i} during $[1 + (i-1)k; k + i + 1]$ for $i \geq 1$. (Sanity check: what would be the starting time of T_{2p}: $1 + (p-1)k$)

Hence: processor P_p executes $(p-1)$ of the tasks $T_{p+1}, ..., T_{2p}$ in the time interval $[1, 1 + (p-1)k]$. At time $(p-1)k$, all the processors $P_1, ..., P_{p-1}$ where available. One of them executes the list of the tasks $T_{p+1}, ..., T_{2p}$ during $[(p-1)K, pK]$.

Then one of the p processors starts task T_{2p+1} at time pk and completes at time $k(2p-1)$, so $MS_{list}(p) = k(2p-1)$. □

Optimal

- At time 0, P_1 executes T_p
- From 1 to $1+k$, each processor executes a task among $T_{p+1}, ..., T_{2p}$
- From $1+k$ through $1+k+k(p-1) = 1+kp$ each processor executes a task among $T_1, ..., T_{p-1}$ and T_{2p-1}

Performance ratio of list schedule:

$$\frac{k(2p-1)}{1+kp} \underset{k\to+\infty}{\longrightarrow} \frac{2p-1}{p} = 2 - \frac{1}{p}$$

4.3 Critical path scheduling

Definition 24 (Critical path)**.** *The critical path of a task is its bottom-level, i.e. the lower bound on the remaining execution time of the graph from the state of the execution of the considered tasks.*

Example: T_1 has no dependencies, T_2 is required for T_4, T_5, T_6 and T_7, T_8 depends on T_3.

task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
weight	3	2	1	3	4	4	3	6
critical path	3	6	7	3	4	4	3	6

With the heuristic of doing first the task with the maximum critical path, we end up with a schedule of 10. This is not optimal, as the optimal schedule weights only 9.

Lower bound on the makespan:

$$\left\lceil \frac{\sum_i \omega(T_i)}{p} \right\rceil = \left\lceil \frac{26}{3} \right\rceil = 9$$

5 Taking communication into account

Macro-dataflow model: let two tasks T and T' with $T \to T'$ ($\to$ meaning that some data produced by T is used by T'):

- If $alloc(T) = alloc(T')$, then there is no communication cost
- If $alloc(T) \neq alloc(T')$, then there is some communication cost $= c(T, T')$: does not depend on the choice of $alloc(T)$ and $alloc(T')$.

Assumptions (implicit ones

- Complete graph
- There are no contention between communications, i.e. the time of the communication is the same if there is one or n simultaneous input communications for a unique processor ($\simeq$ infinite input bandwidth)

Scheduling and communications: A communication DAG in a four-uplet $G = (V, E, \omega, c)$ with:

- V: tasks
- E: set of dependencies between tasks
- ω: execution times of the tasks $\omega : V \to \mathbb{N}^*$
- c : communication costs: $c : E \to \mathbb{N}$

A schedule σ must respect the dependences:

$$\forall e \in E \quad e = (T, T') \begin{cases} \sigma(T) + \omega(T) \leq \sigma(T') & \text{If } alloc(T) = alloc(T') \\ \sigma(T) + \omega(T) + c(T, T') \leq \sigma(T') & \text{otherwise} \end{cases}$$

6 Pb(∞) with communications

Problem:

- Sequential execution: we pay the cost of each task
- Each task on its own processor: we pay all the communications

6.1 NP-completeness of Pb(∞)

Decision problem Given a communication DAG $G = (V, E, \omega, c)$, and an execution bound k, is it possible to execute/schedule G in a time no greater than k?

Reduction from 2-Partition. n positive integers $a_1, a_2, ..., a_n$ with $\sum_{i=1}^{n} a_i = \alpha$. Is there a subset I of $\{1, ..., n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

We build an instance I_2 from the scheduling problem: tasks $T_1, ..., T_n$ (weights $2a_i$) depends on T_0 (weight A); and task T_{n+1} depends on $T_1, ..., T_n$.

The $2n$ edges has the same communication cost: C, any integer in the interval $]\alpha - \min_{1 \leq i] leq n} 2a_i, \alpha[$. $\forall i \ a_i > 0 \Rightarrow a_i \geq 1$ length of the open interval ≥ 2, so C exists. Let $K = 2A + C + \alpha$. The size of I_2 is polynomial in the size of I_1.

If I_1 has a solution I, then I_2 has a solution of makespan $\leq k$ by taking on P_0 the task of I and T_0, and on P_1 the tasks not in I and T_{n+1}.

Lemma 10. *T_0 and T_{n+1} are not executed on the same processor.*

Proof. By contradiction: T_0 and T_{n+1} are executed on the same processor, say P_0. Can all tasks be executed on P_0? $M = \omega(T_0) + \sum_{i=1}^{n} \omega(T_i) + \omega(T_{n+1} = 2A + 2\alpha$.

By definition $\alpha > C$. $M = 2A + 2\alpha > 2A + \alpha + C = k \ \Rightarrow$ at least one task T is not executed on P_0. As $T_0 \to T \to T_{n+1}$, we have to pay both communication costs.

$$\begin{aligned} M &\geq A + C + \omega(T) + C + A \\ &\geq 2A + 2C + \min_{1 \leq i \leq n} 2a_i \\ M &> 2A + 2C + (\alpha - C) = 2A + C + \alpha = K \end{aligned}$$

Contradiction. □

Let P_0 execute T_0 and P_1 execute T_{n+1}.

Lemma 11. *Each task is executed either by P_0 or P_1.*

Proof. By contradiction. There is a task T executed neither by P_0 nor P_1. But $T_0 \to T \to T_{n+1}$, the two communication take place, and $M \geq 2A + 2C + \omega(T) > K$. □

- Each task is executed either on P_0 (like T_0) or P_1 (like T_{n+1}). Let I be the set of indices of task among $T_0, ..., T_n$ executed on P_0, let I be the set of indices of task among $T_0, ..., T_n$ executed on P_1. $I \cup J = \{1, ..., n\}$
- Consider I:

$$\begin{aligned} K \geq M &\geq \omega(T_0) + \omega(I) + C + \omega(T_{n+1}) \\ &= 2A + \omega(I) + C \end{aligned}$$

- Consider J:

$$\begin{aligned} K &\geq \omega(T_0) + C + \omega(J) + \omega(T_{n+1}) \\ K &\geq 2A + C + \omega(J) \\ K &= 2A + \alpha + C \\ \alpha &\geq \omega(I) \\ \alpha &\geq \omega(J) \\ \omega(I) + \omega(J) &= 2\alpha \end{aligned}$$

  Hence, $\omega(I) = \alpha$ and I defines a solution to I_1.

□

In fact, it is NP-complete in the strong sense, even if all executions times are equal to 1 and all communication costs times are equal to 1 (UET-UCT).

7 List heuristics for Pb(p) with communications

Question How to extend the notion of critical path?

Solution Compute critical paths assuming that all communication take place.

7.1 Naive critical path

List schedule with bottom-levels *including* communications and processes are always considered on the same order.

7.2 Modified critical path

Principle Schedule the tasks of the processor that will enable to start it (= complete it) the earliest.

7.3 Two-step clustering heuristics

Clustering partitioning of the tasks

Given a clustering we compute bottom-levels and top level levels including a communication cost between two tasks if and only if they belong to 2 different clusters.

The task in a same cluster will be executed on the same processor.

Let $\mathcal{C}$ be a clustering.

$EPT(\mathcal{C})$ = estimated parallel time of $\mathcal{C} = \max_{v \in V}(tl(v) + bl(v))$.

Recall (case of a clustering)

- $tl(u) = \max(max_{in}, max_{out})$ with

$$max_{in} = \max_{\substack{v \in Pred(v) \\ \mathcal{C}(u)=\mathcal{C}(v)}} \{tl(v) + \omega(v)\}$$
$$max_{out} = \max_{\substack{v \in Pred(v) \\ \mathcal{C}(u)\neq\mathcal{C}(v)}} \{tl(v) + \omega(v) + C(u,v)\}$$

- $bl(u) = \max(max_{in}, max_{out}) + \omega(u)$ with

$$max_{in} = \max_{\substack{v \in Succ(v) \\ \mathcal{C}(u)=\mathcal{C}(v)}} \{bl(v)\}$$
$$max_{out} = \max_{\substack{v \in Succ(v) \\ \mathcal{C}(u)\neq\mathcal{C}(v)}} \{bl(v) + C(u,v)\}$$

I behave as if I had an infinite number of processors.

Kim and Browne linear clustering

- Take one longest dependence path in the graph; define a cluster from it; obtain a new cluster $\mathcal{C}'$; keep $\mathcal{C}'$ if and only if $EPT(\mathcal{C}) \leq EPT(\mathcal{C})$
- Iterate with the remainder of the graph

Sarkar's greedy clustering

- Sort edges by non-increasing communication costs
- For each edge on that order, merge the clusters containing the two extremities of the edges if this does not increase the EPT (initialise $\mathcal{C}_0 = \{\{T_1\}, ..., \{T_n\}\}$).

In the case of $\{T_3, T_4, T_5\}$ with $T_3 \to T_4$ and $T_3 \to T_5$, to compute the EPT, we need to sequentialize T_4 and T_5 (add virtual edge) because in the end they are going to be executed on the same processor.

Dominant Sequence clustering

- Initially: all the edges are marked non-examined
- While there remain non-examined tasks:
  - Pick a dominate sequence (DS)
  - Zero an edge in the DS
    - The edge that decrease the EPT the most (expensive)
    - an edge of maximum weight
    - the first edge

    We keep the clustering if EPT does not increase

Once we have clusters, we need to decide on which processors to map the clusters (and still some scheduling problem to solve if 2 clusters are mapped on the same processor).

Part VII
Automatic parallelization: the case of Lamport's hyperplane method

Hope

- to start from an existing sequential case/algorithm
- to automatically detect the parallelism present in the code/algorithm
- transform the code/rewrite it, to expose the parallelism
- execute the code efficiently

Limited context here:

- Uniform loop nests

1 Uniform loop nests and dependence analysis

Fragment of code:

$$S_1 : a \leftarrow b + 1$$
$$S_2 : b \leftarrow a - 1$$
$$S_3 : a \leftarrow c - 2$$
$$S_4 : d \leftarrow c$$

a: written by S_1 and, later, read by $S_2 \Rightarrow$ *flow* dependence
S_2: reads a and, later, S_3 overwrites it $\Rightarrow$ *anti* dependence
S_1 writes a and S_3, later on, overwrites it $\Rightarrow$ *output* dependence

Uniform loop nests: Set of instructions, all surrounded by the same set of loops (set of perfectly nested loops).

```
1 for i=0 to N do
2 |   for j=0 to N do
3 |   |   S1(i,j) : a(i,j) = b(i,b-6) + d(i-1,j+3)
4 |   |   S2(i,j) : b(i,j) = c(i+2,j+5) + 1
5 |   |   S3(i,j) : c(i+3,j-1) = a(i,j+2)
6 |   |   S4(i,j) : d(i,j-1) = a(i,j-1) - 1
```

Each loop has a loop counter and all counters are incremented by steps of 1.

Definition 25 (Iteration vector)**.** *The iteration vector I is the vector of loop counters.*

Remark Here, $I = \begin{pmatrix} i \\ j \end{pmatrix}$

Definition 26 (Iteration domain)**.** *The iteration domain Dom is defined as the set of values of the iteration vector.*

Remark Here,

$$Dom = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \ 0 \leq i, j \leq N \right\}$$

Definition 27 (Operation)**.** *An operation is an execution of an instruction for a particular value of the iteration vector.*

Original sequential order of execution Let $S(I)$ and $T(J)$ be two operations, I, J the iteration vector, S, T the instructions.

Definition 28 (Sequential order)**.**

$$S(I) <_{seq} T(J) \Leftrightarrow \Big((I = J) \ and \ (S <_{text} T) \Big) \ or \ \Big(I <_{lex} J \Big)$$

Dependences: There is a dependence from $S_i(I)$ to $S_j(J)$ (or $S_j(J)$ depends on $S_i(I)$) if:

- $S_i(I)$ is executed before $S_j(J)$
- $S_i(I)$ and $S_j(J)$ refers to a same memory location M, and at least one access is a write
- The memory location M is not written between the execution of $S_i(I)$ and of $S_j(J)$.

Definition 29 (Dependence vector)**.** *The dependence vector of $S_i(I) \to S_j(J)$ is defined by*

$$d_{i,I,j,J} = J - I$$

Definition 30 (Uniform loop nest)**.** *The loop nest is called uniform if the dependence vectors are independent of I and J.*

Remark A dependence vector is either null or lexicographically positive (i.e., its first non null component is positive).

Variable a: $S_1(i,j)$ 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 $S_i(i,j)$ to $S_4(i,j+1)$ and its dependence vector is $\begin{pmatrix} i \\ j+1 \end{pmatrix} - \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, which is a uniform dependence (constant vector, independent of i and j.

We must have $0 \leq i, j \leq N$ and $0 \leq i, j+1, N$.

Automation: For each statement pair and each variable accessed by both statements, we look whether there is a dependence.

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

$S_1(I)$ access $a(i,j)$, $S_3(J)$ access $a(i', j'+2)$

There is a dependence if and only of $i = i'$ and $j = j' + 2$ (both operations access the same memory location)

$$\begin{pmatrix} i \\ j \end{pmatrix} \preceq_{lex} \begin{pmatrix} i' \\ j' \end{pmatrix} \Leftrightarrow (i < i') \text{ or } (i = i' \text{ and } j = j')$$

Because $S_1(I)$ must precede $S_3(J)$.

No solution ! No flow dependence from S_1 to S_3 because of a.

Is there some dependence from $S_3(I)$ to $S_1(J)$ because of a ?

$$I = \begin{pmatrix} i \\ j \end{pmatrix} \preceq_{lex} J = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

$S_3(I)$ reads $a(i, j+2)$, $S_1(J)$ writes $a(i', j')$, $i = i$'and $j+2 = j$'.

There is anti-dependence from $S_3(I)$ to $S_1(J)$ of dependence $J - I = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$

- $S_1 \to S_4$ flow dependence $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$
- $S_3 \to S_1$ anti dependence $\begin{pmatrix} 0 \\ 2 \end{pmatrix}$
- $S_2 \to S_1$ flow dependence $\begin{pmatrix} 1 \\ 5 \end{pmatrix}$
- $S_3 \to S_2$ flow dependence $\begin{pmatrix} 1 \\ -6 \end{pmatrix}$
- $S_4 \to S_2$ flow dependence $\begin{pmatrix} 1 \\ -4 \end{pmatrix}$

Dependence matrix

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

Dependence graph One vertex per instruction.

The graph is an approximate representation, but this cause no problem because it is conservative, i.e. it is an over-approximation including more dependencies.

2 Lamport's hyperplane method

Basically We look for a schedule (once again) that satisfies the dependences.

$$Dom = \{p \in \mathbb{Z}^n \mid Ap \leq b, A \in \mathbb{Z}^{a \times n}, b \in \mathbb{Z}^a\}$$

Where n is the number of loops. Let D be the dependence matrix $(d_1, ..., d_m)$

$$\begin{aligned} Dom &= \left\{ \begin{pmatrix} i \\ j \end{pmatrix},\ 0 \leq i, j \leq N \right\} \\ &= \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \Big| \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ N \\ 0 \\ N \end{pmatrix} \right\} \end{aligned}$$

Let $p_1, p2 \in Dom$. If $\exists d \in D$ such that$p_2 = p_1 + d$ (p_1 should be executed before p_2).

Definition 31 (Schedule)**.** *A function $\sigma : Dom \to \mathbb{N}$ is a schedule if and only if*

$$\forall p_1, p_2 \in Dom, p_1 \prec p_2 \Rightarrow \sigma(p_1) \leq \sigma(p_2)$$

Definition 32 (Makespan of σ)**.**

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

The Lamport's method focus on linear schedule: a schedule σ can be defined by a n-dimensional vector $\pi \in \mathbb{Q}^n$.

$$\sigma_\pi(p) = \lfloor \pi \cdot p \rfloor$$

π defines an hyperplane.

Lemma 12. *If $f_d \in D$, $\pi \cdot d \geq 1$, then σ_π is a linear schedule.*

Proof. Let d be any dependence vector. Let p_1 and p_2 be such that $p_r = p_1 + d$

$$\begin{aligned} \pi \cdot d &\geq 1 \\ \Rightarrow \pi \cdot_1 + \pi \cdot d &\geq 1 + \pi \cdot p_1 \\ \Rightarrow \pi(p_2) = \pi \cdot (p_1 + d) &\geq 1 + \pi \cdot p1 \\ \Rightarrow \pi \cdot p_2 &\geq \lfloor 1 + \pi \cdot p_1 \rfloor = 1 + \sigma_\pi(p_1) \\ \Rightarrow \sigma_\pi(p_2) &\geq 1 + \sigma_\pi(p_1) \end{aligned}$$

And we need only a sufficient condition. □

Constructing schedule We use the property that the dependence vector is lexicographically positive. Assume matrix D is sorted in lexicographic order. Let k_1 be the first non-null component of d_1. $\pi_{k_1} = 1$ and $\pi_{k_2+1} = ... = \pi_n = 0$.

For all dependence vectors $d_1, ..., d_j$ whose first non-null component is k_1, we have $\pi \cdot d_l \geq 1 \quad (q \leq l \leq j)$.

Let k_2 be the first non null component of d_{j+1}. Let y_2 be the index of the last dependence vector whose first non null component is k_2.

$$\pi_{k_2+1} = ... = \pi_{k_1-1} = 0$$

$$\begin{aligned} k_2 - 1 &\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \\ k_2 &\to > 0 \\ &\vdots \end{aligned}$$

$$\begin{aligned} \pi_{k_2} &= x \\ (> 0) \star x + \alpha &\geq 1 \end{aligned} \qquad \text{(scalar product)}$$

Take for x $\max(1-\alpha)$

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

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

$$\pi_2 = 1$$

$$1 \star \pi_1 - 6 \geq 1$$

$$1 \star \pi_1 - 4 \geq 1 \Leftrightarrow \pi_1 \geq 7, \pi_1 = 7 \quad \begin{pmatrix} 7 \\ 1 \end{pmatrix}$$

$$1 \star \pi_1 + 5 \geq 1 \pi = \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\pi \cdot d \geq 1$$

$$\begin{cases} y \geq 1 \\ 2y \geq 1 \\ x - 6y \geq 1 \\ x - 4y \geq 1 \\ x + 5y \geq 1 \end{cases} \Leftrightarrow \begin{cases} y \geq 1 \\ x \geq 1 + 6y \end{cases}$$

Makespan:

$$\begin{aligned} T_{\sigma_\pi} &= 1 + \max_{p \in Dom} \{\sigma_\pi(p)\} - \min_{p \in Dom} \{\sigma_\pi\} \\ &= 1 + \max_{0 \leq i,j \leq N} \{\lfloor xi + yj \rfloor)\} - \min_{0 \leq i,j \leq N} \{\lfloor xi + yj \rfloor\} \\ &= 1 + \lfloor xN + yN \rfloor \\ &= 0 \end{aligned}$$

We want to minimize the makespan, so we want to minimize

$$1 + \lfloor (x+y)N \rfloor$$

with $y \geq 1$ and $x \geq 1 + 6y$ The optimal solution is $\begin{pmatrix} 7 \\ 1 \end{pmatrix}$

Code writing: the idea is to write under the form:

```
1 for time=time_min to time_max do
2 |   for all p ∈ Dom such that σ_π(p) = time do
3 |   |   S_1(p)
4 |   |   S_n(p)
```

With $\pi = \begin{pmatrix} 7 \\ 1 \end{pmatrix}$, $time = 7i + j$

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

Transform old coordinates in term of new ones

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

We had $0 \leq i, j \leq N$, $time = 7i + j$. Hence $time_{min} = 0$ and $time_{max} = 8N$.
$i = proc$ hence $0 \leq proc \leq N$, and $j = time - 7proc$, so $0 \leq time - 7proc \leq N$ So

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

```
1 for time=0 to 8N do
2 |   for all proc = max(0, ⌈(time−N)/7⌉) to min(N, ⌊time/7⌋) do
3 |   |   S_1(proc, time − 7proc) = b(proc, time − 7proc − α + d(proc − 1, time − 7proc + 3)
4 |   |   S_q...
5 |   |   ⋮
```