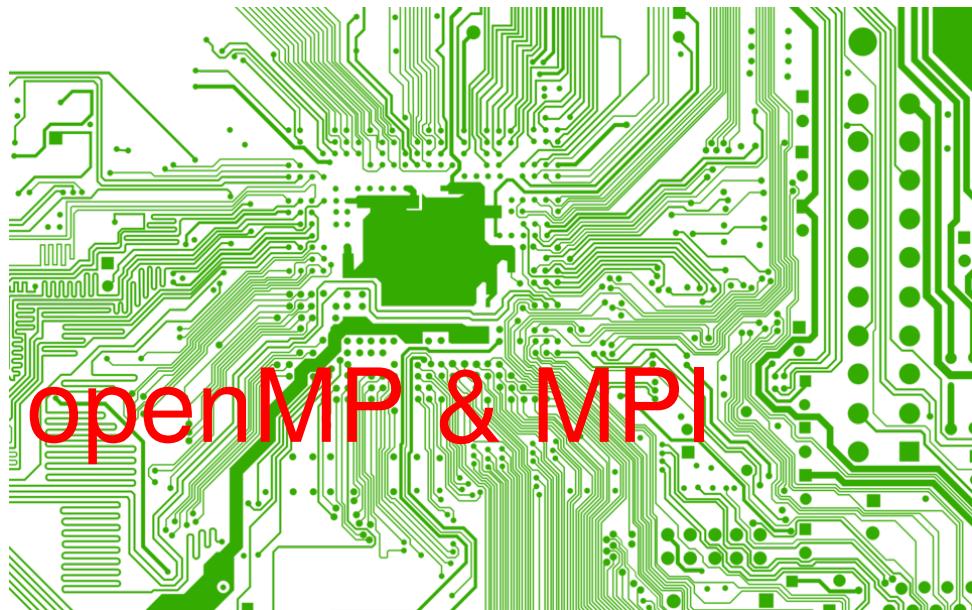


Faculté
des Sciences
& Techniques



Université
de Limoges

Master 1^{ère} année



Parallélisme & Application



P-F. Bonnefoi

Version du 14 janvier 2022

Table des matières

1	Les machines parallèles : différentes architectures	3
	Différentes approches matérielles	5
	Réseau InfiniBand de la société Mellanox	7
2	Recherche de la performance	35
	Accélération et efficacité	36
	Loi d'Amdahl	38
	Speedup	40



1 Les machines parallèles : différentes architectures

3

Notions de flot de calcul et de flot de données

Sur tout type de machine, un algorithme consiste en un **flot d'instructions** à exécuter sur un **flot de données**.

On a **quatre modèles** de calcul suivant qu'il existe un ou plusieurs de ces flots :

- ▷ Modèle SISD : *Single Instruction Single Data* ;
- ▷ Modèle MISD : *Multiple Instructions Single Data* ;
- ▷ Modèle SIMD : *Single Instruction Multiple Data* ;
- ▷ Modèle MIMD : *Multiple Instruction Multiple Data*.

Classification de Flynn

		Flot de données	
		Unique	Multiple
Flot d'instructions	Unique	SISD (Von Neumann)	SIMD (tab de processeurs)
	Multiple	MISD (pipeline)	MIMD (multiprocesseurs)



SISD

Notre ordinateur ? mais il est déjà superscalaire, multi-coeur...

MISD

Les machines vectorielles multi-processeurs :

- peut exécuter plusieurs instructions en même temps sur la même donnée (processeurs vectoriels et architectures pipelines)
- faible nombre de processeurs puissants (1 à 16)
- mémoire partagée
- limite atteinte, coût important

SIMD

Les machines **synchrones** :

- très grand nombre d'éléments de calcul (4096 à 65536) de faible puissance avec une toute petite mémoire locale
- un programme unique : exécution d'une même instruction sur des données différentes : GPU

MIMD

Les multi-processeurs à **mémoires distribuées** :

- grand nombre de processeurs ordinaires à mémoire locale
- communication par envoi de messages à travers des réseaux de communication
- chaque processeur a son propre programme

Les multi processeurs à **mémoire partagée** :

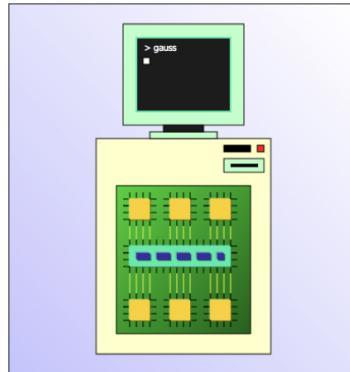
- Si le temps d'accès est égal pour chaque processeur à la mémoire, on parle de UMA, «*Uniform Memory Access*», ou «*Symmetric Multiprocessors*» (SMP) Exemple : un Core 2 Duo ou multi-cores...
- Si le temps d'accès n'est pas le même on parle de NUMA : «*Non Uniform Memory Access*».



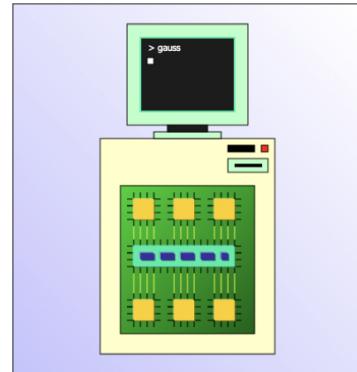
Différentes approches matérielles

5

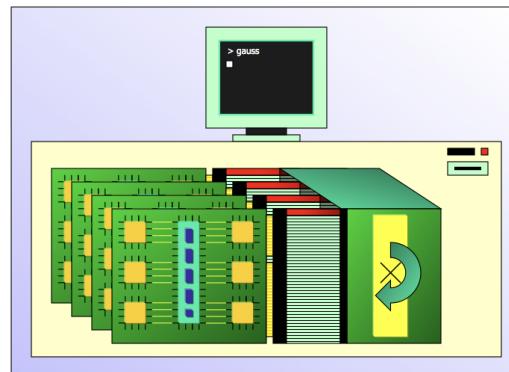
machine à **mémoire partagée**



machine à **mémoire distribuée**



machine **hybrides** : «*Non-Uniform Memory Access*»



Ferme, grappe ou cluster

6

Un ensemble de machines

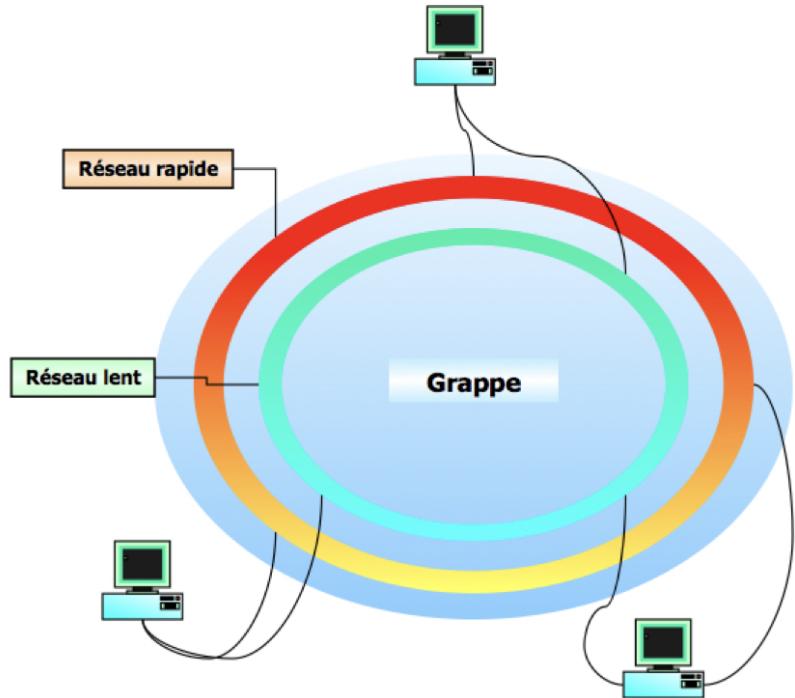
- des PC du commerce

Un réseau classique :

- lent
- réservé à l'administration

Un réseau rapide

- temps de transfert réduit
- débit élevé
- réservé aux applications



[Wikipedia](#)

InfiniBand (IB) is a computer networking communications standard used in high-performance computing that features **very high throughput** and **very low latency**.

As of 2014, it was the most commonly used interconnect in supercomputers. In 2016, Ethernet replaced InfiniBand as the most popular system interconnect of TOP500 supercomputers.

Characteristics										
		SDR	DDR	QDR	FDR10	FDR	EDR	HDR	NDR	XDR
Signaling rate (Gbit/s)		2.5	5	10	10.3125	14.0625	25.78125	50	100	250
Theoretical effective throughput (Gb/s)	for 1 link	2	4	8	10	13.64	25	50	100	250
	for 4 links	8	16	32	40	54.54	100	200	400	1000
	for 8 links	16	32	64	80	109.08	200	400	800	2000
	for 12 links	24	48	96	120	163.64	300	600	1200	3000
Encoding (bits)		8b/10b		64b/66b				PAM4	t.b.d.	
Adapter latency (μs)		5	2.5	1.3	0.7	0.7	0.5	less?	t.b.d.	t.b.d.
Year		2001 2003	2005	2007	2011	2011	2014	2018	2021	after 2023?

⇒2019: Nvidia acquired Mellanox for \$6.9B



Efficacité d'un algorithme



Un problème peut avoir **plusieurs solutions**, dont il faut pouvoir **sélectionner la meilleure** suivant différents critères.

Notion d'algorithme

Exemple 1 : Trier une séquence d'entiers de taille N dans l'ordre ascendant.

La solution est la séquence triée.

pour la séquence [10, 7, 11, 5, 13, 8] de taille 6, la solution est [5, 7, 8, 10, 11, 13]

Exemple 2 : Déterminer si le nombre X est dans la séquence S de N entiers.

La réponse est « oui » ou « non »

pour X = 5 et la séquence [10, 7, 11, 5, 13, 8] de taille 6, la solution est « oui »

Un **algorithme**, pour l'exemple 2, décrit la recette (méthode) mise en oeuvre, sous forme de suite d'instructions, exécutées une par un une successivement :

- commencer à partir du 1er élément de S,
- comparer chaque élément à X jusqu'à trouver une égalité (réponse = « oui ») ou bien jusqu'à l'épuisement de tous les éléments de S (réponse = « non »).

Évaluation d'un algorithme

Différents éléments à prendre en compte :

- langage de programmation ;
- structures de données ;
- l'efficacité de l'algorithme proposé : **complexité en temps d'exécution** et en **espace mémoire**.

Programme = algorithme + structures de données dépendance entre les deux...



Efficacité des algorithmes et amélioration des machines

- augmentation de la vitesse des processeurs ;
- augmentation de la quantité de RAM ;
- diminution du prix associé ;

Mais, l'efficacité des algorithmes reste **important** : augmentation du nombre de tâches réalisées, programmes de plus en plus conviviaux, etc.

Le choix d'un algorithme pour la résolution d'un problème

Les algorithmes candidats se distinguent par leur niveau d'utilisation de deux ressources :

- le temps CPU ;
- la place mémoire.

Un **algorithme efficace** utilise aussi peu que possible chacune de ces ressources.

Compromis espace/temps

Souvent, un gain dans l'utilisation de l'une des ressources se traduit par une perte au niveau de l'autre.

Temps d'exécution estimé

Exemple : recherche d'un élément dans une séquence ordonnée S,
on préfère une méthode de **recherche binaire** plutôt qu'une **recherche séquentielle**.

Pourquoi ?

Le nombre de comparaisons

Taille de S	Recherche séquentielle	Recherche binaire
$128=2^7$	128	8
$1024=2^{10}$	1024	11
$1\ 048\ 576=2^{20}$	1 048 576	21
$4\ 294\ 967\ 296=2^{32}$	4 294 967 296	33



Sur l'exemple : recherche d'un élément dans une séquence ordonnée S, on préfère une méthode de **recherche binaire** plutôt qu'une **recherche séquentielle**.

Intuition : la seconde méthode est (beaucoup) plus performante que la première :

- dans le premier cas, recherche séquentielle, chaque test réduit l'espace de recherche" d'un élément ;
- dans le second, recherche binaire, chaque test le réduit de moitié.

Définition rapide : lors de la recherche d'une **information aléatoire** dans un **tableau aléatoire**, mais **trié**, la première méthode demande, *statistiquement*, un nombre de tests proportionnel au nombre d'éléments stockés dans le tableau : il faudra **en moyenne** parcourir la **moitié** de ce dernier pour obtenir la réponse cherchée (l'information est ou non dans le tableau).

Pour la seconde méthode, comme on **divise par deux** la taille du tableau à explorer après chaque test, il est facile de se persuader que le nombre de tests à effectuer est **proportionnel** au logarithme de la taille du tableau initial.

Quantification de la différence :

si l'on traite une grande quantité d'informations :

- par exemple un dictionnaire (1 000 000 mots et définitions) ;
- un test d'égalité de mots prend une milliseconde de temps de calcul ;

La première méthode fournit sa réponse en $5 \cdot 10^5 * 10^{-3} = 500$ s, soit environ **huit minutes**.

La seconde méthode réclame $10^{-3} * \log_2(10^6)$ ms, soit environ **20 millisecondes**...



Indépendance vis à vis du contexte

Les **mesures réalisées** doivent être indépendantes d'un contexte particulier :

- on ne tient pas compte du **nombre de cycle d'horloge** du CPU (CISC ou RISC etc.) ;
- on ne calcule pas le **nombre d'instructions exécutées** car cela dépend du langage, du style de programmation (impérative, fonctionnelle, logique) et du compilateur ;

Proportionnalité

Le **temps d'exécution** augmente avec la **taille des données** en entrée.

Le temps total d'exécution est proportionnel au nombre de fois que l'on exécute les opérations de base (comparaison par exemple).

Le nombre de fois où les opérations de base sont effectuées est fonction de la taille des entrées, voire fonction de chaque valeur d'entrée.

La **complexité en temps** $T(n)$ doit tenir compte de tous les cas possibles, «*Every case complexity*».

La notion d'opération de base

- ajout d'un élément dans un tableau
calcul de la séquence de Fibonacci, $\text{fib}(0) = 0$, $\text{fib}(1) = 1$ et $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

$$T(n) = n$$

- comparaison de deux éléments
tri par échange, $T(n) = n(n - 1)/2$
- la multiplication de deux valeurs
multiplication de matrices

$$T(n) = n^3$$



Exemple : le tri par échange

13

Soit le tableau

```
□―― xterm ━━  
9 8 7 6 5 4  
8 9 7 6 5 4  
8 7 9 6 5 4      n - 1 comparaisons  
8 7 6 9 5 4  
8 7 6 5 9 4  
8 7 6 5 4 9  
  
7 8 6 5 4 9      n - 2  
7 6 8 5 4 9  
7 6 5 8 4 9  
7 6 5 4 8 9  
  
6 7 5 4 8 9  
6 5 7 4 8 9      n - 3  
6 5 4 7 8 9  
  
5 6 4 7 8 9      n - 4  
5 4 6 7 8 9  
  
4 5 6 7 8 9      n - 5  
  
6 * 5 / 2 = 15 comparaisons
```

$$\sum_{i=1}^n i = n \frac{n-1}{2}$$



Exemple : la **suite de fibonacci**

- $\text{fib}(0) = 0$
- $\text{fib}(1) = 1$
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

Pour calculer $\text{fib}(n)$, il existe deux méthodes parmi les plus connues :

- la **solution récursive** qui conduit à de nombreux calculs redondants
Le nombre de termes calculés est de l'ordre de $2^{(n/2)}$
- la **solution utilisant un tableau F** où, pour $2 \leq n$, $F[i] \leftarrow F[i-1]+F[i-2]$ avec $F[0] = 0$ et $F[1] = 1$
Le nombre de termes calculés $T(n)$ est $n + 1$

Comparaison du temps d'exécution en fonction de n sur une machine qui calcule chaque terme en une nano seconde (10^{-9} sec) :

n	$2^{n/2} \text{s}$	Temps Algo linéaire ($n+1$) termes	Temps Algo récursif
60	1.110^9	61ns	1s
80	1.110^{12}	81ns	18min
100	1.210^{15}	101ns	13 jours
120	1.210^{18}	121ns	36 années
200	1.110^{30}	201ns	4.10^{13} années

D'autres problèmes sont complexes : factorielle, test de primalité $2^{p-1} \bmod p = 1$, pour $p > 2$, etc.



Évaluation de $T(n)$

Si $T(n)$ est impossible, ou trop difficile à calculer, on calcule le pire cas «*Worst-case complexity*».

Exemple : dans le cas du tri par échange, la complexité du pire des cas passe de $n(n - 1)/2$ à n^2 .

Le calcul du **temps moyen** d'un algorithme est difficile :

- ▷ on utilise la complexité générale en temps $T(n)$ sinon le pire des cas pire ;

Pourquoi ?

Le calcul de la **complexité moyenne** a besoin d'assigner des **probabilités** à toutes les informations disponibles en entrée.



Exemple : recherche de X dans S[1..n]

- ▷ chaque élément de S a une chance de $1/n$ d'être X ;
- ▷ la probabilité pour que, $1 \leq k \leq n$, $S[k] = X$ est $1/n$;
- ▷ le nombre d'opérations pour trouver l'élément tel que $S[k] = X$ est k ;
- ▷ pour arriver à l'indice k , on aura fait un nombre de comparaisons égal à :

on fait la somme des coûts de chaque événement pondéré par la probabilité (ou exclusif)

$$A(N) = \sum_{k=1}^n (k * \frac{1}{n}) = \frac{1}{n} * \sum_{k=1}^n k = \frac{1}{n} * \frac{n*(n+1)}{2} = \frac{n+1}{2}$$

espérance mathématique

- ▷ Il faut évaluer le cas où X n'est pas dans S :
 - ◊ la probabilité pour que X soit dans $S = p$;
 - ◊ la probabilité pour que X soit à $S[k]$ est p/n (et probabiliste, avec événements dépendants) ;
 - ◊ la probabilité pour que X ne soit pas dans $S = 1-p$;

Sachant que l'on fait k comparaisons si $S[k] = X$ et n comparaisons si X n'est pas dans S on a la complexité moyenne : soit l'élément est dans l'ensemble, soit il ne l'est pas (dépend de p)

$$A(N) = \sum_{k=1}^n (k * \frac{p}{n}) + n * (1 - p) = \frac{p}{n} * \frac{n*(n+1)}{2} = n * (1 - \frac{p}{2}) + \frac{p}{2}$$

ce qui donne :

- ◊ Si $p = 1$, $A(n) = (n + 1)/2$, X est dans S
- ◊ Si $p = 1/2$, $A(n) = 3n/4 + 1/4$, ce qui veut dire que seul $\frac{3}{4}$ de S est recherché en moyenne !

Remarques : on a supposé une **probabilité identique** pour chaque élément du tableau. Sinon, il faut modifier le calcul en tenant compte d'une distribution différente des valeurs : *telle valeur est plus fréquente ou une table de noms/mots avec la fréquence des premières lettres (E plus courant que Z)...*



Autres remarques dans le cas du calcul de la complexité moyenne

Pour le calcul, les données peuvent varier en fonction du **domaine d'application** (par exemple d'une entreprise à l'autre).

Il faut donc avoir une idée de la **distribution** et la **probabilité** de la taille N des données avant de pouvoir faire le calcul !

Pour le calcul de la complexité, on pourra dire que l'on fait au moins n opérations et au plus m opérations...

Il est difficile de déterminer n et m pour une taille N donnée.

Il faut également trouver quelle est la moyenne de N (distribution, probabilité, etc).

C'est pourquoi on s'intéresse au pire des cas pour obtenir une **borne supérieure** de la complexité !

Il est également nécessaire de **formaliser** ce que l'on évalue et tenir compte de la **nature de l'algorithme** (comment il est constitué).



Le calcul précis de la complexité d'un algorithme est en général impossible, il faut travailler sous un ensemble **d'hypothèses simplificatrices**. Les **hypothèses** sont les suivantes :

- on choisit explicitement **une ou plusieurs opérations** considérées comme **élémentaires** (ou fondamentales), c-à-d nécessitant une **unité de temps**.

Dans la plupart des cas : une **affectation** ou une **comparaison** de scalaires.

La mesure de la complexité se fera dans cette unité.

- les algorithmes présentent dans leur déroulement des **bifurcations** (clauses conditionnelles), ce qui empêche de calculer **exactement** la complexité de manière statique : les branches de l'alternative peuvent ne pas toutes avoir le même coût !

Pour simplifier, on se place dans trois situations représentatives :

- ◊ le **meilleur cas** : on considère que le programme emprunte systématiquement la branche la moins coûteuse de l'alternative,
- ◊ le **pire des cas** : on considère qu'il emprunte systématiquement la plus coûteuse ;
- ◊ le **cas moyen**, qui repose sur une **estimation probabiliste** du passage dans chacune des branches et effectue une somme pondérée des coûts de passage dans chaque branche.

L'intérêt de l'estimation de la complexité dans le cas moyen est évident, mais c'est bien sûr le **calcul le plus difficile** à effectuer.

C'est pourquoi on calcule souvent la **complexité dans le pire des cas**, puisqu'elle fournit une **borne supérieure** au temps de calcul du programme.



Borne supérieure

On choisit le **cas pire** pour obtenir une **borne supérieure** de la complexité, notée $O(f(N))$

Exemple : la recherche d'un élément dans un tableau de N éléments est $O(N)$, ne signifie pas que pour chaque élément du tableau on exécute une instruction.

La mesure $O(f(N))$ donne un **ordre de croissance** de l'algorithme.

Exemple : $O(N)$ signifie que l'algorithme croît proportionnellement à N .

Le «best case», c-à-d le plus favorable serait $B(N)$.

Exemple : recherche d'un élément dans un tableau, $B(N) = 1$!

Définition

but : disposer d'une mesure indépendante de la machine et du langage, pour avoir une idée du comportement et quelles ressources (temps et espace) il aura besoin. *On refusera les algos qui prendront 10 ans de calcul ou plusieurs giga-octets.*

La **complexité d'un algorithme A** est une définition $C_A(N)$ donnant le **nombre d'instructions** caractéristiques exécutées par A dans le **pire des cas**, pour des données de taille N .

Ce calcul, comme on l'a vu, peut être optimiste (best case), pessimiste (worst case) ou moyen (difficile à déterminer).

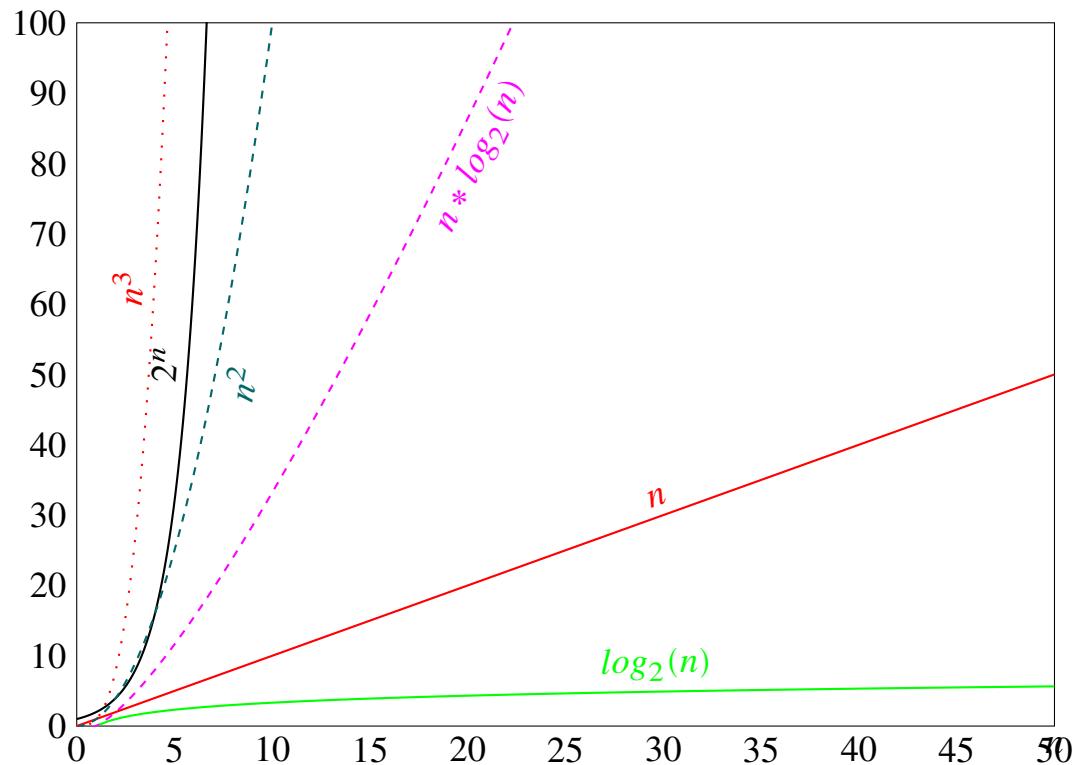
La notation $O()$ permet de mesurer les performances, elle donne une limite supérieure modulo un facteur constant (ce facteur correspond à la nature de la machine).

Exemple : recherche d'un élément dans un tableau de taille N

- optimiste : $O(1)$, on trouve en premier ;
- pessimiste : $O(N)$, on trouve en dernier ;
- moyen : voir calcul de $A(N)$ fait précédemment.

Le **cas pessimiste** permet de **garantir un délai** (application temps réel).





Les ordinateurs ne peuvent résoudre tous les problèmes combinatoires.

Il faut se méfier des **méthodes énumératives** qui examinent tous les cas possibles, elles sont souvent **exponentielles** :

- l'énumération des parties d'un ensemble de n est $O(2^n)$;
- celle d'affectation de n objets à n autres est $O(n!)$;
- n décisions successives avec K possibilités à chaque fois génèrent k^n possibilités.

La croissance relative

Dans l'ordre croissant :

- c constante \Rightarrow accès à un élément dans un tableau ;
- $\log(N)$, logarithmique \Rightarrow couper un ensemble en 2 parties égales, recouper...
- \sqrt{N} \Rightarrow utilisation dans les nombres premiers ;
- $\log^2(N)$, log au carré \Rightarrow recherche dans un B-arbre ;
- N , linéaire \Rightarrow parcours d'un ensemble de données ;
- $N * \log(N)$ \Rightarrow couper un ensemble en 2 + parcours de chaque partie ;
- N^2 , quadratique \Rightarrow parcourir un ensemble une fois par élément d'un autre ensemble de la même taille ;
- N^3 , cubique \Rightarrow triple boucle ;
- 2^N , exponentiel \Rightarrow générer tous les sous-ensembles d'un ensemble de données ;
- $N!$, exponentiel \Rightarrow toutes les permutations d'un ensemble de données



Exemple 1

22

Soit T un tableau de N entiers.

Créer le tableau A tel que $A[i] = \text{moyenne de } T[0] .. T[i]$.

$$A[i] = \frac{\sum_{j=0}^i T[j]}{i+1} \quad i \text{ commence à 0}$$

Pour chaque élément de A , on calcule la moyenne.

```
□ — xterm —
int *moyennes1(int *T, int N)
{
    int *A = (int *) malloc(sizeof(int) * N);
    for (int i=0; i<N; i++)
    {
        a=0;
        for (int j=0; j<=i; j++)
            a = a + T[j];
        A[i] = a / (i+1);
    }
    return A;
}
```

Calcul de la complexité ?



Exemple 1

23

Soit T un tableau de N entiers.

Créer le tableau A tel que $A[i] = \text{moyenne de } T[0] .. T[i]$.

$$A[i] = \frac{\sum_{j=0}^i T[j]}{i+1} \quad i \text{ commence à 0}$$

Pour chaque élément de A , on calcule la moyenne.

```
□ — xterm —
int *moyennes1(int *T, int N)
{
    int *A = (int *) malloc(sizeof(int) * N);
    for (int i=0; i<N; i++)
    {
        a=0;
        for (int j=0; j<=i; j++)
            a = a + T[j];
        A[i] = a / (i+1);
    }
    return A;
}
```

Calcul de la complexité ?

- ▷ la boucle externe est exécutée N fois ;
- ▷ la boucle interne est exécutée au pire N fois et au mieux 1 fois :

$$1 + 2 + \dots + N = n(n + 1)/2$$

- ▷ La complexité est $O(n^2)$.



Exemple 1 : autre proposition

24

On conserve en permanence (dans S) la somme des éléments $T[0] .. T[i]$.

```
□ — xterm —
int *moyennes2(int *T, int N)
{
    int *A = (int *) malloc(sizeof(int)*N);
    int s=0;
    for (int i=0; i<N; i++)
    {
        s=s+T[i];
        A[i] = s / (i+1);
    }
    return A;
}
```

Calcul de la complexité ?



Exemple 1 : autre proposition

25

On conserve en permanence (dans S) la somme des éléments $T[0] .. T[i]$.

```
□ — xterm —
int *moyennes2(int *T, int N)
{
    int *A = (int *) malloc(sizeof(int)*N);
    int s=0;
    for (int i=0; i<N; i++)
    {
        s=s+T[i];
        A[i] = s / (i+1);
    }
    return A;
}
```

Calcul de la complexité ?

- ▷ la boucle est exécutée N fois;
- ▷ la complexité est en $O(n)$!



Soit deux tableaux T_1 et T_2 d'entiers de taille N dont les éléments sont uniques.

Vérifier que les deux tableaux **contiennent les mêmes éléments**.

Première version

- dans cette première version, on vérifie que chaque élément de T_1 figure dans T_2 .
- on sait que les deux tableaux ont la même taille.

```
□—— xterm ——  
bool egaux(int *T1, int *T2, int N)  
{  
    for (int i=0; i<N; i++)  
    {  
        for (int j=0; j<N; j++)  
            if (T1[i] == T2[j]) break; //on a trouve, sortie boucle interne  
  
        if (j == N) // on a jamais trouve T1[i] dans T2  
            return false;  
    }  
    return true;  
}
```

Calcul de la complexité ?

Soit deux tableaux T_1 et T_2 d'entiers de taille N dont les éléments sont uniques.

Vérifier que les deux tableaux **contiennent les mêmes éléments**.

Première version

- dans cette première version, on vérifie que chaque élément de T_1 figure dans T_2 .
- on sait que les deux tableaux ont la même taille.

```
□—— xterm ——  
bool egaux(int *T1, int *T2, int N)  
{  
    for (int i=0; i<N; i++)  
    {  
        for (int j=0; j<N; j++)  
            if (T1[i] == T2[j]) break; //on a trouve, sortie boucle interne  
  
        if (j == N) // on a jamais trouve T1[i] dans T2  
            return false;  
    }  
    return true;  
}
```

Calcul de la complexité ?

- ▷ la boucle externe est exécutée N fois ;
- ▷ la boucle interne est exécutée au pire N fois $\Rightarrow O(n^2)$



Exemple 2 : autre version

28

Dans cette deuxième version, on peut utiliser un algorithme efficace de tri des tableaux en $O(N * \log(N))$ puis une comparaison $O(N)$.

```
□ — xterm —
bool egaux(int *T1, int *T2, int N)
{
    int * Temp1 = (int *) malloc (sizeof(int)* N) ;
    int * Temp2 = (int *) malloc (sizeof(int)*N) ;
    sort(T1, Temp1);
    sort(T2, Temp2);
    // en cas d'égalité des deux, Temp1[i] = Temp2[i], pour i=0..N-1
    for (int i=0; i<N; i++)
        if (T1[i] != T2[i]) return false;
    return true;
}
```

Calcul de la complexité ?

Exemple 2 : autre version

29

Dans cette deuxième version, on peut utiliser un algorithme efficace de tri des tableaux en $O(N * \log(N))$ puis une comparaison $O(N)$.

```
└── xterm └──
    bool egaux(int *T1, int *T2, int N)
    {
        int * Temp1 = (int *) malloc (sizeof(int)* N) ;
        int * Temp2 = (int *) malloc (sizeof(int)*N) ;
        sort(T1, Temp1);
        sort(T2, Temp2);
        // en cas d'égalité des deux, Temp1[i] = Temp2[i], pour i=0..N-1
        for (int i=0; i<N; i++)
            if (T1[i] != T2[i]) return false;
        return true;
    }
```

Calcul de la complexité ?

- ▷ le tri des deux tableaux : $2 * O(N * \log(N))$
- ▷ la boucle $O(N)$
- ▷ la complexité $O(N * \log(N))$

Pire ? trier les deux tableaux et appliquer la première méthode !

Encore pire ? trier les deux tableaux en utilisant un algorithme en $O(N^2)$ et la première version de l'algorithme !

Que se passe t il si les éléments ne sont pas forcément uniques ?

Exercice : vérifier que deux chaînes sont des anagrammes l'une de l'autre. calcul de complexité.



Il existe des algorithmes en $O(n * \log^2(n))$ pour trier, si l'on utilise comme opération de base la comparaison entre deux nombres.

Remarques : tous les logarithmes sont du même ordre et on peut utiliser le lograithme népérien
 $\log_b(x) = \log_a(x)/\log_a(b)$

Exemple : $\log_{10}(x) = \log_2(x)/\log_2(10)$ ce qui vaut à peu près $\frac{1}{3}\log_2(x)$

Tri à bulle

```
□ — xterm —
for (i = 1 ; i <= N-1 ; i++)
    for(j = i + 1; i <= N ; j++)
        if (T[i] > T[j]) échanger (T, i, j);
```

Complexité ?



Tri par insertion

```
for (j=2 ; i <= N; j++)
{
    x = T[j]; // T[j] à insérer dans T[1..j-1]
    i=j-1; // position de l'élément à décaler
    while (i>0 && T[i] > x)
    {
        T[i+1]=T[i]; // décalage
        i = i - 1;
    }
    T[i+1]=x; // Insertion de x à sa place
}
```

Complexité ?



Si on décompose l'exposant en binaire :

```
Initialiser V = 1  
Pour chaque bit de l'exposant B, en commençant par les poids forts :  
    - éléver V au carré  
    - si ce bit vaut 1, multiplier V par A
```

Exemple :

$$35 = 100011$$

soit :

étape 0 : $V = 1, A = 6$

étape 1 : $V = (1 * 1) * 6 = 6$ (bit à 1)

étape 2 : $V = 6 * 6 = 36$ (bit à 0)

étape 3 : $V = 36 * 36 = 1296$ (bit à 0)

étape 4 : $V = 1296 * 1296 = 1679616$ (bit à zéro)

étape 5 : $V = 1679616 * 1679616 * 6$
 $= 2821109907456 * 6 = 16926659444736$ (bit à 1)

étape 6 : $V = 16926659444736 2 * 6 = 1719070799748422591028658176$

Complexité ?



Et dans le cas du parallélisme ?



Et l'exploitation du parallélisme ?



Accélération ou speedup

Accélération = gain de temps obtenu lors de la parallélisation du programme séquentiel.

Définition :

- ▷ Soit T_1 le temps nécessaire à un programme pour résoudre le problème A sur un ordinateur séquentiel ;
- ▷ Soit T_p le temps nécessaire à un programme pour résoudre le même problème A sur un ordinateur parallèle contenant p processeurs ;
- ▷ Alors l'accélération «Speed-Up» est le rapport : $S(p) = T_1/T_p$

Cette définition n'est pas très précise

Pour obtenir des résultats comparables il faut utiliser les mêmes définitions d'**Ordinateur Séquentiel** et de **Programme Séquentiel**.

Ordinateur Séquentiel :

- Ordinateur // configuré avec un seul processeur ;
- Ordinateur séquentiel d'une puissance similaire à l'ordinateur //.

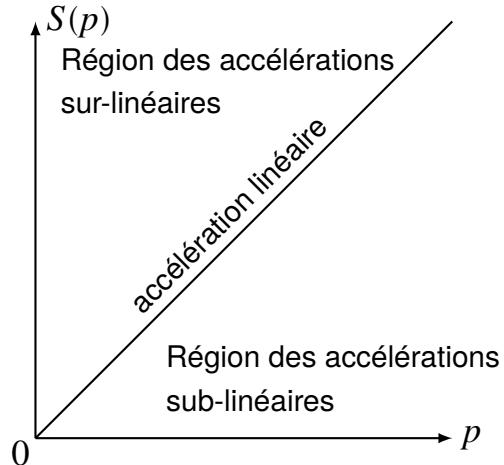
Programme Séquentiel :

- Programme // configuré pour s'exécuter sur un seul processeur ;
- Programme séquentiel utilisant le même algo que le programme // ;
- Programme séquentiel le plus rapide connu utilisant le même algo que le programme // ;
- Programme séquentiel (ou le plus rapide) résolvant le même pb.

Beaucoup de combinaisons, il faut préciser dans chaque cas.



Accélération



Efficacité

- Soit $T_1(n)$ le temps nécessaire à l'algorithme pour résoudre une instance de problème de taille n avec un seul processeur,
- Soit $T_p(n)$ celui que la résolution prend avec p processeurs
- Soit $S(n, p) = T_1(n)/T_p(n)$ le facteur d'accélération.

On appelle **efficacité** de l'algorithme le nombre

$$E(n, p) = S(n, p)/p$$

Efficacité = normalisation du facteur d'accélération



Multiplication de matrices : algorithme A moins bon que algorithme B

Algorithme A

- Temps en séquentiel : 10 minutes
- Nombre de processeurs : 10
- Temps en // : 2 minutes
- Accélération** : $10/2 = 5$ (l'application va 5 fois plus vite)
- Efficacité** : $5/10 = 1/2 = 0,5$

Algorithme B

- Temps en séquentiel : 10 minutes
- Nombre de processeurs : 3
- Temps en // : 4 minutes
- Accélération** : $10/4 = 5/2 = 2,5 < 5$
- Efficacité** : $(5/2)/3 = 0,8 > 0,5$



Le temps d'exécution T_1 d'un programme séquentiel peut être décomposé en deux temps :

- T_s consacré à l'exécution de la partie intrinsèquement séquentielle
- $T_{//}$ consacré à l'exécution de la partie parallélisable

$$T_1 = T_s + T_{//}$$

Seul $T_{//}$ peut être diminué par la parallélisation.

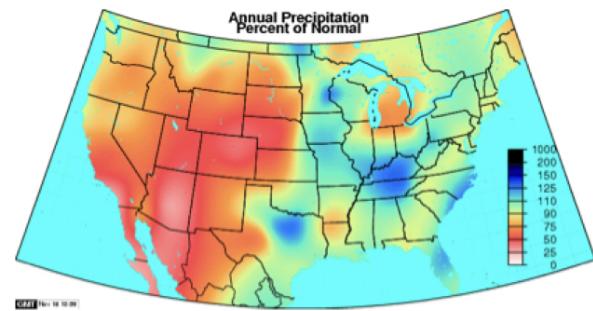
Dans le **cas idéal**, on obtiendra **au mieux** un temps $T_{//}/p$ pour la partie parallélisée.

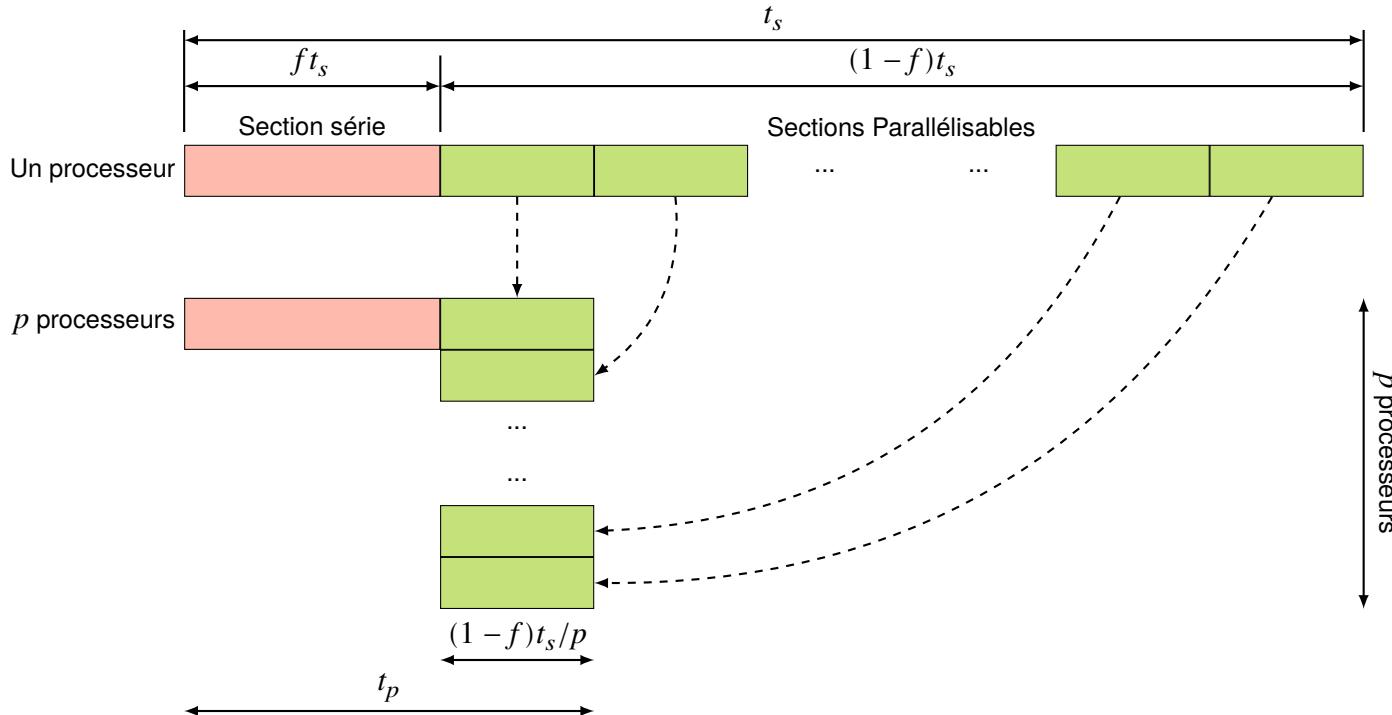
$$T_p \geq T_s + T_{//}/p$$

⇒ L'accélération d'un programme est limitée par le pourcentage de code intrinsèquement séquentiel qu'il contient.

Exemple : filtre graphique parallèle

- partie intrinsèquement séquentielle
 - ◊ capture
 - ◊ chargement sur le serveur
- partie parallélisable
 - ◊ découpage
 - ◊ calculs pour le traitement de l'image ...





La fraction f exprime le rapport entre la partie séquentielle et parallèle par rapport au temps complet t_s :

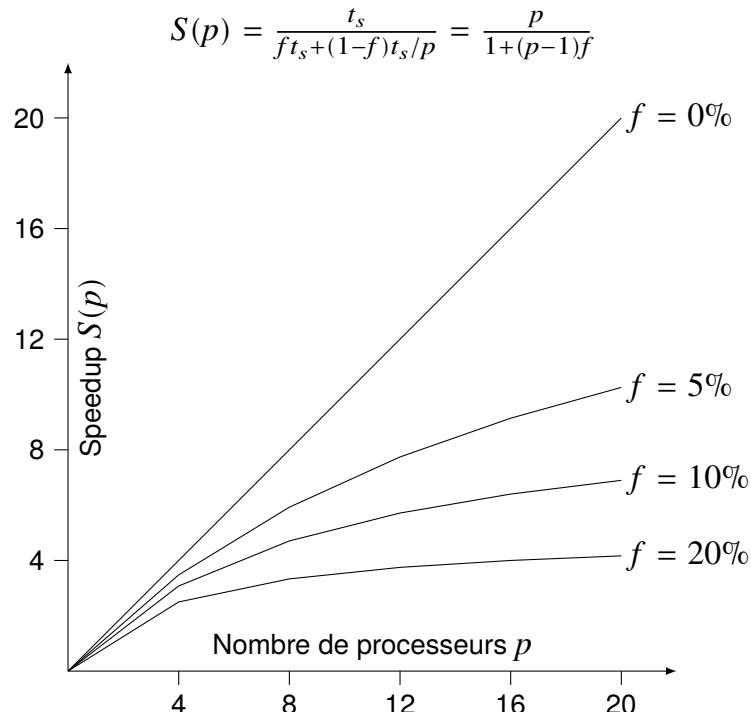
- ▷ ft_s pour le temps de la partie séquentielle ;
- ▷ $(1-f)t_s$ pour le temps de la partie parallèle.



Même avec un nombre infini de processeurs l'accélération maximale est de $1/f$

Exemple : avec seulement 5% de calcul séquentiel, le speedup maximal est de 20.

Calcul du speedup avec f :



f exprime le pourcentage de la partie séquentielle.



Une **accélération linéaire** correspond à un gain de temps égal au nombre de processeurs (100%activité)

Une **accélération sub-linéaire** implique un taux d'activité des processeurs < 100 %(communication, coût du parallélisme...)

Une **accélération sur-linéaire** implique un taux d'utilisation des processeurs > à 100 %ce qui paraît impossible (en accord avec la loi d 'Amdhal).

Cela se produit parfois (architecture, mémoire cache mieux adaptée que les machines mono-processeurs, utilisation de pipeline...)



Comment Paralléliser un algorithme ?



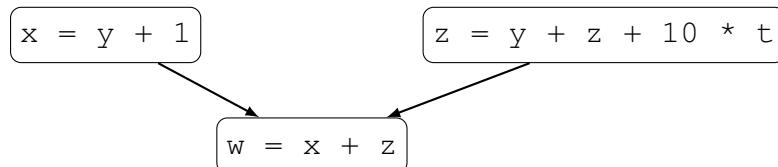
```

1 x := y + 1
2 z := y + z + 10 * t
3 w := x + z
    
```

L'obtention du résultat à partir de x , y et z nécessite **plusieurs étapes** dont certaines peuvent **s'exécuter en parallèle**.

Graphe de précédence

- les nœuds sont les **opérations à réaliser** pour résoudre le problème ;
- les **arcs orientés** sont les **contraintes de précédence** entre les opérations.



⇒ On obtient un **ordre partiel** et les **opérations non ordonnées** par cet ordre partiel sont :

- ▷ **indépendantes**
- ▷ capables de s'exécuter **en parallèle**.

Sur l'exemple :

- ▷ **1** et **2** peuvent s'exécuter **en parallèle** ;
- ▷ *par contre* **3** doit attendre la fin de **1** et **2** avant de débuter.

Le **graphe de précédence** donne une **analyse statique du parallélisme fonctionnel** exploitable :

- la **longueur de son plus long chemin** donne le nombre d'opérations **nécessairement séquentielles** ;
- la **largeur du graphe** donne le **nombre maximum d'opérations exécutables en parallèle** (le degré moyen des nœuds du graphe est également important).



Deux types de source de parallélisme

- le **parallélisme fonctionnel** ou **parallélisme de contrôle** ;
- le **parallélisme de données**.

Parallélisme fonctionnel

Il correspond à :

- ▷ découper un problème en **tâches** (opérations élémentaires) ;
- ▷ indiquer **l'ordonnancement** de ces tâches (graphe de précédence).

Exemple : produit itératif de n éléments

```
P := a(0) ;
Pour i in 1 .. n - 1 faire
    P := P * a(i) ;
```

P est le produit du premier élément avec le produit des $n-1$ éléments.

Analyse :

- le temps d'exécution est linéaire en n , soit $O(n)$;
- son **graphe de précédence** est une **chaîne** : il ne peut être parallélisable.

Pourtant il est **facile** de concevoir un **algorithme parallèle** à l'aide de n processus en $O(\log_2(n))$.

Mais ici, on exploite une **propriété de la multiplication** : son **associativité**.



Syntaxe normale des algorithmes parallèles avec deux instructions pour décrire les opérations parallèles :

- ▷ Si plusieurs étapes sont faites en parallèle, on écrit :

```
faire étapes i à j en parallèle
    étape i ;
    ...
    étape j ;
fin faire
```

ou avec le « co » pour concurrent :

```
co étape 1 ;           co [i = 0 to n-1] {
//   ...                   a[i] = 0 ; b[i] = 0;
//   étape j ;           }
oc
```

- ▷ Si plusieurs processeurs doivent exécuter le même type d'opérations en parallèle, on écrit :

```
Pour i := 1 à n faire en parallèle
S = { r, s, t, ... } /* S est l'ensemble des données */
```

```
opérations réalisées par Pi
fin pour
```



Exploitation (automatique) du Parallélisme

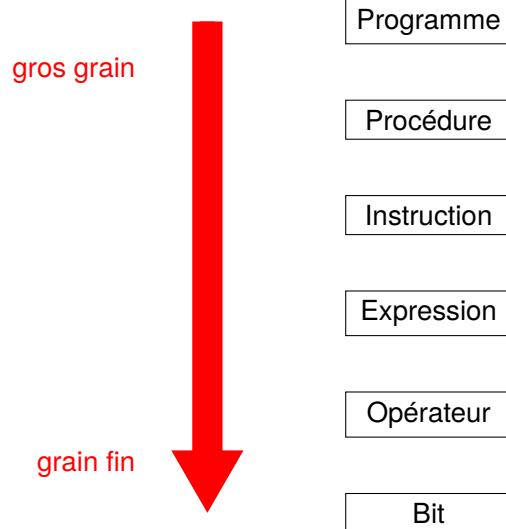


Définition

Le «*grain*» correspond à la taille moyenne des tâches élémentaires.

Le choix du grain lié à l'architecture de la machine (quel type de grain peut être exploité efficacement?).

Si le grain = programme alors on exploite du parallélisme de type système réparti.



Le «*degré*» de parallélisme reflète le **nombre de processeurs** pouvant être utilisés (degré 1 pour les parties purement séquentielles).

Il peut varier dans les différentes parties des programmes : *degré maximal, minimal, moyen d'un programme*.

Le degré maximal constitue une **borne supérieure** au nombre de processeur qu'on utilisera : avec ce nombre exécution rapide mais efficacité médiocre



en effet si le degré max > degré moyen alors des processeurs seront inactifs pendant une partie de l'exécution du programme



Parallélisme de données

- ▷ les données sont souvent plus nombreuses que les processeurs.
- ▷ la régularité de structures de données permet de distribuer de façon régulière ces données sur les différents processeurs : par exemple pour une matrice $n * n$ sur p processeurs :
 - ◊ chaque processeur possède n/p lignes ;
 - ◊ le processeur est dit **propriétaire** de ces lignes ;
 - ◊ il est **responsable** de la réalisation de toutes les tâches les concernant.

Les **schémas de distribution** les plus classiques sont :

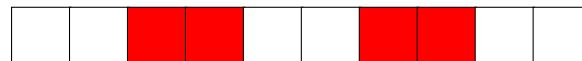
- répartition par **bloc** :



- répartition **cyclique** :



- répartition par **blocs cycliques** :



Exécution synchrone

Le parallélisme de données s'adapte facilement sur un modèle SIMD avec une grande importance donnée à la **synchronisation**.

Exemple :

```
pour i = 0 à n - 1 faire en parallèle  
    a[i] = 2.i  
pour i = 0 à n - 1 faire en parallèle  
    b[i] = a [i] + a[n - 1 - i]
```

Répartition par bloc :

- ▷ $b[0]$ est réalisé sur P_0 et nécessite : $a[0]$ et $a[n - 1]$
- ▷ **a[0]** calculé sur P_0 mais $a[n-1]$ sur P_{n-1}
- ▷ **attention** : pour $b[0]$ certaines valeurs de a doivent être disponibles
- ▷ la **synchronisation** est assurée par l'utilisation du **modèle SIMD**.

Utilisation de machine du modèle MIMD

Il est possible d'exploiter le **parallélisme de données** sur des machines MIMD :

- ▷ il faut veiller à ce que les processeurs travaillent de **manière coordonnée**

Dans l'exemple, il suffit d'attendre que tous les processeurs aient fini de calculer les valeurs du vecteur a avant que l'un d'entre eux ne commence le calcul de b .

On **synchronise** donc l'ensemble des processeurs **entre les deux boucles**.

Le parallélisme de données sur des machines MIMD implique un travail de synchronisation de la part du programmeur.



Dépendance des données

Reprenons l'exemple précédent dans une forme condensée :

```
a[0] = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire
    a[i] = 2.i
    sp[i] = a[i] + sp [i - 1]
```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Notions de variables lus et modifiées pour une instruction S

- ▷ $L(S)$ ensemble des variables utilisées en **lecture** ;
- ▷ $M(S)$ ensemble des variables qui subissent une **modification** (une affectation).

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie, ou RAW, « S_2 Read-After- S_1 Write»
- $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance, ou WAR, « S_2 Write-After- S_1 Read»
- $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie, ou WAW, « S_2 Write-After- S_1 Write»

Ces 3 équations doivent être **vérifiées** pour pouvoir réaliser S_1 et S_2 en **parallèle**.

Il est possible de définir :

- ▷ S comme une **séquence d'instructions** au lieu d'une seule instruction ;
- ▷ les ensembles L et M en les calculant sur les instructions de chaque séquence.

La **condition de Bernstein** exprime la possibilité de calculer en **parallèle** les deux séquences d'instructions S_1 et S_2 .



Dépendance des données

Exemple:

```

    a[0] = 0
    sp[0] = a[0]
    pour i = 1 à n - 1 faire
        a[i] = 2.i
        sp[i] = a[i] + sp [i - 1]

```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie
- $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance
- $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie

Ces 3 équations doivent être vérifiées pour pouvoir réaliser S_1 et S_2 en parallèle.

Exemple

On numérote S_1 , et S_2 les instructions de la boucle :

```

 — xterm —
    a[0] = 0
    sp[0] = a[0]
    pour i = 1 à n - 1 faire
        S1: a[i] = 2.i
        S2: sp[i] = a[i] + sp [i - 1]

```

- $\triangleright M(S_1) = \{a[i]\}$
 - $\triangleright L(S_2) = \{a[i]\}$
- $\left. \right\} \text{alors } M(S_1) \cap L(S_2) \neq \emptyset, \text{on a une «dépendance vraie» ou RAW.}$



Dépendance des données

Exemple:

```

    a[0] = 0
    sp[0] = a[0]
    pour i = 1 à n - 1 faire
        a[i] = 2.i
        sp[i] = a[i] + sp [i - 1]
  
```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie
- $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance
- $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie

Ces 3 équations doivent être vérifiées pour pouvoir réaliser S_1 et S_2 en parallèle.

Exemple

On note S_1 les instructions de la boucle et S_2 les instructions de l'occurrence suivante :

	xterm
<pre> a[0] = 0 sp[0] = a[0] pour i = 1 à n - 1 faire Soit j une occurrence de la boucle S₁: a[j] = 2.j sp[j] = a[j] + sp [j - 1] S₂: a[j+1] = 2.(j+1) sp[j+1] = a[j+1] + sp [j] </pre>	

«Dépendance vraie» entre les instances de l'instruction S_1 (pour j) et S_2 (pour $j + 1$) dans la boucle $M(S_1) \cap L(S_2) = \{sp[j]\} \neq \emptyset \Rightarrow$ Condition de Bernstein **non vérifiée**



Modifications pour tirer parti du parallélisme

— xterm —

```
a[0] = 0
sp[0] = a[0]
pour i = 0 à n - 1 faire
    a[i] = 2.i
    sp[i] = a[i] + sp [i - 1]
```

Peut être réécrit en :

— xterm —

```
a[0] = 0
sp[0] = a[0]
pour i = 0 à n - 1 faire
    a[i] = 2.i
pour i = 0 à n - 1 faire
    sp[i] = a[i] + sp [i - 1]
```

Et enfin :

— xterm —

```
a[0] = 0
sp[0] = a[0]
pour i = 1 à n-1 faire_en_parallel
    a[i] = 2.i

pour i = 1 à n - 1 faire
    sp[i] = a[i] + sp [i - 1]
```

Au final :

- ▷ Le première boucle peut être **exécutée en parallèle** !
- ▷ La seconde boucle établie une **dépendance temporelle** entre une occurrence de la boucle et sa suivante :
⇒ **pas d'exécution parallèle possible** !

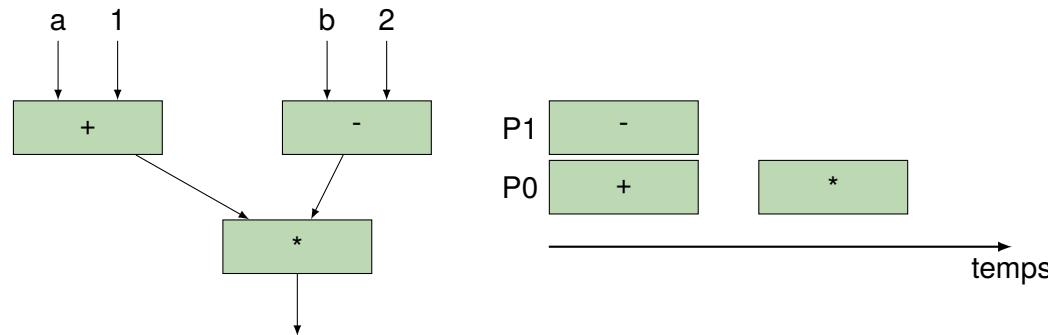


Exploitation du parallélisme

- ▷ 1ère phase : découpage en tâches ;
- ▷ 2ème phase : étude des tâches pour déterminer celles qui peuvent s'exécuter en parallèle et celles qui doivent s'exécuter séquentiellement ;

Exemple :

graphe de $(a + 1) * (b - 2)$



Choix du grain

La taille des tâches, le «*grain*» obtenu lors du découpage est important pour l'accélération.

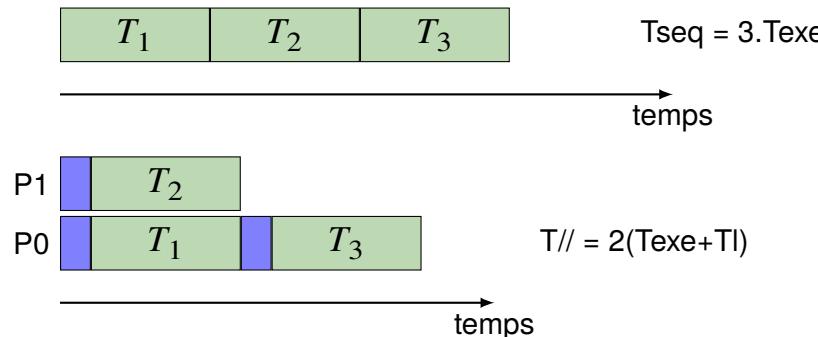
Le grain doit être adapté à la machine sous-jacente :

- ◊ si le **lancement d'une tâche** prend un temps important, on ralenti l'exécution en définissant un grand nombre de tâches
- ◊ les **temps d'exécution des tâches** tournant en parallèle doivent être **voisins**.



Soient 3 tâches T_1 , T_2 et T_3 avec T_3 dépend de T_1 et T_2

- Texe : temps d'exécution de la tâche
- TI : temps de lancement de la tâche



Si $\text{TI} > \text{Texe} / 2$ alors $T_{//} > T_{\text{seq}}$!

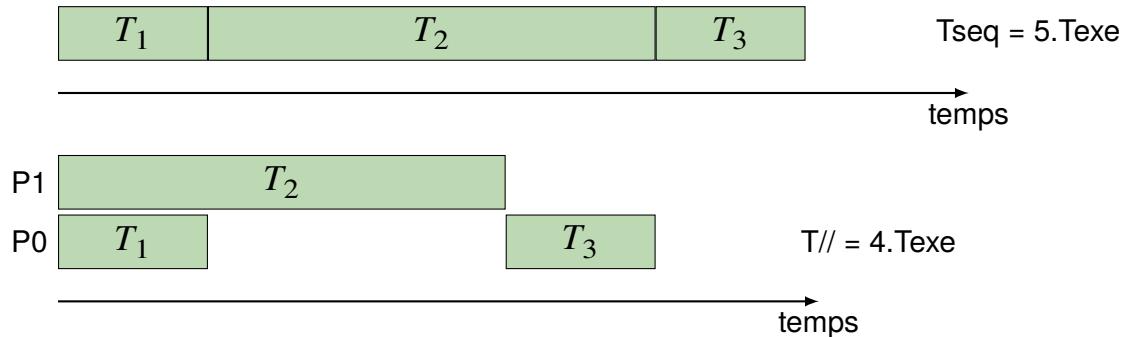
Que faire ?

- ▷ Soit on diminue TI en changeant de machine //
- ▷ Soit on augmente le **grain** des tâches en découplant le programme autrement : si TI est **négligeable** devant Texe, **l'accélération** et **l'efficacité** sont **meilleures**.



Supposons :

- T1 est négligeable devant Texe
- T2 prend 3.Texe



▷ Accélération = 5/4

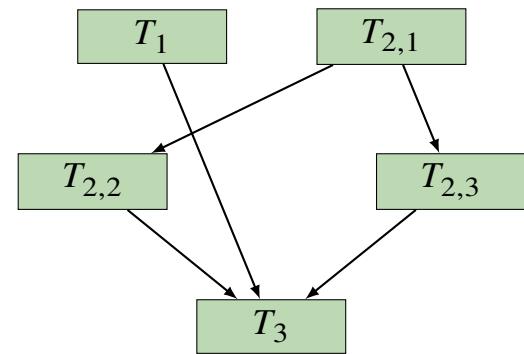
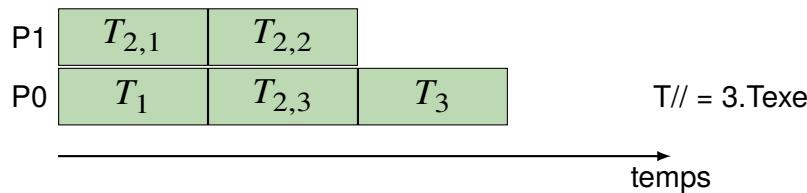
▷ Efficacité = 5/8

Ce qui n'est pas très bon !



Solution 1 :

- découper T2 en 3 sous-tâches // (T2,1 ; T2,2 ; T2,3) de durée Texe
- T2,1 précède les deux autres qui peuvent être exécutées **en parallèle**.



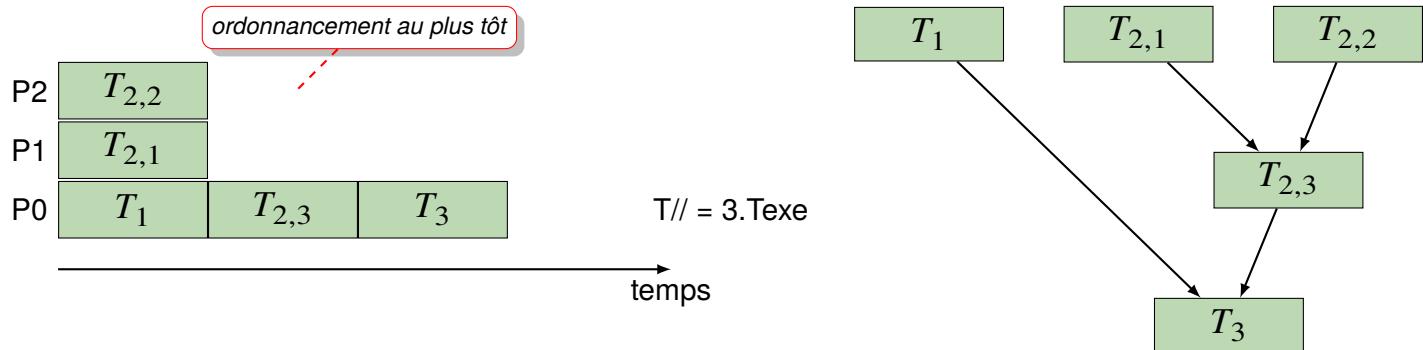
- ▷ Accélération = 5/3
- ▷ Efficacité = 5/6

Ce qui est meilleur !



Solution 2 :

- découper T_2 en 3 sous-tâches // ($T_{2,1}$; $T_{2,2}$; $T_{2,3}$) de durée T_{exe}
- $T_{2,1}$ et $T_{2,2}$ indépendantes et précédant $T_{2,3}$.



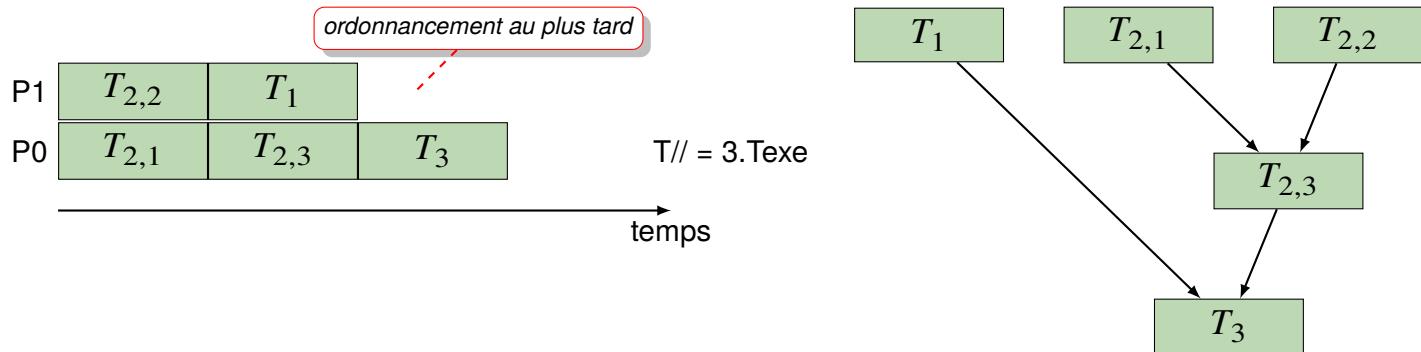
- ▷ Accélération = $5/3$
- ▷ Efficacité = $5/9$

On notera que l'arbre de précédence exprimant les dépendances temporelles entre les différentes tâches est différent de celui du transparent précédent : un nouveau découpage de T_2 a été trouvé.



Solution 2 :

- découper T2 en 3 sous-tâches // (T2,1 ; T2,2 ; T2,3) de durée Texe
- T2,1 et T2,2 indépendantes et précédant T2,3.



- ▷ Accélération = 5/3
- ▷ Efficacité = 5/6



Le coût des communications

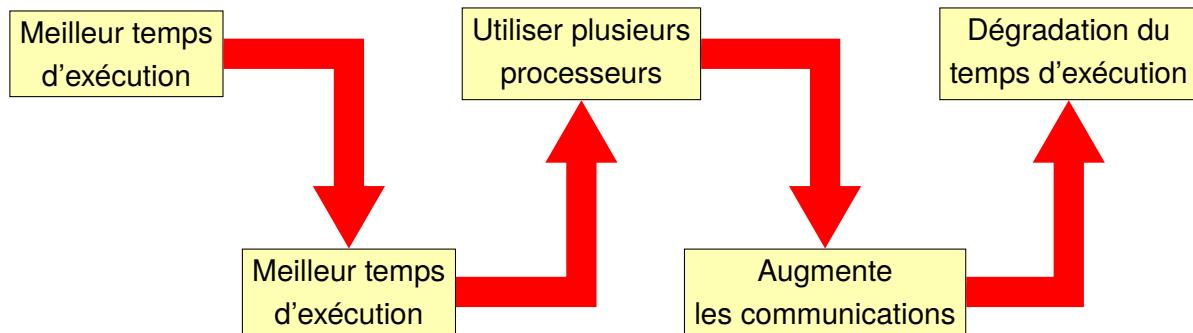
Sur une machine à mémoire distribuée, le coût des communications est important.

Si deux tâches dépendantes sont sur des processeurs distincts, le résultat de la 1ère doit être émis vers la 2ème.

Le meilleur temps d'exécution correspond à la répartition des tâches de manière à :

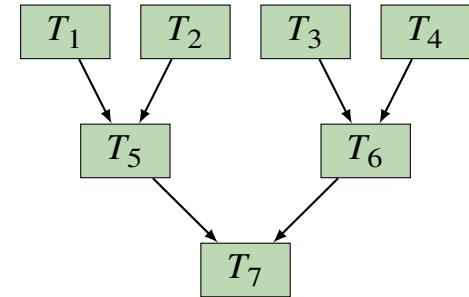
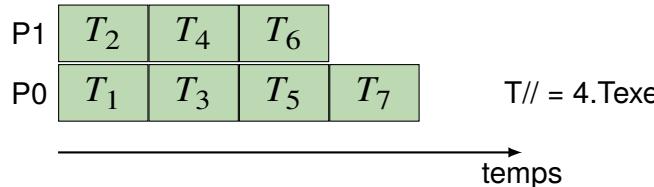
- ▷ minimiser les temps de calcul
- ▷ minimiser les temps d'attente
- ▷ minimiser les communications

Importance du découpage et du placement des tâche



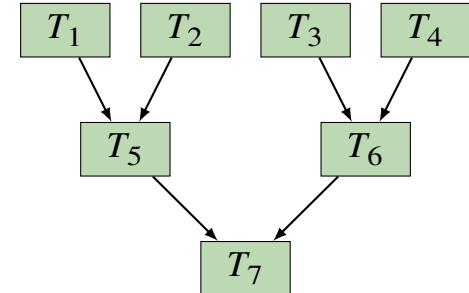
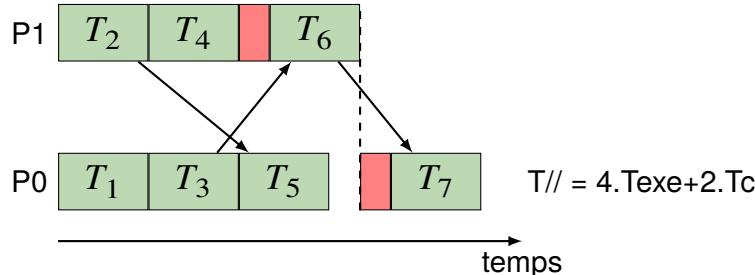
Placement des tâches

Exemple : 7 tâches de même durée Texe, sur le schéma : placement optimal au niveau du temps de calcul

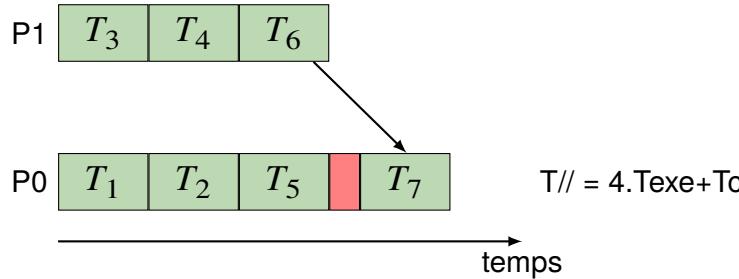


Temps de communication : $T_c < T_{exe}$

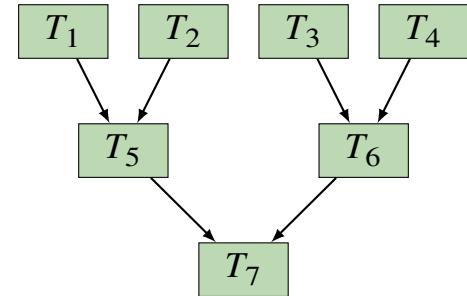
Communication par des circuits spécialisés



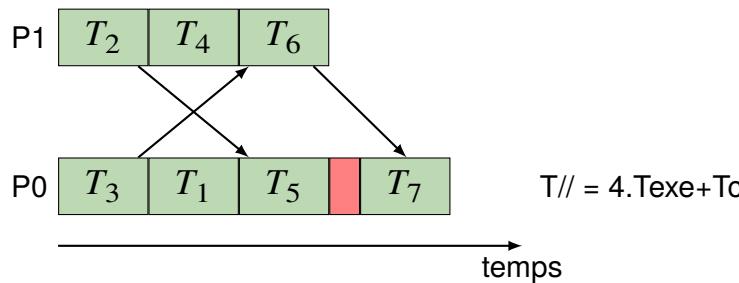
Optimisation : Déplacement des tâches entre les processeurs pour diminuer les communications



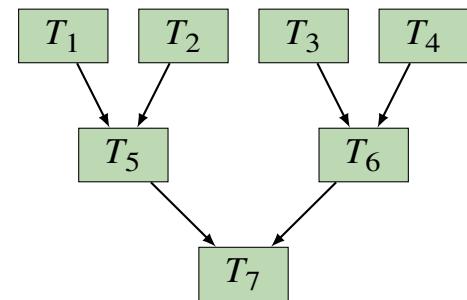
$$T// = 4.Texe + Tc$$



Réordonnancement des tâches sur un même processeur pour les faire communiquer **pendant** qu'ils calculent



$$T// = 4.Texe + Tc$$



Bilan sur le processus de parallélisation

- ▷ Le choix des tâches définit le **degré** de parallélisme :

plus le nombre de tâches ↗, plus le grain ↘

- ▷ La **taille d'une tâche** doit être **suffisamment grande** pour que **TI** soit négligeable devant **Texe** ;
- ▷ La **taille des tâches** qui s'exécutent simultanément doit être «voisines» ou «identiques» ;
- ▷ L'ordonnancement des différentes tâches doit être fait de manière choisie : probablement de manière **empirique** par évaluation sur différentes exécutions ;
- ▷ Il faut trouver un **placement efficace** des différentes tâches sur les nœuds de la machine ;
- ▷ Sur une machine à **mémoire distribuée**, il faut également tenir compte des **communications** (encore plus sur un cluster de station de travail) :

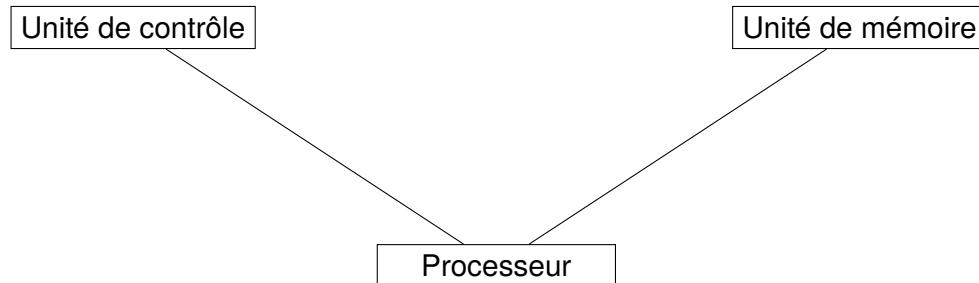
⇒ **recouvrement calcul/communication**



Et par rapport aux architectures parallèles ?



- un flot d'instructions ;
- un flot de données ;



- ▷ À chaque étape de calcul, l'unité de contrôle produit une instruction qui opère sur une donnée obtenue à partir de l'unité mémoire.
- ▷ Il n'y a qu'un seul processeur \Rightarrow **modèle séquentiel**.

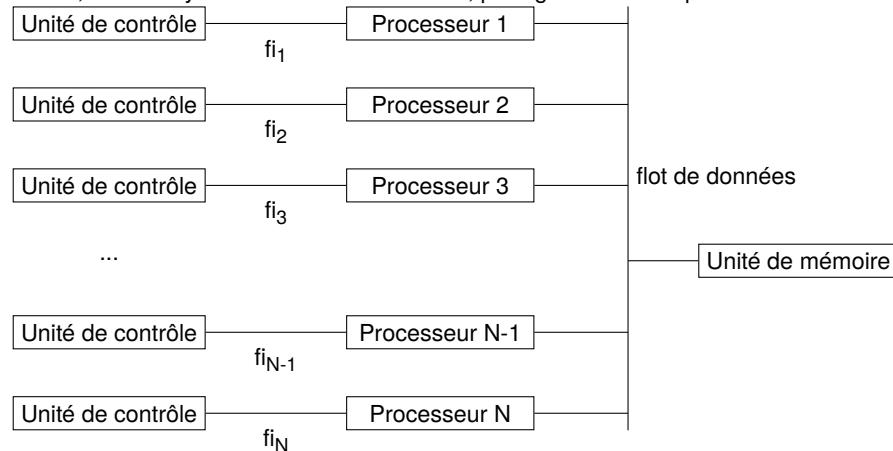


A chaque étape, une donnée est :

- fournie par l'unité mémoire ;
- traitée parallèlement par les N processeurs qui exécutent chacun une instruction spécifique.

On a plusieurs **flocs d'instructions**, f_i , opérant sur un seul **flot de données**.

On a N processeurs, $N > 1$, chacun ayant son **unité de contrôle**, partageant une unique **unité mémoire**.



Exemple

On veut savoir si un nombre z est premier.

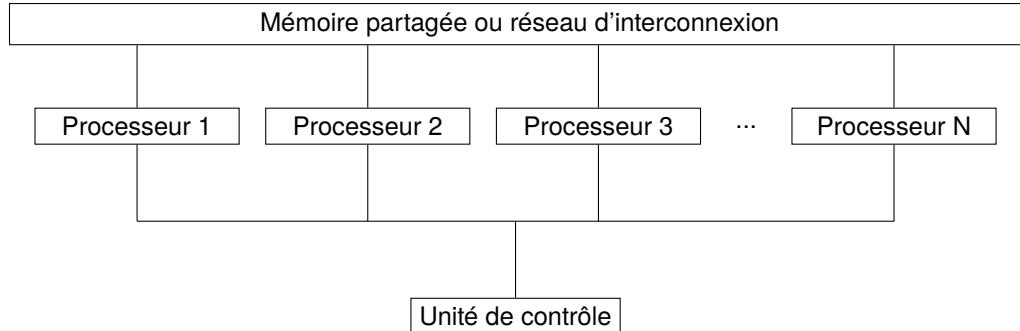
On peut associer à chaque processeur un diviseur potentiel de E (nombres entre z et $z-1$).

Un processeur :

- reçoit z ;
- fait la division par son le diviseur potentiel ;
- répond par « oui » ou « non » dans une zone mémoire fixe, connue par tous les processeurs.

Il n'existe pas de calculateur fonctionnant avec ce modèle !

On dispose de N processeurs, tous identiques ayant chacun une mémoire locale où on peut stocker des données et des instructions.



On suppose la mémoire locale intégrée au processeur.

Tous les processeurs opèrent sous le contrôle d'un unique flot d'instruction (ceci est équivalent à avoir dans chaque mémoire locale une copie d'un unique programme).

Les processeurs opèrent de manière **synchrone** (horloge globale) sur les N flots de données (à chaque pas de temps, ils exécutent tous la même instruction).

À certaines étapes, certains processeurs peuvent rester inactifs, en fait ils exécutent « une instruction d'attente » ayant la durée voulue (nécessité de conserver une totale synchronisation entre tous les processeurs).

Les processeurs communiquent (échangent des données) pour coordonner leurs résultats. Ceci se fait de deux manières différentes :

- **mémoire partagée** SM SIMD (shared memory) ;
- **réseau d'interconnexion** (interconnection network).



Le modèle est connu sous le nom de modèle **PRAM**, «Parallel Random Access Machine».

Les processeurs utilisent la mémoire la **mémoire partagée** de la manière suivante :

« *si le processeur i veut communiquer une donnée avec le processeur j, alors i écrit le nombre en mémoire partagée dans une zone connue par j, puis j va lire le nombre* »

Quand les N processeurs vont faire ces actions en même temps, il y aura des problèmes de concurrence d'accès.

On aura quatre sous classes du modèle SM SIMD :

- ▷ EREW, Exclusive Read Exclusive Write ;
 - ▷ CREW, Concurrent Read Exclusive Write ;
 - ▷ ERCW, Exclusive Read Concurrent Write ;
 - ▷ CRCW, Concurrent Read Concurrent Write.
- Puissance du modèle
↓

A priori les accès concurrents en **lecture** ne posent pas de problème.

Par contre, en **écriture**, il faut une règle pour trancher le problème :

- a. le processeur ayant le plus petit numéro parmi ceux voulant accéder en écriture sera prioritaire ;
- b. l'écriture est interdite sauf si tous les processeurs écrivent la même valeur ;
- c. on écrit la somme des valeurs que les processeurs veulent écrire.



On considère un fichier contenant n enregistrements distincts rangés dans un ordre quelconque.

On veut savoir si un enregistrement x donné est présent dans le fichier ou pas.

Dans un modèle SISD, (machine séquentielle), ceci prend au pire des cas n comparaisons et $n/2$ en moyenne.

On considère le modèle EREW SM SIMD avec N processeurs et $N \leq n$.

On note les processeurs P_1, \dots, P_N .

Pour commencer, tous les processeurs doivent connaître x .

Ceci se fait par une opération de « broadcasting » (one to all) à partir de P_1 .

- étape 1 : P_1 lit x et le communique à P_2 : $P_1 \xrightarrow{x} P_2$
- étape 2 :
 - ◊ $P_1 \xrightarrow{x} P_3$
 - ◊ $P_2 \xrightarrow{x} P_4$ en parallèle
- étape 3 :
 - ◊ $P_1 \xrightarrow{x} P_5$
 - ◊ $P_2 \xrightarrow{x} P_6$
 - ◊ $P_3 \xrightarrow{x} P_7$
 - ◊ $P_4 \xrightarrow{x} P_8$ en parallèle
- etc.
- étape t : les processeurs P_i , $1 \leq i \leq 2^t$ connaissent x

On suppose que le fichier est découpé en N sous-fichiers de taille n/N et que chaque processeur P_i recherche x dans le $i^{\text{ème}}$ sous fichier.

Les N recherches se font en parallèle donc il faut au pire des cas n/N étapes de comparaison.

La réponse finale est fournie (si elle est positive) par l'unique processeur ayant trouvé x dans son sous- fichier.



il faut donc au plus une étape pour fournir le résultat dans la mémoire partagée (dans une variable prédéfinie initialisée à Faux).

Soit au pire des cas, on a $\log_2(N) + n/N + 1$.

Amélioration ?

On regarde ce qui se passe avec une approche « moyenne », i.e. on arrête dès que x est trouvé.

On considère une variable booléenne partagée F ($F \in P_1$) et initialisée à faux.

Quand un processeur trouve x , il met F à vraie.

À chaque étape, tous les processeurs recevront F et s'arrêtent si F est vraie (point de synchronisation forte : coûteux sauf si on a un très bon réseau d'interconnexion).

En moyenne on a :

$$\frac{1}{2}[\log_2(N) + \frac{n}{N}(1 + \log_2(N)) + 1 + \log_2(N)] = \log_2(N) + \frac{n/N+1}{2}(1 + \log_2(N))$$

⇒ Ce qui est moins bon !

Pour mieux exploiter cet arrêt possible, on considère le modèle CREW SM SIMD :

⇒ Les **lectures simultanées** vont permettre de remplacer des termes $\log_2(N)$ par 1.

⇒ On aura alors $n/N + 2$ étapes comme dans le pire des cas.

Remarque : si x est présent plusieurs fois, il faut un modèle CRCW pour avoir le même résultat.



On veut pouvoir simuler des accès concurrents à l'aide du modèle EREW.

On a N processeurs P_1, \dots, P_N qui veulent simultanément lire le contenu d'une variable A , ou écrire dans A .

- ▷ **lecture**: $\log_2(N)$ étapes sont nécessaires
- ▷ **écriture**: on appelle a_i , $1 \leq i \leq N$, la valeur que veut écrire le processeur P_i .

On suppose que les N processeurs pourront écrire dans A que si tous les a_i sont égaux.

1. Pour tout i , $1 \leq i \leq N/2$ faire simultanément:

Si $a_i = a_{i+N/2}$ alors P_i met à vrai la variable b_i sinon P_i met à faux la variable b_i

2. Pour tout i , $1 \leq i \leq N/4$ faire simultanément:

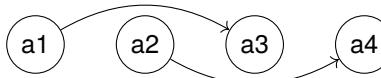
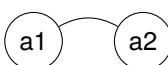
Si b_i et $b_{i+N/4}$ vraies et $a_i = a_{i+N/4}$ alors P_i met à vrai la variable b_i sinon P_i met à faux la variable b_i

On itère en divisant l'intervalle par 2 à chaque étape et après $\log_2(N)$ étapes P_1 sait si tous les a_i sont égaux.

⇒ Si oui P_1 écrit a_1 dans A et sinon il ne fait rien (A reste à faux).

Exemple

$N = 4$

- ▷ 1^{ère} étape : comparaisons 
- ▷ 2^{ème} étape : comparaison 
- ▷ 3^{ème} étape : écriture 



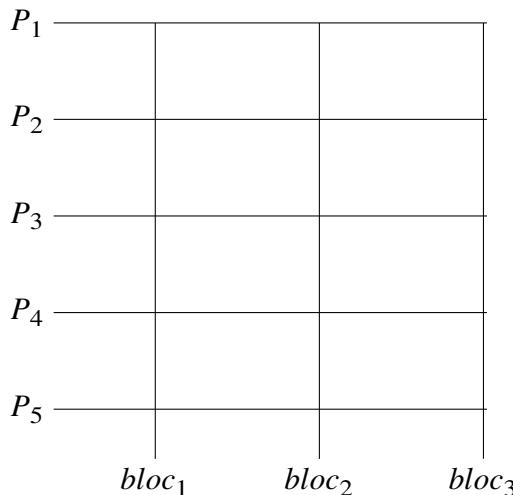
Ce modèle est très puissant mais il est peut réaliste à cause de la complexité des circuits d'accès à la mémoire.

Si la mémoire possède M zones élémentaires, le coût du circuit permettant un calcul d'adresse pour un processeur varie comme $f(M)$ ou f est croissante.

Dans le cas de N processeurs partageant la mémoire, le coût du circuit global varie comme $N * f(M)$.

⇒ Ceci est **irréaliste** pour N et M grands.

Une solution pour diminuer ce coût est de diviser la mémoire en R blocs de taille M/R chacun.



La **concurrence** se fait au niveau du bloc.

Un processeur quelconque peut accéder à un bloc quelconque.

Il y a $N * R$ «switches» : un à chaque intersection bus mémoire/bus processeur.

L'accès se fait par une logique « ligne-colonne » à travers les switches.

Chaque bloc de mémoire dispose d'un circuit d'adressage interne (M/R possibilités)

Le coût est de : $R * f(M/R) + (N * R) * \text{« coût d'un switch »}$

Le coût d'un switch est négligeable.

⇒ Le modèle est **moins puissant** à cause de l'accès au niveau des blocs.



On veut étendre l'idée précédente :

Les M éléments de mémoire sont distribués sur les N processeurs, chacun ayant localement M/N mémoire.

Chaque **paire de processeurs** est connectée par un lien bidirectionnel et à chaque étape un processeur quelconque P_i peut :

- ▷ recevoir une donnée d'un processeur P_j
- ▷ envoyer un autre message à P_k (k peut être égal à j).

Chaque processeur doit avoir un circuit d'adressage de coût $f(N - 1)$ lui permettant de sélectionner n'importe lequel des autres processeurs.

Il doit avoir un circuit d'adressage de coût $f(M/N)$ pour permettre un accès à sa propre mémoire.

On aura en tout :

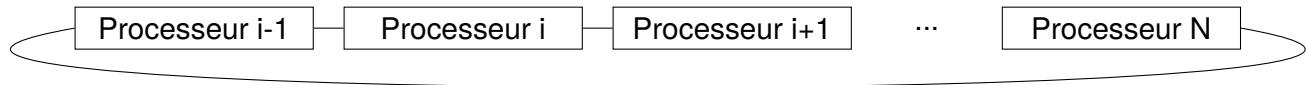
$$N * (f(N - 1) + f(M/N))$$

Le coût est **trop important** mais le modèle est **plus puissant** que celui avec découpage en blocs de la mémoire.

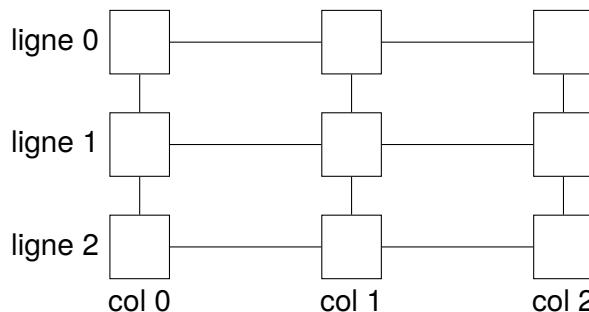
⇒ On a besoin de réseaux « **faiblement couplés** » par opposition au **graphe complet**.



La chaîne «1D»



La grille «2D»



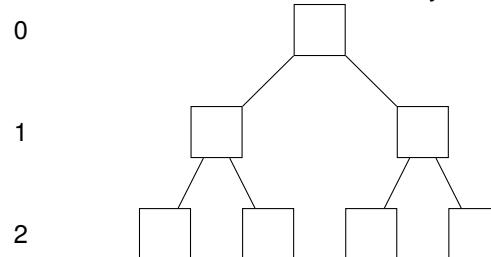
Processeur $P_{i,j}$:

- ▷ ligne i ;
- ▷ colonne j ;



L'arbre binaire

On a un arbre binaire complet avec d niveaux numérotés de 0 à $d - 1$ et il y a $N = 2^d - 1$ processeurs.



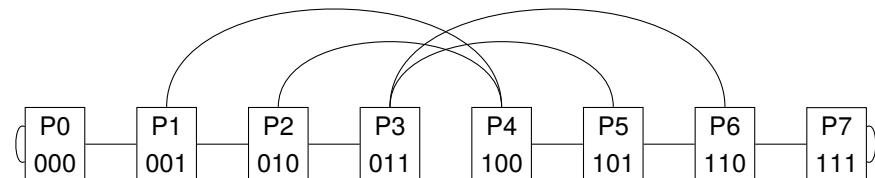
Le «Perfect Shuffle» ou mélange parfait

On a N processeurs numérotés P_0, \dots, P_{N-1} , avec $N = 2^P$

P_i est connecté à P_j si et seulement si :

- ▷ $j = 2 * i$ pour $0 \leq i \leq N/2 - 1$;
- ▷ $j = 2 * i + 1 - N$ pour $N/2 \leq i \leq N - 1$;

⇒ L'écriture binaire de j se déduit de celle de i par un décalage circulaire à gauche : liens « shuffle ».



On rajoute des liens d'échanges entre tout processeur ayant un numéro pair et son suivant.

On a le « shuffle exchange ».

Ce type de réseau est bien adapté pour le calcul de la **transformée de Fourier rapide** ou FFT en parallèle.



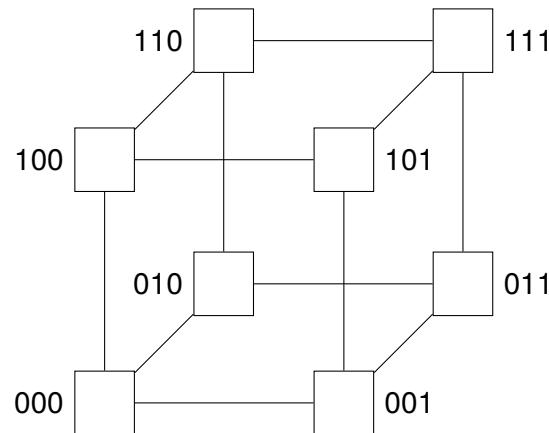
Le cube

On considère $N = 2^q$ processeurs numérotés P_0, P_1, \dots, P_{N-1} ($q \geq 1$)

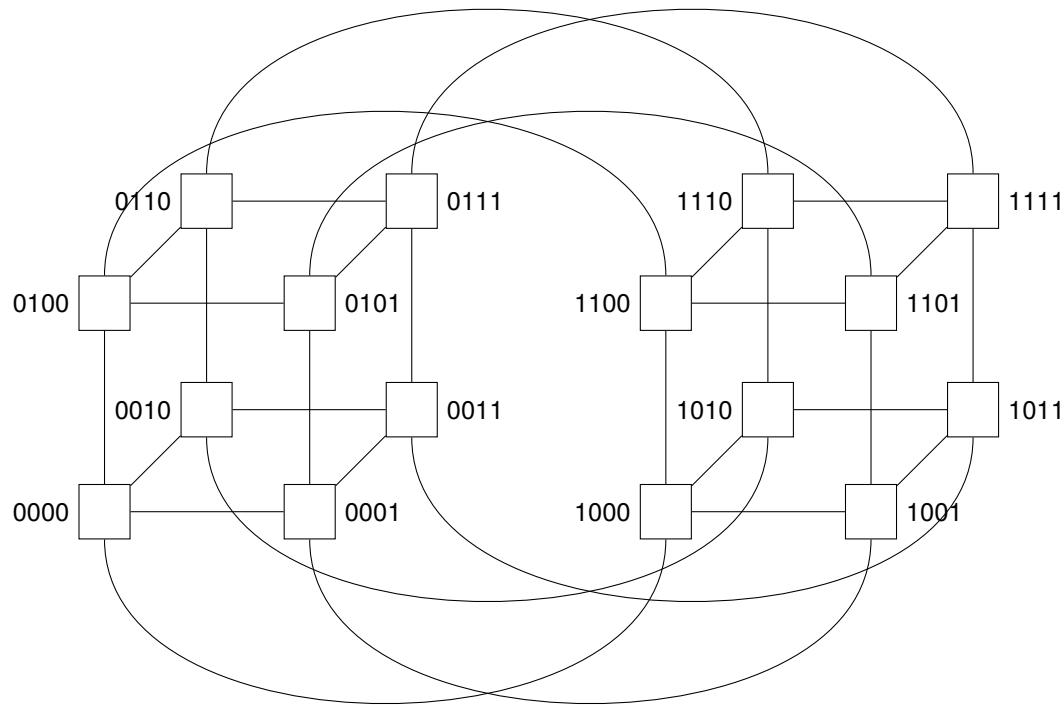
Un cube de dimension q (hypercube) est obtenu en connectant un sommet à exactement q voisins.

Ce qui donne un degré logarithmique.

P_i est connecté à P_j si et seulement si les écritures binaires de i et de j ne diffèrent que d'un chiffre binaire.



Le cube de degré 4



Exemple sur l'arbre binaire

On veut faire la somme de n nombres x_1, x_2, \dots, x_n .

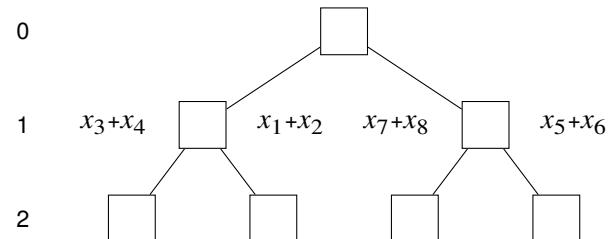
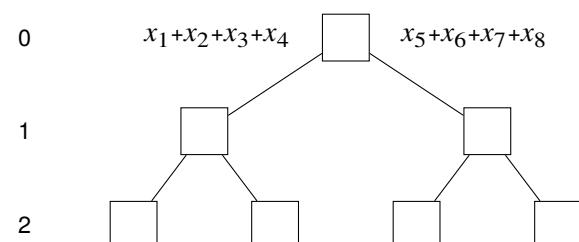
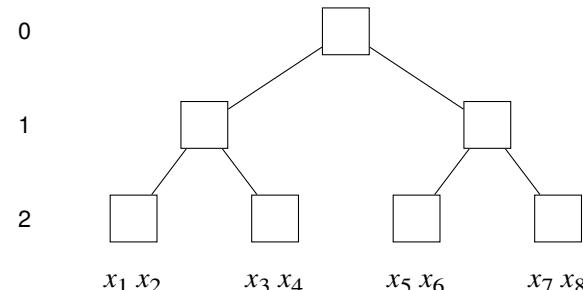
Dans un modèle SISD, il faut n étapes ($n-1$ additions et 1 affectation).

En utilisant un arbre ayant $\log_2(n)$ niveaux et $n/2$ feuilles, le calcul peut se faire en $\log_2(n)$ étapes.

Chaque feuille a au départ 2 nombres, elle fait la somme et l'envoie au père.

Pour les processeurs des autres niveaux, on fait la chose suivante :

- ▷ recevoir une donnée de chacun des deux fils ;
- ▷ faire la somme ;
- ▷ envoyer le résultat au père.



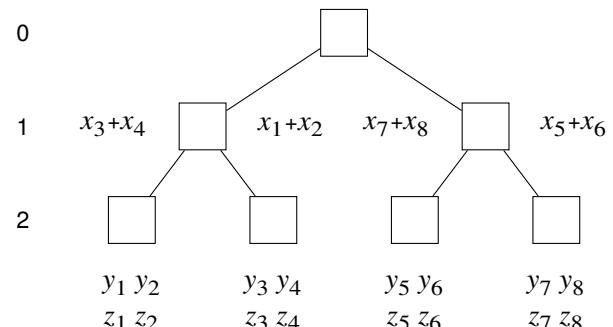
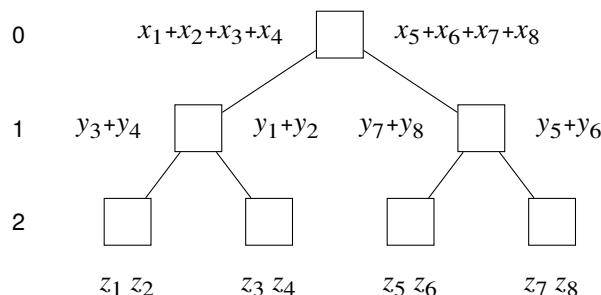
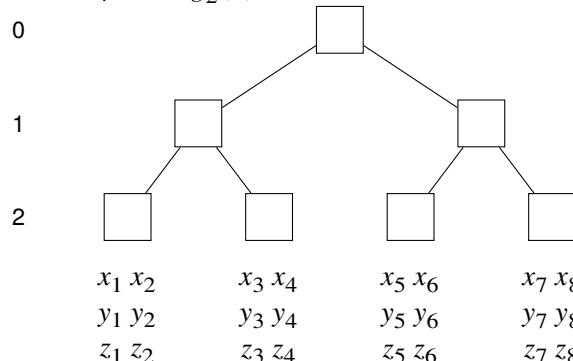
La racine n'a plus qu'à faire la **somme finale** :

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$

Effet pipeline

Si on veut faire m sommes de n nombres, on « pipeline » les différents calculs.

Le calcul prend $\log_2(n) + m - 1$.



La racine fait la première **somme** :

$$x_1+x_2+x_3+x_4 + x_5+x_6+x_7+x_8$$

Puis à l'étape suivante, la seconde somme :

$$y_1+y_2+y_3+y_4 + y_5+y_6+y_7+y_8$$

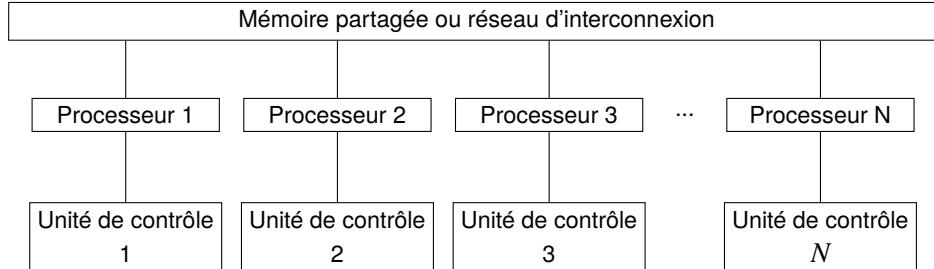
Puis à l'étape suivante, la troisième somme :

$$z_1+z_2+z_3+z_4 + z_5+z_6+z_7+z_8 \text{ etc.}$$



C'est le modèle **le plus général**.

Il y a N processeurs avec N flots de données différents.



Chaque processeur dispose d'une **mémoire locale**.

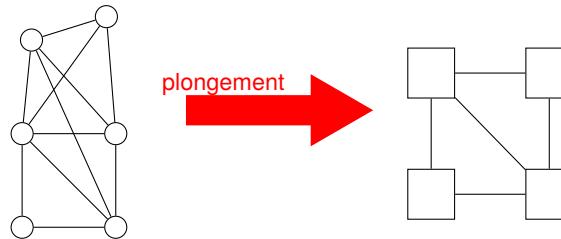
Pour le modèle avec mémoire partagée, on a la même classification qu'en SIMD au niveau de la concurrence des accès.

On a aussi tous les réseaux vus en SIMD : anneau, grille, hypercube, ...

Les algorithmes s'expriment en terme de processus communicants par **SM**, «*Shared Memory*», ou par **transmission de messages**.

On a généralement n processus sur N processeurs avec $N \leq n$

- ▷ **placement des processus** sur les processeurs (problème NP-complet) ;
- ▷ placement **statique** ou **dynamique** ;
- ▷ **équilibrage de la charge** des processus (dynamique ou statique) ;
- ▷ **minimisation** de la **communication** et du **routage**.



Et l'évaluation de nos algorithmes parallèles
sur ces modèles ?



On veut des **critères** pour évaluer la **qualité** des algorithmes parallèles.

Les **métriques** classiques sont :

- le temps d'exécution ;
- le nombre de processeurs utilisés ;
- le coût.

Le temps d'exécution :

- le temps entre le début et la fin du calcul ;
- temps écoulé entre le moment où le premier processeur démarre et le moment où le dernier arrête.

Les unités sont :

- celles de calcul : calcul arithmétique ou logique ;
- celles de routage : on doit faire intervenir la distance entre les processeurs dans le schéma de communication dans le réseau.

On notera $t(n)$ le temps d'exécution parallèle dans le pire des cas pour un problème de taille n .

On calcule des bornes inférieure et supérieure : notations (Ω, σ) .

On évalue la qualité de l'algorithme par **l'accélération** : «*speedup*».

$$\text{speedup} = \frac{\text{temps d'exécution dans le pire des cas du meilleur algorithme séquentiel}}{\text{temps d'exécution dans le pire des cas de l'algorithme parallèle} \Rightarrow t(n)}$$

Si on a N processeurs valant $t(n)$ on a un speedup $\leq N$.



On note $p(n)$ le nombre de processeurs pour l'algorithme parallèle traitant le problème de taille n .
C'est un critère économique.

Pour un problème de taille n , le coût d'un algorithme parallèle est :

$$c(n) = t(n) * p(n)$$

Si tous les processeurs exécutent la même masse de travail, $c(n)$ est le nombre total d'opérations du problème.

Sinon, on a un majorant de ce nombre.

Si une borne inférieure de ce nombre est connue et si elle correspond au coût de l'algorithme séquentiel, on dit que **l'algorithme parallèle est à coût optimal**.

Un algorithme parallèle n'est **pas à coût optimal** s'il existe un algorithme séquentiel dont la complexité est **inférieure** à son coût.

Problème dit de la « scalabilité » ou extensibilité

Est-ce que $t(n)$ **diminue** lorsque $p(n)$ **augmente** ?



Sur les exemples

1. Recherche dans un fichier à n éléments avec N processeurs

Modèle «*CREW SM SIMD*» :

- ◊ $t(n) = o(n/N)$
- ◊ $p(n) = N$ alors $c(n) = o(n) \Rightarrow$ **coût optimal**

2. Somme de n nombres sur un arbre à $n - 1$ processeurs :

- ◊ $t(n) = \log_2(n)$
- ◊ $p(n) = n-1$ alors $c(n) = o(n * \log_2(n)) \Rightarrow$ **pas à coût optimal**

pour m sommes de n nombres avec l'effet pipeline :

- ◊ $t(n) = m-1 + \log_2(n)$
- ◊ $p(n) = n-1$ alors $c(n) = o(m.n) \Rightarrow$ **coût optimal**

Notion d'efficacité

On introduit aussi l'**efficacité** :

$$\text{efficacité} = \frac{\text{speedup}}{p(n)} \leq 1$$

