

## **Chapitre 1 : But et définition du parallélisme**

### 1.1 Introduction

**Parallélisme** : avoir plusieurs processeurs qui coopèrent à la solution d'un même problème dans le but de résoudre plus rapidement le problème et de pouvoir résoudre de problème plus grand (trop grand pour la mémoire d'un processeur).

Regard tourné vers performances. Mais pour co-opérer, il faut se co-ordonnées.

**Overhead** : Coordination  $\Leftrightarrow$  échange d'information entre les processeurs.

Le parallélisme a toujours été présent en informatique, mais pas toujours un succès : pas fiable, trop de panne.

C'est là que l'architecture de Von Neumann s'est imposée : succès car besoin de performance continu.

CPU  $\Leftrightarrow$  mémoire (prog + donnée)

Exécution séquentielle.

$\Rightarrow$  Convergence des modèles de programmation, matériel et conception de méthode de résolution.  $\Rightarrow$  succès des ordis séquentiels.

Besoin de performances : On a tjrs besoin de performance.

**Limite Von Neumann** : pour augmenter perf, suffit d'augmenter vitesse d'un proc séquentiel. Mais cela augmente chaleur produite qu'il faut évacuer. Et si chaleur augmente, propriété semi-conductrice des circuits disparaît  $\Rightarrow$  RIP.

**Von Neumann bottleneck** (goulet d'étranglement, « memory-call ») : la connexion mémoire-CPU est lente et le processeur ne peut en général pas fonctionner à sa perf max, car données pas dispo. On peut quantifier cet effet avec le « roofline model ».

**HPC** : High Performance Computing : calcul à haute performance. On doit mesurer perf de calcul pour évaluer efficacité.

**Flops** : Métrique standard de mesure de performance : flops = FLOATING POINT OPERATION PER SECOND.

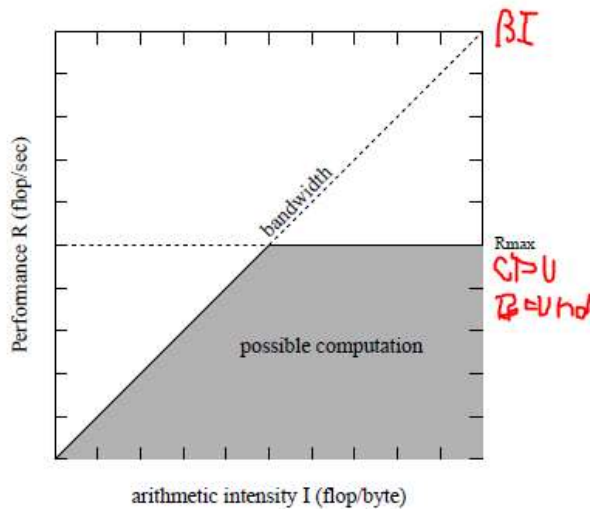
Maintenant, environ Petaflops  $10^{15}$  flops (150Pflops avec 10 Millions de cœurs = 15MWatts)

**Bande passante mémoire CPU** : bêta en byte/sec

**Intensité arithmétique** : I en flop/byte, donne le nombre d'opérations arithmétique qui sont réalisées pour chaque nouvelle donnée.

**Rmax** : perf max du processeur en flops/s

**Performance :  $R = \beta \cdot I$**



## 1.2 Où trouver des performances

On a 3 leviers possibles :

- Technologie : Cela désigne ici les processus physiques utilisés pour faire du calcul. Actuellement, les circuits sont basés sur les **semi-conducteurs au silicium**. Cette technologie a permis des progrès exponentiels durant de nombreuses années.
  - Loi de Moore : la puissance des ordis double tous les 18 mois (à prix constant).
  - Jusqu'en 2006, progrès garantis par la loi de réduction (scaling de Dennard) :
    - Soient :
      - $P$ , la puissance dissipée par le circuit
      - $F$ , la fréquence d'utilisation
      - $V$ , le voltage
    - On a que  $p = v^2 \cdot f$  avec  $v$  qui est une fonction de la fréquence.
    - Pendant longtemps,  $f$  et  $v$  étaient liées à la finesse du trait de photo lithogravure (graver sur pierre).
    - $F$  environ  $1/8$
    - Voltage diminue si  $\Delta$  diminue ceci garantissant une puissance constante. Mais on ne peut pas tjrs diminuer  $\Delta$  (effet de la physique : transistors auront des problèmes pour fonctionner)
- Architecture : en améliorant façon dont différentes étapes de calculs sont organisées et structurées par unités de tritement, on peut considérablement augmenter performances d'un processeur.
- Algorithme :

Le cerveau a ses limites :  $10^{11}$  neurones avec  $10^4$  synapses

Faut-il mieux 1 proc. Très rapide ou beaucoup de processeurs très lents ?

On peut regarder ce problème du point de vue énergétique.

On veut regarder le rapport  $R/p$  où  $R$  est la puissance de calcul et  $p$ , la puissance consommée.

$R$  environ  $\gamma$ ,  $p$  environ  $\gamma^2$ ,  $\gamma$  est la fréquence du proc.

Si on a  $n$  processeurs, on a que  $R/p = n \cdot \gamma / (n \cdot \gamma^2) = 1/\gamma$

Pour maximiser  $R/p$ , il faut que  $\gamma \rightarrow 0$ .

Mais pour avoir une machine qui calcule, il faut prendre  $n \rightarrow \infty$ .

Augmenter performance : modifier architecture des processus

Solutions :

- Parallélisme solution architectural au défi des performances
- Mémoire cache
- Processeurs vectoriels
- ...
- Faire calculs là où on les trouvent les données (des constructeurs pensent faire ça)

Autre sol pour gagner perf. :

- Optimiser code
- Inventer algos

### 1.3 Evolution des architectures :

On peut diviser évolution en période de 10 ans. Chaque période est caractérisée par un type d'architecture dominant.

1970 ILLIAC IV météo 64 proc.

1975-1990 : ère des superordinateurs vectoriels (bcp ralenti le dév du parallélisme)

1988-2000 : boom machine parallèle, SIMD, MIMO, MPP

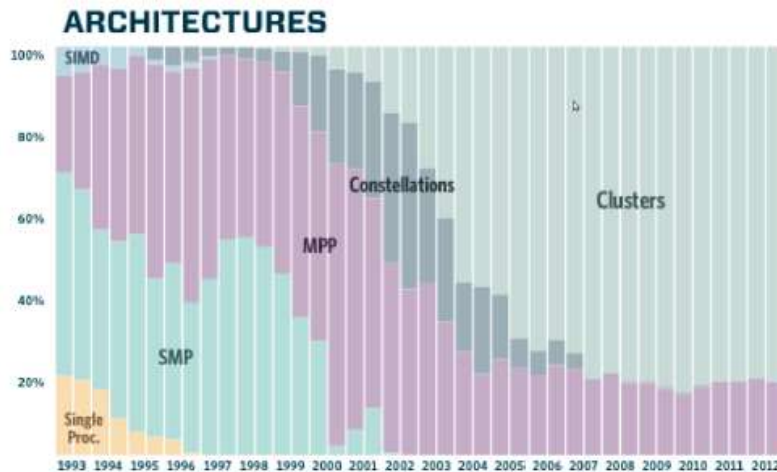
1995 : cluster de PC (beowulf)

2000: GRID, Cloud

2010 : GPU  $\rightarrow$  Green computing exascale  $\rightarrow 10^{18}$  flops/s

Maintenant, parallélisme partout, à l'intérieur de chaque processeurs, entre processeurs, etc...  
Tout cela grâce aux progrès technologiques et architecturaux.

### 1.4 Systèmes parallèles et répartis.



Maintenant, on a des **systèmes répartis (distribués)** partout : les ordinateurs sont interconnectés et constante interaction.

Différences principales :

Système **répartis** : architecture composé de plusieurs processeurs interconnectés dont le but est **la résolution simultanée de plusieurs problèmes** plus ou moins reliés, par une mise en commun de ressource.

Système **parallèle** : machine composée de plusieurs processeurs fortement coupés qui travaillent en même temps et coopère à **la solution d'un même problème**.

pour le parallélisme, on met en commun des ressource qui sont sous contrôle et parfaitement connues. Les proc. Se reconnaissent et tout est fait pour avoir des performances.

	Couplage entre proc	Mise en commun délibérée	Hétérogénéité des ressources	Hypothèse sur le système	Connaissance mutuelle	Fiabilité	Problème de sécurité
Parallèle	Fort	Oui	Non	Oui	Oui	Oui	Non
Réparti	Faible	Non	Oui	Non	Non	Non	Oui

### 1.5 Exploiter le parallélisme

Tirer parti de plusieurs processeurs

Comment faire coopérer à la solution d'un même problème.

Plusieurs stratégies de parallélisation avec exemple de la vie de tous les jours.

Famille dupont faire des tartines :

Tâches :

I1 : couper des tranches de pain

I2 : beurrer les tranches

I3 : mettre confiture

I4 : placer tartines sur un plat

1<sup>er</sup> stratégie (**SEQUENTIELLE ou Von Neumann**)

Mme Dupont fait tout de manière séquentielle (un puis l'autre)

$$T_{seq} = N * 4 * I$$

I = temps d'une opérations

2<sup>ème</sup> stratégie (**TRAVAIL A LA CHAINE PIPELINE**)

Attention : toutes les étapes doivent faire la même durée.

Mme. D : I1

Mr. D : I2

Fils 1 : I3

Fille 1 : I4

$T_{pipeline} = 4T + (N-1)*T$  (il reste N-1 qui sont prêts une à une chaque étape  $t_n$  vaut  $N*t$  si N est grand)

$$T_{seq}/T_{pipeline} = 4*t*n/t*n = 4 // 4 \text{ fois plus vite avec 4 travailleurs}$$

Pour que ça fonctionne, il faut que I1, I2, I3, I4 soient de même durée.

NB : il y a du temps potentiellement perdu pour passer la tartine l'un à l'autre.

Il n'y a pas de place pour une 5<sup>ème</sup> personne pour accélérer encore la fabrication nb-personnes = nb-tâches

Pas de tolérances aux pannes

3<sup>ème</sup> stratégie (**SIMD : Single Instruction flow, Multiple data flow**)

Faire plusieurs tartines en parallèle.

Supervisé par Mme. DUPONT, le mari, les deux enfants, confectionne.

Pour estimer  $T_{simd}$ , on va supposer qu'on a p travailleurs.

En,  $4t$ , on produit p tartines.

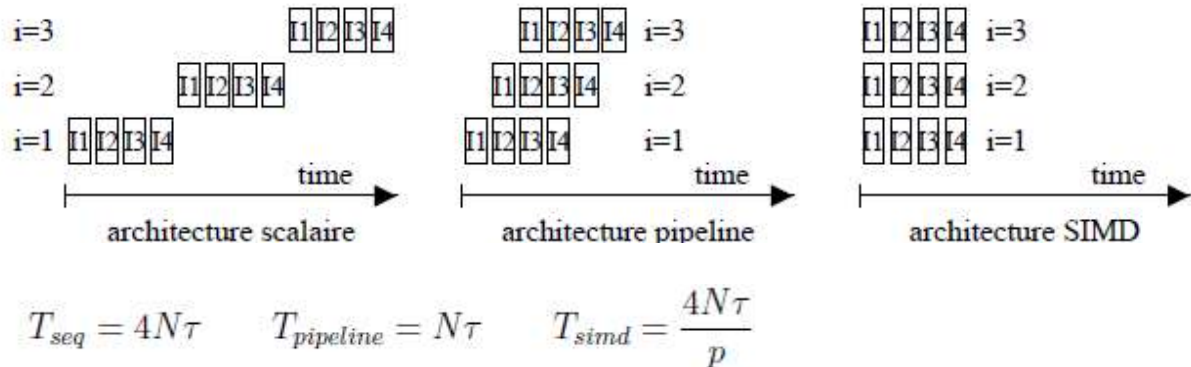
Pour faire  $n$  tartines, il faut  $(N/p)*4*t$  étages de temps :

$$T_{sind} = (4*t*N)/p$$

Ici  $p$ , peut varier de nombreux travailleurs.

Potentiel de parallélisme, beaucoup plus grand

Cette technique fonctionnera bien si chaque travailleur a ses propres données (sa hache de pain, plaque de beurre, ...) → principe de **mémoire distribuée** pour les processeurs.



4<sup>ème</sup> stratégie (**MIMD**) :

Chaque utilisateur est autonome et à la recette de fabrication de tartines. Chacun peut être dans un état (state) différent de la fabrication. Tous les travailleurs sont symétriques. Le plus naturel est d'avoir les données (beurre, pain, confiture) en commun dans une zone partagée. Cela peut évidemment créer des conflits. Si plusieurs personnes veulent le beurre en même temps certains vont devoir attendre → perte de performance.

Cette approche est dite de **mémoire PARTAGÉE**, par opposition à la mémoire **distribuée** proposé avec SIMD.

La mémoire partagée n'est pas **scalable** car avec beaucoup de processeurs, il y a trop de conflits d'accès.

Mais on peut avoir un système MIMD à mémoire distribuée. Dans notre exemple, on est dans un cas dit **SPMD** (Single Program Multiple Data), mais ce n'est pas une obligation du MIMD.

Problème de cette organisation : Comment savoir quand s'arrêter, sachant que l'on veut par exemple 100 tartines ? Il faut imaginer un processus de coordination sophistiqué. Soit il faut communiquer tout ensemble pour décider si ça vaut le peine de faire encore une tartine → compliqué.

Sinon, on force chacun à faire un nombre prédéfini de tartines, mais certains vont plus vite que d'autres et on sera limité par le plus lent. C'est le problème du **déséquilibre de charge** [load balancing] qui est un gros soucis du parallélisme.

On peut aussi avoir des situations de deadlock (interblocage), par exemple si celui qui a le couteau ne le rend pas tant qu'il n'a pas le beurre et vice versa.

Le MIMD donne plus de souplesse que le SIMD, mais peut être source de nombreux problèmes de coordination.

Hint : dans les problèmes réels, il y a beaucoup de problèmes de parallélisme dans les données (énorme masse de données à traiter) mais peu dans les tâches.

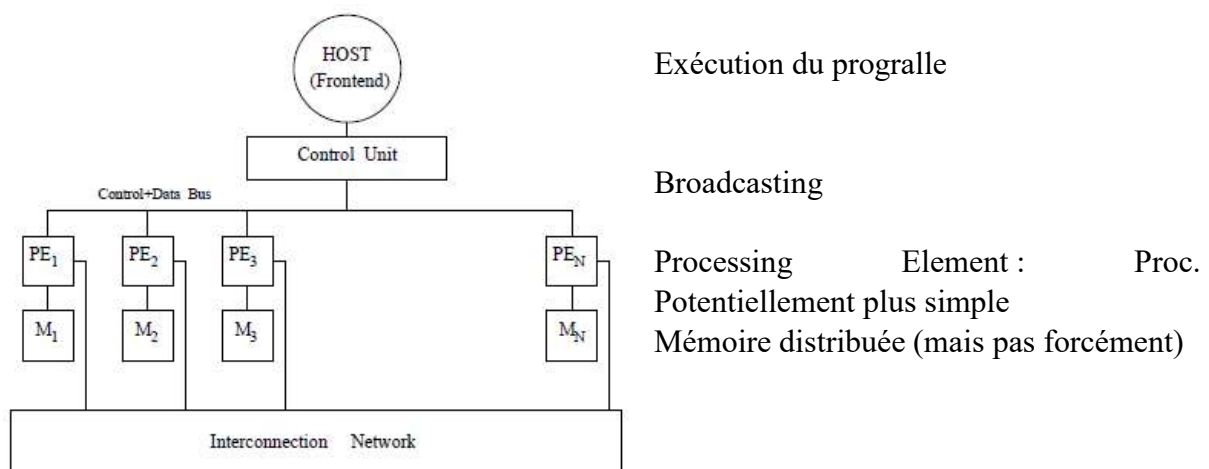
- ➔ Parallélisme de mémoire est profitable
- ➔ Parallélisme de contrôle limité

## Chapitre 2 : Architectures à haute performance

		Flow of Data	
		Single	Multiple
Instruction Flow	Single	SISD (von Neumann)	SIMD
	Multiple	MISD (pipeline ?)	MIMD

FIGURE 2.1 – Diagramme représentant la classification de Flynn  
Architecture de base SIMD (single instruction, multiple data)

### 2.1 SIMD



Maintenant, plus SIMD pour architecture parallèle mais MIMD. Mais SIMD efficace pour certains problème et peuvent justifier leur constructions encore.

Tous les PE sont **synchrones** (attente de finir une tâche avant de passer à la prochaine). Ils travaillent sur leurs propres données. Il existe un réseau d'interconnexion qui permet aux PE d'échanger des données.

Avantages : simplicité/fiabilité et performance

Désavantage : peu efficace pour des problèmes irréguliers

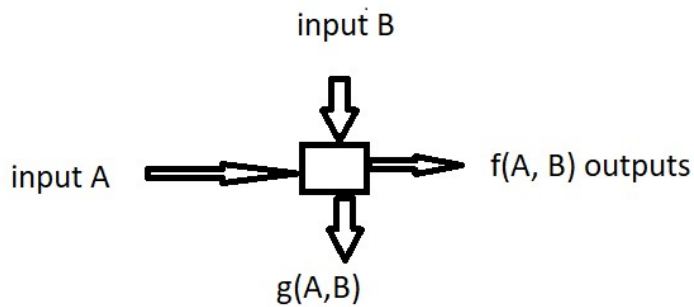
Idéal pour une solution itérative comme des équations différentielles par différence finie. (se trouve dans les GPU modernes).



Maintenant, le SIMD, apparaît seulement dans des composantes décotées et spécialisés comme les GPU et réseaux systoliques.

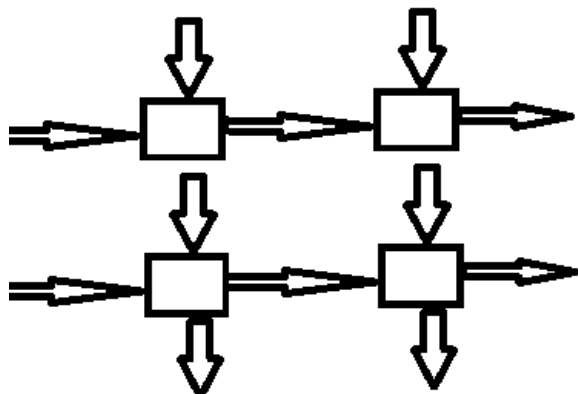
Les réseaux systoliques sont intéressants pour le traitement de signal des systèmes embarqués :

Idée :



Un réseau systolique est une architecture SIMD de type particulier.

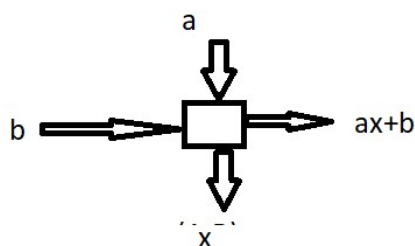
Un réseau systolique est un réseau interconnecté de ces cellules simples (ces cellules travaillent en parallèle) :



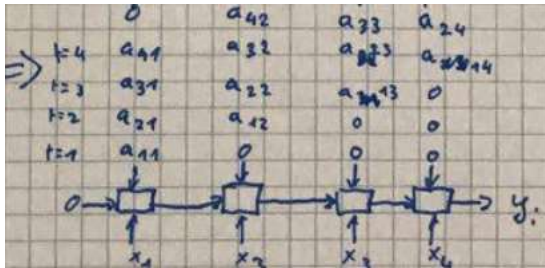
On peut réaliser des multiples de matrices avec cette architecture.

C'est SIMD car fortement synchronisé et tout le monde fait la même chose. Chaque cycle d'horloge correspond à un calcul + communication.

Illustration : multiplication matrice vecteur :  $Ax = y$ ,  $A \in M4*4$ ,  $y, x \in M4$



A la 4<sup>ème</sup> étape, on a obtenu  $y_1$ , puis à chaque cycle, on obtient un autre  $y_i$ .

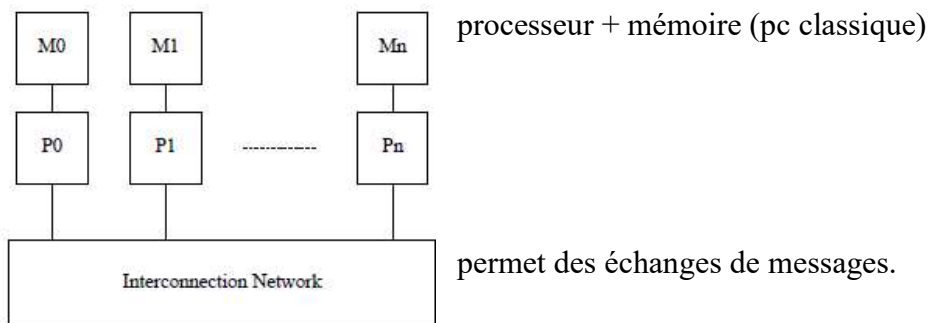


$T_{\text{statistique}}(n) = n \cdot T + (n-1) \cdot T = (2n-1) \cdot T$ ,  $T$  = temps d'un cycle

$T_{\text{seq}}(n) = n \cdot (n + (n-1)) \cdot T = n \cdot (2n-1) \cdot T \Rightarrow n$  fois plus lent

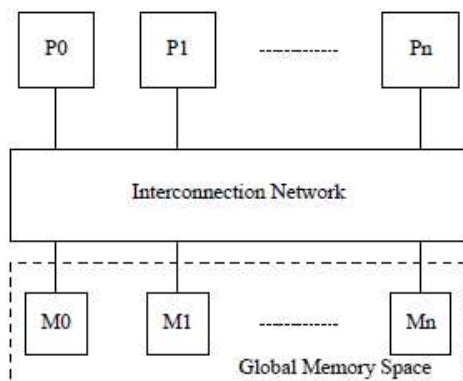
## 2.2 Architecture MIMD :

Processeurs autonomes et asynchrones, chacun avec son propre programme.  
mémoire **distribuée**



Cluster

Schéma mémoire (multiprocesseur) MIMD:



Un processeur n'accède qu'un banc de mémoire à la fois et un banc mémoire ne peut accepter qu'un proc. A la fois.

Une difficulté importante de l'architecture à mémoire partagée est que les processeurs ont des mémoires caches. Une variable peut donc être copiée à plusieurs endroits et potentiellement modifiés de façon incohérents.

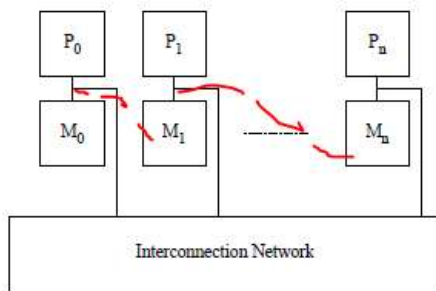
Il faut ajouter en hardware des processus (protocoles) de cohérence des caches qui invalide les copies périmées et qui ne permet qu'une seule modification à la fois (processus atomique). Les processeurs sont continuellement en communication pour assurer cette cohérence → solution peu scalable

Architecture à haute performance :

SIMD,

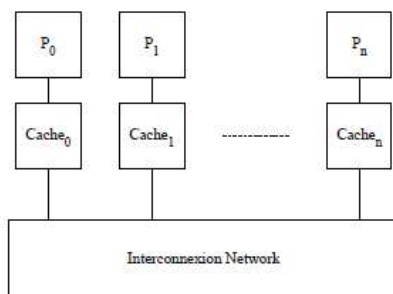
MIMD (mémoire distribuées => scalable =>  $10^6$  cœurs // mémoire partagés => peu scalable => < 100 cœurs)

**NUMA** : Non uniform Memory Access : mémoire virtuellement partagée, physiquement distribuée. On peut accéder directement les autres mémoires, c'est plus lent.



**COMA** : Cache Only Memory Access

Memory associative adressés par le contenu. Les données n'ont pas d'emplacement fixe. Elles migrent de processeurs en processeurs, selon les besoins. Il s'agit de systèmes hybrides ou hiérarchique.



COMA

Multicoeur à mémoire partagée interconnecté de façon distribué

GPU : on l'utilise aussi de plus en plus pour du calcul HPC (High Performance Computing)

C'est une machine MIMD à mémoire partagée avec des processus SIMD aussi à mémoire partagée.

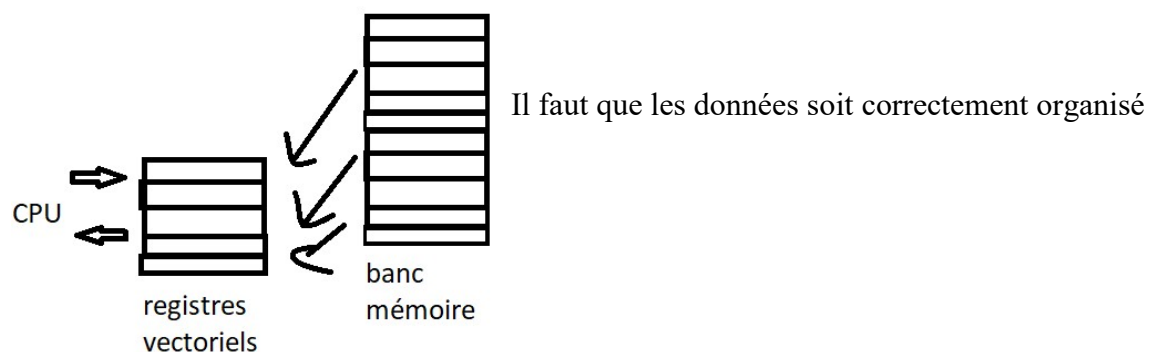
---

## 2.4 Architecture vectorielle

### 3 caractéristiques :

- Registres vectoriels
- Pipeline
- Unités d'exécution multiples

**Registres vectoriels** : accéder à la mémoire par des vecteurs : plusieurs données à la fois (512). Ceci pour faire face au goulet d'étranglement de Von Neumann (lenteur d'accès à la mémoire).



Gain ?

Temps pour traiter N données sur une machine scalaire (séquentielle) et vectoriel :

$$T_{\text{scalar}} = N \cdot T_{\text{fetch}} + N \cdot T_{\text{cpu}} + N \cdot T_{\text{store}}$$

$$T_{\text{vector}} = T_{\text{fetch}} + N \cdot T_{\text{cpu}} + T_{\text{store}}$$

$$T_{\text{scalar}}/T_{\text{vector}} = N \cdot (T_{\text{fetch}} + T_{\text{store}}) / (N \cdot T_{\text{cpu}}) = 2 \cdot T_{\text{fetch}}/T_{\text{cpu}} \gg 1$$

$$T_{\text{cpu}} \ll T_{\text{fetch}} \text{ environ } T_{\text{store}}$$

$$N \cdot T_{\text{cpu}} \gg T_{\text{fetch}} \text{ car } N \text{ grand}$$

Toutes les unités d'exécutions sont faites en 9 étages pour permettre une exécution à la chaîne

$$9T + (N-1) \cdot T = T_{\text{scalar}}/q$$

9T temps de la 1<sup>ère</sup> donnée

(N-1)T temps des N-1 données restantes \* T le temps cycle d'une machine

$$D = A \cdot B + C$$

A, B, C, D, des vecteurs de taille N.

1 cycle pour chaque valeur de D

La vectorisation échoue dans plusieurs cas :

- Dépendance de contrôle (dans le flot d'instruction)
- Dépendance de donnée (comme equation chaleur)

⇒ Casse les pipeline et oblige à traiter les données 1 à 1.

Exemple : dépendance de contrôle

Les conditions if détruit le flot d'instruction et on doit finir le test avant de savoir quel branche prendre.

Pour maintenir le flot d'instruction, on doit transformer id en qqch d'autres.

Dépendance de donnée : produit matriciel de matrice vecteur

Autre ex de dep de données :

S=0 ;

For i=1..n

    S = S + y(i) ; // s = dépendance de donnée

End for

```
s=0
```

```
do i=0,n,q
```

```
  do k=1,q
```

```
    t(k)=t(k)+y(i+k)
```

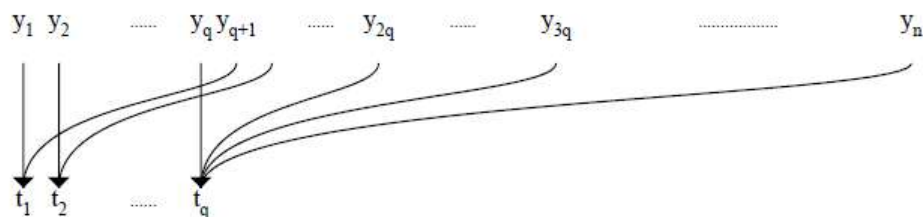
```
  enddo
```

```
enddo
```

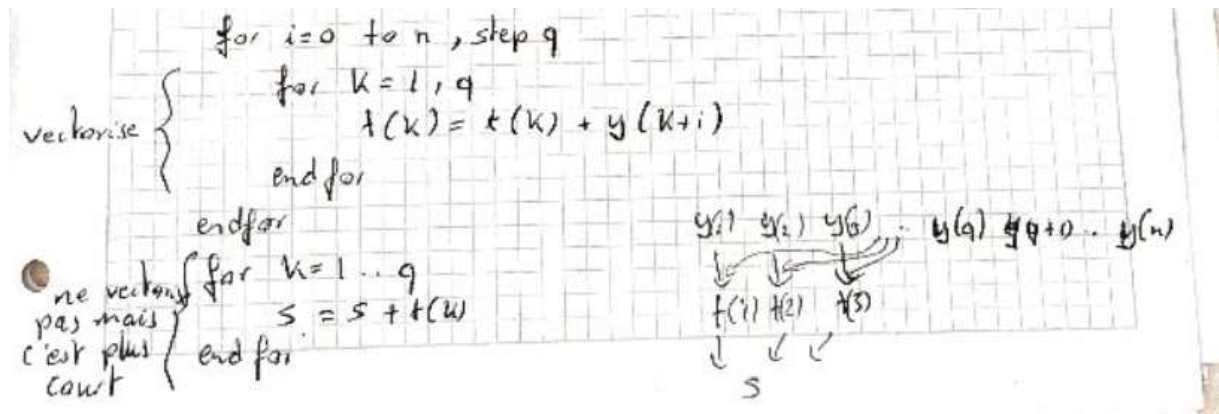
```
do k=1,q
```

```
  s=s+t(k)
```

```
enddo
```



Technique de déroulement de boucle



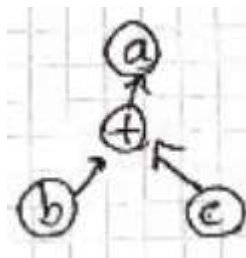
En  $\log_2(n)$ , on arrive à dérouler la boucle.

## 2.5 Architecture Dataflow

Habituellement, ordinateurs sont « contrôles-driver » ou « instruction-driver ». On peut avoir d'autre modèle dans lesquels, ce sont les données qui pilotent l'exécution (**data flow**)

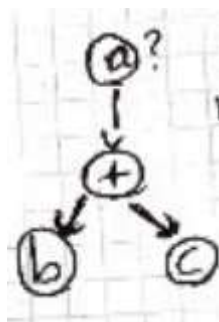
2 possibilités :  $a = b + c$

Data driver

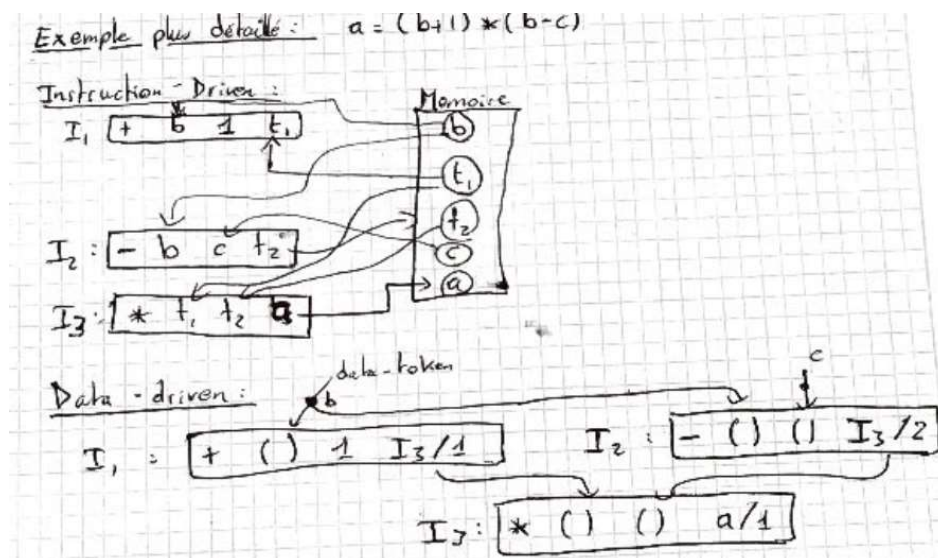


disponibilité de b et c, va activer opération qui donne c.

Demand driver :



la demande de la valeur a, va activer l'opération de recherche des valeurs b et c.



Si  $b$  et  $c$  sont dispo,  $I_1$  et  $I_2$  sont exécutés en parallèle, spontanément, sans avoir à le préciser au niveau programme.

## 2.6 Parallélisme interne

Techniques architecturaux pour augmenter performance. Ex : mémoire cache, pour réduire goulet d'étranglement de Von Neumann.

Dans les processus modernes, il y a du **parallélisme au niveau des instructions** (ILP-processors: Instruction Level Parallelism)

Cela se décline de plusieurs façons

ILP scalaire : CPI = 1 (Cycle Par Instruction : chaque instruction doit être exécuté en 1 cycle machine).

Il faut faire du **pipelining** : utiliser une pipeline d'exécution

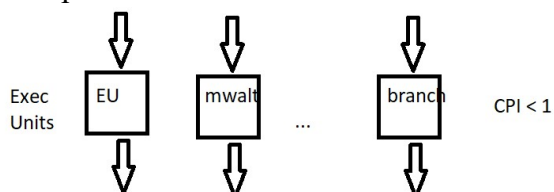
➔ Fetch (des instructions) ➔ decode ➔ execute ➔ write back (résultat dans registres)

RISC CPI = 1

CISC CPI >> 1

**Processus superscalaire** : autrement dit **unités fonctionnelles multiples**

On a plusieurs unités de calcul activables en même temps



P. exemple 0.25 car il y a souvent 4 EU

Comment les exploiter ?

- VLIW : Very Long Instruction word  
I1 = ADD, I2 = NONE, I3 = MVT, I4 = NONE

C'est le compilateur qui extrait le parallélisme à disposition.

C'est une instruction qui en contient 4 chacune pour une unité extrait le parallélisme à disposition.

Mais dans les superscalaire, cela est fait en HW, grâce à des stations de reservations :

➔ Données (Stations de reservation) --- EU

L'instruction est exécuté dès que les données correspondantes sont (data flow) dispo dans la station de reservation

On ne respecte pas forcément l'ordre d'exécution présent par l'ordre des instructions dans le programme.



## Chapitre 3 : Réseaux d'interconnexion

### 3.1 Introduction et définition

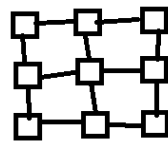
Une machine parallèle, ce sont des processeurs qui coopèrent et ils le font en **communiquant**.

Les communications interprocesseurs sont le fait du **réseau d'interconnexion**. (colonne vertébrale d'une machine parallèle).

Essentiel : rapidité sinon bénéfice parallélisme perdu.

Il y a plusieurs façons d'interconnecter des processeurs.

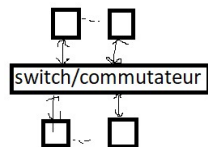
**Topologies :**



Un carré c'est un proc+routeur

Statique :

graphe dont les nœuds sont les proc + routeurs et les arcs sont les fils de connexion.



Dynamique :

Les processeurs, la périphérie d'un switch et d'établir des connexions entre paire de processeur.

But des topologies : grande bande passante, faible latence

**Propriétés d'un réseau d'interconnexion :**

- Le diamètre D : longueur minimal qui relie deux processeurs les plus éloignés
- Le degré : nombre de liens (fil) qui arrive sur un processeur
- Bande passante : Débit du réseau en byte/sec
- Latence : temps qui sépare msg envoyé et arrive a destination (latence physique + latence préparation du message)
- Largeur bisectionnelle : mesure débit global dans réseau (il faut diviser le réseau en 2 moitié égales et en comptant le nombre de liens entre ces 2 moitiés) : nombre de lien quand tu sépare réseau en deux.
- Prix et/ou complexité : mesure en fonction du nombre de processeurs, qté de matériel
- Scalabilité : peut on tjrs faire croitre ou ya limite ?

**Routage :** algorithme d'acheminement des messages à travers le réseau. Dans une machine parallèle, on va exploiter au maximum la connaissance du réseau d'interconnexion et algorithme de routage pourra varier selon la topologie.

Algo gère mieux les congestions de reseau en redirigeant des msg.

Routage local asynchrone.

**Technicien de commutation :** comment les messages se déplacent-ils à travers le réseau.

**STORE AND FORWARD** : (mauvaise solution)

Message entièrement reçu par nœud intermédiaire avant d'être envoyé au nœud suivant.

W est la bande passante. Il faut  $1/W$  pour passer 1 Byte

Temps store and forward pour m bytes =  $m \cdot l/W$  avec l la longueur du chemin

**Commutation par train de bits** : (bonne méthode) (worm hole/ cut through)

On laisse le message traverser les nœuds intermédiaires sans l'arrêter, à la façon d'un locomotive qui tire ses wagons. Locomotive ouvre chemin et le réserve jusqu'au passage du dernier wagon.

$T_{cut\ through} = l/w + (m-1)/w$  (à chaque cycle les m-1 morceaux restant arrivent  
 $l/w \rightarrow$  temps du premier morceau

$T_{out\ through}$  environ  $M/W$  si M est grand

Modèles de performance pour les communications :

$T_{communication} = M/W + t_s$  ou  $t_s$  est le temps de latence (ou startup précédemment défini)

Indépendamment de la distance qui sépare l'expéditeur du destinataire.

Si M est assez grand, on pourra négliger  $t_s$  par rapport à  $M/W$  et on aura  
 $T_{communication} = M/W$

### 3.2 Primitives de communication

**Communication point-à-point** : un proc  $p_i$  envoie un message à un proc  $p_j$ . Ceci se fera grâce à l'algorithme de routage qu'on vient de discuter.

En MPI, les Send, Receive. Mais souvent, on a des patterns de communication qui implique tous les processeurs : **communications collectives**.

**Permutation** : chaque proc.  $P_i$  envoie à un proc  $p_{j(i)}$  où la relation  $i \rightarrow j(i)$  est une bijection  
( $p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n \rightarrow p_1$ )

**Broadcast (one-to-all)** : diffusion : un proc. Envoie une valeur à tous les autres

**Réduction (many-to-one or all-to-one)** : Tous les proc envoient une valeur à un seul, ou toutes ces valeurs seront combinées par une opération arithmétique ou logique (addition ex).

Gather : rassemble, scatter : divise

**Echange total** : (multi broadcast) chaque processeur envoie une donnée à tous les autres

**Echange total personnalisé** : tous les processeurs envoient une données différente à tout les autres : multi-scatter

**Opération de scan ou préfixe-parallèle** : combinent communication et calcul de sorte que la donnée finale du ième proc soit la somme (ou autre op) : contrairement a reduce, le calcul est fait de manière équitable par chaque proc.

Si réseau d'interconnexion bon, scan s'exécute en  $O(p)$

### 3.3 Réseaux Statiques

Plusieurs topologies statiques d'interconnexions

#### Anneau :

Diamètre :  $N/2$  (croit linéariement avec  $N$  le nb de proc)

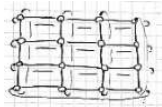
Degré : 2 indép. De  $N$

Largeur bissectionnelle : 2 indépendant de  $N$

Prix :  $O(N)$

Scalabilité : Oui

#### Grilles :



2D :

Diamètre :  $O(\sqrt{N})$

Degré : 4 indépendant de  $N$

Largeur bissectionnelle :  $\sqrt{N}$

Prix :  $O(4N)$ , 4 = nombre de connexions

Scalabilité : oui

3D :



Diamètre :  $O(N^{1/d})$  racine d-ème (IBM BQQ :  $d=5$ )

Degré :  $2d$

Largeur bissectionnel :  $N^{1/d} \rightarrow$  tend vers  $N^{1/d} \rightarrow N^{1/d} \rightarrow N^{1/d}$

Prix :  $O(b*d)$

Scalabilité : Oui

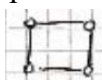
Routeage : d'abord on s'aligne à j puis on monte/descend à i

**Hypercube** : grille de dimension  $d$  mais d'arrête 2. Intressant de voir un processus de construction différent. Caractérisé par sa dimension  $d$ .

$D = 0$  : 1 seul proc


$D = 1$  : 2 proc °-°

$D = 2$  :



4 proc

D = 3 :  8 proc

D = 4 :  16 proc

Le nombre de processeurs (nœuds) N est relié à la dimension  $N = 2^d \Leftrightarrow d = \log_2(N)$

Pour construire un hypercube de dim. K, on prend deux hypercubes de dim k-1 et on relie les nœuds correspondants.

Diamètre : d, on suit chacune des arêtes du cube en dim d. (d est le nombre d'arêtes de l'hypercube).  $D = \log_2(N)$  donc diamètre :  $O(\log(N))$

Croissance la plus faible vu jusqu'à maintenant

Degré :  $\log_2(N)$  donc augmente avec la taille

Largeur bisectionnelle :  $N/2$  car par construction d'un hypercube de dim d-1 avec  $2^{(d-1)} = N/2$  nœuds. Comme on l'a dit, on relie deux cubes de dim d-1, donc ça correspond à ça

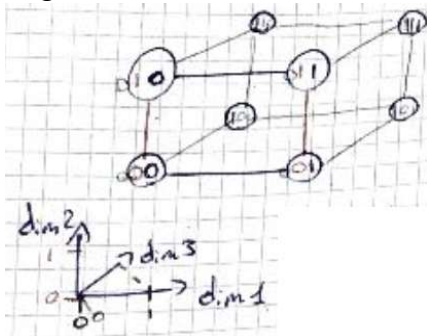
Prix :  $dN$  (nombre de lien par nœud fois nombre de nœud) =  $O(N \cdot \log(N))$  croissance plus rapide que linéaire

Scalabilité : non car il faut des nœuds avec de + en + de lien et donc on ne peut pas simplement combiner.

IBM Blue Gene grille de dim 5

Numérotation des nœuds d'un hypercube :

Si grille d'arête 2, dim d. Il faut coordonnées à dim d., dont chacune à 2 val. Possible 0 ou 1



Avec numérotation décimale, donnée par MPI\_Rank, on ne peut pas construire chaîne de bit et savoir position du proc. Dans topologie virtuelle d'hypercube.

CSQ : 2 nœuds voisins sur hypercube si leur 10 diffère que de 1 bit

On envoie à voisin de dim. En général, il y a  $d = \log_2(N)$  étapes

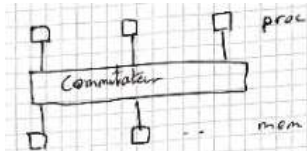
Routage : on prend adresse origine et XOR avec destination. Les 1 restants indiquent dimensions à traverser.

Dans topologie de grille, on augmente le nb de processeurs en augmentant la taille des arêtes de la grille, mais on garde la dimension constante. Pour hypercube, c'est le contraire : on a toujours des arêtes de taille R mais on augmente D.

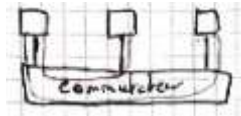
... un peu fucked up parce que description algo nul à chier ...

### 3.4 Réseaux dynamiques

Essentiellement switches ou commutateurs qui relient les proc. Ensembles, où les proc. Et mém. Ensembles.



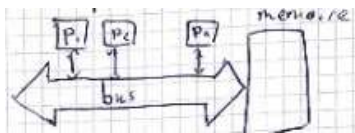
Mém partagé



le carée a meme et proc

3 catégories de réseaux dynamiques (plus ça descend, plus il y a de fonctionnalité et prix) :

- Connexion par bus : beaucoup utilisé en informatique mais pas vraiment pour le parallélisme (mémoire partagée)

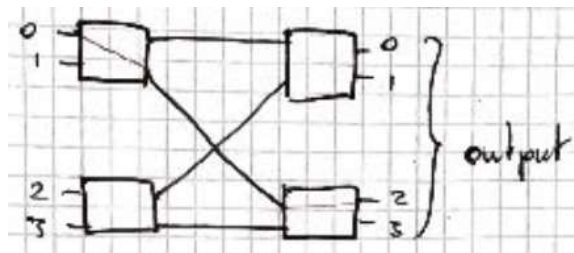


plusieurs proc partage même bus pour accéder mémoire.  
=> un seul proc à la fois

Bande passante  $O(1/N)$ , donc décroît, plus il y a de processeur. PAS SCALABLE

En plus, il faut mettre en place, un FIFO pour accès mémoire.

- Réseaux multi-étages : bon compromis entre coût et performance.



Même switch 2\*2 que tout à l'heure

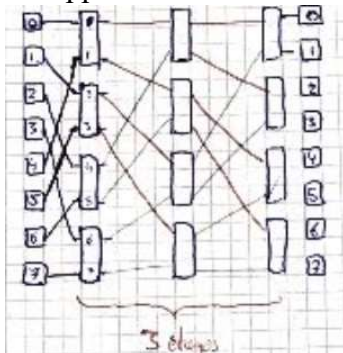
Chaque entrée peut être connecté à n'importe quel sortie.

Mais un lien entrée sortie contraint les autres possibilités.

Si  $0 \rightarrow 2$ , 1 ne pourra aller que sur 0 ou 1. C'est pourquoi, on dit qu'il est **bloquant**.

Solution pour diminuer cela : ajouter des chemins avec couches supplémentaires.

On appelle cela un réseau **OMEGA**. C'est un parmi plusieurs réseau multi étages possible.



Nombre d'étage  $\log_2(N)$

Il faut réfléchir en binaire pour les connexions : ex pour aller de 2 à 5 ( $010 \rightarrow 101$ )

1<sup>er</sup> étage : 1 0 1

2<sup>ème</sup> étage : 0 1

3<sup>ème</sup> étage : 1

C'est un shift circulaire vers la gauche des bits d'entrées.

Cette permutation : **shuffle inverse**.

Algo routage : adresse destination : chaque étage, on regarde bit dominant, 0 sort par haut, 1 par bas peu importe l'origine

Preuve : Soit adresse  $O1O2\dots Ok$  où  $k = \log_2(N)$  ici  $k=3$

Soit  $d1d2\dots dk$  adresse destination

1<sup>er</sup> ligne : permutation cyclique à gauche

On part de  $e_1e_2\dots e_k \xrightarrow{\text{cyc}} e_2e_3\dots e_k e_1$   
 $\rightarrow \begin{cases} e_2e_3\dots e_k 0 & \text{sortie haute } (d_1=0) \\ e_2e_3\dots e_k 1 & \text{sortie basse } (d_1=1) \end{cases}$   
 $= e_2e_3\dots e_k d_1 \rightarrow e_3e_4\dots e_k d_1 d_2$   
 $\rightarrow e_3\dots e_k d_1 d_2$   
 $\rightarrow d_1d_2\dots d_k \text{ après } k \text{ étages de shuffle-exchange}$

Haut

Bas

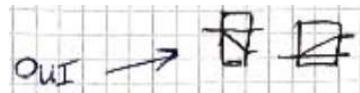
Shuffle

Sortie commutation

Cela montre qu'il faut un nombre  $k$  d'étage (nombre de bit d'adresse)  $= \log_2(N)$   
 Réseau oméga reste bloquant car un choix de chemin empêche certains autres d'être réalisé.

Latence  $= O(\log_2(N))$

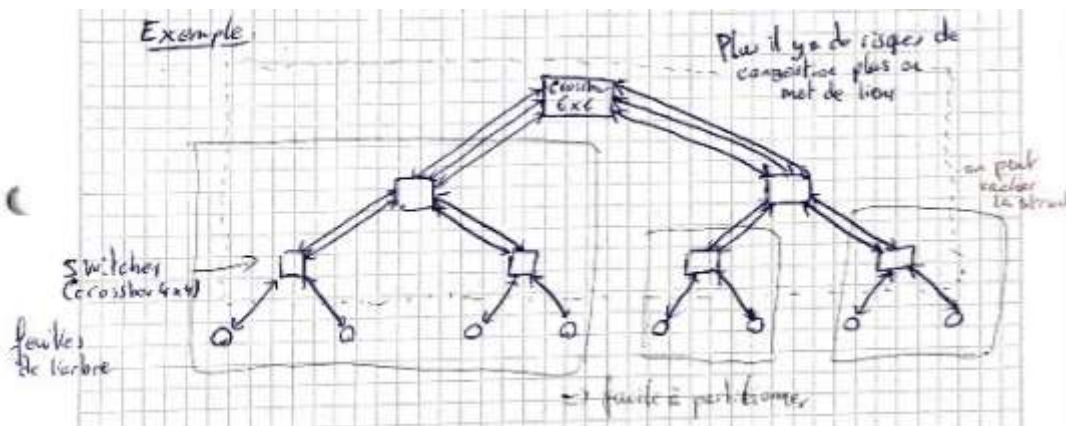
Comment faire un broadcast ? Il faut broadcast sur commutateurs élémentaires



Prix :  $(n/2 \text{ par étage}) * \log_2(N) \text{ (nb étage)} = O(N * \log(N))$

Scalabilité : il faut tout recablé car entre  $N$  et  $2N$ , le shuffle est différent. On ne peut pas prendre 2 oméga de taille  $N$  pour en faire un de taille  $2N$

**FATTREE** : Un type de réseau multi-étage couramment utilisée :



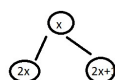
Chaque nœud de l'arbre est un commutateur.

Feuille : input et output

**Arbre gras** car branches s'épaississent à mesure que l'on monte vers la racine, ou la bande passante est de plus en plus nécessaire.

Réseau qui permet aussi de facilement créer de partitions de processeurs indépendants les uns des autres.

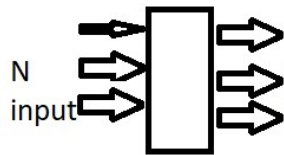
Routing dans un arbre binaire :



(en binaire : valeur binaire : deux enfants : valeur parent suivi de 0 ou 1)

Pour aller d'un nœud à un autre (12 à 7), il faut remonter jusqu'à ancêtre commun, (même préfixe) et on redescend selon la valeur.

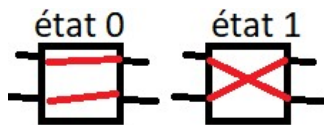
■ Crossbar : cher mais performant



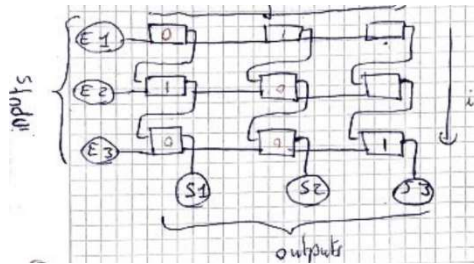
Il peut relier les input aux outputs : toutes les permutations de N entrées sur les N sortie sont réalisables.

Si permutation  $i \leftrightarrow j$ , pas de conflits entre chemin.  $N!$  pattern de connexions. Mais il ne peut pas avoir deux entrées sur la même sortie.

Commutations :



Exemple d'un crossbar 3x3



On peut ainsi faire toutes les permutations. Il n'y aura jamais deux commutateurs de la même colonne dans l'état 1 car deux entiers ne peuvent pas se connecter à la même sortie.

Dans un vrai, il faut ajouter gestion de conflits et files d'attente, donc coût plus élevée.

Désavantage : prix croît comme  $N^2$  puisqu'il contient

$N \times N$  éléments. Solution courante pour relier cœurs d'un proc multicœur à mémoire.

Il y a eu des 64\*64 crossbar



## Chapitre 4 : Travail et Speedup

Puissance du processeur noté R (pour exécution Rate) : sa « vitesse » ou le nombre d'instructions (ou opérations) qu'il réalise par seconde. Par exemple :  $R = 1 \text{ Gflops/s}$

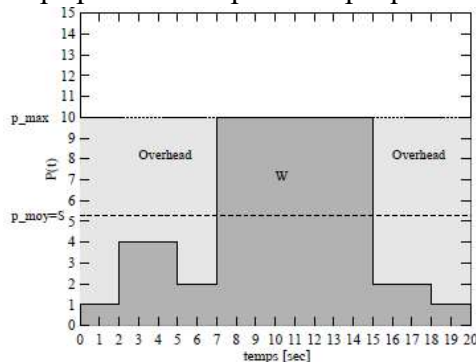
Travail W (pour work) : travail nécessaire pour résoudre un problème. Nombre d'instructions utilisés pour résoudre le problème.

Degré de parallélisme  $p(t)$  : nombre de processeurs actifs au temps t (ce qui font vraiment du travail)

On a aussi  $p = p_{\max}$ , le nombre total de proc. Utilisés.

Temps séquentiel  $T_{\text{seq}}$  : tps nécessaire à 1 seul pour résoudre le problème

Temps parallèle :  $T_{\text{par}}$  : temps qu'il faut au p processeurs utilisés.



$\Delta W = R \cdot \Delta T$  : travail fait par 1 proc. Pendant  $\Delta T$  est égal à  $\Delta W$

$$W = \int_{t_{\min}}^{t_{\max}} \underbrace{p(t)}_{\text{nb proc actifs}} \underbrace{R}_{\text{nb d'instructions résolues pendant dt}} dt$$

Pour faire ce travail en séquentiel :  $T_{\text{seq}} = W/R = \int_{t_{\text{start}}}^{t_{\text{end}}} p(t) dt$

$T_{\text{par}} = T_{\text{end}} - T_{\text{start}}$

**Speedup =  $T_{\text{seq}}/T_{\text{par}}$  :** 
$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{(W/R)}{t_{\max} - t_{\min}} = \frac{1}{(t_{\max} - t_{\min})} \int_{t_{\min}}^{t_{\max}} P(\tau) d\tau$$

=  $P_{\text{moyen}}$  = degré parallélisme moyen

Idéalement  $s = p = p_{\max}$ . Mais il faut que tout les proc. Soit tout le temps actif

**NB : algo séquentiel : faut prendre le meilleur algo possible**

Pour certains problème actuel de grande taille, impossible de résoudre en séquentiel.

**Efficacité :  $E = S/P = T_{\text{seq}}/(P \cdot T_{\text{par}})$**

Idéalement, on aimerait  $E = 1 = 100\%$ , mais généralement  $< 1$

Travail parallèle :  $W_{\text{par}} = p \cdot T_{\text{par}}$  (un facteur  $\cdot R$  a été enlevé car intéressant)

**Overhead** : différence entre le travail parallèle et travail utile. Généralement positif  $W_{\text{par}} > W_{\text{seq}}$  car en parallélisme, il y a de la coordination entre processus (tps de communication, tps d'attente entre les processus)

$$W_{\text{par}} - W_{\text{seq}} = p \cdot T_{\text{par}} - T_{\text{seq}}$$



Si objectif, aller plus vite que séquentiel : speedup>1 intéressant (24h de calcul pour trouver météo du lendemain, on accélère ça).

Mais pour des raisons d'utilisations des ressources (coût), on préfère des E>60%

Souvent 2 situations pour discuter de limites et bénéfice du parallélisme.

### Loi d'Amdhal et Loi de Gustafon

**Amdhal** : On a un travail W à réaliser avec P processus à disposition. Fraction alpha de ce travail ne se parallélise pas alors le reste le reste 1-alpha se parallélise idéalement.

$$T_{par} = \frac{\alpha W}{R} + \frac{(1-\alpha)W}{pR} \quad \text{Première partie séquentiel. L'autre travail restant divisé sur p proc.}$$

$$T_{seq} = W/R$$

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \leq \frac{1}{\alpha}$$

Speedup :

speedup limité par fraction code séquentiel

Vision pessimiste du parallélisme

**Gustafon** : Vision optimiste du parallélisme.

Pendant une fraction beta du temps parallèle, on a 1 seul proc. Qui travaille, alors que pendant le temps restant (fraction 1-beta), on a p proc utilisés

Travail total effectué :  $W = \beta T_{par} R + p(1-\beta) T_{par} R$  (premier : partie séquentiel, deuxième ; réalisé par p proc)

$$\text{Speedup : } S = \frac{T_{seq}}{T_{par}} = \beta + p(1-\beta) = O(p)$$

$$T_{seq} = W/R = \beta T_{par} + p(1-\beta) T_{par}$$

Différence entre Amdahl et gustafon : dans le 1<sup>er</sup> cas, on essaye de paralléliser un travail donné, constant, alors qu'avec Gustafon, on augmente le travail proportionnellement à P (plus la machine est grande, plus le problème sera grand).

1 femme 9 mois → 9 femmes ne peuvent pas avoir un enfant en un mois

9 femmes peuvent avoir 9 enfants en 9 mois.

**CONCLUSION : parallélisme n'est pas seulement une façon d'aller plus vite, c'est surtout une bonne façon de traiter des problèmes plus grand dans le même temps.**

Il faut ajouter la taille du problème au nombre de processeurs

Weak scaling : on augmente W de sorte que le travail par proc soit constant

Strong scaling : on garde W constant et on augmente p

Speedup < p (nombre processeurs)

$$W_{seq} = R * T_{seq}$$

$$W_{par} \rightarrow R * T_{par}$$

En général  $W_{par} > W_{seq}$  à cause communication, synchronisation etc (overload du parallélisme)

En général, on a  $E < 1$

### Speedup surlinéaire :

Cas particulier où on observe  $S > P$

Pour deux raisons :

- 1) Matériel : problème trop grand pour entrer en mémoire cache d'un seul proc. Mais une fois divisé sur  $p$  proc., il se peut que les données tiennent toutes les caches respectifs. Ainsi accès mémoire plus rapide en parallèle qu'en séquentiel. On peut décrire cela en disant qu'en parallèle la puissance effective est supérieure à  $R$ .

### 4.2 Effet algorithmique

On suppose qu'on recherche une donnée dans un tableau de taille  $N$  qui est distribués sur  $p$  proc.

Exemple

### Systèmes hétérogènes

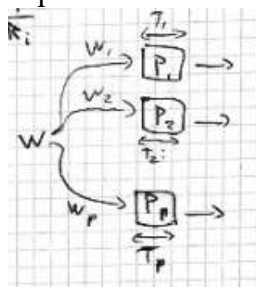
Comment définir speedup si processeurs différents.

A quoi comparer execution parallèle.

Soit  $W$ , le travail total, divisé en  $p$  morceaux  $W_i$ , sur  $p$  proc.

$$\sum_{i=1}^p W_i = W$$

On pose  $T_i$  comme le temps du proc  $P_i$  pour réaliser  $W_i$  :  $T_i = W_i / R_i$



Execution optimale est celle où tous les  $T_i$  sont égaux disont  $T_{opt}$ .

Si un  $T_i$  est plus court que  $T_j$ , on donne son travail de  $P_j$  à  $P_i$  pour équilibrer.

$T_{par}$  : proc le plus lent

Charge  $W_i$  est telle que dans une exec optimale,  $T_{opt} = W_i / R_i$

$$W_i = R_i * T_{opt}$$

$$T_{opt} = W / \text{somme}(R_i)$$

On peut alors définir l'efficacité  $E = T_{opt} / T_{par} = W / \text{somme}(R_i) * 1 / T_{par} = W / (P * R_{mag} * T_{par}) = T_{seq} / p T_{par}$  ou  $T_{seq} = W / R_{mag}$

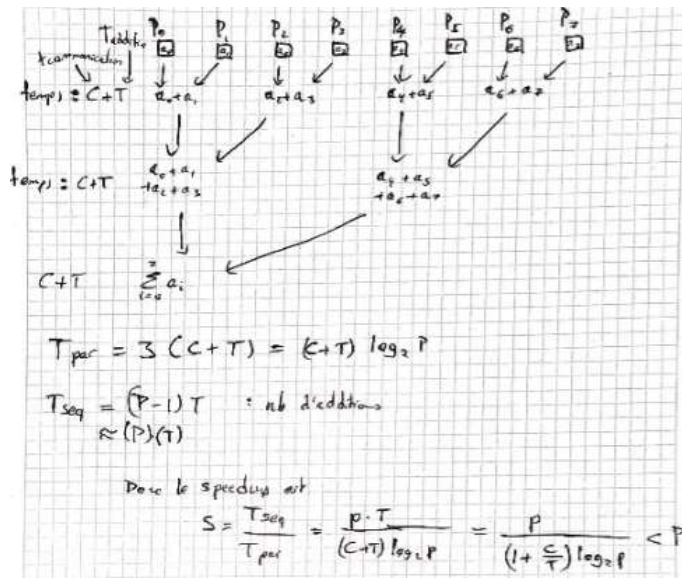
Cela suggère que bonne façon de définir speedup est de comparer le  $T_{par}$  au  $T_{seq}$  obtenu par un proc de puissance  $R_{mag} = \text{somme}(R_i) / p$

## Chapitre 5 : Modèles de performances

Ici, on va considérer deux app parallèle et donner expression du temps de calcul en fonction de la taille du problème, du nombre de proc, leur puissance et des perfs du réseau d'interconnexion.

Le premier problème : sommer n nombre répartis sur p processeurs.

1<sup>er</sup> cas :  $n=p$  : 1 valeur par proc. On veut recevoir le résultat de la somme de toutes ces valeurs :



Speedup inférieur à  $p$  :  
communication qui fait que  $1 + T_{\text{com}}/T_{\text{cal}} > 1$  et en général  $T_{\text{com}} \gg T_{\text{cal}}$ .

Même si  $T_{\text{com}} = 0$ , on a toujours  $s \ll p$

Raison est que l'on utilise de moins en moins de processeur pour ce travail. Il y a des cycles de perdu.

$p/\log_2 p$ , nombre moyen de processeur utilisé durant le calcul.

(Mapping sur hypercube)

2<sup>ème</sup> cas : on a  $n \gg p$  valeurs distribués sur  $p$  proc.

Il y en a  $n/p$  par proc. (on suppose que  $p$  divise  $n$ ).

Mauvaise algo : on répète  $n/p$  fois l'algo procédant avec 1 val. Par proc. On obtien  $n/p$  valeurs dans  $P_0$  qu'on somme ensuite.

**Modèle de performance :**

... TODO ... juste des calculs

Bon algorithme : On fait d'abord somme locale à chaque proc. Sur les  $n/p$  proc. Ensuite somme parallèle

... TODO ... juste des calculs

Ici, le temps de calcul  $n/p$  sera contraint par le prix avec 4 données. Le temps de calcul sera pareil, déterminé par les proc de charge 4.

Mais on peut s'attendre à ce que la solution à 5 proc, ait moins d'overhead car overhead croît avec proc.

## 2<sup>ème</sup> algorithme possible :

Résolution de l'équation de la chaleur (laplace)

Chaque sous domaine de taille  $l^3$  sera traité par un autre proc. On a donc  $N^3 = p \cdot l^3$

Temps d'exécution en parallèle d'une itération est :  $T_{par}(N^3, p) = T_{cal} + T_{com}$  où  $T_{cal}$  est le temps pour itérer sur  $l^3$  points et le  $T_{com}$  est le temps pour accéder aux voisins des sous domaine.

Soit  $T$ , le temps d'une opération arithmétique et  $C$  le temps de communication pour 1 donnée.

$$T_{par}(N^3, p) = l^3 \cdot 6T \text{ (5 addition + 1 multiplication)} + 6 \cdot l^2 \cdot C \text{ (taille d'une face).}$$

$$= 6 \cdot l^3 / R + 6 \cdot l^2 \cdot C \text{ où } R \text{ est la vitesse du proc } R = 1/T$$

$$T_{seq}(N^3) = 6 \cdot N^3 / R$$

$$\text{Speedup} = T_{seq} / T_{par} = 6 \cdot N^3 / R = p / (1 + (RC/l)) \text{ optimal } p$$

$$P = N^3 / l^3$$

Speedup tend vers  $p$  si  $l \rightarrow \infty$ , ie. Problème assez grand dans chaque processeurs.

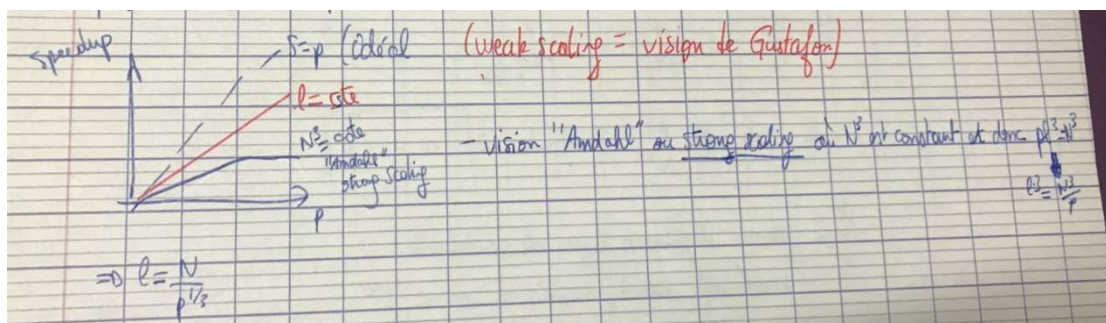
On a aussi  $RC/l$  qui égal à  $T_{com} / T_{cal} = 6 \cdot l^2 \cdot c / (6 \cdot l^3 / R) = RC/l$

Ici, vraiment les temps de communication  $c \neq 0$  qui limite speedup. On voit plus les sous domaines sont grand, plus le rapport  $T_{com} / T_{cal}$  diminue car surface/volume  $\rightarrow 0$  si volume croît. Sphère serait la forme la plus favorable pour ce rappel mais pas utilisable en pratique (découpage en tranche est moins favorable que le cube).

Courbe de speedup :  $p \rightarrow S(p) = p / (1 + (RC/l))$

Deux approches possible :

On dit que  $l$ , indépendant de  $p \Rightarrow$  taille  $N^3$  du problème soit avec  $p$ .



**Scalabilité** : idée de base : augmenter nb proc, augmente perf ? Ici, on va se concentrer sur **scalabilité d'application** : comment perf changent avec taille du problème et nombre de processeurs.

Cette notion peut se formaliser à travers concept d'iso-efficacité.

Fonction d'iso-efficacité  $f_E(p)$  donne taille  $n$  du problème qu'il faut en fonction de  $p$  pour garder efficacité  $E = S/p$  constante.

Comment varier  $n$  en fonction de  $p$  pour garder  $E$  constant ?

Il faut que  $p \cdot \log(p)/n = \text{constante}$  ( $1/8$ )  $\rightarrow n = \gamma \cdot p \cdot \log(p)$ , fonction d'iso efficacité.

Si pas de  $n$  dans la fonction, la fonction d'iso efficacité n'existe pas.

Equation de la place :

$S = P / (1 + (RC/l)) = P / (1 + RC(P/N^3)^{1/3})$  où  $N^3 = n$  est la taille du problème.

$E = S/P = \text{cste} \rightarrow P/N^3 = \text{cste} \rightarrow n = \gamma \cdot p$

Cela donne différents type de scalabilité :

- Scalabilité idéal/linéaire :  $n = \gamma \cdot p \rightarrow$  favorable au parallélisme
- Scalabilité polylogarithmique :  $n = \gamma \cdot p \cdot \log^k(p)$   $k \geq 1 \rightarrow$  favorable au parallélisme
- Scalabilité faible :  $n = \gamma \cdot p^k$   $k > 1$
- Non-scalabilité : fonction d'iso-efficacité n'existe pas.

## **Chapitre 6 : Tâches, partitionnement, ordonnancement et équilibrage de charge**

Granularité d'une tâche : sa taille, nombre d'instruction ou taille de E et S (ensemble de données entrée et sorties)

Granularité fine (avec 1 seule variable) : granularité grossière : programme complet

$T_i$ , caractérisé par ensemble de variable d'input  $E_i$  et variable output  $S_i$ .

### **Indépendances entre tâches :**

Si  $T_i$  et  $T_j$  sont indépendant (ensemble de Julia), on peut les exécuter dans n'importe quel ordre (voir simultanément).

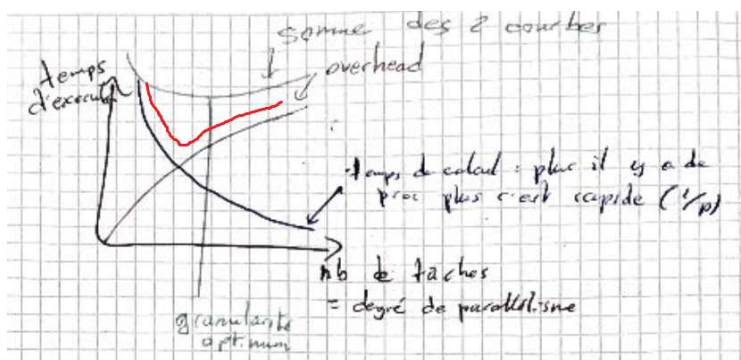
Si  $T_i$  et  $T_j$  ne sont pas indépendant, il faut spécifier ordre dans laquelle elles doivent être spécifiés. On notera :  $T_i < T_j$ , si  $T_j$  doit s'exécuter après  $T_i$ .  $T_j < T_i$  sinon.

Si on viole ces dépendances, on obtient des résultats faux. On peut aussi définir les tâches consécutives.

$T_j$  est consécutif à  $T_i$ , s'il n'existe aucune autre tâche  $T_k$  tel que :  $T_i < T_k$  et  $T_k < T_j$ .

On peut donc faire graphe de précédence.

**Le partitionnement** : découpage d'un problème en tâche.



Granularité optimum : granularité des tâches de plus en plus fin.

Souvent partitionnement, découpage du domaine. Partitionnement optimale difficile.

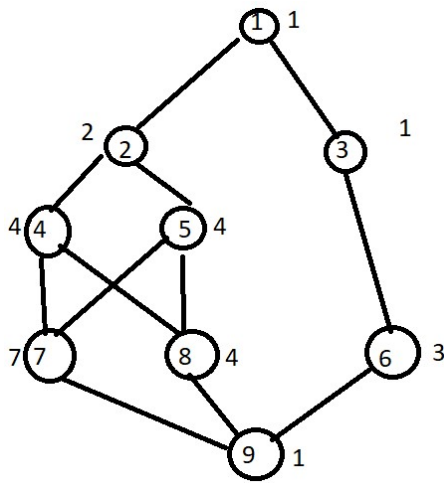
**Le placement** : choix du processeurs qui vont exécuter les tâches obtenus après partitionnement.

**Objectif** : Trouver un placement qui minimise l'overhead comme communication ou temps de synchro.

Tâches indépendantes mais de durée différentes. Il faut limiter inactivité du proc. (mauvais équilibrage de charge).

**Scheduling** : temps auquel chaque tâche peut s'exécuter sur le proc. Qui la détient de sorte à ne pas violer la dépendance. Méthode des temps au plus tôt et au plus tard.

On va utiliser cette méthode d'optimisation pour illustrer difficultés du placement et du scheduling sur un problème défini par son graphe de tâche.



On veut paralléliser cett app. Sur quel proc mettre quoi ?

Combien de proc peut on utiliser pour efficacité.

Négligeons temps de communication entre 2 tâches dépendantes

On fait un tableau avec colonne : au plus tôt (le plus tot que l'on peut le lancer), au plus tard (le temps le plus tard que l'on peut le lancer)

Pour certaine tâche, on a de la marge, en ce qui concerne leur fourchette de temps. Pour d'autre, y'a pas. Elles doivent commencer au plus tôt = au plus tard.

On met d'abord ce qui n'ont pas de marge puis ce qui ont en

Tâche	T1	T2	T3	T4	T5	T6	T7	T8	T9
Au Plus tot	0	1	1	3	3	2	7	7	5 (t3),11 (t8),14 (t7)
Au plus tard	0	1	10	3	3 (T2)	11	7	10	14 (temps optimum de au plut tôt)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P1	T1	T2	T2	T4	T4	T4	T7	T7	T7	T7	T7	T7	T7	T7	T9
P2			T3	T5	T5	T5	T5	T6	T6	T6	T8	T8	T8		

2 processeurs nécessaire et suffisent pour garantir temps minimal.  $T_{par} = 15$   $T_{seq} = 27$

Speedup =  $27/15 = 1.8$  (pas 2 car p2 inactif a certains moments)

Efficacité =  $1.8/2 = 0.9$

Ici, on ne peut qu'exploiter qu'au plus 2 proc. Mais on pourrait partitionner encore le problème de façon plus fine (7 devient 7a et 7b exécutable en parallèle).

### 6.3 Equilibrage de charge :

$T_{par}$ , temps du proc. Le plus lent.

Proc inactif : source de perte de performance. On aimerait que tout soit utilisé et finir en meme temps. Dans Julia, certaines zones sont plus difficile à calculer (si **partitionnement statique uniforme**)

Pour équilibrer : **partitionnement statique**, ou **partitionnement dynamique**.

**Statique** : on peut estimer charge à l'avance. Il existe algos qui découpe un problème en morceaux de travail égal.

Ex : partitionnement de graphe (METIS) : trouver un découpage p sous-graphe pour qu'ils aient tous le même nombre de sommet et minimiser nombre d'arc entre sous-graphe (communication).

### LOAD BALANCING STATIQUE :

Nécessite connaissance de temps de calcul et donc charge. Problème irrégulier sont à partitionner de façon à rééquilibrer les charges (donne à chaque proc une charge moyenne équivalente).

- Bisection récursive
- Partitionnement de graphe
- Découpage cyclique ou modulo

### LOAD BALANCING DYNAMIQUE :

Les charges de calcul changent en cours d'exécution ou bien elles sont inconnus.

- Ensemble de Julia, trafic routier, météo, etc..

Il faut :

- Détecter déséquilibre (mesurer charge au cours d'exécution)
- Décider : faut-il rééquilibrer les charges ? est-ce-que ça vaut la peine ? Est-ce que tps de répartition se justifie ?
- Migrer tâche si on pense que rééquilibrage nécessaire.

Faut il avoir un processus de mesure/décision/migration local (entre processeurs voisins) ou global (maître esclave)

Local : scalable, marche pour un nombre aléatoire de processeurs. Plus compliqué à mettre en œuvre et peut ne pas converger si la charge change trop rapidement.

Global : plus précise, plus simple à implémenter, mais non scalable car proc maître est un bottleneck.

Il faut distinguer 2 types d'applications :

- **Problèmes itératifs** (équation de la chaleur) : même calcul répété sur plusieurs étapes, et tout les proc. Doivent se synchro à chaque étape.  
Processus de **loadbalancing** sera activé après chaque itération. Une itération est un bon prédicteur de la charge de la prochaine itération.
- **Problèmes non-itératifs** : chaque donnée du problème n'est calculé qu'une seule fois (ensemble de Julia).

Quand un processeur à fini, il n'a en principe plus rien à faire. **Approche centralisé** : tous le travail encore à faire est stocké dans une zone accessible à tous. Proc qui a fini, demande travail à cette source. NON-scalable.

Version locale : proc qui demande travaille à un autre processeur de groupe. On le choisit au hasard ou systématiquement. En cas de refus (si plus de travail pour lui, ou trop peu), il faut redemander ailleurs. Pb : proc qui travaille doivent s'interrompre pour répondre aux requêtes. Ils enverront du travail au demander : s'ils en ont suffisamment (assez long pour justifier envoi) : Envoi de la moitié du travail restant.



## Chapitre 7 : Modèles de programmation

Il y a plusieurs modèles principaux de programmation parallèle :

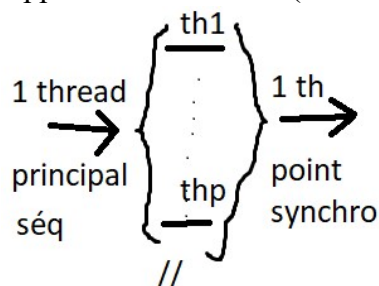
- Echange de message (MPI, PVM, ...)
- Multithreading/sharedMemory (thread PQSIX, Open MP)
- Data parallelism (HPF) : structure de données parallèles (ex : vecteurs), flot d'instruction séquentielle mais instruction qui agit sur des structures de données parallèles.

### **Le modèle par échange de message :**

- Coopération entre processeur se fait par des communications
- Le problème doit être partitionné sur chaque processeur
- Pas d'espace mémoire globale, espace fragmenté sur les processeurs
- Modèle « do it yourself » mais assez bien contrôlé par l'utilisateur
- Il faut changer sa façon de voir le problème, penser « parallèle »
- Modèle clair et on sait ce qui se passe
- Solution scalable (→ millions de cœurs)
- Matériel pas excessivement rendu compliqué par le parallélisme
- Coordination naturelle entre processeur par l'envoi/réception de message (on ne peut pas recevoir un message avant qu'il n'ait été envoyé, pas de donnée partagée)

### **Le modèle à mémoire partagée**

Approche multithread (1 thread par cœur)



Coopération se fait par modification de variables communes.

On a un espace mémoire unique et cohérent

Programmation proche de séquentiel mais trompeusement simple.

Coordination entre processeurs est nécessaire : race condition (proc. En compétition pour ressource et ordre d'arrivée modifie le résultat).

On parle de **cohérence séquentiel** : code parallèle doit donner même résultat que code séq. Qui suit ordre instructions.

**Régions critiques** : 1 seul proc peut passer à la fois

**Barrières de synchronisation** : implantées pour problème

Matériel plus compliqué : il faut assurer cohérence des caches et implémenter primitive de coordination.

Solution NON-SCALABLE avec bcp de proc.

Exemple de parallélisation avec OpenMP

```
#program omp parallel           parallélise facile car donnée indépendant
For i=1 :n                     code identique en séquentiel
    Y [i] = a*x[i] + b
End for
#program omp end parallel
```

Modèle programmation mémoire partagé :

➔ Nécessite coordonner threads pour garantir exécution correct des programmes.

Contrôle d'accès : situation qui arrivent quand plusieurs threads essayent de modifier une même variable partagée.

Coordination peut être garantie par primitives telle que lock/unlock.

Réalise exclusion mutuelle.

Il faut aussi des primitives de coordination pour contrôler la séquence des événements.

**Contrôle de séquence** : il faut empêcher un proc. D'aller trop vite et de commencer à utiliser des variables qui n'ont pas encore été mises à jour par d'autres processeurs.

Ex : ... TODO ... exemple

Utilisation de barrière de synchronisation : on attend que tous les proc. Atteint avant de continuer.

### **Primitives de synchronisation :**

Comment implémenter contrôle d'accès et séquence au niveau du processeur.

On fait grâce aux primitives de bas niveau, disponible au niveau matériel. Basé sur le principe d'opération atomiques ou indivisible.

Ensemble d'instruction qui ne peut être exécutés que par un seul proc. Et ne peut pas être interrompu.

Au niveau matériel, plus simple d'implémenter ces primitives en forçant sérialisation temporelle : FIFO. Primitives classique, qu'on retrouve aussi dans les OS multitâches sont les sémaphore, le test-and-set ou compare and swap.

### **Primitive fetch-and-add**

Du point de vue fonctionnel, fetch\_and\_add est une fonction qui peut s'écrire :

fetch\_and\_add(S(var commune), I(var privée à chaque thread))

atomic {tmp = S, S+=I} return tmp // ici comme si un seul proc a la fois.

Ordre des proc est imprévisible.

On peut proposer implémentation parallèle de fetch-and-add qui permet à plusieurs proc de l'exécuter en même temps sans pour autant violer la fonctionnalité souhaitée.

... TODO ... dessin

Fetch and add, qq soit son implémentation, peut s'utiliser pour ;

- Equilibrage de charge dynamique
- Section critique
- Barrière de synchronisation

Ex :

For i=1 :n

Compute(a[i])

End for

Si on parallélise cette boucle de façon naïve, chaque thread aura des indices  $i$  prédéfinis, et pour certains, ce sera calcul rapide et pour d'autres → déséquilibre de charge.

Version équilibré :

I=1 ; // var commun

Repeat

    J = f\_and\_a(i,1) ; // j indice local à chaque proc

    If(j>n) break

    Compute(a[j])

End repeat

Ainsi, dès qu'un proc est dispo, il va recevoir la prochaine indice à calculer, et si plusieurs proc sont en compétition, f\_and\_a le résout.

### Implémentation d'une sémaphore avec fetch\_and\_add.

S = 1 ; // sémaphore (S=1 SC, S<1 : pas SC)

While fetch\_and\_add(S,-1)<1

    Fetch\_and\_add(S, 1) ;

End while

SC

f-a-a(S,1) // unlock

lock le premier proc a executer f-a-a (celui qui a 1 saute le while

les autres proc tournent dans le while mais

n'arriveront pas à mettre S à 1 car décrémente

juste avant

### Barrière de synchronisation avec fetch and add :

X = 0 ;

...

f-a-a(1)

wait until x == n // n nb de thread a synchronisé

plusieurs barrières avec le même (mauvaise solution) :

x= 0

...

If f-a-a(x,1) == N-1 then x=0

Wait for x ==0

...

x != 0

Solution mauvaise car il e pourrait que quand x passe à zéro, certain proc atteignent 2<sup>ème</sup> barrière avant que tous les proc encore à la première barrière n'et testé que x avait passé à zéro. Maintenant x !=0. Les poc bloqués restent bloqués à la premire barrière et ceux arrivés à la seconde, le seront aussi car f-a-a == N-1 n'aura jamais lieu

BONNE SOLUTION :

X=0

...

Local\_flag = (X<N)

If f-a-a(X,1) = 2N-1 then X=0

Wait until local\_flag != X < N

...

Local\_flag = x < n

...  $\leftarrow$  on passe dès que les N proc ont atteint 2<sup>ème</sup> barrière ce qui fait que x = 0 et local\_flag != x < n

Point clé est que première barrière reste ouverte jusqu'à ce que tous les proc aient atteint la 2<sup>ème</sup> barrière.

Première barrière 'ouvre dès que n proc arrivent car alors x = n et local\_flag != x < n

Historique de PC ?

Différence entre système parallèle et système répartis ?

Expliqués différents types de truc que l'on a vu ? (séquentiel, Pipeline, SIMD, MIMD, SPMD)

Qu'est ce qu'un réseau systolique ?

Expliquer le NUMA et le COMA ?

A quoi sert l'architecture vectoriels ? Quels sont les avantages et les inconvénients ?

Explique les deux types de dataflow ? (datadriven et demand driver)

Topologie statique et dynamique ?

Décrivez les 6 propriétés d'un réseau d'interconnexion ?

Décrivez technique de communication (store and forward, cut through)

Décrivez les primitives de communications (point à point, permutation, broadcast, réduction, gather, échange total, échange total personnalisé)

Réseaux statiques : que pouvez vous me dire ? (anneaux, grille 2D, 3D, Hypercube)

Catégories de réseaux dynamiques ? (connexion par bus, multi étage (omega, fattree), crossbar)

Weak scaling : augmenter nombre itérations

Strong scaling : augmenter le nombre de CPU

Qu'est ce que l'overhead ?

Qu'est-ce-qu'un speedup ? Quel est la différence entre Amdhal et Gustafon ?

Weak scaling ? strong scaling ?

Qu'est ce qu'un speedup superlinéaire ? Pourquoi observe-t-on cela ?

Qu'est ce qu'un système hétérogènes ?

Présentez les systèmes d'équilibrages de charge ? (statique, dynamique)

Il y a deux types d'applications. Que sont-ils ? Quels sont les contraintes ?

Qu'est ce que l'approche centralisé pour l'équilibrage de charge dynamique ? Différence entre ça et la version locale ?

Citez moi quelques modèles principaux de programmation parallèle ?

Donner les caractéristiques du modèle par échange de message ?

Qu'est ce que le contrôle de séquence ?