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 ⇔ échange d'information entre les processerus.

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 ⇔mémoire (prog + donnée)
Exécution séquentielle.

- ⇨ Convergence des modèles de programmation, matériel et conception de méthode de résolution. => 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 => RIP.

Von Neumann bottleneck (goulet d'étranglement, « memory-call ») : la connexxion 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.

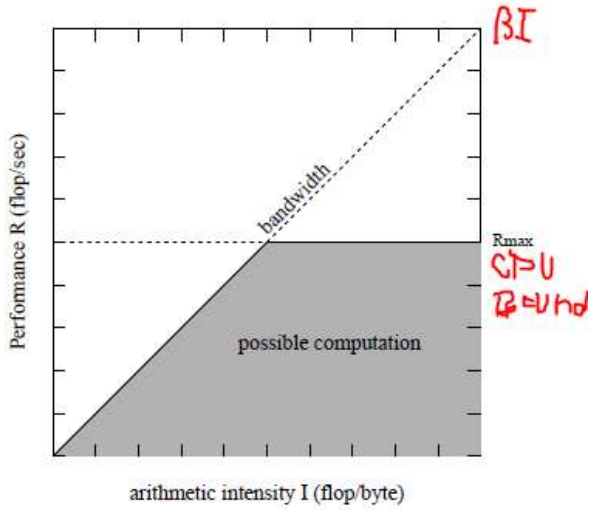
Maintenant, environ Petaflops 10^15 flops (150Pflops avec 10 Millions de cœurs = 15MWatts)

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 = bêta * 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*f avec v qui est une fonction de la fréquence.
        - Pendant longtemps, f et v était 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 structurés 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*gamma/(n*gamma^2) = 1/gamma

Pour maximiser R/p, il faut que gamma → 0.

Mais pour avoir une machine qui calcule, il faut prendre n→infini.

Augmenter performance : modifier architecture des processus

Solutions :

- Parallélisme solution architectural au défi des performances
- Mémoire cache
- Processeurs vectoriels
- …
- Faire calculs la 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

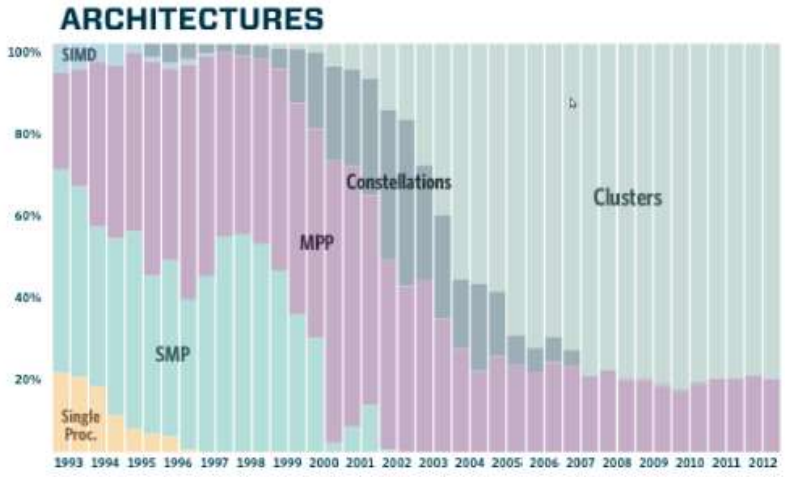
1995 : cluster de PC (beowulf)

2000: GRUD, Cloud

2010 : GPU → Green computing exascale → 10^18 flops/s

Maintenant, parallélisme partout, à l'intérieur de chaque processeurs, entre processueurs, 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 problèm**e.

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 délibéré	Hétérogénéité des ressources	Hypothèse sur le système	Connaissance mutuelle	Fiabilité	Problème de securité
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 : coupes des tranches de pain
I2 : beurrer les tranches
I3 : mettre confiture
I4 : placer tartines sur un plat

1er stratégie (**SEQUENTIELLE ou Von Neumann**)
Mme Dupont fait tout de manière séquentielle (un puis l'autre)

Tseq = N * 4*I
I = temps d'une opérations

2ème 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

Tpipeline = 4T + (N-1)*T (il reste N-1 qui sont prête une à une chaque étape tn vaut N*t su N est grand)

Tseq/Tpipeline = 4*t*n/t*n = 4 // 4 fois plus vite avec 4 travailleur

Pour que ça fonctionne, il faut que I1, I1, 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 5ème personne pour accélerer encore la fabrication nb-personnes = nb-tâches

Pas de tolérances aux pannes

3ème stratégie (**SIMD : Single Instruction flow, Multiple data flow**)

Faire plusieurs tartines en parallèle.

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

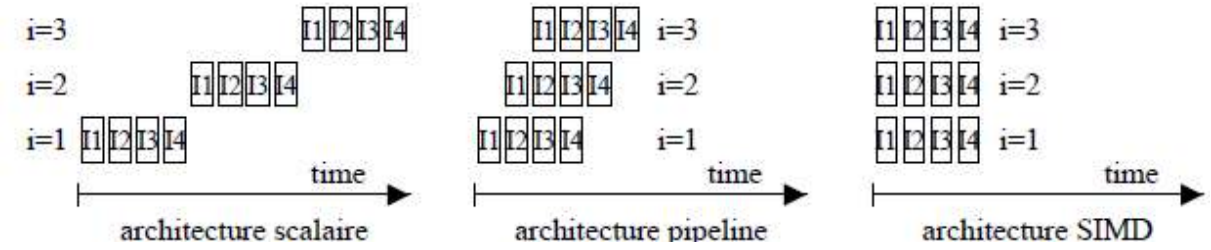
Pour estimer Tsimd, on va supposer qu'on a p travailleurs.

En, 4t, on produit p tartines.

Pour faire n tartines, il faut (N/p)*4*t étaoes de temps :
Tsimd = (4*t*N)/p
Ici p, peut varier de nombreux travailleur.

Potentiel de parallélisme, beaucoup plus grand

Cette technique fonctionnera bien si chaq travailleur a ses propres données (sa hache de pain, plaque de beurre, …) ➔ principe de **mémoire distribués** pour les processeurs.



$$T_{seq} = 4N\tau \qquad T_{pipeline} = N\tau \qquad T_{simd} = \frac{4N\tau}{p}$$

4ème 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 PARTAGEE**, 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 probèmes de coordination.

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

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

<u>Chapitre 2</u> : **Architectures à haute performance**

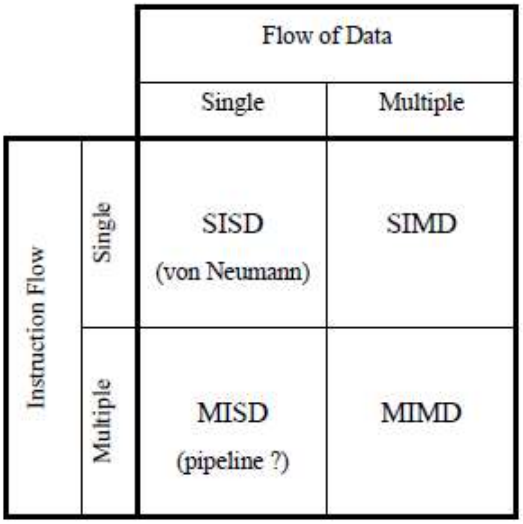
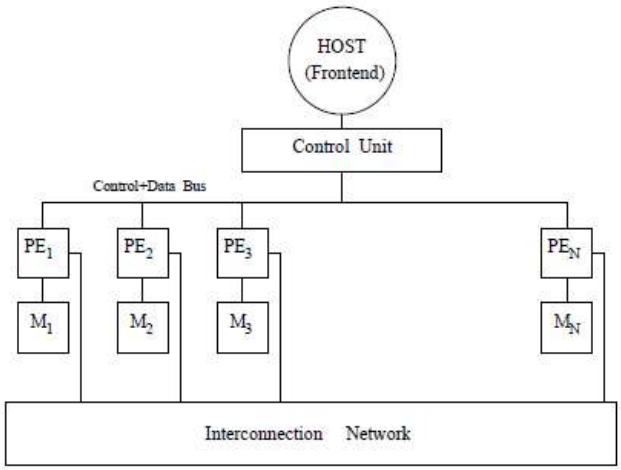


FIGURE 2.1 – *Diagramme représentant la classification de Flynn*

Architecture de base SIMD (single instruction, multiple data)

2.1 SIMD



Exécution du progralle

Broadcasting

Processing Element : Proc.
Potentiellement plus simple
Mémoire distribuée (mais pas forcément)

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** (attende 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.

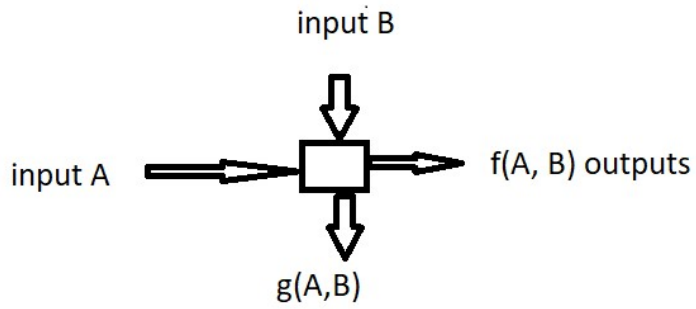
Avantages : simplicité/fiabilité et performance
Désavantage : peu efficace pour des problèmes irréguliers

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

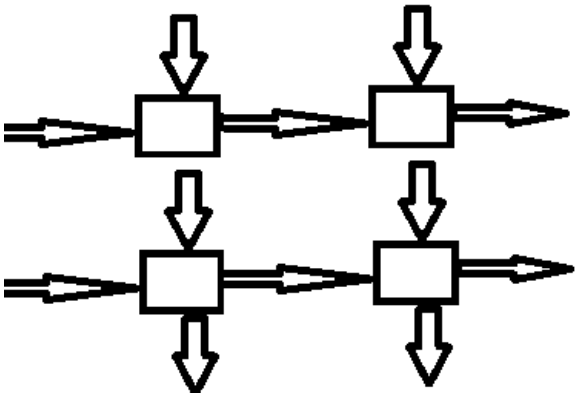
Maintenant, le SIMD, apparait seulement dans des composantes décotées et spécialisés comme les GPU et réseaux systoliques.

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

Idée :



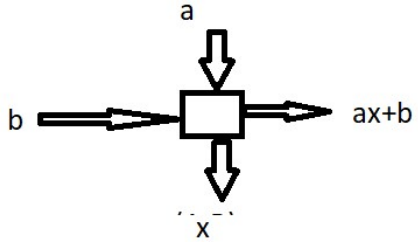
Un réseau systolique est une architecture SIMD de type particulier.
Un réseau systolique est un réseau interconnecté de ces cellules simples (ces cellules travaillent en parallèle) :



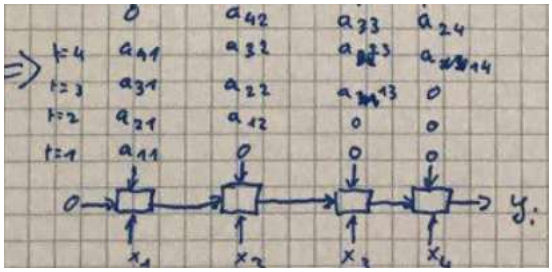
On peut réaliser des multiples de matrices avec cette architecture.

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 € M4*4, y, x € M4



A la 4ème étape, on a obtenu y1, puis à chaque cycle, on obtient un autre yi.

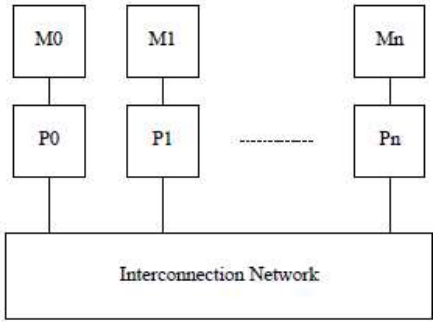


Tsystatique(n) = n*T + (n-1)*T = (2n-1)*T, T = temps d'un cycle

Tseq(n) = n*(n + (n-1))*T = n*(2n-1)*T => n fois plus lent

2.2 Architecture MIMD :

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

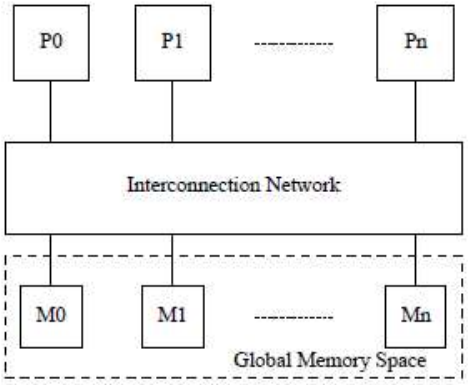


processeur + mémoire (pc classique)

permet des échanges de messages.

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.

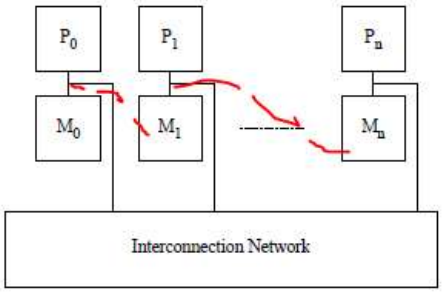
Une difficulté importante de l'architecture à mémoire parttagée est que les processeurs ont des mémoires caches. Une variable peut donc être copiée à plusieurs endroits et potentiellement modifiés de façon inconstitents.

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 (procesus atomique).
Les processeurs sont continuellement en communication pour assurer cette cohérence → solution peu scalable

Architecture à haute performance :

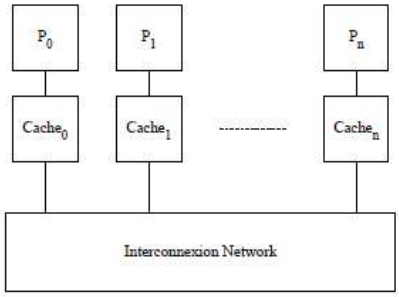
SIMD,
MIMD (mémoire distribuées => scalable => 10^6 cœurs // mémoire partagés => peu scalable => < 100 cœurs)

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 processerues, 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)

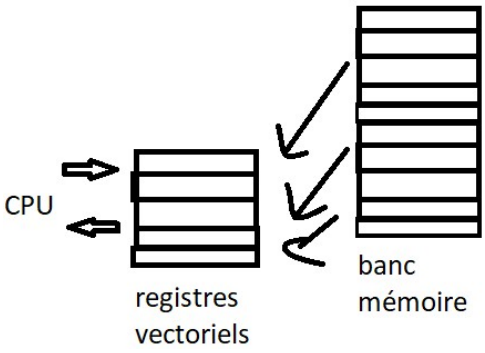
C’est une machine MIMD à mémoire partagée avec des processus SIMD aussi à mémoire partagée.

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).



Il faut que les données soit correctement organisé

Gain ?

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

Tscalar = N*Tfetch + N*Tcpu + N*Tstore
Tvector = Tfectch + N*Tcpu + Tstore

Tscalar/Tvector = N*(Tfetch + Tstore)/(N*Tcpu) = 2*Tfetch/Tcpu >> 1

Tcpu << Tfetch environ Tstore
N*Tcpu >> Tfetch car N grand

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

9T + (N-1)*T = Tscalar/q

9T temps de la 1ère donnée

(N-1)T temps des N-1 dp,,ées restantes * T le temps cycle d’une machine

D = A*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.

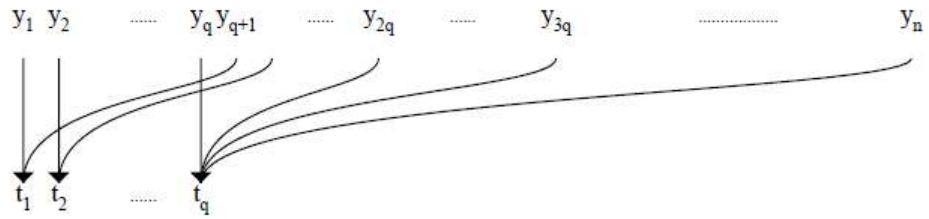
Dépendance de donnée : produit matriciel de matrice vecteur

Autre ex de dep de données :
S=0 ;
For i=1..n
S = S + y(i) ; // s = dépendance de donéne
End for

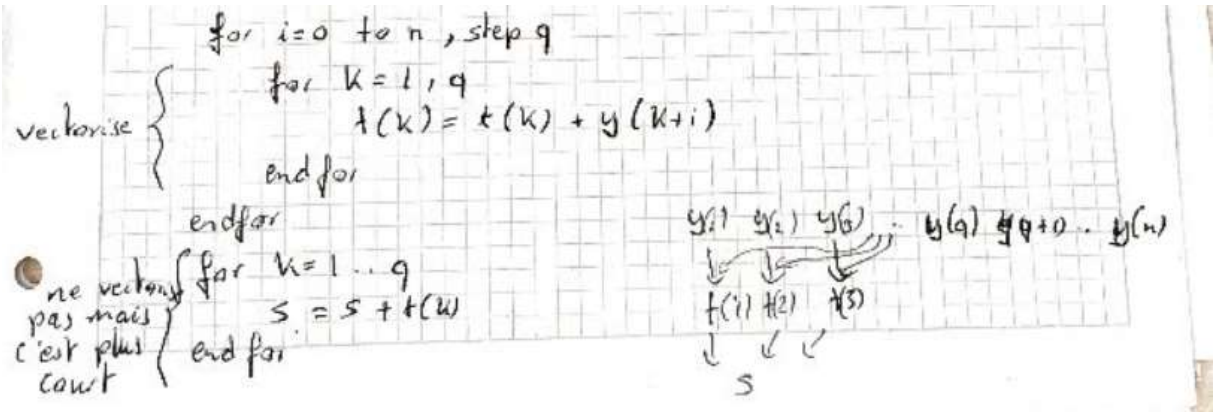
```
s=0

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



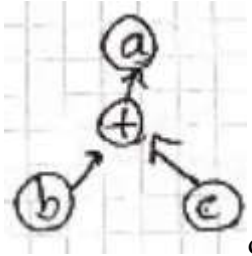
En log2(n), on arrive a dérouler la boucle.

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**)

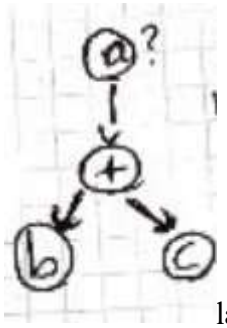
2 possibilités : a = b+c

Data driver

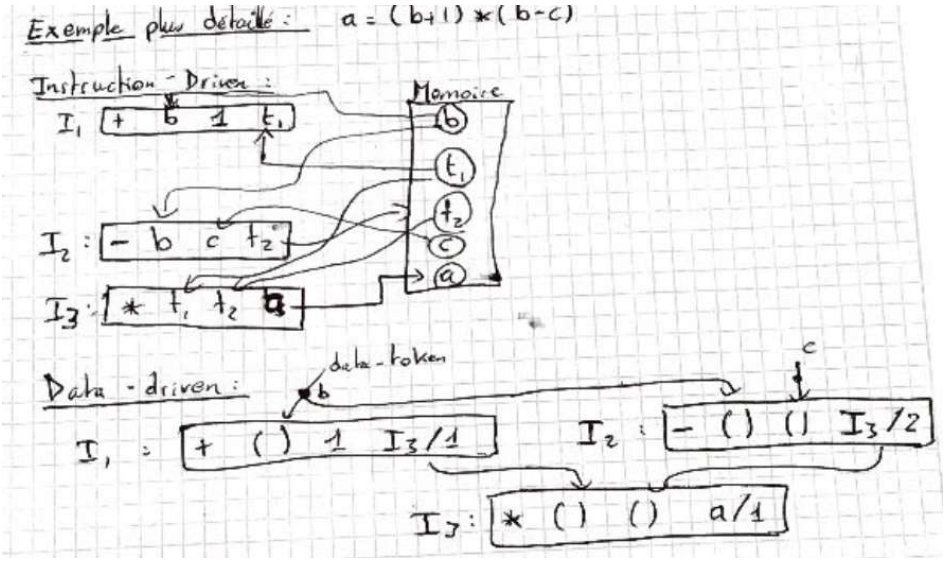


disponibilité de b et c, va activer opération qui donne c.

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, I1 et I2 sont exécutes 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).

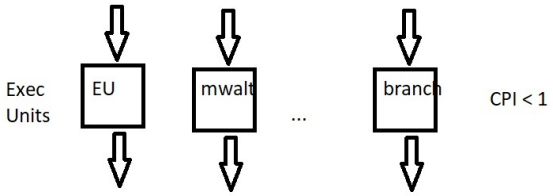
Il faut faire du **pipelining** : utiliser une pipeline d'exécution

➔ 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 ward

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'instrcution est executé dès que les données correpondantes 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

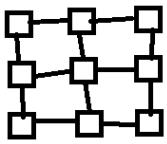
Une machine parallèle, ce sont des processeurs qui coopèrent et ils le font en **communiquant**.

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 :

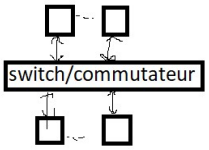


Un carré c'est un proc+routeur

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 bisectionelle : 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 sépares 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