Notions sur l'architecture des calculateurs

Formation d'Ingénieurs de l'Institut Galiléee MACS 2

Philippe d'Anfray

Philippe.d-Anfray@cea.fr

CEA Délégation Calcul Intensif

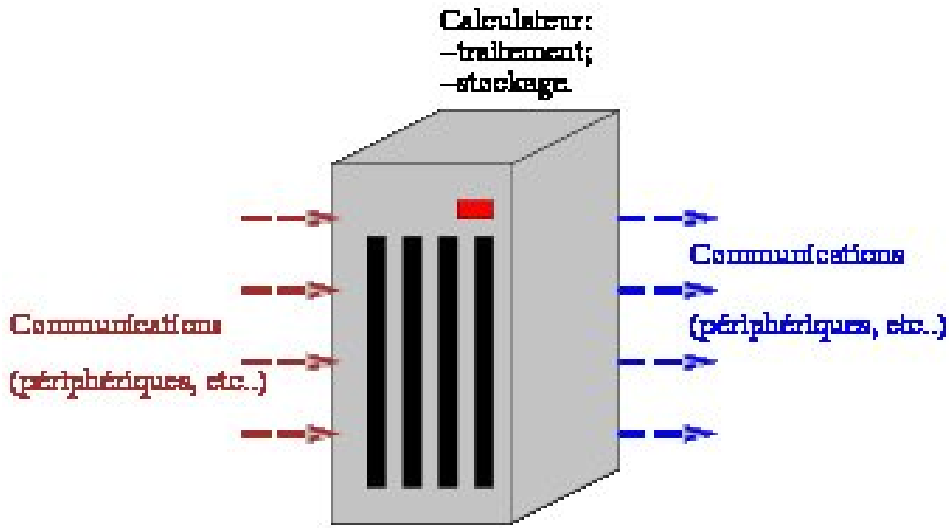
2009-2010

Plan de l'exposé :

- Vue globale
- L'unité centrale
- La mémoire
- Caractérisation
- Evolution des technologies
  - Pipe-line instruction
  - Le reste (...)
- Conclusions

Vue globale (1)

Un calculateur est une entité qui interagit, communique avec le monde extérieur pour traiter des informations.



Le traitement de l'information consistant à faire agir des instructions sur des données.

Vue globale (2)

Un calculateur est un système complexe qui comprend plusieurs millions de composants.

Les fonctionnalités présentes dans un calculateur sont :

- traiter l'information (e.g. des calculs) ;
- stocker l'information ;
- transférer de l'information ;
- et. . . contrôler que tout se passe bien.

Vue globale (3)

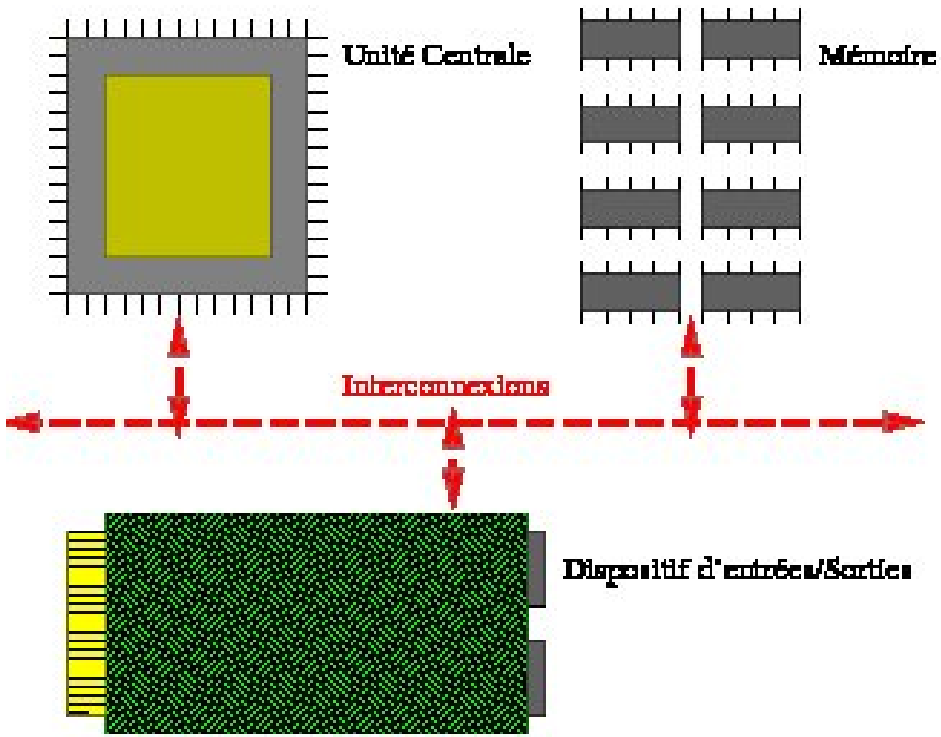
Un calculateur est un système hiérarchique.

On peut décomposer la machine en quatre grandes parties :

- l'**unité centrale** (*processeur*)
  contrôle les opérations réalisées par la machine, effectue le traitement ;
- la **mémoire centrale** (stockage de l'information) ;
- les inter-connexions (*bus*) transfert de l'information entre parties du calculateur ;
- les dispositifs entrées/sorties (`I/O`)
  transfert de l'information entre le calculateur de le monde extérieur.

Vue globale (4)

Premier zoom, un calculateur, les grandes parties :



Nous allons détailler les parties les plus complexes :
– l'unité centrale – la mémoire.

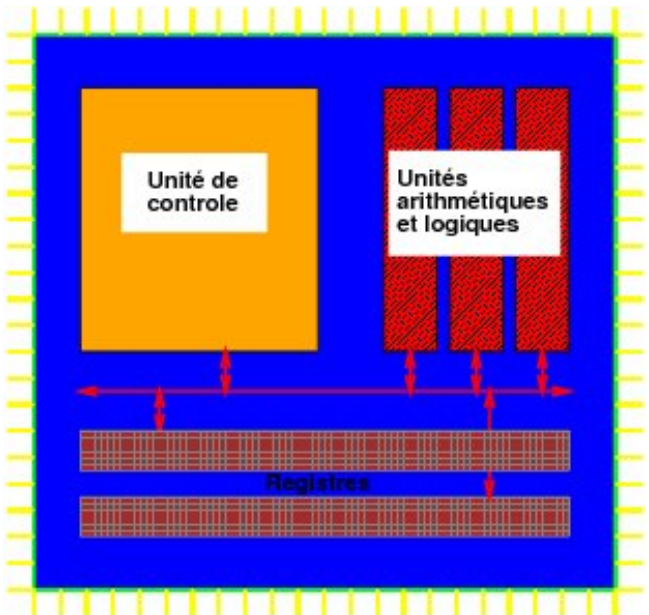
L'unité centrale (1)

L'unité centrale ou processeur comprend :

- l'unité de contrôle
  lit et décode les instructions à exécuter ;
- l'unité arithmétique et logique
  effectue les calculs ;
- les registres
  stockage temporaire -et limité- d'information ;
- l'inter-connexion.

L'unité centrale (2)

Un calculateur, zoom sur l'unité centrale, les grandes parties :



Ce qui nous intéressera :
–Unité(s) de calcul –(différents types de) registres.

La mémoire (1)

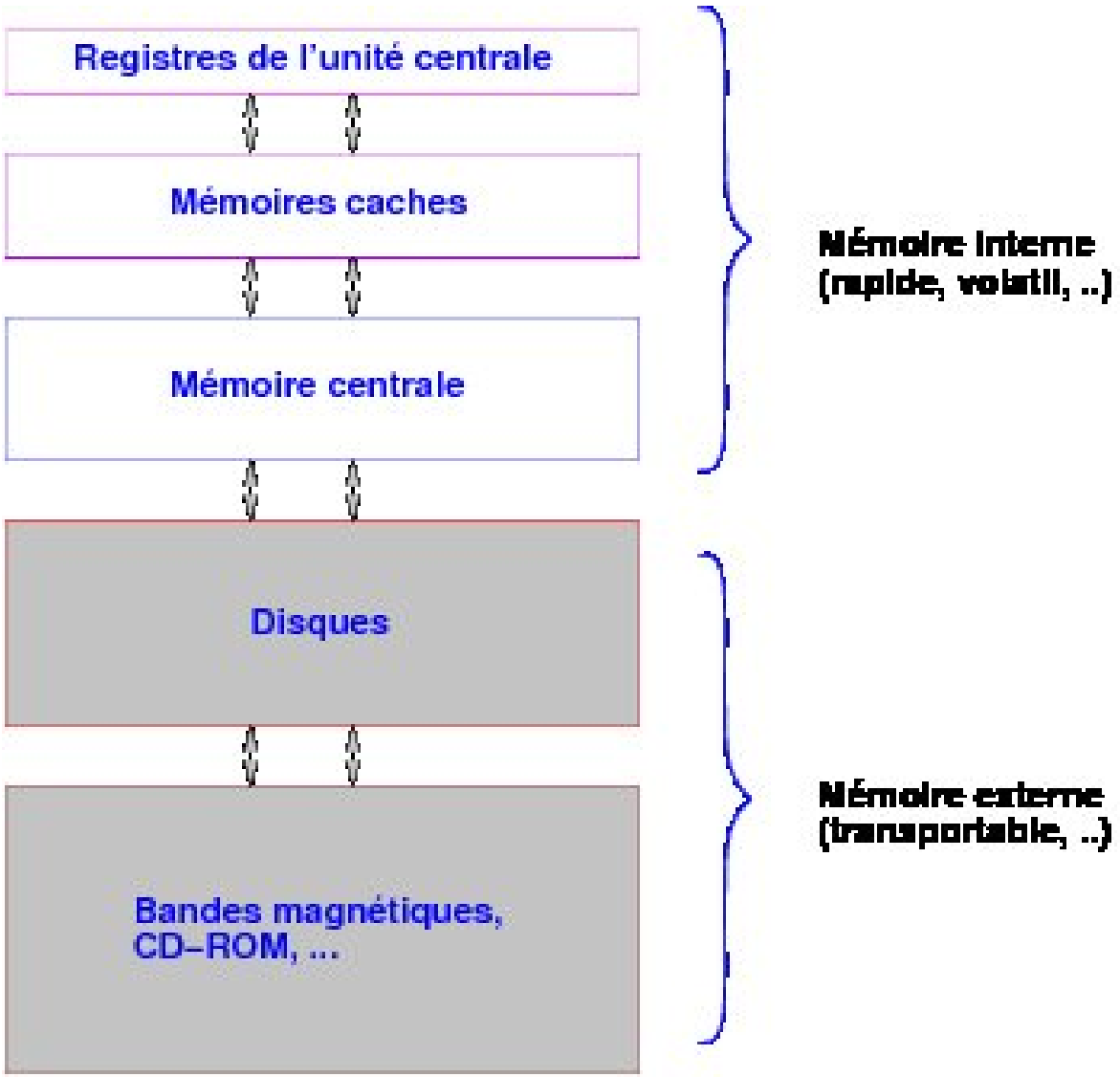
Le constructeur d'un calculateur doit répondre à trois questions :

1. Quelle taille mémoire *(en octets)*
2. Quel débit *(i.e. le temps d'accès)*
3. Et surtout. . . *Quel prix ? ?*

Le resultat est un compromis entre la taille et le temps d'accès.

En fait, on trouve sur les calculateurs une **hiérarchie de mémoires**.

La mémoire (2)


Registres de l'unité centrale
Mémoires caches
Mémoire centrale
Disques
Bandes magnétiques,
CD–ROM, ...
Mémoire interne
(rapide, volatil, ..)
Mémoire externe
(transportable, ..)

La mémoire (3)

Dans cette hiérarchie,

- si *coût/octet* décroit ;
- la capacité augmente mais...
- le temps d'accès augmente lui-aussi.

Problème : gérer les transferts entre niveaux pour diminuer le coût moyen des accès à la mémoire.

La mémoire (4)

Ordres de grandeur typiques :

	Registres	Cache
taille	dizaine de Ko	centaines de Mo
t accès	ns	dizaine de ns
débit	dizaine de Go/s	quelques Go/s
	Mém centrale	**Disques**
taille	dizaines de Go	jusqu'à quelques To
t accès	qqs dizaines de ns	qqs **ms** (10^6 ns)
débit	qqss centaines de Mo/s	centaine de Mo/s

La mémoire représente une part importante du coût d'une station de travail

Caractérisation (1)

Pour la simulation numérique, on s'intéresse surtout

- à la puissance de traîtement.
  Liée au temps de cycle, ou à la fréquence.
- à la taille de la mémoire centrale

Le nombre d'instructions exécutées à chaque temps de cycle est directement lié à l'organisation de la machine. (nombre de processeurs, d'unités de calcul, jeu d'instructions etc...).

Caractérisation (2)

Il y a plusieurs unités de mesures de la puissance notamment :

Le MIPS millions d'instructions/secondes.
Dépend du jeu d'instruction de la machine, peut varier selon les programmes. Pire, peut varier en fonction inverse de la performance constatée.
Pour le calcul scientifique, le MIPS est une unité inadaptée.

Le MFLOPS million d'ops flottantes/seconde (GFLOPS, TFLOPS, PFLOPS).
On ne s'intéresse qu'aux opérations arithmétiques. La bonne mesure pour la simulation numérique.

il y en a d'autres, e.g. GLOPS opérations logiques, etc. . .

Technologie (1)

La "montée en puissance" des composants de base des calculateurs se heurte à des **limites technologiques**.

- le temps de commutation (nécessaire à une porte logique pour changer d'état) ;
- la dissipation de chaleur (quelques mW par composant) ;
- enfin, les connexions sont **plus lentes** que les temps de commutation : la lumière ne parcourt que 30cm en 1ns. . . !.

Autres soucis, la consommation d'éléctricité, etc...

Technologie (2)

Entre les années 50 et 80 la technologie permettait de gagner pour le temps de cycle un facteur 10 tous les 5 ans.

Depuis une vingtaine d'années, ces limites se font sentir.

le meilleur temps de cycle qui était de 12.5 ns en 1981 (CRAY-1) n'a pu être ramené Ìa fin des années 90 qu' à 4 ns (1998 NEC SX 5) ou 2.66 ns (1999 IBM RS 6000)

Certains processeurs (vectoriels) ont néanmoins des puissances de l'ordre de 10 GFLOPS (NEC, Cray, . . .).

Technologie (3)

Comment faire mieux ? modifications architecturales.

- Créer des Pipe-lines pour le recouvrement “partiel” des traitements :
  - pipe-line instruction,
  - pipe-line exécution.
- Faire fonctionner en parallèle les composants de la machine.
- Multiplier les composants de base :
  - sur chaque processeur, les unités fonctionnelles (additionneur, multiplieur, etc. . .) ;
  - et les autres : canaux d’entrée/sorties etc. . .

Technologie (4)

Comment faire mieux ? modifications architecturales, et aussi :

- Multiplier les processeurs ;
- Découper la mémoire en bancs adressables simultanément pour "alimenter" ces unités fonctionnelles et les processeurs ;
- Enfin "distribuer" la mémoire centrale sur les processeurs.

Toutes ces modifications induisent certaines formes de parallélisme accessible (ou-non) à l'utilisateur.

Technologie (5)

Fonctionnement concurrent, parallèle

On qualifie de *concurrent* le fonctionnement simultané des différentes parties de la machine
unité centrale, dispositifs d'entrées/sorties, Idots. Cela existe sur tous les calculateurs même le plus simple des *PC*.

On obtient du parallélisme :

- Entre travaux,
- Entre parties d'un même travail (Entrées/Sorties, calculs,. . .)

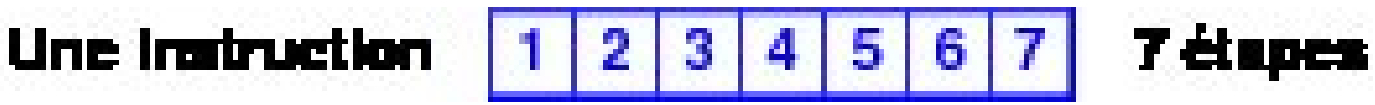
Ce parallélisme n'est pas, en général, accessible à l'utilisateur.

Pipe-line instruction (1)

Décomposer le traitement d'une instruction par l'unité centrale en étapes élémentaires effectuées par des dispositifs différents.

1. Lecture de l'instruction (transfert mémoire/registre)
2. Incrémentation du compteur d'instruction (compteur ordinal)
3. Décodage de l'instruction,
4. Préparation des opérandes,
5. Exécution de l'instruction,
6. Rangement des résultats,
7. Test du signal d'interruption.

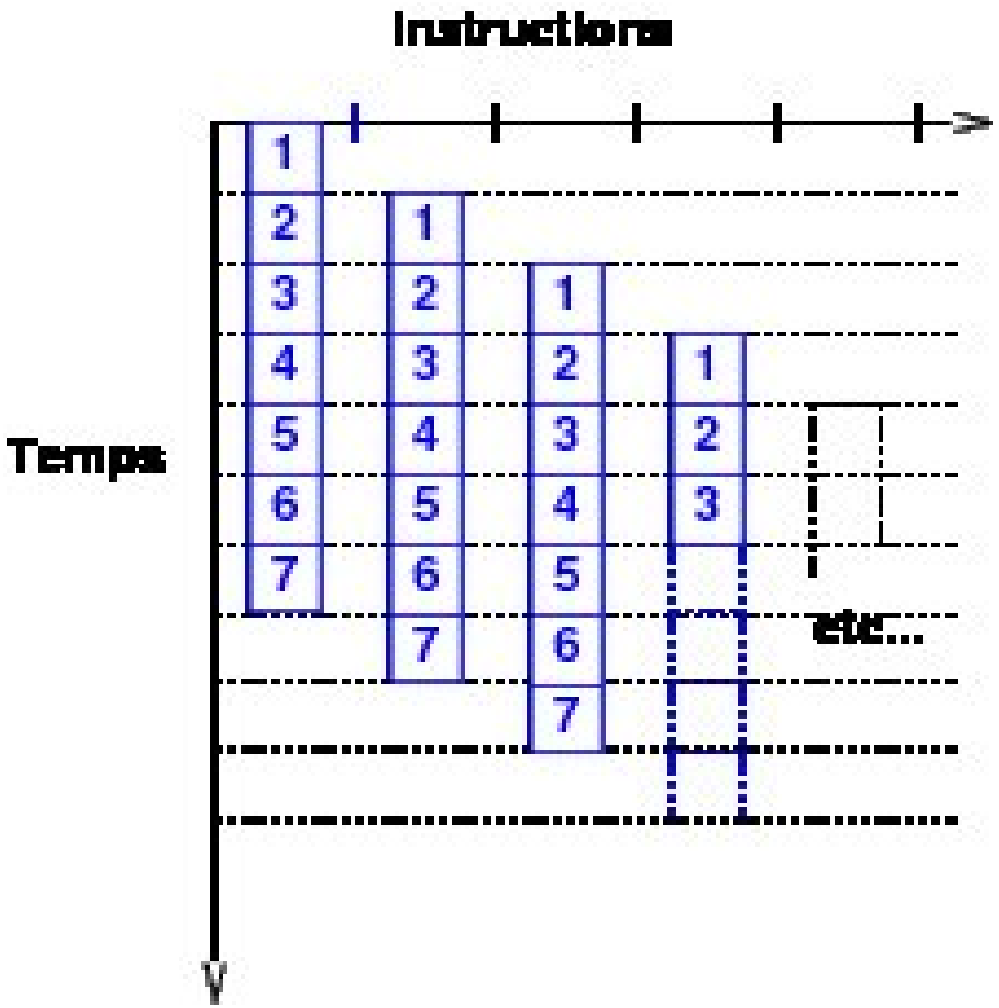
Pipe-line instruction (2)



- Les instructions sont traitées selon le principe de la "chaîne de montage"
- Ce parallélisme entre instructions n'est pas, en général, accessible à l'utilisateur.

Le pipe-line instruction existait dès les années 60 : CDC6600 (1964) ou IBM 360 (1967).

Pipe-line instruction (3)



Les autres (1)

Les autre formes de parallélisme sont exploitables par l'utilisateur, et ont un impact sur la programmation :

Plusieurs niveaux de parallélisme :

- Dans un même programme, entre différentes parties du code,
- A l'intérieur des boucles entre "séries d'instructions".
- Entre opérations élémentaires.

La recherche de performances nécessite d'exploiter ces formes de parallélisme. (modèles de programmations, outils, langages)

Plan de l'exposé :

- Classer
- SISD
- SIMD
- MIMD
- Performances
- Remarques

Architectures Parallèles (1)

Celles qui ont un impact pour le programmeur.

On adopte la classification dite de FLYNN (1972)

- qui est complétement dépassée ;
- largement inadaptée ;
- et . . . très utilisée.

soit : SISD, SIMD, MIMD pour

Single ou **Multiple Instruction** flow et **Single** ou **Multiple Data** flow.

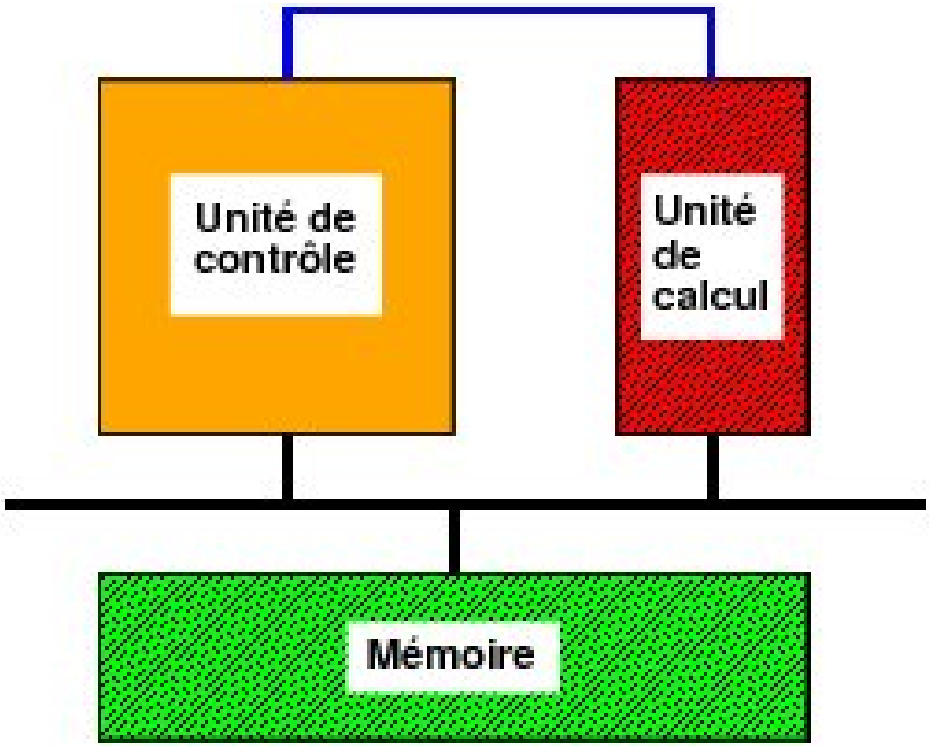
(ne tient pas compte de la nature des processeurs, ni de l'organisation de la mémoire)

SISD (1)

La machine séquentielle classique :

- à **un** instant donné,
- la machine exécute **une** instruction,
- portant sur **une** donnée.

SISD (2)



SISD (3)

L'instruction :

. C=A+B ;

sera décomposée en :

- lire A,
- lire B,
- effectuer A+B,
- ranger C.

Il faudra attendre que cette instruction soit terminée pour en commencer une deuxième.

Il n'y a plus vraiment de processeur SISD, ils font tous du recouvrement partiel entre instructions.

SIMD (1)

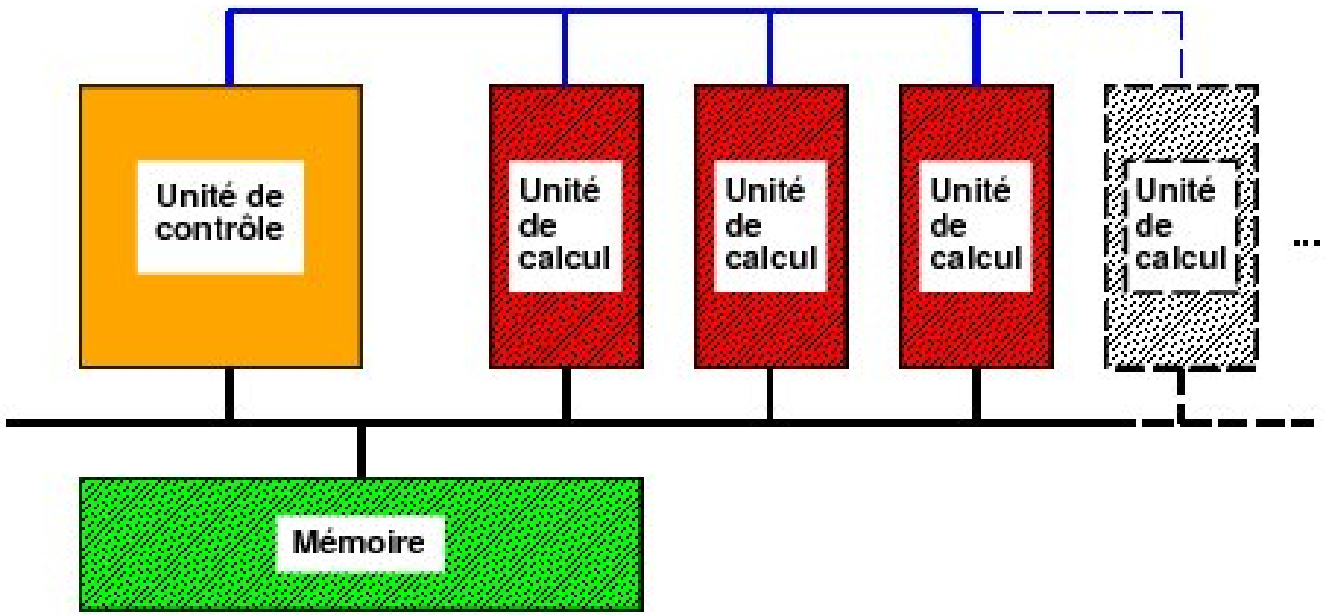
Ce sont des machines conçues pour faire un grand nombre d'opérations.

On duplique les unités de calcul pour effectuer simultanément les mêmes opérations (élémentaires) sur des données distinctes.

Application typique (à l'époque) : le traitement d'images. Le premier : ILLIAC IV, années 60, USA, Connection Machine (CM2) jusqu'à 64k processseurs , "array processors"

Architecture abandonnée mais techniques logicielles "redécouvertes" avec les GPU (même si on fait mieux que des opérations élémentaires)

SIMD (2)



L'UC délégue aux unités de calcul des opérations (ou des séquences d'opérations) identiques

SIMD (3)

En pratique, une série d'opérations identiques est une **boucle** :

```
.     Pour i=1, n
.       C(i)=A(i)+B(i) ;
```

On peut espérer que :
tps d'exécution= tps sur une unité / nombre d'unités

Cette boucle utilise des données rangées dans des tableaux, on ne doit pas avoir de récursivité :


```
.     Pour i=2, n
.       X(i)=2.0*X(i-1) ;
```

La boucle ci-dessus présente une **dépendance de donnée**.

SIMD (4)

Quelques remarques :

- accélération non uniforme (taille de la boucle/nombre d'unités)
- problèmes d'accès mémoire (débit)
- temps d'amorçage croit avec le parallélisme
- pas de recursivité

Bancs Entrelacés (1)

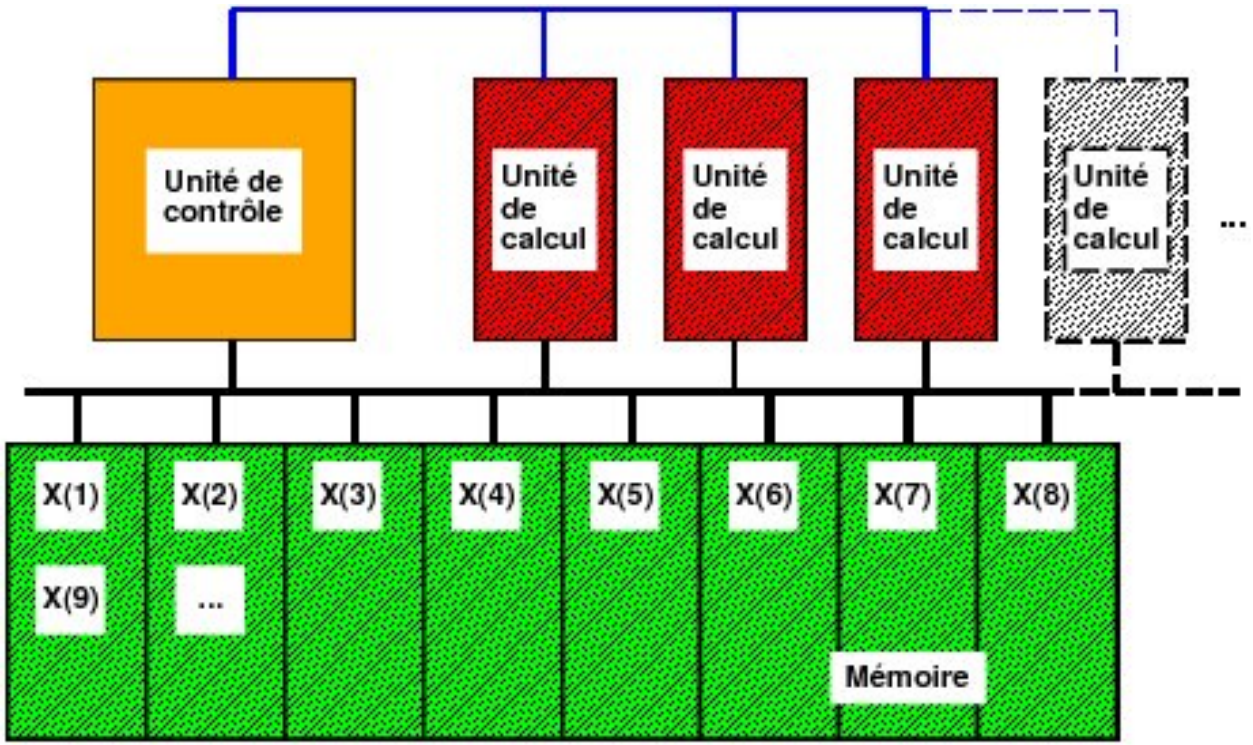
Une réalisation pratique d'une telle machine nécessite de découper la mémoire en *bancs* accessibles simultanément.

Si les unités de calcul peuvent accéder à des bancs différents.

```
.      Pour i=2, n
.         X(i)=2.0*y(i) ;
```

toutes les données sont accédées très rapidement.

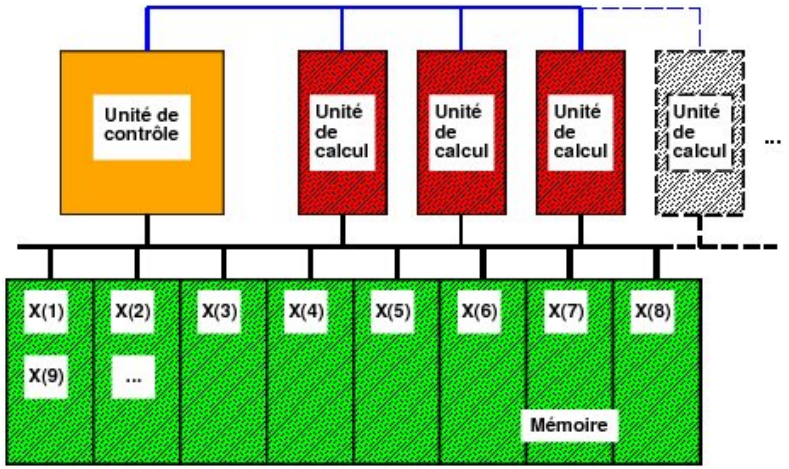
Bancs Entrelacés (2)



Cette disposition est maintenant classique.

Bancs Entrelacés (3)

Les bancs entrelacés n'empêchent pas les conflits d'accès :



. Pour i=2, n pas 8
. X(i)=2.0*Y(i) ;

ici toutes les données sont sur le même banc.

re-SIMD (1)

Le réseau d'interconnexion entre les bancs mémoire et les unités de calcul est très difficile à gérer ! ! !

Il n'existe plus de machines de ce type ! ! ! (dernière BSP 1977)

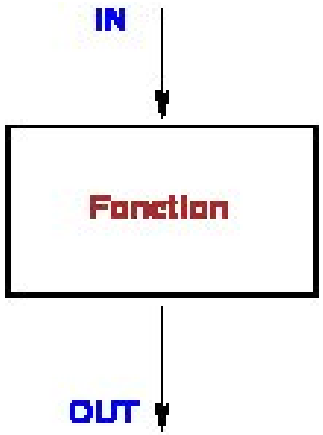
Les machines SIMD qui ont suivi n'ont plus de mémoire commune mais de petites mémoires locales attachées aux unités de calcul.

Ces machines ne sont pas révélées adaptées au calcul scientifique mais les techniques sont redécouvertes dans le cadre des GPU.

Machines spécialisées : traitement d'images, I.A., calcul symbolique...

Pipe-line exécution (1)

Pour calculer plus rapidement des fonction de base :

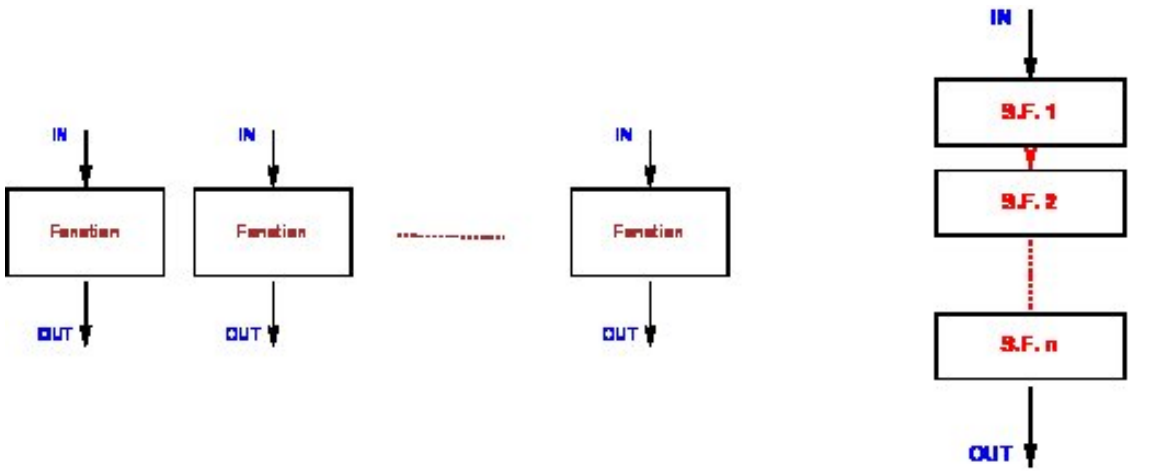


L'idée est de faire les calculs en même temps, deux possibilités :

- traiter la même fonction sur des données différentes en dupliquant les unités fonctionelles, c'est l'approche SIMD (déjà vue) ;
- décomposer la fonction en sous-fonctions traitant simultanément une partie du problème. C'est l'idée déjà vue de la chaîne de montage. Ici **pipe-line exécution**.

Pipe-line exécution (2)

C'est ce que l'on trouve dans les machines **super-scalaires** et **vectorielles**.



SIMD — Pipe-line exécution

Le nombre de sous-fonctions dans un pipe-line exécution est appelé le nombre d'**étages** du pipe-line.

Pipe-line exécution (3)

Il faut présenter à la machine des suites d'opérations identiques portant sur des données *régulières*.

En pratique, ce seront des boucles portant sur des éléments de tableaux :

```
pour i=1 jusqua n  a(i) =b(i) + c(i);
```

L'accès à la mémoire constitue un goulot d'étranglement qui limite la performance maximale possible mais

- si en SIMD les données étaient accédées par paquets.
- en pipe-line exécution il suffit d'assurer un flot continu de données.

Pipe-line exécution (4)

Flot continu... accès réguliers : c'est à dire la régularité des accès à la mémoire

Mais cela pose aussi de multiples problèmes. . . par exemple les *dépendances de données* :

```
pour i=2 jusqua n  x(i) = x(i-1) + C(i);
```

La boucle ci-dessus présente une forme de récursivité qui interdit l' exécution parallèle du corps de boucle pour les valeurs successives de `i`.

Super scalaire (1)

- On appelle super-scalaire un calculateur possèdant des unités fonctionnelles pipelinées , mais une hiérarchie mémoire “classique”.
- On appelle vectoriel un calculateur possèdant des unités fonctionnelles pipelinées et des registres spéciaux (vectoriels).

Une grande partie de ce que l’on dira sur les calculateurs vectoriels s’appliquera aux calculateurs super-scalaires.

Vectoriel (1)

Un calculateur vectoriel possède des unités fonctionnelles pipe-linées.

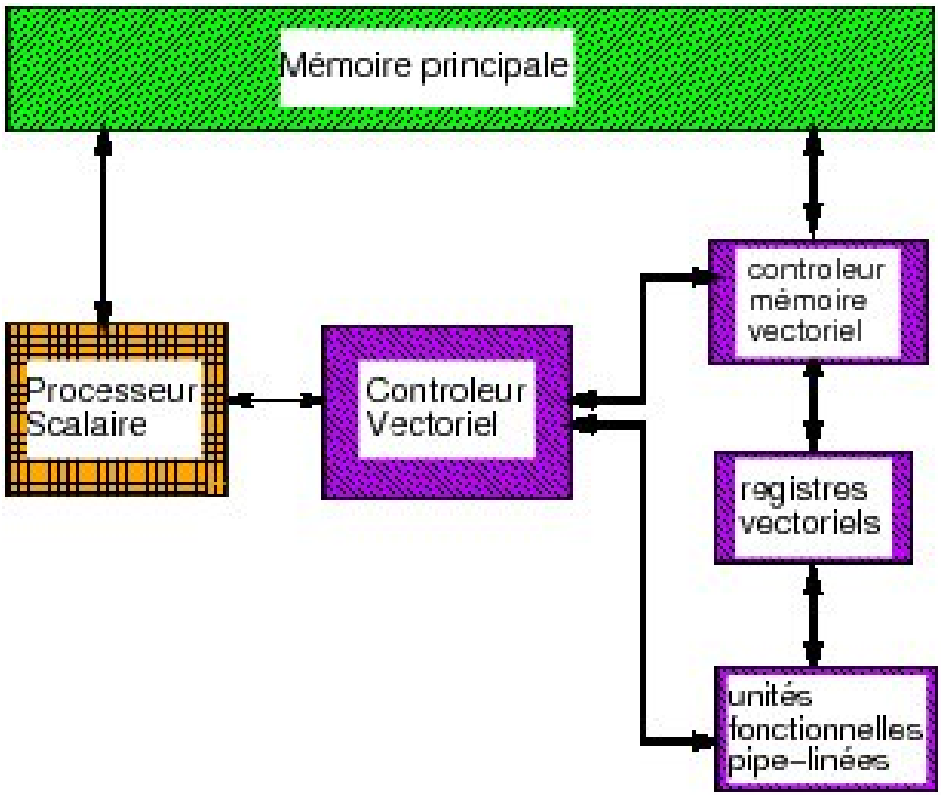
Pour “alimenter” ces unités, la mémoire doit fournir un flux continu de données.

On fait appel pour cela à un nouveau stockage intermédiaire : les **registres vectoriels**.

Pour les utiliser, la machine possèdera aussi un jeu d’instructions spéciales dites instructions vectorielles.

Vectoriel (2)

L'organisation (très générale) d'un calculateur vectoriel est la suivante :



Vectoriel (3)

Mémoire principale :

lors du traitement des instructions vectorielles elle doit satisfaire à la demande. On a recours à l'entrelaçage pour augmenter le débit.

Processeur scalaire :

traite tout ce qui n'est pas vectoriel (opérations scalaires, I/O, contrôle du programme...).

Controleur vectoriel :

décode les instructions vectorielles, calcule les adresse des opérandes, alloue le controleur mémoire vectoriel et les unités pipelinées.

Controleur mémoire vectoriel :

gère les requêtes d'accès mémoire, controle l'alimentation des pipe-lines.

Registres vectoriels :

les registres vectoriels sont utilisés comme *mémoire tampon* pour alimenter les pipelines.

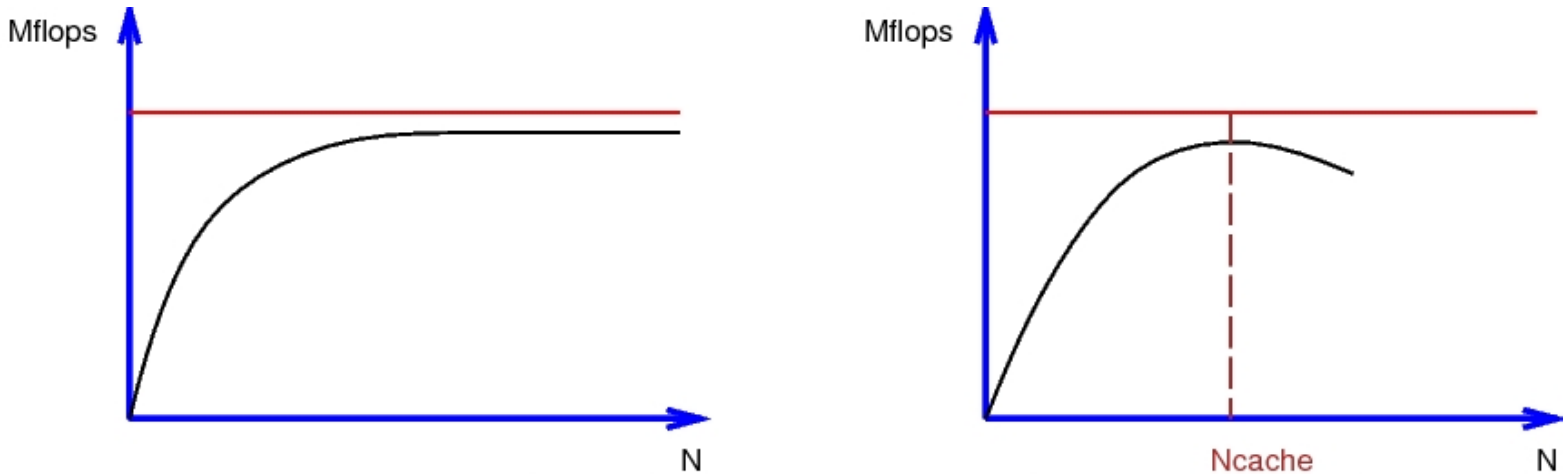
Vectoriel *vs* Super Scalaire (1)

Sur un calculateur vectoriel, la performance croit avec la taille de la boucle (quantité de données accédées) jusqu'à une asymptote : la performance crête de la machine pour la boucle envisagée.

Sur un calculateur superscalaire il en est de même.. **tant que les données traitées tiennent dans le cache**. Si il faut *flusher* le cache (le recopier vers la mémoire centrale), la performance s'écroule !.

Vectoriel *vs* Super Scalaire (2)

En vectoriel, le flux continu de données permet de conserver la puissance maxi sur une section de code quelque soit la taille N des données accédées.



Puissance mesurée/taille de la boucle, cas vectoriel vs cas superscalaire

Avec un cache, une fois la taille Ncache atteinte la puissance diminue, les architectures multicache permettent d'améliorer les choses.

MIMD (1)

La machine possède plusieurs processeurs indépendants.

Processeur = unité de contrôle **et** unité(s) de calcul.

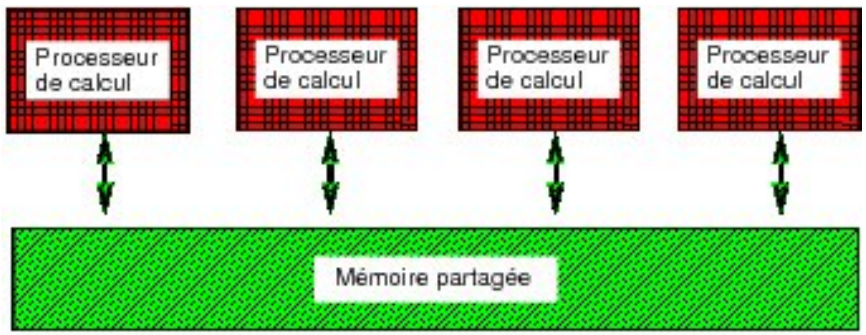
Chaque processeur traite son flot d'instructions propre (*processus*) sur ses données propres.

Souvent un calculateur *hôte* s'occupe des I/O, du lancement des processus ...

La mémoire peut être partagée ou distribuée.

MIMD (2)

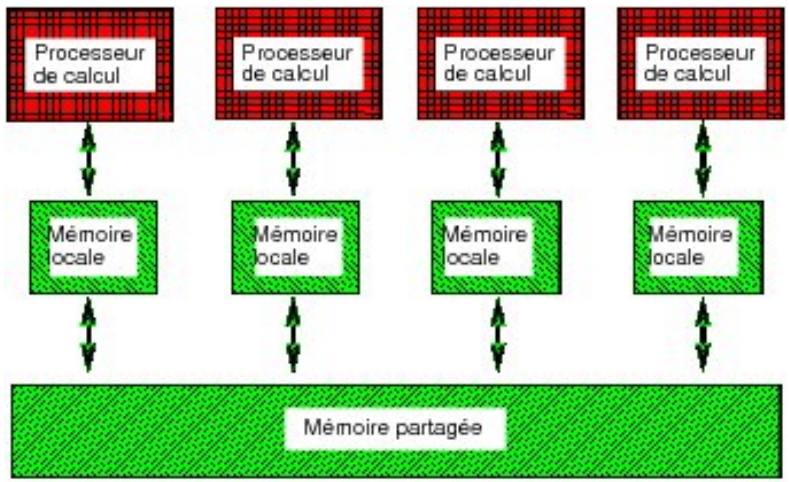
Mémoire partagée :



C'est l'architecture des supercalculateurs vectoriels CRAY, NEC, etc... Le nombre de processeurs est limité (4, **8**, 16...) par le débit mémoire et les conflits d'accès.

MIMD (3)

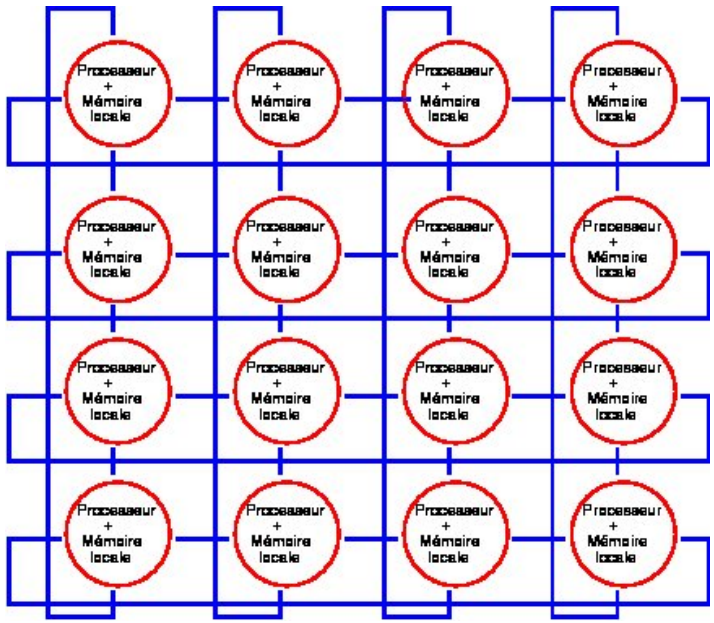
Il a existé aussi des machines à hiérarchie mémoire :



BBN. Chaque processeur de calcul à sa mémoire locale (rapide ! !) pour stocker les données utiles au calcul. (cf. les caches).

MIMD (4)

Processeur + mémoire locale forment les nœuds d'un réseau, plus de contrôle centralisé.



La machine est caractérisée par ses processeurs mais aussi par la topologie et le débit du réseau d'interconnexion : machines INTEL iPSC Paragon, n-Cube, CRAY T3D, T3E, et ... clusters de PCs.

MIMD (5)

Caractéristique fondamentale :
non-uniformité du coût d'accès à la mémoire

Architecture NUMA

Conséquence :

Ils faut rechercher des algorithmes mettant en évidence des structures de données locales et privilégient les calculs sur ces données.

LOCALITE

Performances (1)

Cas des machines SIxx

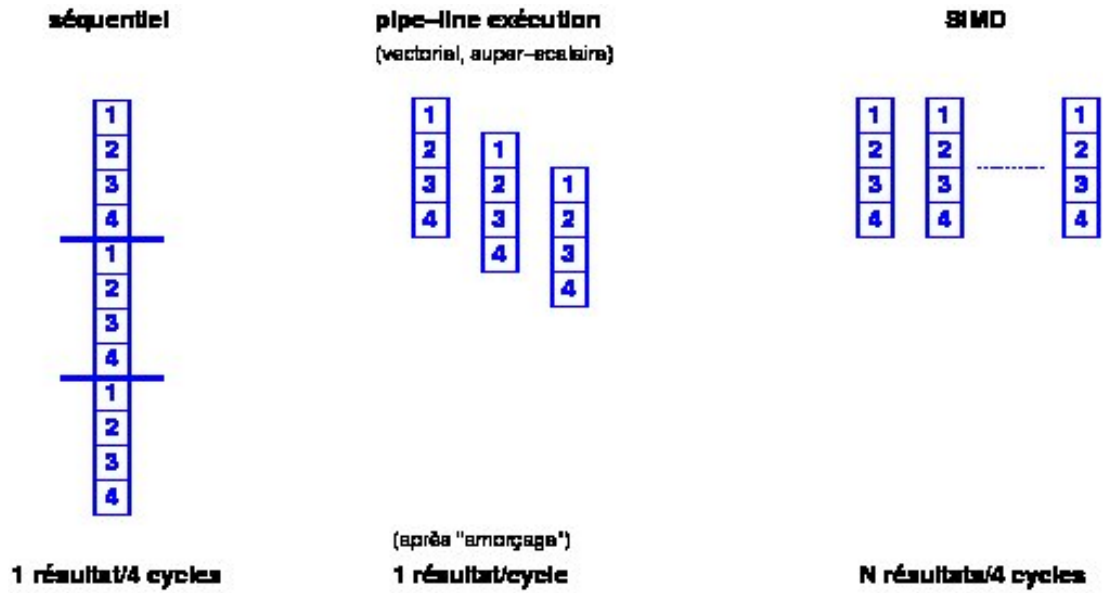
Supposons qu'une addition se décompose en quatre étapes :

1. Comparer les exposants,
2. Décaler la mantisse,
3. Additionner les mantisses,
4. Normaliser le résultat,

chacune durant un cycle de base de la machine.

Performances (2)

On aura les performances suivantes :



Nous allons détailler lecas vectoriel.

Performances (3)

Cas Vectoriel

Soit a le temps de traitement d'une opération (ex addition)
Pour N opérations il faudra un temps $T(N)$:

$$T(N) = S + a * N$$

où Sest le temps de démarrage (*startup*) chargement et à l'amorçage du pipe-line.

Entre le temps observé : $T(N)$ et le temps idéal $a * N$ on a un *facteur de pénalisation* $\rho(N)$:

$$T(N) = (1 + S/(a * N)) * a * N \quad \text{et} \quad \rho(N) = 1 + S/(a * N)$$

Performances (4)

Deux quantités caractérisent les performances du calculateur :

- r_∞, la vitesse asymptotique ($T(N)$ pour N *grand*). r_∞ caractérise la machine d'un point de vue technologique.
- $n_{1/2}$, la longueur de "demi-efficacité". C'est la valeur de N qui donne une vitesse égale à la moitié de r_∞. $n_{1/2}$ mesure le taux de parallélisme de l'architecture.

exemple :

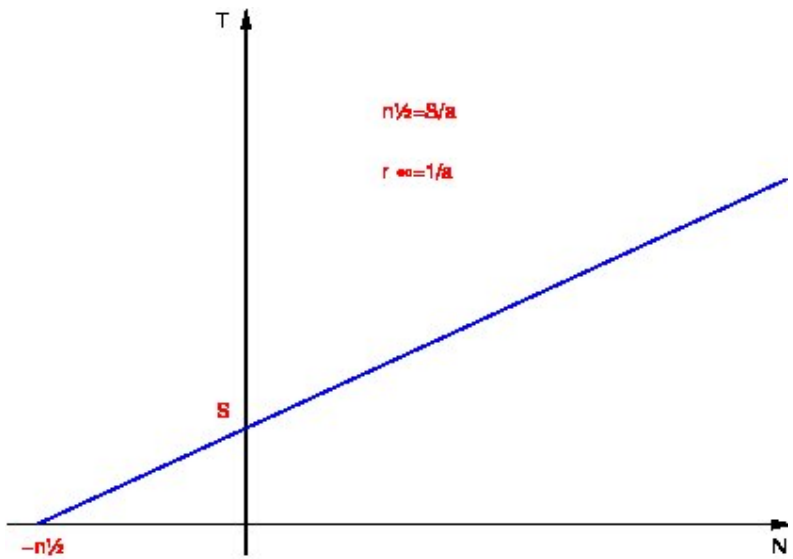
```
.     Pour i=1, N
.       C(i)=A(i)+B(i) ;
```

Performances (5)

En utilisant les notations précédentes, on obtient :

$r_\infty = 1/a$ et $n_{1/2} = S/a$

La courbe T en fonction de N à cette allure :



L'efficacité du pipe-line croit avec le nombre d'étages mais il y a bien sûr une limite au nombre de sous-fonctions qui partitionnent une opération donnée.

Performances (6)

On peut aussi chercher à évaluer le "poids" des parties scalaires face aux parties vectorielles.

- t_s est le temps de traitement en scalaire d'une opération,
- t_v est le temps de traitement (moyen) en vectoriel de cette opération, et
- f le pourcentage d'opérations vectorielles.

Pour effectuer N opérations, le coût sera de :

$$T(N) = N * f * t_v + N * (1 - f) * t_s$$

Performances (7)

$P(f)$ est le facteur de dégradation par rapport au coût idéal $N * t_v$.

$$T(N) = N * t_v * (f + (1 - f) * t_s/t_v) \qquad \text{et}$$

$$P(f) = f + (1 - f) * t_v/t_s$$

Application numérique "réaliste" :

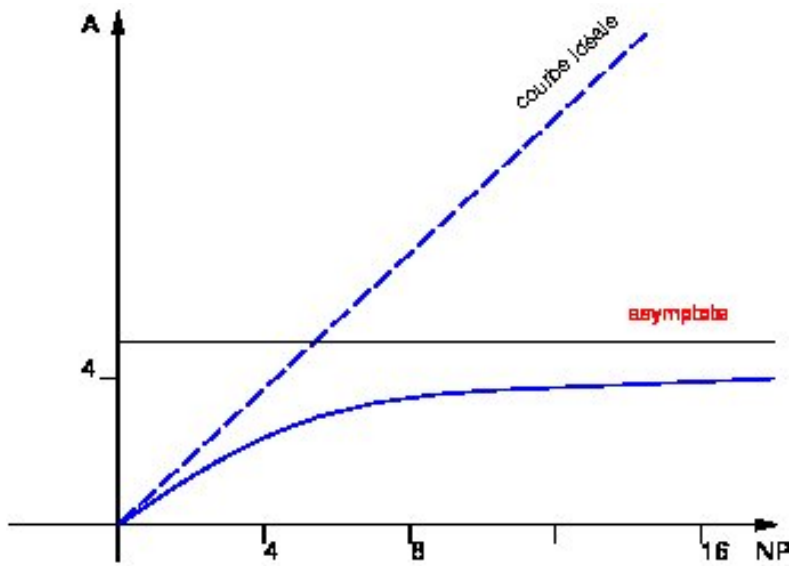
$t_s/t_v = 6$ et $f = 80\%$ nous donnent :
$P(f) = 2!!!!!$

Les parties scalaires coûtent cher...

Performances (8)

Dans le cas MIMD, ce qui précède est valable pour un processeur.

Pour l'ensemble de la machine, on s'intéressera surtout à l'accélération (*speedup*) obtenue en "parallélisant" le code. Accélération (A) / nbe de processeurs (NP) :



Le coût de contrôle du parallélisme croit avec le nombre de processeurs.

Performances (9)

Code parallélisable à 80%, le temps **de résidence** est ramené à 40% sur 4 processeurs, 30 % sur 8 et 20 % sur. . . une infinité de processeurs.

Le calcul de l'accélération est un problème non-trivial :

Il n'est pas toujours possible de passer le code sur un processeur.

On distingue deux modes de calculs selon le type de problèmes.

Performances (10)

Problèmes à taille fixe

où le domaine de calcul est constant et le nombre de processeurs variables.

si T_{par} est le temps **"parallélisable"**, on obtient :

$T_{1proc} = T_{seq} + T_{par}$ et

$T_{Pproc} = T_{seq} + T_{par}/P$

le gain aymptotique est borné :

$T_{1proc}/T_{\infty proc} = 1 + T_{par}/T_{seq}$

Problèmes à échelle

où le domaine de calcul est constant par processeurs.

T_{par}, cette fois-ci le temps **"parallélisé"**.

$$T_{Pproc} = T_{seq} + T_{par} \quad \text{et}$$
$$T_{1proc} = T_{seq} + T_{par} * P$$

Ici le gain $1 + (P-1)T_{par}/(T_{seq} + T_{par})$ croit avec P, mais T_{1proc} n'est, en général, pas mesurable directement.

Remarques (1)

Dans un centre de calculs on fonctionne rarement en "machine dédiée", les performances réelles sont fonctions de la charge de la machine...

L'exécution en parallèle vise à diminuer le **temps de résidence** des travaux mais augmente en général le **temps CPU** consommé.

Le découpage du programme en tâches équilibrées est indispensable et amène à repenser les algorithmes et les méthodes.

Remarques (2)

À chaque architecture correspond un modèle de programmation et donc des outils et des langages adaptés.

- vectoriel.
- SIMD $\Longrightarrow$ GPU
- MIMD à mémoire partagée.
- MIMD à mémoire distribuée.

Les machines actuelles permettent d'utiliser ces modèles simultanément. par exemple : un réseau dont chaque nœud est constitué de plusieurs processeurs vectoriels sur une mémoire partagée.

A quel niveau optimiser un code en fonction de quelle caractéristique technique ?