École Nationale Supérieure
d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, FRANCE

Architecture parallèle vs séquentielle

Niveau	2ème **année**
Parcours	**Informatique**
Unité d'enseignement	**2I2AC3 - Architectures parallèles**
Création	1er **Septembre** 2005
Révisions	2 **Décembre** 2007
	9 **Mai** 2012
	22 **Octobre** 2014
	17 **Aout** 2015
	5 **Juillet** 2020
Responsable	**Emmanuel Cagniot**
	emmanuel.cagniot@ensicaen.fr

Table des matières

1 Modèle d'exécution séquentiel

Les grandes lignes de conception d'une machine électronique ont été établies par J. Von Neuman en 1946. Le postulat de base du modèle d'exécution sous-jacent considère que tout problème peut être décrit par une séquence d'instructions opérant sur une séquence de données, les deux séquences constituant le programme informatique.

Quoique ancien, ce modèle continue de régir le mode de fonctionnement interne des ordinateurs monoprocesseurs actuels. Il prévoit trois composants fondamentaux (Figure 1) : le processeur, la mémoire et le bus permettant au processeur de dialoguer avec la mémoire.

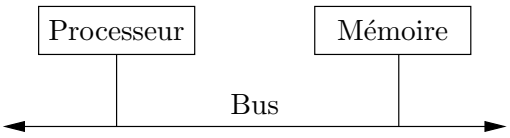

Figure 1 – Modèle d'exécution de J. Von Neuman (1946).

Pour illustrer les notions de séquences d'instructions et de données, considérons le cas d'un langage de programmation dit « compilé machine », par exemple le langage C.

Le programmeur commence par écrire un code source comme celui de la figure 2.

```
1 int globale = 0;
2
3 void
4 main() {
5   while (globale < 8) {
6     globale ++;
7   }
8 }
```

Figure 2 – Exemple de programme C.

Ce code est ensuite transcrit par le compilateur en langage assembleur. La figure 3 présente une traduction possible en assembleur 8086 (Intel) sur un vieux système MS-DOS (exemple utilisé pour sa simplicité). Les séquences de données et d'instructions sont respectivement représentées par les segments de mémoire `.DATA`. et `.CODE`.

In fine, le code précédent est écrit sur disque dans un format binaire directement compréhensible par le processeur. Le fichier exécutable correspondant contient également des informations permettant au chargeur du système d'exploitation de la machine de :

1. réserver une zone de mémoire suffisante pour y recopier la séquence de données. Ces dernières sont qualifiées de données statiques ou persistantes puisqu'elles existent pendant toute la durée d'exécution du programme ;
2. réserver une zone de mémoire suffisante pour y recopier la séquence d'instructions ;
3. réserver deux zones de mémoire additionnelles respectivement appelées pile et tas. La taille de ces zones est fixée via des options de compilation.

La figure 4 présente la structure simplifiée d'un fichier exécutable chargé dans la mémoire de l'ordinateur.

```
1   .MODEL SMALL     ; Modèle mémoire (les 2 séquences limitées à 64Ko).
2
3   .DATA            ; La séquence des données.
4   globale DW 0
5
6   .CODE            ; La séquence des instructions.
7   mov ax, @data
8   mov ds, ax       ; Le registre ds pointe la séquence des données.
9
10  mov cx, globale  ; Le registre cx accueille la valeur de globale.
11
12 loop:             ; Etiquette marquant le début de la boucle.
13  cmp cx, 8
14  je  end_loop     ; (je = jump if equal).
15  inc cx
16  jmp loop
17
18 end_loop:         ; Etiquette marquant la fin de la boucle.
19  mov globale, cx  ; Mise à jour de globale.
20
21  mov ah, 4ch      ; Routine de retour à MS-DOS.
22  mov al, 0
23  int 21h
24
25  END
```

FIGURE 3 – Traduction en assembleur 8086 du programme de la figure 2.

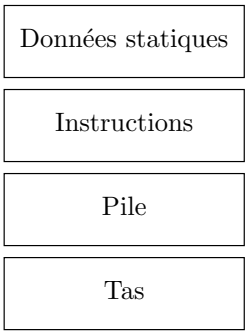

FIGURE 4 – Structure simplifiée d'un fichier exécutable.

La pile est la zone de mémoire dans laquelle sont créées toutes les données qualifiées de locales ou de non persistantes du programme, c'est à dire les données créées automatiquement pour les besoins d'un calcul et détruites automatiquement à l'issue de ce dernier. La pile est gérée selon un principe appelé FIFO (*First In First Out*), c'est à dire que seule la dernière donnée créée est accessible. Deux opérations sont associées à la pile : `push` qui permet d'ajouter le contenu d'un registre en sommet de pile et `pop` qui permet de retirer ce sommet pour le recopier dans un registre.

Le tas est la zone qui accueille les données dynamiques du programme, c'est à dire les données créées et détruites manuellement par le programmeur via des instructions spécialisées et dont la durée de vie est contrôlée par ce dernier. Ces données sont manipulées par l'intermédiaire des pointeurs.

Une fois le programme chargé en mémoire, sa séquence d'instructions est exécutée dans l'ordre (d'où le nom : modèle d'exécution séquentiel) sous le contrôle du compteur ordinal (*instruction pointer*) qui contient l'adresse de la prochaine instruction à exécuter.

L'exécution d'une instruction peut être décomposée en étapes successives :

1. décodage de l'instruction ;
2. mise à jour du compteur ordinal ;
3. calcul de l'adresse des opérandes de l'instruction ;
4. recopie des opérandes depuis la mémoire vers des registres du processeur ;
5. exécution de l'instruction ;
6. calcul de l'adresse du résulat ;
7. recopie du contenu d'un registre vers le résultat en mémoire.

1.1 Déroulage de boucle

Le code des figures 2 et 3 est sous forme dite « canonique » c'est à celle ou sa séquence d'instructions est la plus courte. Cette forme est celle choisie par le compilateur (sur indication du programmeur) lorsque la quantité de mémoire disponible est faible : nous parlons alors d'optimisation en « espace (mémoire) ».

Pourvu que cette quantité de mémoire soit plus importante, le programmeur peut demander au compilateur d'optimiser cette fois en « temps » c'est à dire en durée de calcul. L'idée est ici d'ajouter des instructions supplémentaires à la séquence initiale (d'où une consommation de mémoire plus importante) afin d'accélérer son exécution. Bien sûr il ne s'agit pas de n'importe quelles instructions : nous allons répliquer un certain nombre de fois les instructions des corps de boucles, la technique correspondante étant applée « déroulage de boucle (*loop unrolling*) ».

Supposons que nous répliquions deux fois le corps de boucle du programme C de la figure 2 (on dit « dérouler sur une profondeur valant 2 ») : nous obtenons ainsi le nouveau programme de la figure 5.

Les deux versions sont sémantiquement identiques (elles assurent la même fonction). La différence vient du fait que dans la version optimisée, nous n'effectuons plus que quatre itérations (contre huit dans la version canonique) puisque l'incrémentation de la variable est réalisée deux fois par itération.

```
1 int globale = 0;
2
3 void
4 main() {
5   while (globale < 8) {
6     globale ++;
7     globale ++;
8   }
9 }
```

FIGURE 5 – Déroulage de boucle sur une profondeur valant 2.

La figure 6 présente les modifications apportées à la version assembleur de notre nouveau code.

Dans la version canonique, nous exécutions huit itérations et chaque itération contenait 4 instructions (lignes 13-16), soit $8 \times 4 = 32$ instructions.

Dans la version optimisée, nous exécutons quatre itérations et chaque itération contient 5 instructions (lignes 2-6), soit $4 \times 5 = 20$ instructions. Par conséquent, la forme optimisée termine plus rapidement que la forme canonique puisqu'elle exécute moins d'instructions.

```
1 loop:
2   cmp cx, 8
3   je  end_loop
4   inc cx
5   inc cx
6   jmp loop
```

FIGURE 6 – Modification de la boucle dans la version assembleur.

L'optimisation par déroulage de boucle provoque un résultat à priori contre-intuitif puisque nous ajoutons des instructions pour, in fine, en exécuter moins ... Dans la pratique les compilateurs travaillent directement sur le code assembleur et les profondeurs choisies sont plutôt de l'ordre de 8.

1.2 Opérateurs pipelinés

Certaines instructions telles que les additions et les multiplications flottantes sont présentes en très grandes quantités dans les codes. Ces opérations présentent souvent une particularité intéressante : celle de pouvoir être décomposées en étapes. Dans ce cas, il est possible d'améliorer le modèle séquentiel en faisant assurer l'exécution de ces instructions par des opérateurs particuliers appelés pipelines et directement intégrés au processeur.

Prenons l'exemple d'une addition flottante : celle-ci peut être segmentée en trois étapes consécutives :

1. comparaison des exposants et alignement de la mantisse du plus petit nombre sur celle du plus grand (dénormalisation) ;
2. addition des mantisses en virgule fixe ;
3. normalisation du résultat (IEEE 754).

La figure 7 présente l'opérateur pipeliné correspondant.

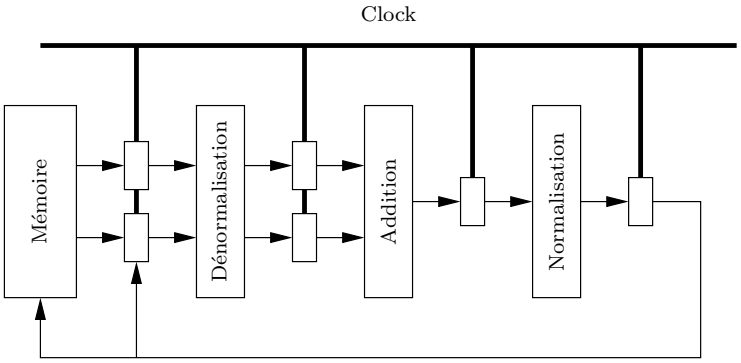

FIGURE 7 – Additionneur en virgule flottante.

Un opérateur pipeliné est constitué d'étages c'est à dire des ressources matérielles souvent uniques (et qui doivent donc être partagées) dotées de registres d'entrée et de sortie. Ces étages sont regroupés par niveaux consécutifs, les sorties du niveau i étant les entrées du niveau $i + 1$. Au sein d'un niveau, tous les étages sont activés simultanément. Le passage des informations depuis un niveau précédent vers un niveau suivant sont synchronisés par une horloge (opérateur pipeliné synchrone). Ainsi, dans l'exemple

de la figure 7, nous avons un pipeline à trois niveaux, chaque niveau étant constitué d'un seul étage.

Considérons une séquence $\mathcal{I}_n = \{op_1, op_2, \cdots, op_n\}$ constituée de n occurrences d'une même opération élémentaire *op* (il s'agit généralement d'une boucle). Cette opération peut être exécutée sur un opérateur pipeliné à trois niveaux d'étages (un étage par niveau) dont la table de réservation est donnée par :

	1	2	3	4
$\mathbf{stage}_3$				×
$\mathbf{stage}_2$			×	
$\mathbf{stage}_1$	×	×		

Une table de réservation est un formalisme dans lequel les lignes représentent les étages et les colonnes les tops de l'horloge. Un symbole est placée dans la case (i, j) si l'étage i est utilisé au top j.

Dans notre cas, la table de réservation indique que les durées de traversée des étages $\mathbf{stage}_1$, $\mathbf{stage}_2$ et $\mathbf{stage}_3$ valent respectivement 2τ (2 tops d'horloge), 1τ et 1τ. Nous allons maintenant caractériser cet opérateur.

Si notre opérateur est utilisé en mode séquentiel alors tous ses niveaux sont activés l'un après l'autre. Dans ce cas, la durée d'exécution de notre séquence est :

$$t_1 = n \times 4 \times \tau. \tag{1.1}$$

À l'inverse, si notre opérateur est utilisé en mode pipeline (c'est le compilateur qui décide à partir d'un critère que nous étudierons en TD) alors tous ses niveaux sont activés simultanément. Dans ce cas, il faut nous intéresser à deux choses :
— la durée d'exécution de la première instruction (la latence) ;
— le rythme d'arrivée des $(n-1)$ résultats suivants.

Dans notre cas, il faut 4τ pour exécuter la première instruction.

Ensuite, il faut essayer de commencer la seconde instruction le plus tôt possible afin de réduire l'écart entre deux instructions consécutives. Ainsi, si la première instruction a démarré au top 0, nous ne pouvons pas commencer la seconde avant le top 2 car l'étage $\mathbf{stage}_1$ n'est pas disponible (la ressource correspondante n'existe qu'en un seul exemplaire).

En injectant la seconde instruction au top 2, nous constatons qu'il n'y a aucun conflit d'accès aux différents étages (ils sont tous à nouveau disponibles) : nous venons donc de déterminer la meilleur vitesse d'injection des instructions suivantes : deux tops plus tard que l'instruction précédente. D'après la table de réservation, cette fréquence d'injection fait qu'une instruction achève son exécution deux tops plus tard que la précédente.

Nous pouvons à présent donner l'expression de la durée d'exécution de notre séquence si l'opérateur est utilisé en mode pipeline :

$$t_2 = \big(4 + (n-1) \times 2\big)\tau = (2n+2)\tau. \tag{1.2}$$

Définissons à présent le facteur d'accélération (*speedup*) de notre opérateur comme :

$$s_3^n = \frac{t_1}{t_2} = \frac{2n}{n+1}, \tag{1.3}$$

et faisons tendre n vers l'infini (la séquence devient très longue) ; nous obtenons :

$$s_3^\infty = 2, \tag{1.4}$$

ce qui signifie que notre séquence sera exécuté deux fois plus rapidement en mode pipeline qu'en mode séquentiel (en fait au rythme du niveau le plus lent).

Ce résultat explique la généralisation de la technique du pipeline à tous les niveaux du modèle séquentiel afin d'améliorer sa « vitesse » générale.

1.3 Mémoires multi-bancs

Le modèle séquentiel ne prévoit qu'un seul banc de mémoire puisqu'un bus ne peut assurer qu'une communication à la fois. De fait, afin d'obtenir une machine relativement performante, il convient de trouver le meilleur compromis possible entre la technologie du processeur, celle du bus et de la mémoire (tout système fonctionne au rythme de son composant le plus lent).

En règle générale, le processeur est bridé par la mémoire. Une façon de remédier à ce problème consiste à installer une hiérarchie de mémoires caches entre le processeur et la mémoire (découpe horizontale de la mémoire).

Cette solution peut être encore améliorée en connectant plusieurs bancs de mémoire sur le bus (découpe verticale de la mémoire). Pour peu que les données ou instructions auxquelles souhaite accéder le processeur se trouvent dans des bancs séparés, il devient possible d'y accéder quasi-simultanément en pipelinant les requêtes. Nous pouvons alors passer du comportement mono-banc de la figure 8 au comportement multi-bancs de la figure 9. Dans le premier cas, le processeur travaille au rythme de son unique banc de mémoire alors que dans le second, les requêtes à la mémoire (et la récupération des résultats correspondants) s'effectuent au rythme du processeur.

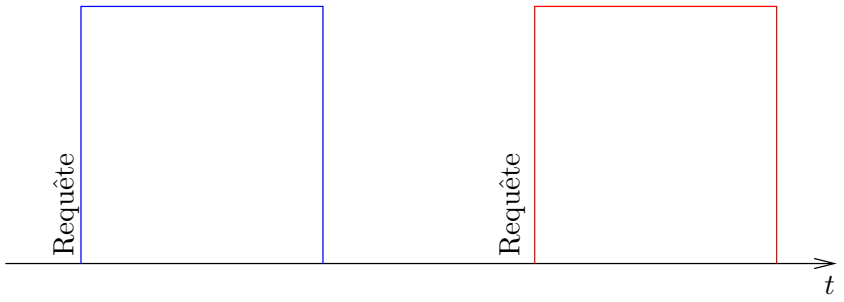

FIGURE 8 – Système mono-banc : les requêtes à la mémoire sont sérialisées.

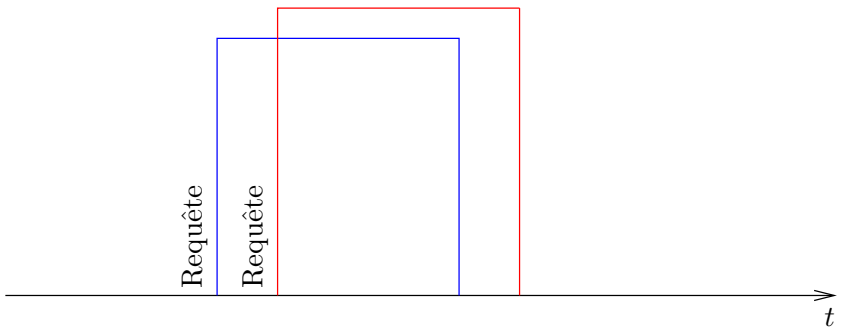

FIGURE 9 – Système multi-bancs : les requêtes à la mémoire sont pipelinées.

Connecter plusieurs bancs de mémoire sur un même bus impose de modifier le format d'adresses utilisé. Dans le cas mono-banc, une adresse mémoire désignait in fine une *offset*. Dans le cas multi-bancs, il faut qu'une adresse contienne à la fois le banc cible de la requête ainsi que l'*offset* à l'intérieur de ce banc.

Les bancs de mémoire sont tout simplement désignés par un entier naturel, la numérotation commençant à 0. Il faut p *bits* pour coder un numéro dans un système comportant 2^p bancs. Ainsi, la capacité de stockage mémoire maximum d'un système 32 *bits* comportant quatre bancs est $2^{32-2} = 2^{30}$ *bits* par

banc soit encore $4 \times 1\,Gb$.

Considérons un registre d'adresse (d'une longueur quelconque). Ce registre peut être lu de deux façons selon l'architecture du processeur :

little endian : la lecture s'effectue des *bits* de poids faibles vers ceux de poids forts ; c'est le cas des processeurs INTEL par exemple ;

big endian : la lecture s'effectue des *bits* de poids forts vers ceux de poids faibles ; c'est le cas des processeurs SPARC par exemple.

Le numéro du banc doit toujours être lu avant l'*offset*. Par conséquent, deux formats d'adresses peuvent être définis :

low order interleaving : le format associé au architectures *little endian* : l'identification du banc se trouve dans la partie basse du registre d'adresse ;

high order interleaving : le format associé au architectures *big endian* : l'identification du banc se trouve dans la partie haute du registre d'adresse.

Reprenons l'exemple de notre système 32 *bits* doté de 4 bancs et supposons qu'il soit *little endian*. Considérons également le tableau d'entiers `t = [ 1, 2, 3, 4, 5 ]`, un entier étant codé sur quatre octets.

Dans la pratique il est impossible d'imposer explicitement le placement d'un élément de ce tableau dans un banc particulier (les langages de programmation ne proposent pas ce type de mécanisme). Cependant, rappelons nous qu'un tableau est une suite d'adresses mémoires contiguës (des adresses et pas des emplacements physiques). S'il n'est pas possible de placer explicitement une donnée dans un banc, il est en revanche possible de demander au compilateur d'implanter un tableau à partir d'une certaine adresse en mémoire.

Supposons que nous demandions au compilateur d'implanter le tableau t à l'adresse de base de la mémoire c'est à dire 0 ; nous avons alors :

Élément	***Offset*** (30 ***bits***, ←)	**Banc** (2 *bits*, ←)
`t[0] = 1`	0...00	00
`t[1] = 2`	0...00	01
`t[2] = 3`	0...00	10
`t[3] = 4`	0...00	11
`t[4] = 5`	0...01	00

d'où la répartition cyclique suivante :

$\mathbf{B}_0$	$\mathbf{B}_1$	$\mathbf{B}_2$	$\mathbf{B}_3$
1	2	3	4
5			

Dès lors, pour une boucle telle que celle de la figure 10, les quatre premières requêtes à la mémoire pourront être pipelinées puisque les éléments concernés se trouvent dans des bancs différents.

```
1 for (int i = 0; i < 5; i ++) {
2   t[i] = 10;
3 }
```

FIGURE 10 – Boucle de parcours des éléments d'un tableau.

1.4 Pipeline d'instructions

Dans le modèle séquentiel, l'exécution d'une instruction est décomposable en étapes. Certaines concernent la mémoire et d'autres le processeur. Qui dit étapes dit pipeline et, par conséquent, même l'exécution d'une instruction peut être pipelinée. Cependant, nous faisons face ici à un écueil : celui des instructions de saut (plus couramment appelées *jumps* ou bien encore *goto*) que nous rencontrons dans les alternatives (`if then`, `if then else`, etc.) et les boucles (`while`, `for`, etc.).

En termes d'efficacité, rien n'est pire que de devoir arrêter un pipeline pour le vidanger puis le réalimenter avec autre chose. Or, dans le cas d'un *jump*, c'est exactement ce qui risque de se produire. Par exemple, si nous regardons le code assembleur de la figure 3, nous constatons la présence de deux instructions de saut :

— l'instruction `je end_loop` (*jump if equal*) en ligne 14 ;
— l'instruction `jmp loop` en ligne 16.

La seconde ne pose aucun problème car elle ne dépend pas d'une condition (saut inconditionnel). Par conséquent, l'instruction suivante est toujours connue : celle qui suit l'étiquette indiquée dans le saut.

À l'inverse, la première est une source de problèmes car elle dépend d'une condition (saut conditionnel) qui ne peut être évaluée à l'avance. Dans notre cas, selon le résultat de la comparaison du contenu du registre `cx` avec 8, nous pouvons soit exécuter la suite du corps de boucle, soit sauter à la fin de cette boucle.

Il existe plusieurs techniques permettant de traiter ce problème. L'une des plus courantes est celle du « branchement retardé » qui consiste à approvisionner le pipeline avec les premières instructions des deux branches possibles puis à continuer avec celle de la bonne branche une fois la condition évaluée. Cependant, cette technique suppose de neutraliser certaines interruptions du système puisque des instructions qui n'auraient pas dû être exécutées (par exemple des divisions par 0) le seront quand même.

1.5 Processeur super-scalaire

Le modèle séquentiel de Von Neuman ne prévoit qu'une seule unité d'exécution au niveau du processeur. Il est cependant possible d'en ajouter d'autres (sous réserve de résoudre les problèmes technologiques correspondants, par exemple en termes de place disponible, de consommation électrique, de dissipation de chaleur, etc.). Chaque unité d'exécution étant indépendante des autres, nous pouvons alors avoir autant d'opérateurs pipelinés dédiés à une opération particulière que d'unités d'exécution : le processeur est alors dit « super-scalaire ».

Supposons que notre processeur soit pourvu de deux unités d'exécution indépendantes mais d'un seul exemplaire de l'opérateur pipeliné de la section 1.2 (c'est un exemple). Dans ce cas, la boucle de la figure 11 est prise en compte par cet unique exemplaire et nous obtenons un *speedup* valant 2 (`op` désigne l'opération concernée).

```
1 for (int i = 0; i < n; i ++) {
2   a[i] = b[i] op c[i];
3 }
```

Figure 11 – Exploitation d'un seul exemplaire de notre opérateur pipeliné.

Installons à présent un deuxième exemplaire de cet opérateur et supposons que la longueur n de notre séquence d'opérations soit un multiple de 2. Nous pouvons alors ré-écrire notre boucle afin que le compilateur exploite simultanément les deux opérateurs. La figure 12 présente cette modification (l'écriture est la même que pour un déroulage de boucle).

In fine, la durée de calcul est déjà divisée par deux au niveau global puisque chaque opérateur prend en

```
for (int i = 0; i < n; i += 2) {
  a[i    ] = b[i    ] op c[i    ];
  a[i + 1] = b[i + 1] op c[i + 1];
}
```

FIGURE 12 – Exploitation simultanée de deux exemplaire de notre opérateur pipeliné.

charge une moitié de la séquence initiale. D'autre part, l'exécution de chaque demi-séquence est accélérée d'un facteur 2 (*speedup* de l'opérateur). Par conséquent, l'exécution de notre séquence est accélérée d'un facteur 4.

Dans la pratique, le compilateur effectue lui-même cette ré-écriture lorsque ses options d'optimisation sont activées.

Bien qu'un système équipé d'un processeur super-scalaire ne soit pas considéré comme parallèle (il faut au moins deux processeurs), nous jouons sur les mots car c'est bel et bien le cas.

1.6 Architecture vectorielle et super-vectorielle

Un processeur vectoriel est exclusivement conçu pour le traitement des tableaux (ou vecteurs), traitement basé sur une exploitation intensive des pipelines (il s'agit donc d'un processeur super-scalaire à la base). Plus précisément, ces opérations de traitement peuvent être classées en quatre catégories :

1. `Vecteur` $\rightarrow$ `Vecteur`;
2. `Vecteur` $\rightarrow$ `Scalaire`;
3. `Vecteur` $\times$ `Vecteur` $\rightarrow$ `Vecteur`;
4. `Scalaire` $\times$ `Vecteur` $\rightarrow$ `Vecteur`.

Tout système équipé de ce type de processeur est qualifié d'architecture vectorielle. La figure 13 présente un exemple d'architecture dotée de deux opérateurs pipelinés (addition et multiplication entière ou flottante) et donc les accès mémoire sont également pipelinés (mémoire multi-bancs).

Considérons l'algorithme 1 faisant intervenir une opération de type `Vecteur` $\times$ `Vecteur` $\rightarrow$ `Vecteur` et une autre de type `Scalaire` $\times$ `Vecteur` $\rightarrow$ `Vecteur` (k désigne la constante scalaire). L'implémentation se fera selon l'algorithme 2 sur un système séquentiel (modèle d'exécution de VON NEUMAN) et selon l'algorithme 3 sur une architecture (ou système) vectorielle.

Algorithme 1 (`Vecteur` $\times$ `Vecteur` $\rightarrow$ `Vecteur`) et (`Scalaire` $\times$ `Vecteur` $\rightarrow$ `Vecteur`).

for $i = 1$ **to** n **do**
  $A(i) \leftarrow B(i) + C(i)$
  $D(i) \leftarrow k * E(i + 1)$
end for

Le fait que les accès mémoire ainsi que le traitement des tableaux soient pipeliné explique la puissance d'une architecture vectorielle par rapport à un système séquentiel (on dit aussi « scalaire » par opposition à « vectoriel »).

Une architecture vectorielle peut encore être améliorée (quitte à répliquer certaines ressources matérielles partagées par ses différents pipelines). En effet, dans une architecture vectorielle (voir algorithme 3), le traitement des opérandes de type tableau ne commence que lorsque ceux-ci ont été chargés dans des registres vectoriels. Si, au lieu d'attendre que cette recopie soit achevée, nous commençons à traiter les

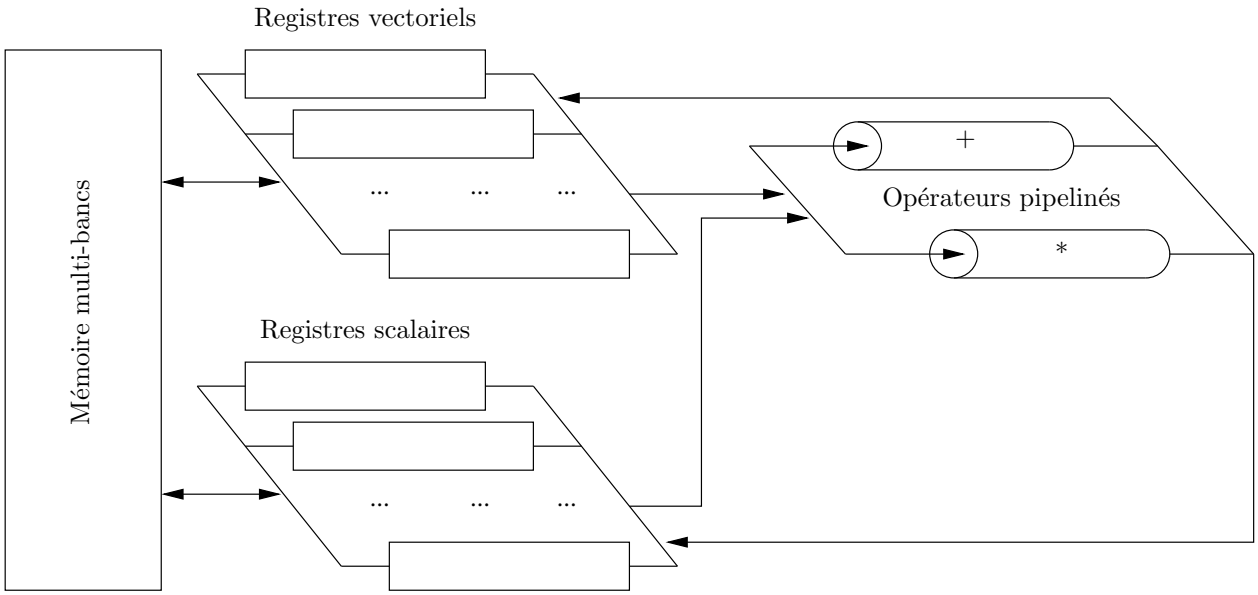

FIGURE 13 – Une architecture vectorielle.

Algorithme 2 Implémentation sur système séquentiel.

1: $i \leftarrow 1$
2: `Load` $B(i)$
3: `Load` $C(i)$
4: `Add` $B(i)$, $C(i)$
5: `Store` $A(i)$
6: `Load` k
7: `Load` $E(i+1)$
8: `Mul` k, $E(i+1)$
9: `Store` $D(i)$
10: $i \leftarrow i+1$
11: **if** $i \leq n$ **then**
12: `Goto` 2
13: **end if**

Algorithme 3 Implémentation sur architecture vectorielle.

1: `Load` $B(1:n)$
2: `Load` $C(1:n)$
3: $A(1:n) \leftarrow B(1:n) + C(1:n)$
4: `Store` $A(1:n)$
5: `Load` k
6: `Load` $E(2:n+1)$
7: $D(1:n) \leftarrow k \times E(2:n+1)$
8: `Store` $D(1:n)$

premières composantes de nos vecteurs en attendant les autres alors l'ensemble de notre architecture devient un gigantesque pipeline : nous parlons alors d'architecture « super-vectorielle » ou, pas abus de langage, de « super-calculateur vectoriel ».

Illustrons ces notions sur l'exemple d'architecture de la figure 13 en supposant que :

1. l'unité de temps est le cycle de base de la machine ;
2. un accès à l'un des bancs de mémoire nécessite $\mathcal{C}$ cycles ;
3. le pipeline d'addition de la machine est idéal (même durée de traversée pour chaque niveau) et sa latence vaut $\mathcal{P}_+ = 4$ cycles ;
4. le pipeline de multiplication est lui-même idéal et sa latence vaut $\mathcal{P}_\times = 5$ cycles.

L'algorithme que nous devons implémenter sur cette architecture est :

Algorithme 4 Algorithme à implémenter.

```
1: for i = 1 to n do
2:     Y(i) ← a * X(i) + B(i)
3: end for
```

Dans le cas d'une implémentation scalaire, l'exécution d'une itération de la boucle nécessite la lecture des trois opérandes a, $X(i)$ et $B(i)$ soit 3 accès consécutifs à la mémoire. L'addition et la multiplication nécessitent respectivement $\mathcal{P}_+$ et $\mathcal{P}_\times$ cycles. Le stockage du résultat $Y(i)$ en mémoire nécessite $\mathcal{C}$ cycles. La durée totale est donc $3\mathcal{C} + \mathcal{P}_+ + \mathcal{P}_\times + \mathcal{C} = 4\mathcal{C} + \mathcal{P}_+ + \mathcal{P}_\times$ cycles. Par conséquent, l'exécution de la boucle nécessite :

$$t_1 = n \times (4\mathcal{C} + \mathcal{P}_+ + \mathcal{P}_\times) = n \times (4\mathcal{C} + 9) \text{ cycles.} \tag{1.5}$$

Dans le cas des implémentations vectorielle et super-vectorielle, le pipeline d'accès mémoire est considéré comme idéal et sa latence vaut $\mathcal{C}$ cycles.

Dans une implémentation vectorielle, l'exécution de la boucle nécessite :
— le chargement du scalaire a soit $\mathcal{C}$ cycles ;
— le chargement du vecteur $X(1:n)$ soit $(\mathcal{C} + n - 1)$ cycles ;
— la multiplication du scalaire a par le vecteur $X(1:n)$ soit $(\mathcal{P}_\times + n - 1)$ cycles ;
— le chargement du vecteur $B(1:n)$ soit $(\mathcal{C} + n - 1)$ cycles ;
— l'addition des vecteurs $a \times X(1:n)$ et $B(1:n)$ soit $(\mathcal{P}_+ + n - 1)$ cycles ;
— le stockage en mémoire du vecteur $Y(1:n)$ soit $(\mathcal{C} + n - 1)$ cycles.

En sommant toutes ces quantités, nous obtenons la durée d'exécution de notre boucle dans une implémentation vectorielle :

$$t_2 = 5n + (4\mathcal{C} + \mathcal{P}_+ + \mathcal{P}_\times - 5) = 5n + (4\mathcal{C} + 4) \text{ cycles.} \tag{1.6}$$

Nous remarquons que le terme dominant de l'équation précédente est $5n$ contre $n \times (4\mathcal{C} + 9)$ dans l'implémentation scalaire : la boucle s'exécute déjà beaucoup plus rapidement dans son implémentation vectorielle.

Passons maintenant à une implémentation super-vectorielle ou l'architecture n'est plus qu'un gigantesque pipeline. Dans celle-ci, l'obtention du premier résultat nécessite le chargement de a, $X(1)$ et $B(1)$, la multiplication $a \times X(1)$, l'addition $a \times X(1) + B(1)$ et la recopie de $Y(1)$ en mémoire, ce qui représente une latence valant $(3\mathcal{C} + \mathcal{P}_\times + \mathcal{P}_+ + \mathcal{C})$ cycles. Les résultats suivants sont obtenus au rythme d'un résultat par cycle de base. De fait, la durée d'exécution de notre boucle dans une implémentation super-vectorielle est :

$$t_3 = (3\mathcal{C} + \mathcal{P}_\times + \mathcal{P}_+ + \mathcal{C}) + n - 1 = n + (4\mathcal{C} + 8) \text{ cycles.} \tag{1.7}$$

Cette fois-ci, le terme dominant de l'équation précédente est n, c'est à dire que notre boucle s'exécute cinq fois plus rapidement que dans une implémentation vectorielle.

1.7 Et encore d'autres améliorations

Il est encore possible d'améliorer un système séquentiel ; citons, pêle-mêle :
- les unités d'exécution SIMD (jeux d'instructions vectoriels) ;
- l'*hyper-threading* (INTEL) ;
- les co-processeurs spécialisés ;
- etc.,

mais également ... des problèmes de sécurité :
- l'exécution spéculative (branchements retardés) à l'origine des failles SPECTRE ou MELTDOWN ;
- etc.

2 Architecture parallèle

La distinction entre système parallèle et distribué est parfois floue. Nous pouvons cependant donner les deux petites définitions suivantes :

système parallèle : un ensemble de processeurs élémentaires qui coopèrent à la résolution d'un problème de grande taille ;

système distribué : un ensemble de processeurs autonomes qui ne partagent pas d'espace mémoire primaire et qui coopèrent par échanges de messages au travers d'un réseau de communication.

Ces définitions montrent qu'un système parallèle est exclusivement conçu pour le calcul intensif, c'est à dire le support d'applications gourmandes en temps de calcul et en espace mémoire. Un système distribué est, quant à lui, beaucoup plus généraliste mais peut également servir au calcul intensif.

Deux approches permettent d'accroître la puissance de calcul des machines. La première, séquentielle, consiste à exécuter les instructions plus rapidement. La seconde, parallèle, consiste à exécuter simultanément plusieurs instructions. Du fait des limites physiques en matière d'intégration électronique et des problèmes liés aux bruits parasites et à la dissipation thermique, les gains obtenus par l'approche séquentielle sont bornés à terme. L'approche parallèle apparaît alors comme une alternative.

Les recherches sur ces deux approches sont menées conjointement. Si les limites physiques de l'approche séquentielle ne sont pas encore atteintes (nous gravons de plus en plus fin), l'approche parallèle s'est généralisée.

Une architecture parallèle contient des processeurs (ou cœurs), des bancs de mémoire et un réseau de communication. Ce dernier peut relier les processeurs entre eux ou les relier aux bancs de mémoire. Selon le type de la machine, la mémoire et le réseau de communication peuvent adopter des architectures très différentes. Le fonctionnement interne de cette machine est régi par un modèle d'exécution décrivant les relations entre instructions et données auxquelles elles s'appliquent.

L'étude de la complexité algorithmique parallèle nécessite la définition d'un modèle de calcul. De nombreux modèles sont dédiés aux machines parallèles, les principaux étant le modèle PRAM (*Parallel Random Access Memory*) et les circuits booléens et arithmétiques.

La programmation d'une machine parallèle est régie par un modèle de programmation étroitement lié à son modèle d'exécution. Deux modèles sont actuellement définis : le modèle à parallélisme de données et le modèle à parallélisme contrôle.

Enfin, la qualité d'une application parallèle est évaluée en fonction de trois critères appelés facteur d'accélération (*speedup*), facteur d'efficacité (*efficiency*) et facteur de d'élasticité (*scalability*).

2.1 Classification de Flynn

Toute architecture, qu'elle soit séquentielle, parallèle ou distribuée doit « rentrer dans une case » c'est à dire être rangée dans une catégorie.

Une première classification possible peut se faire selon les modèles d'exécution. Parmi les classifications existantes, celle de J. Flynn (quoique maintenant ancienne : 1972) est toujours largement utilisée du fait de sa simplicité : elle est basée sur la notion de flux d'instructions et de données, un flux pouvant être simple ou multiple (Figure 14).

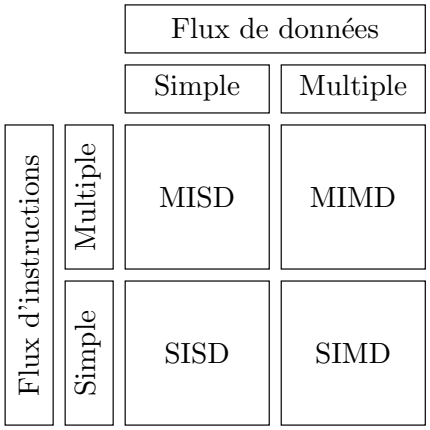

FIGURE 14 – Classification de J. FLYNN (1972).

2.1.1 Classe des modèles SISD

Cette classe est celle des architectures mono-processeur classiques dans lesquelles un flux unique d'instructions est appliqué à un flux unique de données (Figure 15). Elle ne comporte qu'une seule instance : le modèle de J. VON NEUMAN (1946).

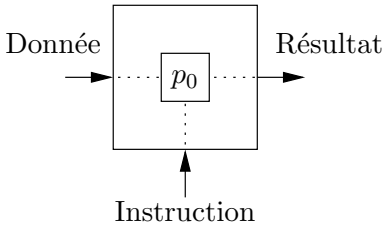

FIGURE 15 – Classe des modèles SISD.

2.1.2 Classe des modèles SIMD

Cette classe est celle des machines parallèles équipées d'une unité de contrôle centralisée. Leur fonctionnement est de type synchrone. L'unité de contrôle envoie la même instruction à tous les processeurs de la machine. Ces derniers l'exécutent simultanément sur leur propre donnée et génèrent leur propre résultat. Le flux d'instructions est donc simple et le flux de données multiple (Figure 16).

Les processeurs de ces machines sont souvent peu puissants mais nombreux. Ce grand nombre de processeurs pose des problèmes au niveau de l'horloge interne de la machine. Le fonctionnement synchrone impose que tous les processeurs reçoivent simultanément le même top d'horloge. Ces difficultés techniques ont peu à peu conduit à l'abandon de ce modèle dans les architectures parallèles mais à son intégration dans les processeurs (initialement sur les processeurs *Pentium MMX* d'INTEL en 1996) puisqu'il est à l'origine des jeux d'instructions dit « vectoriels » et des unités d'exécution dites « SIMD ».

L'idée d'un jeu d'instruction vectoriel est d'exploiter tout l'espace laissant vacant dans un registre lorsqu'une donnée y est copiée pour ensuite servir d'opérande à une instruction assembleur. Ainsi, dans le jeu d'instruction vectoriel SSE 2 (introduit avec les *Pentium IV* d'INTEL en 1999), un registre 128 *bits* peut accueillir, au choix (Figure 17) :

— 16 nombres entiers codés sur 8 bits (type `char`) ;
— 8 nombres entiers codés sur 16 bits (type `short int`) ;

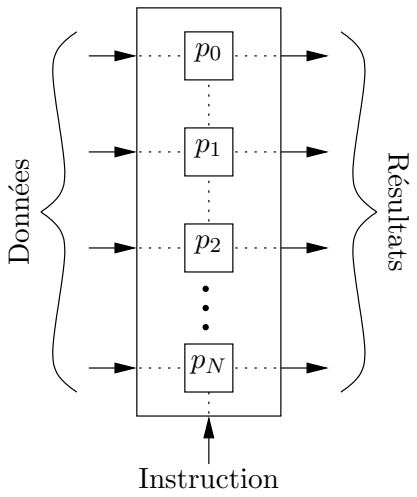

FIGURE 16 – Classe des modèles SIMD.

— 4 nombres entiers codés sur 32 bits (type `int`) ;
— 2 nombres entiers codés sur 64 bits (type `long int`) ;
— 4 nombres flottants simple précision codés sur 32 bits (type `float`) ;
— 2 nombres flottants double précision codés sur 64 bits (type `double`).

Les jeux d'instructions vectoriels sont mis en œuvre via des fonctions de bibliothèque écrites en assembleur : les « *intrinsics* ». Les figure 18, 19 et 20 présente le cheminement de tout programmeur.

La figure 18 présente le point de départ : une boucle sous forme canonique permettant de calculer la somme de deux vecteurs de type `float` dont la taille est un multiple de 4.

Nous supposons maintenant que le jeu d'instructions SSE 2 est disponible sur notre processeur et que le programmeur souhaite l'exploiter. Il va donc commencer par écrire une forme intermédiaire de type « boucle déroulée » sur une profondeur de 4 (Figure 19).
Cette forme intermédiaire fait apparaître les caractéristiques suivantes à chaque itération de la boucle :
— la tranche `B[i:i+3]` est accédée en lecture (4 éléments) ;
— la tranche `C[i:i+3]` est accédée en lecture (4 éléments) ;
— une fois lues, les tranches `B[i:i+3]` et `C[i:i+3]` sont additionnées, la composante `B[k]` étant ajoutée à la composante `C[k]` ;
— le résultat de l'addition est recopié dans la tranche `A[i:i+3]` (4 éléments).

Ne reste plus qu'à écrite la forme « vectorisée » de notre boucle (Figure 20). Pour cela, nous avons besoin du module de bibliothèque `emmintrin` fourni par le compilateur.

On constate que la forme vectorisée est beaucoup moins lisible que la forme « boucle déroulée » puisqu'elle utilise des fonctions *intrinsics*. Cependant, le programmeur dispose maintenant de deux techniques d'optimisation possibles (à lui, ou à son compilateur, de choisir la bonne en fonction du problème traité) :
— déroulage de boucle pour exploiter les possibilités super-scalaire du processeur ;
— vectorisation pour exploiter son unité d'exécution SIMD (il peut en exister plusieurs).

2.1.3 Classe des modèles MISD

Dans la réalité, cette classe (Figure 21) ne possède aucune instance. Par certains aspects, le modèle d'exécution pipeline s'en rapproche mais les données qui circulent entre les niveaux successifs peuvent être différentes.

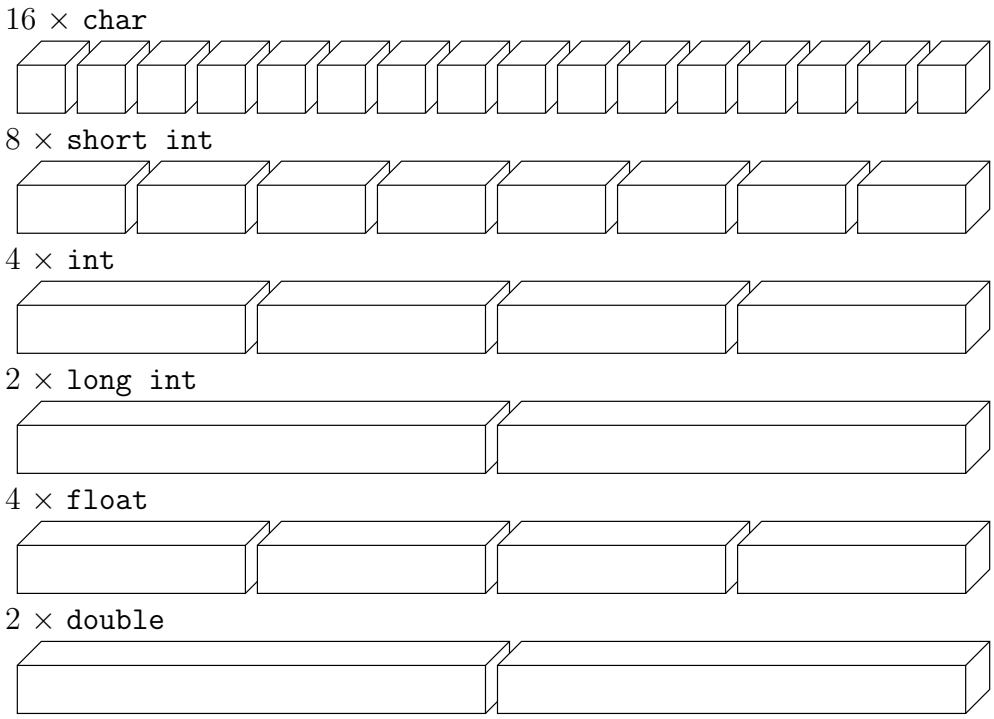

FIGURE 17 – L'un des huit registres 128 *bits* `XMMx` du *Pentium IV*.

```
1  const int N = 4 * 1024;
2  double A[N], B[N], C[N];
3
4  // ... initialisation de B et C ...
5
6  for (int i = 0; i < N; i ++) {
7    A[i] = B[i] + C[i];
8  }
```

FIGURE 18 – Le point de départ : une boucle sous forme canonique.

```
1  const int N = 4 * 1024;
2  double A[N], B[N], C[N];
3
4  // ... initialisation de B et C ...
5
6  for (int i = 0; i < N; i += 4) {
7    A[i    ] = B[i    ] + C[i    ];
8    A[i + 1] = B[i + 1] + C[i + 1];
9    A[i + 2] = B[i + 2] + C[i + 2];
10   A[i + 3] = B[i + 3] + C[i + 3];
11 }
```

FIGURE 19 – Forme intermédiaire de type « boucle déroulée ».

```
#include <emmintrin.h>

const int N = 4 * 1024;
double A[N], B[N], C[N];

// ... initialisation de B et C ...

for (int i = 0; i < N; i += 4) {
  __m128 reg_b = _mm_load_ps(&B[i]);
  __m128 reg_c = _mm_load_ps(&C[i]);
  __m128 reg_a = _mm_add_ps(reg_b, reg_c);
  _mm_store_pd(&A[i], reg_a);
}
```

FIGURE 20 – Forme définitive « vectorisée ».

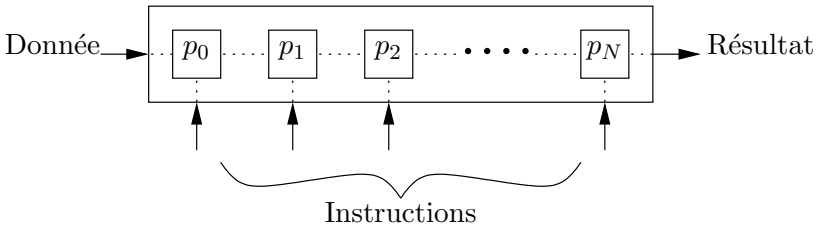

FIGURE 21 – Classe des modèles MISD.

2.1.4 Classe des modèles MIMD

Cette classe est celle des machines parallèles équipées de plusieurs unités de contrôle totalement indépendantes les unes des autres. Leur fonctionnement est de type asynchrone. Chaque processeur est autonome et gère son propre flux d'instructions et son propre flux de données. Les programmes qui s'exécutent sur ces processeurs peuvent être totalement différents. Le flux d'instructions et le flux de données sont donc multiples (Figure 22).

Le mode de fonctionnement asynchrone permet de s'affranchir du problème d'horloge des machines SIMD et donc d'obtenir des architectures dites « massivement parallèles ». Les machines MIMD sont à l'heure actuelle les machines parallèles les plus couramment rencontrées.

La programmation des machines MIMD est plus complexe que celle des machines SIMD puisque c'est au programmeur de gérer explicitement la synchronisation entre les différentes entités de son application (processus ou threads). Il existe de nombreux outils permettant d'écrire des applications pour architectures MIMD. Il est possible, par exemple, de « décorer » un code séquentiel avec des directives de parallélisation que le compilateur interprète pour écrire le code parallèle correspondant.

Les figures 23 et 24 présente l'utilisation de l'un de ces outils : le standard OPENMP permettant d'écrire des applications multi-threadées.

La figure 23 représente le point de départ : une application séquentielle dans laquelle deux fonctions indépendantes sont appelées (elles ne mettent à jour aucune structure de données commune).

Le programmeur, constatant que les deux fonctions `f` et `g` sont indépendantes, va demander au compilateur de faire gérer leur appel par deux threads différents. Pour cela, il va décorer le code précédent pour produire celui de la figure 24.

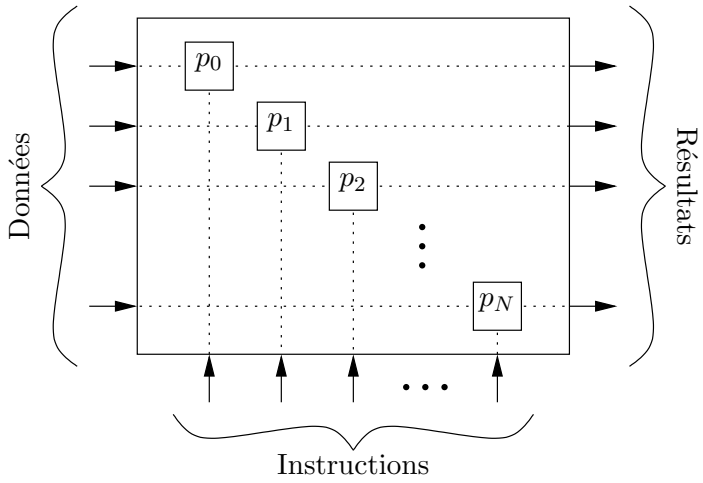

FIGURE 22 – Classe des modèles MIMD.

```
1 int a;
2 double b;
3
4 // Deux fonctions indépendantes.
5 a = f(5);
6 b = g(36.25);
7
8 // ... faire qqch avec a et b.
```

FIGURE 23 – Un code séquentiel potentiellement parallélisable.

```
int a;
double b;

// Deux fonctions indépendantes.
#pragma omp parallel
{
  #pragma omp sections
  {
#pragma omp section // Le premier thread.
    a = f(5);

#pragma omp section // Le deuxième thread.
    b = g(36.25);
  }

} // barrière de synchronisation implicite.

// ... faire qqch avec a et b.
```

FIGURE 24 – Sections parallèles en OPENMP.

Ne reste plus qu'à demander la compilation de ce nouveau code pour obtenir l'exécutable multi-threadé.

2.2 Classification mémoire

Les architectures parallèles peuvent également être classées selon la structure de leur mémoire. Cette classification est chronologique puisqu'elle retrace leur évolution dans le temps.

2.2.1 Mémoire partagée (années 1980-1990)

Dans ce type de machine, tous les processeurs accèdent à une mémoire commune via le réseau de communication. La figure 25 présente l'architecture générale d'une machine à mémoire partagée dans laquelle tout processeur p_i peut accéder à n'importe quel banc de mémoire m_j via le réseau. Ce dernier achemine à la fois données et instructions vers les processeurs.

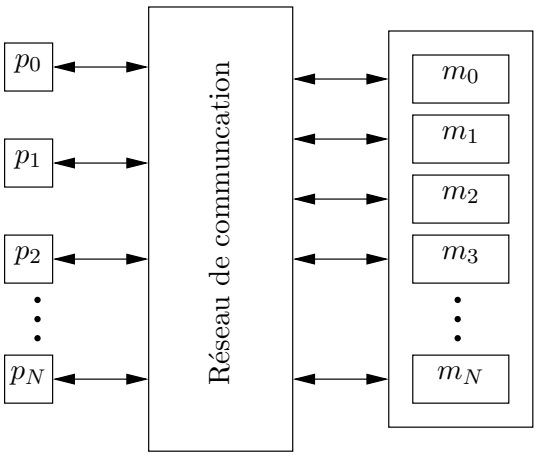

FIGURE 25 – Machine à mémoire partagée.

Le réseau de communication de ces machines est dynamique (un « *switch* ») c'est à dire que les routes partant des processeurs et menant aux bancs de mémoire évoluent dans le temps. Ils sont construits à partir d'une petite brique de base : le commutateur c_{22} possédant deux entrées, deux sorties et deux états de commande (Figure 26).

FIGURE 26 – Commutateur c_{22} et ses deux états de commande.

Un réseau dynamique est caractérisé par l'un des modes de fonctionnement suivants :
— **non bloquant** : une nouvelle connexion entre une entrée libre (un processeur) et une sortie libre (un banc de la mémoire) est toujours possible ;
— **ré-arrangeable** : une nouvelle connexion entre une entrée libre et une sortie libres est toujours possible mais celle-ci peut nécessiter une modification (re-routage) des connexions en cours ;
— **bloquant** : en fonction de connexions en cours, certaines connexions peuvent ne pas être établies du fait de l'absence de routes disponibles.

Le « *crossbar* » est un réseau dynamique non bloquant permettant de relier n entrées à m sorties. Un commutateur c_{22} est placé à chaque intersection de la ligne connectant le processeur et de la colonne connectant le banc. La figure 27 présente un *crossbar* 3×3.

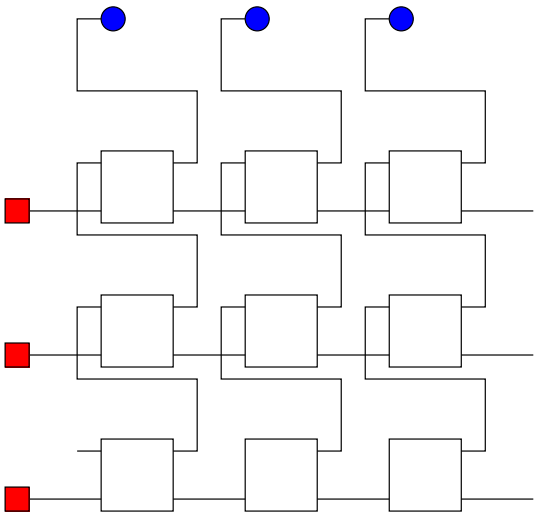

FIGURE 27 – *Crossbar* 3×3

Le nombre de briques de base c_{22} nécessaires à la réalisation d'un *crossbar* $n \times m$ est naturellement $n \times m$, ce qui interdit la réalisation de réseaux de grande dimension. Dans ce cas, le *crossbar* devient lui même une brique de base permettant la réalisation de réseaux dits « multi-étages ».

Les réseaux multi-étages augmentent les durées de connexion puisque la communication traverse cette fois-ci plusieurs étages constitués de *crossbars*. Cependant, le nombre de briques de base c_{22} nécessaires à la réalisation de tous ces *crossbars* est inférieur à celui nécessaire à la réalisation du *crossbar* de dimension équivalente à notre réseau multi-étages. La figure 28 présente un exemple de réseau 6×6 à trois étages appelé « CLOS$(2, 3, 3)$ ».

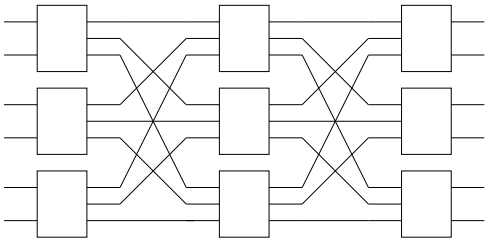

FIGURE 28 – Réseau CLOS$(2, 3, 3)$.

Le modèle de programmation naturel des machines à mémoire partagée est le modèle multi-threads. Lorsqu'un code exécutable est lancé sur la machine (un processus), celui-ci se voit attribuer un nombre maximum de processeurs par le système d'exploitation, nombre dépendant de son « niveau de privilège ». Les threads qui composent le processus sont alors répartis sur ce sous-ensemble de processeurs. Ils communiquent ensuite les uns avec les autres par lecture/écriture des données du processus implantées dans les bancs de mémoire de la machine.

Ce mode de communication entre processeurs pose un gros problème : celui de la cohérence des mémoires caches puisque comme pour un système séquentiel, il existe une découpe verticale de la mémoire c'est à dire une hiérarchie de mémoires caches menant de chaque processeur à chaque banc de mémoire.

Explicitons ce problème au travers d'une petite machine à mémoire partagée ne comportant que deux processeurs. Chaque processeur dispose d'un cache de niveau 1 (cache L_1) privé de petite capacité. Ces deux processeurs partagent un cache de niveau 2 (cache L_2) de plus grande capacité, celui-ci étant relié aux différents bancs de mémoire de la machine. Supposons (c'est un exemple) que :
— les caches L_1 peuvent accueillir 16 blocs de 64 octets ;
— le cache L_2 peut accueillir 1024 blocs de 64 octets.

La taille des blocs du cache L_2 étant de 64 octets, la mémoire est « paginée » en blocs de 64 octets, c'est à dire que lorsque l'un des processeurs accède à une instruction ou une donnée de la mémoire, c'est le bloc qui la contient qui est d'abord recopié dans le cache L_2. Ainsi, si *addr_a* représente l'adresse de cette instruction ou de cette donnée, son numéro de bloc ainsi que son *offset* à l'intérieur du bloc sont respectivement données par :

$$bloc\_num\_a = addr\_a \ / \ 64, \tag{2.8}$$
$$offset\_a = addr\_a \ \% \ 64, \tag{2.9}$$

où % désigne l'opérateur modulo.

Le bloc copié dans le cache L_2 l'est également dans le cache L_1 du processeur concerné.

Supposons à présent que *addr_a* soit l'adresse d'une donnée et que notre processeur en modifie la valeur initiale : il y a donc une incohérence entre la nouvelle valeur dans son cache L_1 et l'ancienne dans le cache L_2 et son banc de mémoire d'origine.

Dans un système séquentiel, cette incohérence ne pose aucun problème car la cohérence entre caches et mémoire est rétablie lorsque la donnée doit quitter le cache L_1 pour faire place à une autre (le processeur n'est en contact qu'avec son cache L_1).

Ce n'est malheureusement pas le cas dans un système parallèle. En effet, supposons que notre second processeur souhaite accéder à une donnée d'adresse *addr_b* et que $bloc\_num\_a = bloc\_num\_b$. Comme le bloc concerné se trouve déjà dans le cache L_2, il devrait être simplement copié dans le cache L_1 de ce processeur. Mais ce n'est pas possible car si tel était le cas, nos deux processeurs auraient deux versions

différentes d'un même bloc dans leurs caches L_1 respectifs c'est à dire que nous aurions maintenant :
— une incohérence entre cache L_1 et cache L_2/banc de mémoire pour le premier processeur ;
— une incohérence entre caches L_1 pour nos deux processeurs.

La seule solution consiste à rétablir immédiatement la cohérence entre les deux caches L_1 et le cache L_2 (la cohérence avec le banc de mémoire sera rétablie plus tard comme dans un système séquentiel). Il faut donc que :
— le bloc correspondant du cache L_2 soit mis à jour puis estampillé « *dirty* » ;
— ce bloc étant indiqué comme corrompu, il doit à nouveau être copié dans les caches L_1 de nos deux processeurs.

Un tel mécanisme (appelé *cache coherence mechanism*) est extrêmement lourd à mettre en œuvre puisqu'il doit être accolé au réseau de communication (le seul à « avoir tout vu » puisqu'il fait transiter à la fois les données et les instructions).

Ce double handicap (réseau multi-étages, cohérence des caches) est une barrière infranchissable pour la réalisation de machines à mémoire partagées massivement parallèles.

Notons que ce problème de cohérences de caches est à l'origine d'un problème bien connu en programmation multi-thread : le *false sharing*. Il s'agit d'une application multi-thread sémantiquement correcte mais aux performances calamiteuses car certains de ses threads travaillent sur des données trop proches en mémoire.

2.2.2 Mémoire distribuée (années 1990-2000)

Dans ce type de machine, chaque processeur dispose d'un banc de mémoire local qu'il est seul à pouvoir adresser. L'espace mémoire global de la machine consiste en la juxtaposition de ces espaces locaux. La figure 29 présente l'architecture générale d'une machine à mémoire distribuée dans laquelle chaque processeur p_i dispose de son propre banc de mémoire m_i et communique avec les autres processeurs via le réseau par échange de messages ne contenant que des données.

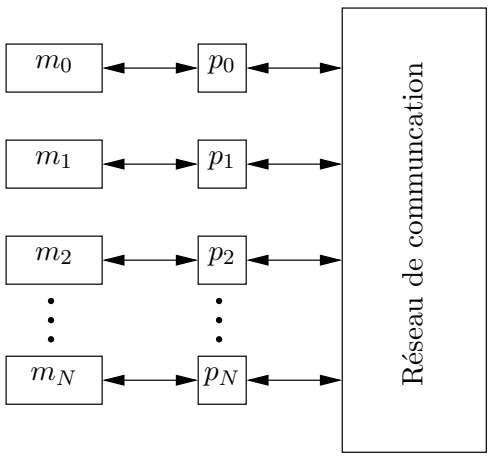

FIGURE 29 – Machine à mémoire distribuée.

Le réseau de communication de ces machines est statique c'est à dire que les routes menant d'un processeur à un autre sont figées. Un réseau statique est caractérisés par un couple (Δ, D) tel que :
— Δ est la connectivité moyenne des nœuds (processeurs) ;
— D est la plus longue distance (en bonds) entre deux nœuds.

Il existe de nombreuses topologies, toutes dédiés à un type d'application particulier.

Un réseau de n processeurs en anneau (Figure 30) est caractérisé par $\Delta = 2$ et $D = \frac{n}{2}$. Il s'agit d'un bus re-bouclé sur lui-même, ce qui assure une petite tolérance aux pannes en cas de rupture (unique) à un endroit du bus. Lorsque le nombre de nœuds augmente, il est possible de l'améliorer en ajoutant de nouveaux liens (cordes) entre certains nœuds.

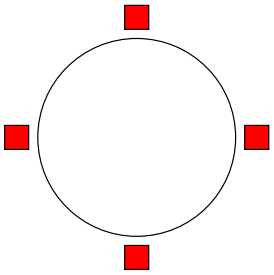

FIGURE 30 – Topologie en anneau.

Un réseau de $n \times n$ processeurs en grille (Figure 31) est caractérisé par $\Delta = 4$ et $D = 2 \times (n - 1)$. Il est possible de l'améliorer en augmentant la connectivité par de nouveaux liens sur la diagonale et en re-bouclant la grille sur elle-même (grille torique).

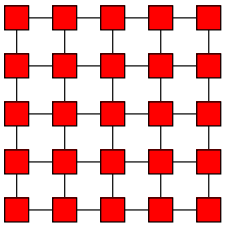

FIGURE 31 – Topologie en grille.

Un réseau de $n = 2^p$ processeurs en arbre (Figure 32) est caractérisé par $\Delta = 3$ et $D = 2 \times \log_2(n)$. Il est possible de l'améliorer en connectant ses feuilles sur un anneau (hyper-arbre).

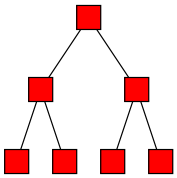

FIGURE 32 – Topologie en arbre.

La difficulté à laquelle on est confronté au moment de choisir une architecture à mémoire distribuée est cette grande variété de topologies. Par exemple, les grilles sont très adaptées au traitement d'image puisque dans ce type d'application, l'action effectuée sur un pixel dépend souvent de ses voisins immédiats. Par contre, elles sont beaucoup moins adaptées aux communications collectives (diffusion, centralisation, etc.) que les arbres. Or, il est rare de faire exécuter un seul type d'application sur une architecture parallèle. D'autre part, les connectivités absolues sont généralement préférées aux connectivités moyennes puisqu'une phase d'un algorithme peut ainsi être appliquées à tous les processeurs de la machine sans

avoir à tenir compte des cas particuliers.

La solution idéale serait un réseau à connectivité absolue permettant de reproduire toutes les topologies possibles : le programmeur n'aurait alors plus qu'à choisir sa topologie en indiquant les numéros de processeurs à utiliser directement dans son code. Un tel réseau est appelé « hyper-cube ».

Un hyper-cube de degré d (ou $\{d\}$-*cube*) est un réseau comportant 2^d nœuds, chaque nœud possédant exactement d voisins. Sa construction est récursive (ce qui s'avère pratique pour démontrer ses propriétés) : un $\{d\}$-*cube* se déduit de deux $\{d-1\}$-*cubes* en reliant chaque nœud de l'un au nœud de même position relative de l'autre. Ce réseau est caractérisé par $\Delta = d$ et $D = d$.

La figure 33 présente le processus de construction récursif d'un hyper-cube de degré 4.

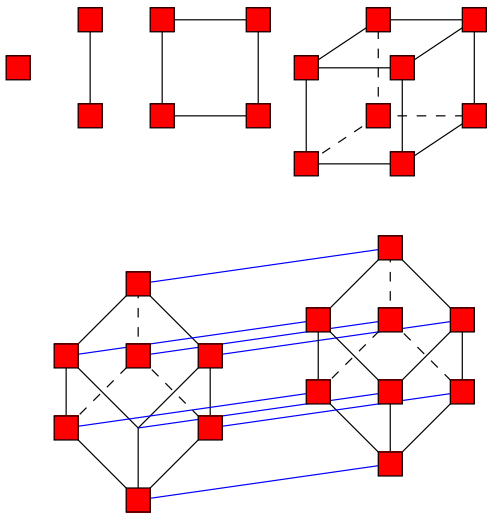

FIGURE 33 – Construction récursive d'un hyper-cube de degré 4.

Le modèle de programmation des machines à mémoire distribuée est le modèle multi-processus communiquants (également appelé modèle « *message passing* »). Comme pour une machine à mémoire partagée, un nombre maximum de processeurs est affecté à l'exécutable par le système d'exploitation selon son niveau de privilège. Les processus qu'il crée y sont répartis et communiquent entre eux par échanges de messages, ceux-ci ne contenant que des données. Ces communications sont assurées par des bibliothèques de primitives synchrones et asynchrones, les plus connues étant PARALLEL VIRTUAL MACHINE ou le standard MESSAGE PASSING INTERFACE.

Le banc de mémoire associé à chaque processeur étant privé, il contient à la fois les données et les instructions du processus qu'il exécute. Si celui-ci à besoin d'autres données alors il les reçoit par messages en provenance des autres processus de l'application. Par conséquent, le problème de la cohérence des caches propre aux architectures à mémoire partagée disparaît lorsque la mémoire devient distribuée. Dès lors, dotée d'un modèle d'exécution MIMD, une machine à mémoire distribuée peut être massivement parallèle.

2.2.3 Mémoire mi-distribuée, mi-partagée (années 2000-)

La fin des années 90 marque aussi celle des grands programmes gouvernementaux, la guerre froide étant terminée. Les grands constructeurs sont alors en difficulté et doivent trouver de nouveaux débouchés. Il faut alors se tourner vers le monde des moyennes et grandes entreprises mais renoncer à leur proposer systématiquement des super-calculateurs.

Cette mutation commence avec l'apparition des SMP (*Symmetric Multi-Processor*). Il s'agissait à l'origine d'une machine à mémoire partagée dotée d'un petit nombre de processeurs (de 4 à 8) et du réseau de communication le plus simple qui soit : le bus. Aujourd'hui, ce bus est de plus en plus souvent remplacé par un *crossbar* ou un réseau d'alignement, ce qui permet d'intégrer un plus grand nombre de processeurs (de 16 à 32 voire 64).

Les caractéristiques principales du SMP sont :
— un coût très faible (tous vos ordinateurs portables multi-cœurs sont des SMP) ;
— un temps d'accès à la mémoire identique pour tous ses processeurs (d'où le qualificatif *symmetric*).

Pour construire des machines plus importantes, il faut utiliser les SMP comme briques de base et les interconnecter : nous parlons alors de « grappe SMP (*SMP cluster*) ». La figure 34 en présente l'architecture générale.

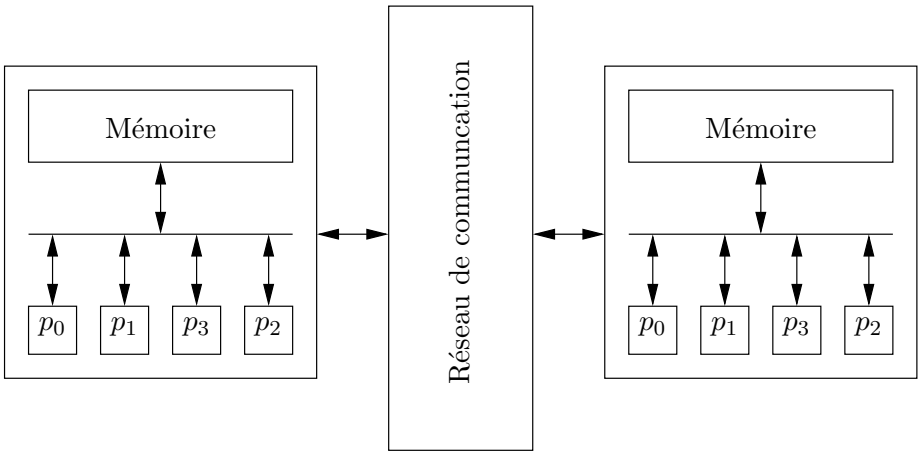

FIGURE 34 – Architecture de type « *SMP cluster* ».

Il existe deux types de grappes SMP selon qu'elles soient destinées à un usage généraliste ou un usage « calcul intensif ».

Dans le cas d'un usage généraliste, le public ciblé est celui des entreprises moyennes ou grandes. Il s'agit de leur fournir une machine parallèle évolutive c'est à dire qu'il est possible de rajouter progressivement des processeurs et de la mémoire sans avoir à changer de machine. Celle-ci se présente donc sous la forme d'un châssis équipé de plusieurs *slots* destinés à accueillir des cartes SMP (généralement quadri-processeurs). Si le nombre de cartes actuel s'avère insuffisant alors il suffit simplement d'en acheter d'autres. De telles machines peuvent atteindre 128 processeurs.

La vocation de ces architectures dites CC-NUMA (*Cache Coherence, Non Uniform Memory Access*) n'est pas le calcul intensif : elles doivent donc être très facile d'accès à un public néophyte (autant utilisateur que programmeur), c'est à dire le propre des machines à mémoire partagée.

Par conséquent, une architecture CC-NUMA possède une mémoire physiquement distribuée (Figure 34) mais logiquement partagée pour son utilisateur. Pour ce faire, le réseau de communication intègre un mécanisme d'adressage global permettant à chaque SMP d'accéder à la mémoire des autres SMP (le temps d'accès aux mémoires distantes est de deux à trois fois plus long que le temps d'accès à la mémoire locale, d'où le qualificatif NUMA). Ce réseau incorpore également un mécanisme permettant de garantir la cohérence des mémoires caches (d'où le qualificatif CC).

Dans le cas d'un super-calculateur de type grappe SMP, il s'agit de reprendre l'architecture d'une machine à mémoire distribuée et de remplacer les couples processeur/banc de mémoire par des SMP (d'où une augmentation vertigineuse du nombre de processeurs et de bancs de mémoire). Ce type d'architecture est celle des super-calculateurs parallèles actuels (qui peuvent même combiner plusieurs types de processeurs). Leur modèle de programmation est un peu délicat puisqu'il est à la fois multi-processus communiquants (sur les SMP de la machine) et multi-threads (à l'intérieur de chaque SMP).

3 Algorithmique parallèle

L'étude de la complexité des algorithmes parallèles nécessite la définition d'un modèle de calcul. Dans le cas d'une machine séquentielle, il s'agit du modèle RAM (*Random Access Machine*). Dans le cas d'une machine parallèle, plusieurs modèles peuvent être définis. De fait, il existe des notions différentes de complexité algorithmique suivant le modèle utilisé.

La thèse du calcul parallèle est basée sur l'idée d'une relation entre la durée de calcul sur les machines parallèles et l'espace de calcul, c'est à dire l'espace mémoire, sur les machines séquentielles. Elle stipule que tout problème pouvant être résolu sur une machine séquentielle « raisonnable » à l'aide d'un espace de calcul polynomial peut être résolu en temps polynomial sur une machine parallèle « raisonnable » et inversement.

La définition de la notion de machine séquentielle ou parallèle « raisonnable » est ardue. Un consensus a cependant été établi par les théoriciens : la thèse de l'invariance. Celle-ci stipule que des machines « raisonnables » peuvent se simuler entre elles avec au plus un accroissement polynomial en temps et une multiplication constante de l'espace.

La complexité algorithmique séquentielle est exprimée en nombre d'opérations élémentaires. Dans le cas de l'algorithmique parallèle, cette complexité peut être exprimée en :
- **temps** : il s'agit ici de sommer la durée des opérations élémentaires et celle du routage de données. La complexité en temps dépend du réseau de communication ;
- **nombre de processeurs** : cette complexité est fonction de la taille du problème à résoudre.

3.1 Modèle de calcul PRAM

Une *Parallel Random Access Machine* (PRAM) est un ensemble de P processeurs séquentiels indépendants (*Random Access Machines*) pourvus chacun de registres, d'une mémoire locale et communiquant via une mémoire partagée de N bancs. Les grandeurs P et N sont fonctions de la taille du problème à résoudre.

Les opérations fondamentales sont exécutées de façon atomique (en une unité de temps c'est à dire qu'elles ne peuvent être interrompues) et synchrone. Ces opérations sont :
- le calcul en mémoire locale ;
- la lecture en mémoire partagée ;
- l'écriture en mémoire partagée ;
- l'attente symbolisée par l'exécution de l'instruction spéciale `nop` (*no operation*). Un processeur exécutant cette instruction est dit inactif.

Le modèle PRAM est un modèle généraliste trop puissant. De fait, des restrictions peuvent être plus réalistes d'un point du vue technologique.

3.1.1 Restriction EREW

Dans cette variation, la lecture et l'écriture sont toutes deux exclusives (*Exclusive Read*, *Exclusive Write*). De fait, un seul et unique processeur peut accéder à une cellule de mémoire partagée pour la lire ou l'écrire.

3.1.2 Restriction CREW

Dans cette variation, la lecture est concurrente (*Concurrent Read*) tandis que l'écriture est exclusive. De fait, une cellule de mémoire partagée peut être lue simultanément par plusieurs processeurs mais ne peut être écrite que par un seul et unique processeur, toute tentative d'écriture simultanée entraînant un blocage.

3.1.3 Restriction CRCW

Dans cette variation, la lecture et l'écriture sont toutes deux concurrentes. Plusieurs méthodes de gestion des conflits d'écriture sont envisageables :

- **priorité** : une priorité statique est associée à chaque processeur. Le processeur actif de plus forte priorité écrit la cellule de mémoire partagée ;
- **commune** : la machine se bloque si les processeurs actifs n'écrivent pas la même valeur dans la cellule de mémoire partagée ;
- **commune à erreur** : le contenu de la cellule de mémoire partagée reste inchangé si les processeurs actifs n'écrivent pas la même valeur. Cette méthode garantit une absence de blocage ;
- **commune restreinte** : les écritures sont autorisées dans certaines cellules si et seulement si les processeurs actifs écrivent la valeur 1 ;
- **collision** : quelles que soient les valeurs écrites, un symbole spécial est écrit dans la cellule de mémoire partagée ;
- **collision$^+$** : commune à erreur avec un symbole spécial écrit dans la cellule de mémoire partagée si les valeurs diffèrent ;
- **collision tolérante** : pas de modification de la cellule de mémoire partagée en cas d'écritures concurrentes ;
- **collision robuste** : le contenu de la cellule de mémoire partagée n'est pas spécifié en cas d'écritures concurrentes (non déterminisme) ;
- **arbitraire** : en cas d'écritures concurrentes, un processeur actif est choisi au hasard (non déterministe) ;
- **combinaison** : en cas d'écritures concurrentes, la valeur de la cellule de mémoire partagée est une combinaison des valeurs écrites. Les opérations concernées possèdent des propriétés d'associativité et de commutativité.

3.1.4 Restriction CROW

Cette variation (*Concurrent Read, Owner Write*) permet de modéliser les machines SIMD à mémoire distribuée :

- chaque cellule de mémoire a un processeur propriétaire ;
- un processeur n'écrit que dans une cellule qui lui appartient ;
- la fonction de propriété peut éventuellement varier dans le temps.

3.1.5 Exemple de la recherche d'un minimum

Supposons que nous écrivions un algorithme de recherche d'un minimum dans un tableau X contenant $N = 2^q$ éléments.

Le meilleur algorithme séquentiel pour résoudre ce problème consiste à considérer comme candidat potentiel le premier élément de ce tableau. Nous parcourons ensuite les éléments suivants. À chaque fois que nous rencontrons un élément plus petit que notre candidat potentiel alors cet élément devient notre nouveau candidat. La complexité de cet algorithme est linéaire c'est à dire en $\mathcal{O}(N)$.

Supposons à présent que nous disposions d'une machine parallèle de type PRAM-EREW à N processeurs $P_{i\geq 1}$. L'algorithme 5 est alors le meilleur pour résoudre notre problème. Il se caractérise par :

- une complexité $\mathcal{O}(log_2(N))$ en temps puisque la boucle parallèle intérieure est exécutée en temps constant, c'est à dire en $\mathcal{O}(1)$ et elle l'est $log_2(N)$ fois ;
- une complexité $\mathcal{O}(N)$ en nombre de processeurs.

3.1.6 Lemme de Brent

L'étude de la complexité d'un algorithme parallèle est réalisée dans un cadre théorique dans lequel le nombre de processeurs est illimité (cas de l'algorithme précédent) : nous parlons alors de parallélisme non borné.

Algorithme 5 Recherche d'un minimum sur machine PRAM-EREW

Entrées : $X(i)$ en mémoire locale de P_i
Sorties : P_1 contient $min(X(1:N))$

1: **for all** $j := 1$ **to** N **do**
2: P_j écrit $X(j)$ en mémoire partagée
3: **end for**
4: **for** $j := 0$ **to** $\lceil log_2(N) \rceil$ - 1 **do**
5: **for all** $i := 1$ **to** N **step** 2^{j+1} **do**
6: P_i lit $X(i + 2^j)$ en mémoire partagée
7: **if** $X(i + 2^j) < X(i)$ **then**
8: $X(i) := X(i + 2^j)$
9: P_i écrit $X(i)$ en mémoire partagée
10: **end if**
11: **end for**
12: **end for**

L'implémentation de cet algorithme doit être réalisée dans un cadre pratique dans lequel le nombre de processeurs est borné : nous parlons alors de parallélisme borné.

Le passage du parallélisme non borné au parallélisme borné introduit la notion de granularité : nous parlons de granularité fine lorsque le nombre de processeurs est supérieur ou égal au nombre de données et de granularité grossière dans le cas contraire. Le passage d'une granularité fine à une granularité grossière s'appuie sur le lemme de Brent.

Lemme de Brent : soit un modèle PRAM donné. Si un algorithme parallèle nécessite T unités de temps et Q opérations pour résoudre un problème donné, alors il existe un algorithme avec P processeurs qui résout le même problème en un temps $\mathcal{O}(T + \frac{Q}{P})$.

Démonstration : soit Q_i le nombre d'opérations exécutées au pas de temps i par l'algorithme initial. Nous avons : $\sum_{i=1}^{T} Q_i = Q$. Avec P processeurs, le pas de temps i peut être sub-divisé en $\lceil \frac{Q_i}{P} \rceil$ pas. Comme $\lceil \frac{Q_i}{P} \rceil \leq \frac{Q_i}{P} + 1$, nous obtenons le résultat en sommant les étapes.

3.2 Performances d'une application

La qualité d'une application séquentielle est généralement évaluée en fonction de sa durée d'exécution. La qualité d'une application parallèle est évaluée de manière plus complexe en fonction de trois facteurs appelés accélération (*speedup*), efficacité (*efficiency*) et élasticité (*scalability*).

3.2.1 Facteur d'accélération

Pour un problème de taille (espace mémoire) N, le facteur d'accélération est défini comme le rapport des durées utilisés par le meilleur algorithme séquentiel et le meilleur algorithme parallèle pour résoudre ce problème. Si $t_1(N)$ désigne la meilleure durée séquentielle et $t_P(N)$ la meilleure durée parallèle sur P processeurs, alors le facteur d'accélération est défini par :

$$s(N, P) = \frac{t_1(N)}{t_P(N)}. \tag{3.10}$$

La figure 35 présente les trois cas possibles pour le facteur d'accélération. Celui-ci peut être :
— linéaire avec $s(N, P) = P$. Il s'agit d'un cas idéal dans la mesure où les processeurs ne font que du calcul et pas de communications ;
— sub-linéaire avec $s(N, P) < P$. Il s'agit du cas le plus courant dans la mesure où les processeurs calculent et communiquent entre eux ;

— sur-linéaire avec $s(N,P) > P$. Il s'agit d'un cas plus rare puisque l'application présente des comportements très différents selon le nombre de processeurs utilisés.

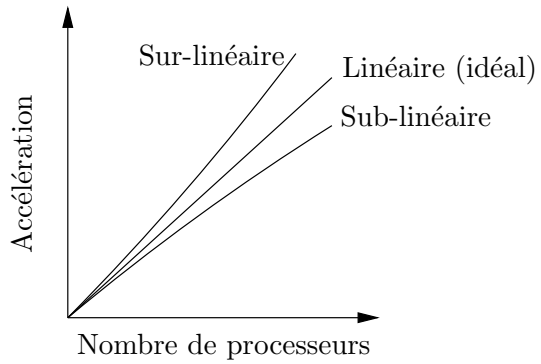

FIGURE 35 – Facteur d'accélération.

3.2.2 Facteur d'efficacité

Cette métrique permet d'établir le taux d'utilisation des processeurs en normalisant le facteur d'accélération obtenu par le facteur d'accélération idéal. Elle est définie comme :

$$e(N,P) = \frac{s(N,P)}{P} = \frac{t_1(N)}{P \times t_P(N)}. \tag{3.11}$$

La figure 36 présente les trois cas possibles pour le facteur d'efficacité. La qualité d'une application parallèle est d'autant meilleure que son efficacité est proche de l'unité, c'est à dire que son accélération est proche du cas idéal.

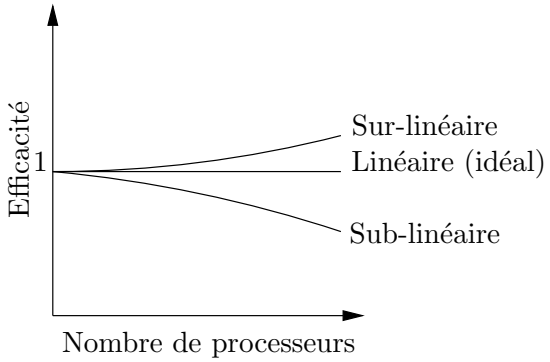

FIGURE 36 – Facteur d'efficacité.

3.2.3 Facteur d'élasticité

Cette métrique représente la capacité de l'application à traiter des problèmes de tailles de plus en plus importantes avec un nombre de processeurs en rapport avec ces tailles. Il s'agit en fait du facteur d'efficacité mesuré en faisant varier proportionnellement la taille du problème et le nombre de processeurs. Plus l'efficacité est proche du cas idéal et plus l'application possède une bonne élasticité.

3.3 Loi d'Amdhal (1967)

Cette loi permet d'établir une borne supérieure pour le facteur d'accélération sur une machine à mémoire partagée. Elle se base sur le fait que toute application parallèle contient une zone de code séquentielle qui ne peut être parallélisée.

Pour un problème de taille N, notons $t_{\text{séq}}(N)$ la durée d'exécution de la zone de code séquentielle et $t_{\text{par}}(N)$ la durée d'exécution de la zone de code parallèle. La durée d'exécution de cette application peut alors s'écrire sous la forme :

$$t_1(N) = t_{\text{séq}}(N) + t_{\text{par}}(N), \tag{3.12}$$

sur une machine mono-processeur et sous la forme :

$$t_P(N) = t_{\text{séq}}(N) + \frac{t_{\text{par}}(N)}{P}, \tag{3.13}$$

sur une machine parallèle à P processeurs.

Si $f(N)$ désigne la proportion de la zone de code non-parallélisable dans l'application complète, nous pouvons également écrire :

$$\begin{aligned} t_{\text{séq}}(N) &= f(N) \times t_1(N), & (3.14)\\ t_{\text{par}}(N) &= \big\{1 - f(N)\big\} \times t_1(N). & (3.15)\\ & & (3.16) \end{aligned}$$

Le facteur d'accélération de cette application est donné par :

$$s(N,P) = \frac{t_{\text{séq}}(N) + t_{\text{par}}(N)}{t_{\text{séq}}(N) + \frac{t_{\text{par}}(N)}{P}} \leq \frac{t_1(N)}{t_{\text{séq}}(N)}. \tag{3.17}$$

Nous pouvons alors écrire :

$$s(N,P) = \frac{1}{f(N) + \frac{1-f(N)}{P}} \leq \frac{1}{f(N)}, \tag{3.18}$$

ce qui démontre que le facteur d'accélération est borné supérieurement par une valeur indépendante du nombre de processeurs et de la structure de la machine. En conséquence, si la zone de code non-parallélisable représente 15% de l'application complète, le facteur d'accélération ne peut excéder $6,67$ quelque soit le nombre de processeurs utilisés.

3.4 Loi de Gustafson (1988)

Cette loi permet d'établir une borne inférieure pour le facteur d'accélération sur les machines à mémoire distribuée.

On considère ici la classe des problèmes dont la durée de calcul et l'espace mémoire requit croissent avec la taille de l'instance considérée. Dans le cas d'une machine à mémoire partagée, la taille de la mémoire n'intervient pas puisque l'instance du problème peut y être résolue avec un ou plusieurs processeurs. De fait, la loi d'Amdhal est applicable. À l'inverse, dans le cas d'une machine à mémoire distribuée, la taille du banc de mémoire de chaque processeur est importante puisque l'augmentation du nombre de processeurs entraîne une augmentation de la taille de la mémoire et donc la possibilité de résoudre des problèmes de tailles plus importantes.

Considérons un algorithme parallèle à P processeurs permettant de résoudre un problème de taille maximale N. Le facteur d'accélération correspondant est alors :

$$S(N,P) = \frac{t_1(N)}{t_P(N)} = \frac{t_{\text{séq}}(N) + P \times t_{\text{par}}(N)}{t_{\text{séq}}(N) + t_{\text{par}}(N)}. \tag{3.19}$$

Comme $t_P(N) = t_{\text{séq}}(N) + t_{\text{par}}(N) = 1$, nous pouvons alors écrire :

$$S(N,P) = t_{\text{séq}}(N) + P \times t_{\text{par}}(N) \geq P \times t_{\text{par}}(N), \tag{3.20}$$

ce qui démontre que le facteur d'accélération est borné inférieurement par une expression croissant avec le nombre de processeurs. Ainsi, si la zone de code non-parallélisable représente 50% de l'application complète, le facteur d'accélération est supérieur à $0,5 \times P$.

Dans la pratique, les quantités $t_{\text{séq}}(N)$ et $t_{\text{par}}(N)$ sont inconnues. Il est cependant possible d'évaluer le facteur d'accélération en considérant :

— $t_{max}(1)$ le temps moyen de calcul pour exécuter une opération élémentaire d'un algorithme séquentiel résolvant le plus grand problème qu'il est possible de stocker sur un processeur ;
— $t_{max}(P)$ le temps moyen de calcul pour exécuter une opération élémentaire d'un algorithme parallèle résolvant le plus grand problème qu'il est possible de stocker sur P processeurs.

En supposons (hypothèse purement théorique) que notre problème de taille maximale N puisse être stocké sur un seul processeur, nous avons alors :

$$S(N,P) = \frac{t_1(N)}{t_P(N)} = \frac{N \times t_{max}(1)}{N \times t_{max}(P)} = \frac{t_{max}(1)}{t_{max}(P)}. \tag{3.21}$$