Université de Caen Basse-Normandie
U.F.R. des Sciences
Département d'informatique

Bâtiment Sciences 3 - Campus Côte de Nacre
F-14032 Caen Cédex, FRANCE

Support d'enseignement

Niveau	**L3**
Parcours	**Informatique**
Unité d'enseignement	**INF5E - Informatique Industrielle**
Élément constitutif	**INF5E2 - Parallélisme**
Création **Révisions**	1er **Septembre** 2005 2 **Décembre** 2007 9 **Mai** 2012 22 **Octobre** 2014 17 **Aout** 2015 5 **Juillet** 2020
Responsables	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr** **Alexandre Niveau** **alexandre.niveau@unicaen.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.

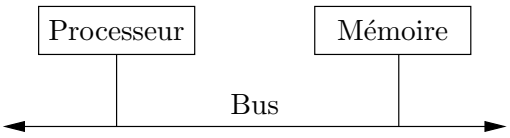

Figure 1 – Modèle d'exécution de J. Von Neuman (1946).

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.

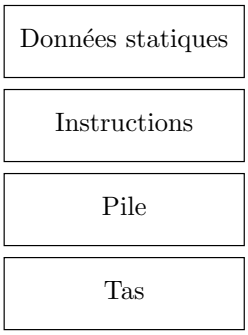

FIGURE 4 – Structure simplifiée d'un fichier exécutable.

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.

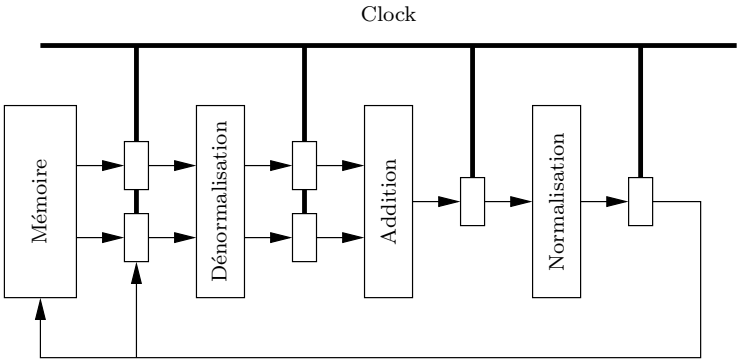

FIGURE 7 – Additionneur en virgule flottante.

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.

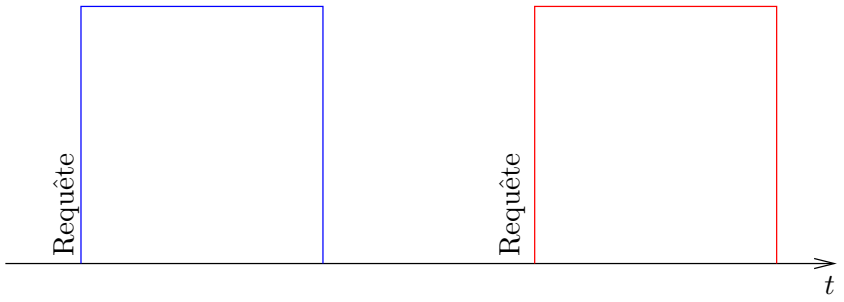

FIGURE 8 – Système mono-banc : les requêtes à la mémoire sont sérialisées.

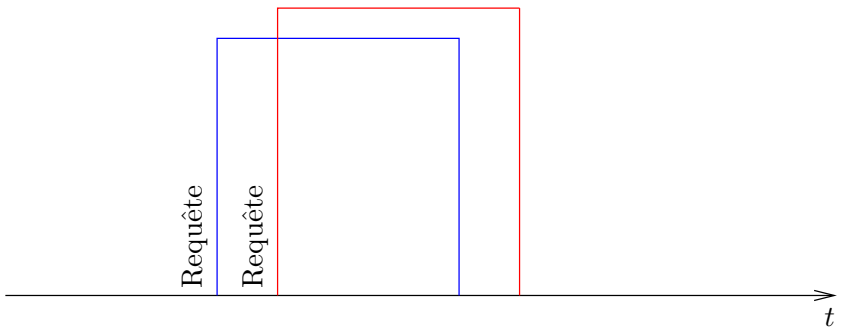

FIGURE 9 – Système multi-bancs : les requêtes à la mémoire sont pipelinées.

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

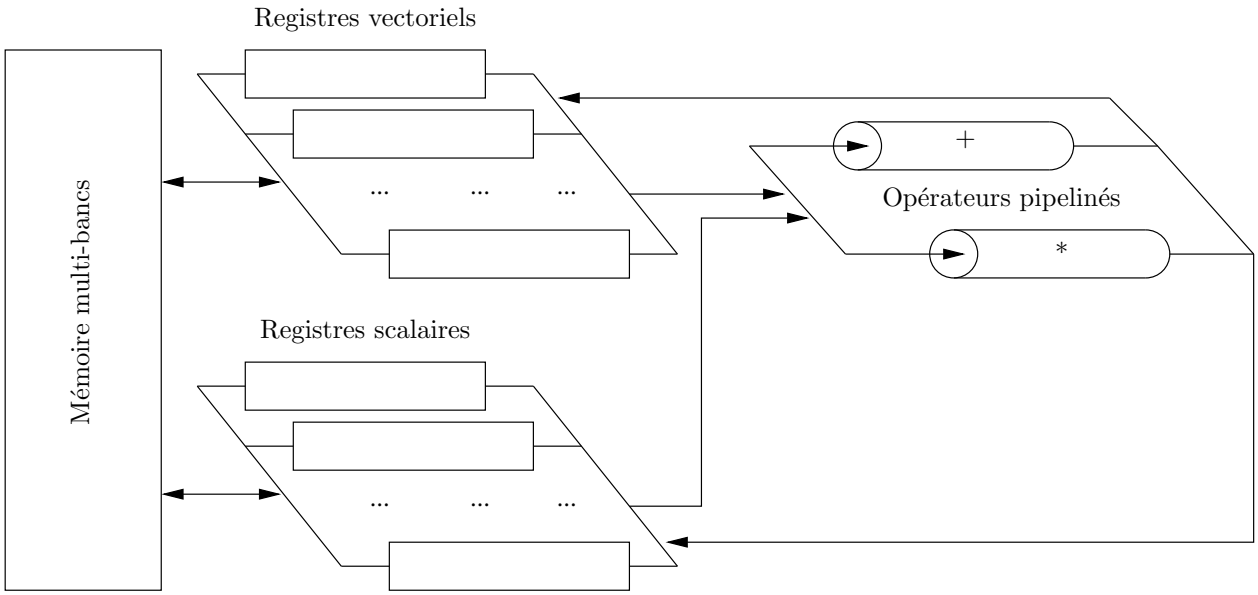

FIGURE 13 – Une architecture vectorielle.

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).

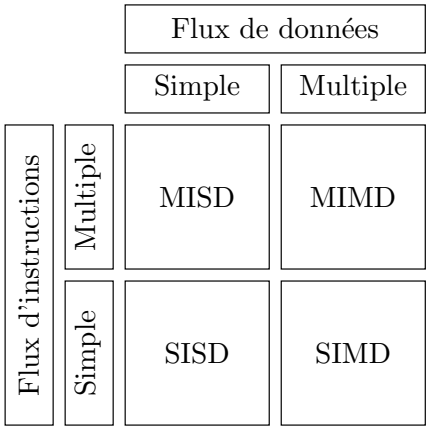

FIGURE 14 – Classification de J. FLYNN (1972).

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).

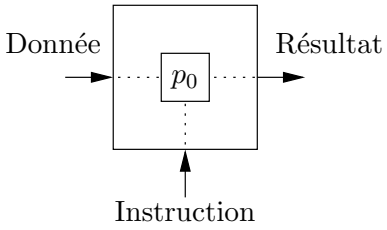

FIGURE 15 – Classe des modèles SISD.

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`) ;

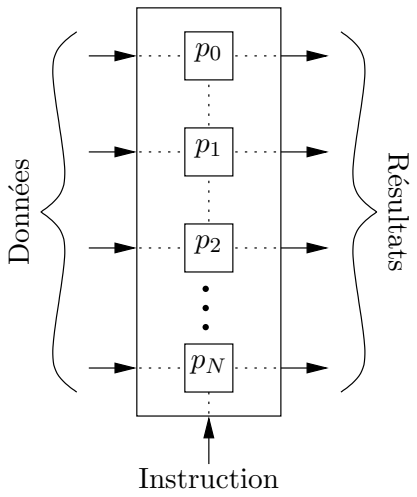

FIGURE 16 – Classe des modèles SIMD.

— 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.

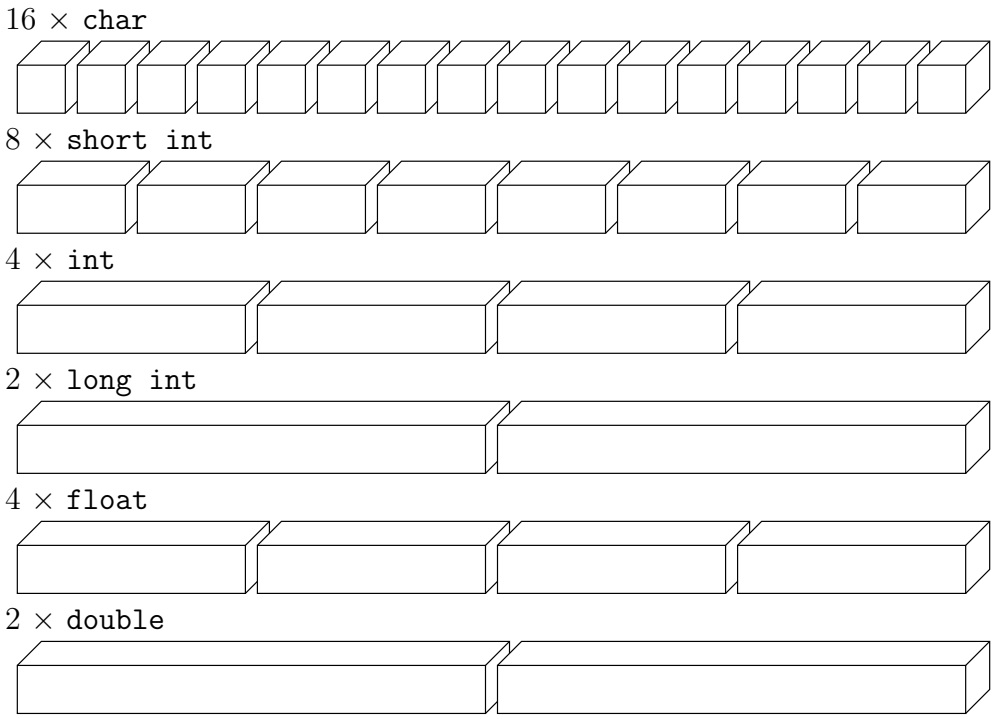

FIGURE 17 – L'un des huit registres 128 *bits* `XMMx` du *Pentium IV*.

```
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 ».

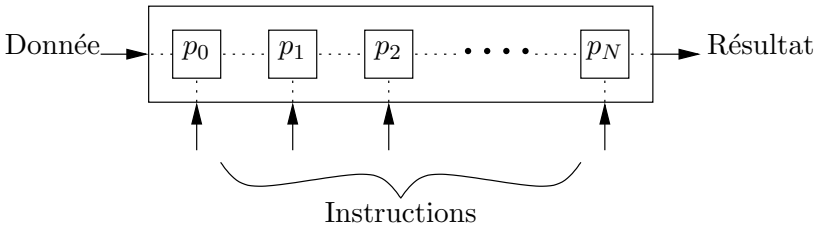

FIGURE 21 – Classe des modèles MISD.

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.

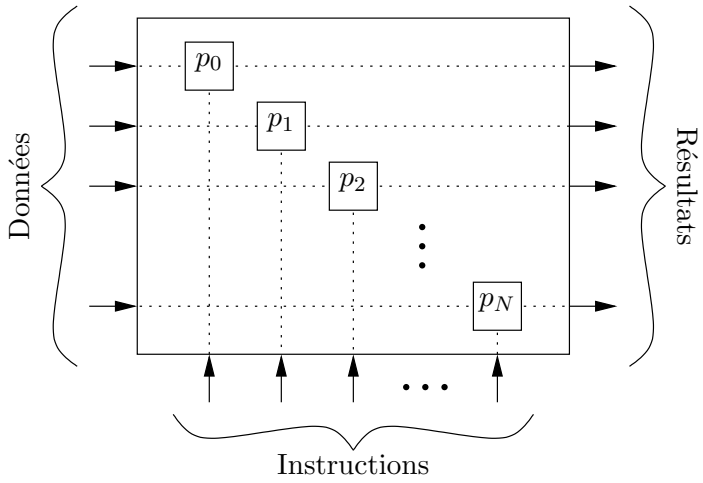

FIGURE 22 – Classe des modèles MIMD.

```
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.

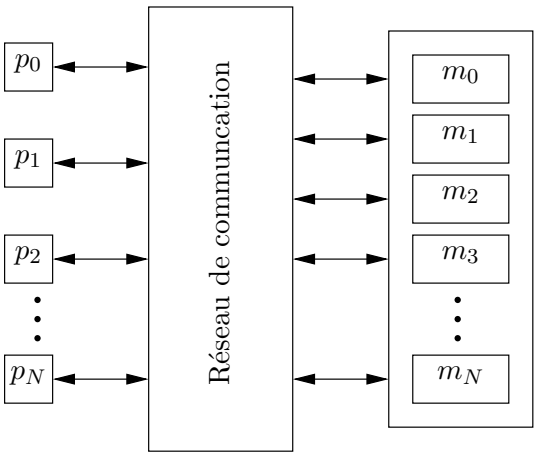

FIGURE 25 – Machine à mémoire partagée.

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.

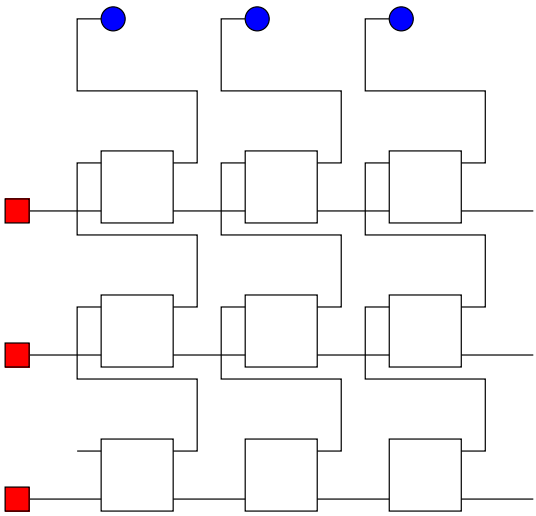

FIGURE 27 – *Crossbar* 3×3

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$(2, 3, 3)$ ».

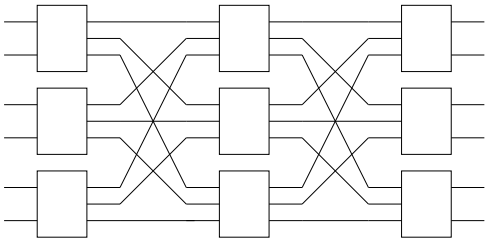

FIGURE 28 – Réseau CLOS$(2, 3, 3)$.

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.

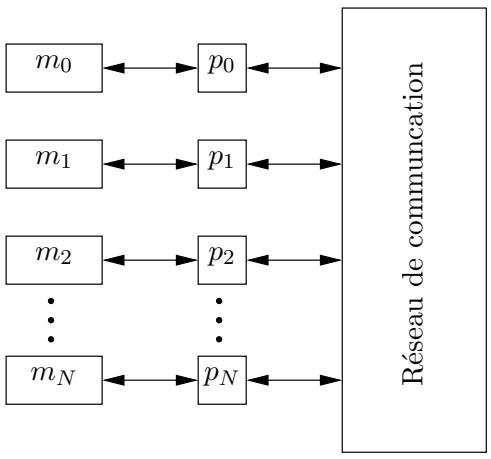

FIGURE 29 – Machine à mémoire distribuée.

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.

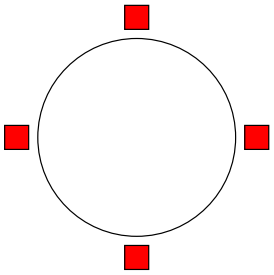

FIGURE 30 – Topologie en anneau.

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).

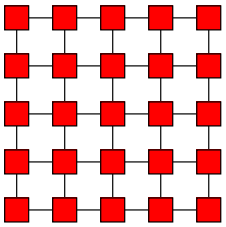

FIGURE 31 – Topologie en grille.

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).

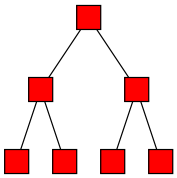

FIGURE 32 – Topologie en arbre.

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.

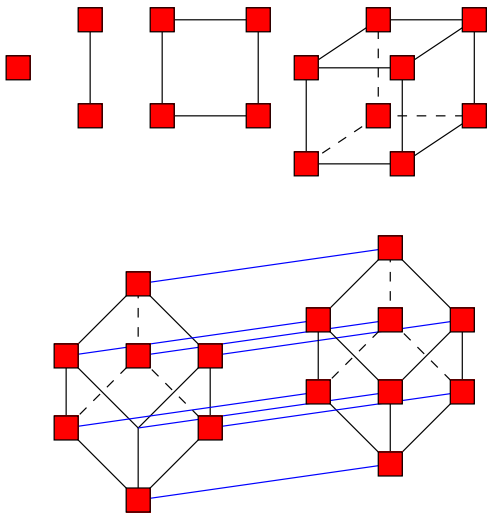

FIGURE 33 – Construction récursive d'un hyper-cube de degré 4.

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.

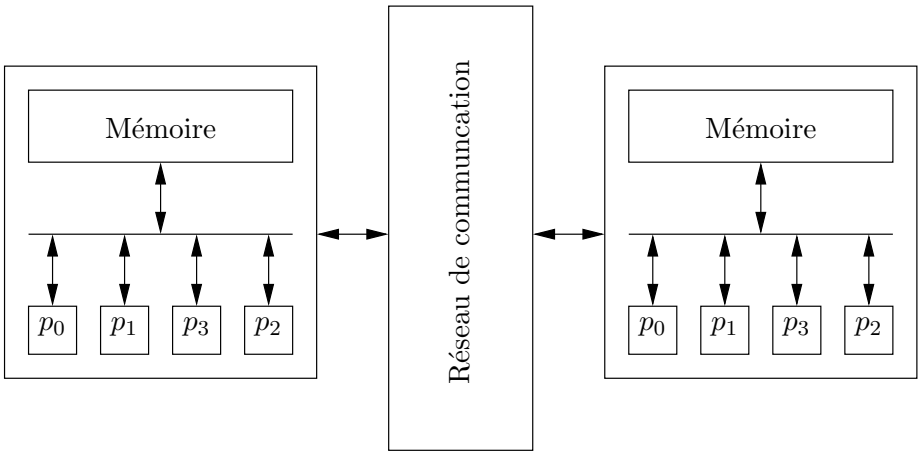

FIGURE 34 – Architecture de type « *SMP cluster* ».

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.

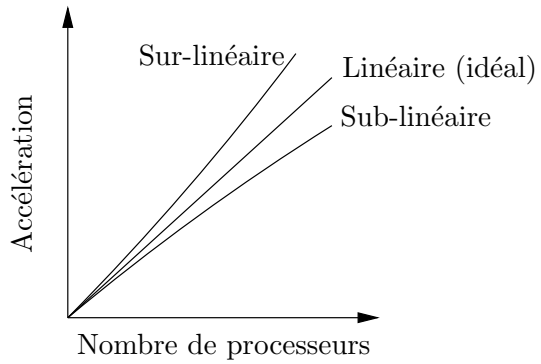

FIGURE 35 – Facteur d'accélération.

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.

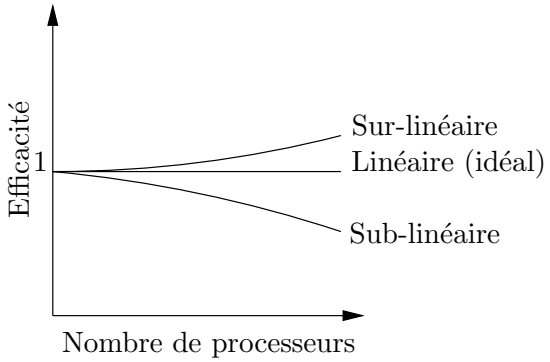

FIGURE 36 – Facteur d'efficacité.

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 :

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

4 Multi-threading en OpenMP (C et C++)

Le modèle de programmation naturel d'une machine SMP est le modèle à parallélisme de processus légers ou multi-threading. Dans ce dernier, un processus lourd est composé de processus légers ou threads, c'est à dire des groupes d'instructions qui s'exécutent de manière séquentielle ou concurrente sur l'ensemble des processeurs de la machine (modèle de programmation dit « à parallélisme de contrôle »).

Les threads communiquent entre eux par lecture/écriture des données du processus père. À tout thread est associée une pile système dans laquelle sont créées ses données privées c'est à dire les paramètres, les résultats et les données locales de ses fonctions.

Bien que ce modèle permette de tirer pleinement partie des possibilités de la machine, il est perçu comme relativement difficile à mettre en œuvre par les non initiés : risques d'inter-blocages, mécanismes d'exclusion mutuelle garantissant la cohérence des écritures, mécanismes de synchronisation, etc.

Une manière de contourner cette difficulté consiste à ne pas écrire directement une application multi-threadée via les bibliothèques de threads standardisés Posix mais plutôt à transformer un code séquentiel existant en un code multi-threadé en y ajoutant des directives destinées au compilateur, ce dernier se chargeant alors de la génération du code multi-threadé correspondant.

Une telle manière de procéder offre des avantages importants :
— l'application est développée en tant qu'application séquentielle, ce qui facilite sa mise au point ;
— les connaissances nécessaires se bornent aux grandes lignes de fonctionnement d'une machine SMP ainsi qu'aux directives proposées et à leurs effets. De fait, tous les publics sont concernés et non plus seulement celui des spécialistes.

Les inconvénients ne sont cependant pas négligeables :
— le seul parallélisme exploitable est celui de l'application séquentielle. Or, un bon algorithme séquentiel se révèle bien souvent être un mauvais algorithme parallèle. De fait, le programmeur doit posséder une certaine connaissance de la machine, notamment du fonctionnement des mémoires caches, afin d'exploiter pleinement le parallélisme. En conséquence, lorsqu'il conçoit son application séquentielle, il doit le faire en prévision de sa parallélisation ;
— le compilateur étant responsable de l'application des directives, l'efficacité de l'application est liée à la qualité de ce dernier.

Ainsi, toute parallélisation basée sur l'utilisation de directives doit être vue comme une façon de tirer un minimum de performances d'une machine SMP.

OpenMP est un standard pour la parallélisation des applications C, Fortran ou C++ par le biais de directives. Adopté en octobre 1997 par une majorité d'industriels et de constructeurs, ses spécifications appartiennent à l'OpenMP Archictecture Review Board, seul organisme habilité à les faire évoluer.

4.1 Éléments de base

4.1.1 Régions parallèles et séquentielles

Le nombre de threads est, par défaut, égal au nombre de processeurs (ou cœurs) de la machine. L'utilisateur peut néanmoins choisir le nombre de threads nécessaires à l'exécution de son application par le biais de la variable d'environnement `OMP_NUM_THREADS`. Le programmeur peut également imposer ce nombre par le biais de la fonction `omp_set_num_threads` tandis que la fonction `omp_get_num_threads` lui permet de connaître le nombre de threads demandés.

Le processus père compte pour un thread : il est appelé thread maître tandis que les autres sont appelés threads fils. Les threads sont affectés aux différents processeurs par le système d'exploitation. Si le nombre de threads indiqué par l'utilisateur est supérieur au nombre de processeurs, plusieurs threads sont affectés

à un même processeur.

Une application OpenMP est caractérisée par une alternance de régions séquentielles et parallèles. Une région séquentielle est exécutée par le thread maître tandis qu'une région parallèle est exécutée par l'ensemble des threads.

La figure 37 présente le principe des régions séquentielles et parallèles dans le cas d'un bi-processeur.

Toute zone de code ne faisant pas l'objet d'une directive désigne une région séquentielle (zones 1 et 3 dans l'exemple de la figure 37).

À l'inverse, toute zone de code à l'intérieur d'un bloc `omp parallel` désigne une région parallèle (zone 2 dans l'exemple de la figure 37). Dans cet exemple particulier, les deux threads se synchronisent à la sortie de la région parallèle puis seul le thread maître poursuit l'exécution de la dernière région séquentielle. Le modèle d'exécution utilisé par OpenMP est donc le modèle classique « *fork-join* ».

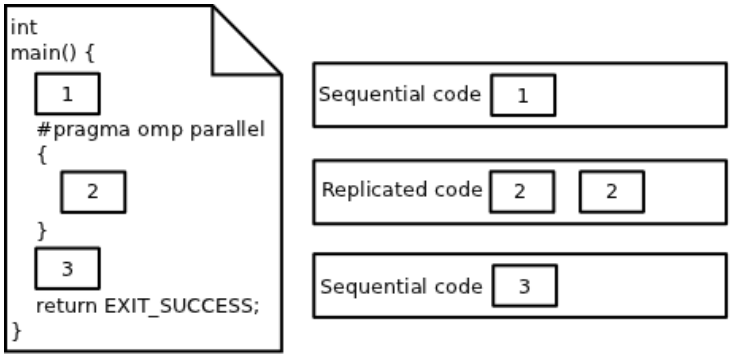

Figure 37 – Régions séquentielles et parallèles sur un bi-processeur.

Chaque thread peut obtenir son identifiant entier via la fonction `omp_get_thread_num`. L'identifiant du thread maître est classiquement la valeur zéro.

Il est possible d'imbriquer des régions parallèles, c'est à dire que les threads qui exécutent une région parallèle peuvent eux-mêmes créer d'autres threads lorsqu'ils rencontrent une région parallèle interne : nous parlons alors de parallélisme imbriqué. Par défaut, ce type de construction est désactivée mais l'utilisateur peut contrôler son activation ou sa désactivation en positionnant respectivement à `TRUE` ou `FALSE` la variable d'environnement `OMP_NESTED`.

Le programmeur peut faire de même via la fonction `omp_set_nested` tandis que `omp_get_nested` lui permet de savoir si la construction est activée ou pas.

L'utilisation du parallélisme imbriqué est problématique car la création d'un thread est une opération coûteuse qui doit être rentabilisée dans la quantité de travail que ce thread devra effectuer. Par conséquent, créer des threads supplémentaires lorsque la charge de travail est insuffisante ne peut amener qu'à une dégradation des performances de l'application. Lorsque la construction est désactivée, le compilateur ignore simplement la directive.

La clause `if (condition)` peut être ajoutée à la directive `omp parallel`. Son argument est une condition booléene. Si cette condition est vérifiée alors la région est considérée comme parallèle. Dans le cas contraire, elle est considérée comme séquentielle.

Il est également possible de fixer le nombre de threads qui vont exécuter une région parallèle en adjoi-

gnant le clause `num_threads(n)` où `n` désigne le nombre maximum de threads autorisés.

Les threads qui exécutent une région parallèle peuvent se répartir la charge de travail de cette dernière. Ce partage peut prendre trois formes.

4.1.2 Sections parallèles

Un bloc `omp sections` annonce la découpe d'une région parallèle en plusieurs zones de code délimitées par des sous-blocs `omp section` (zones 3 et 4 dans l'exemple de la figure 38).

Par défaut, une barrière de synchronisation implicite est placée à la fin d'un bloc `omp sections`, cette barrière ne pouvant être franchie que lorsque tous les threads l'ont atteinte. Pour des considérations d'optimisation, il est possible de supprimer cette barrière via la clause `nowait`.

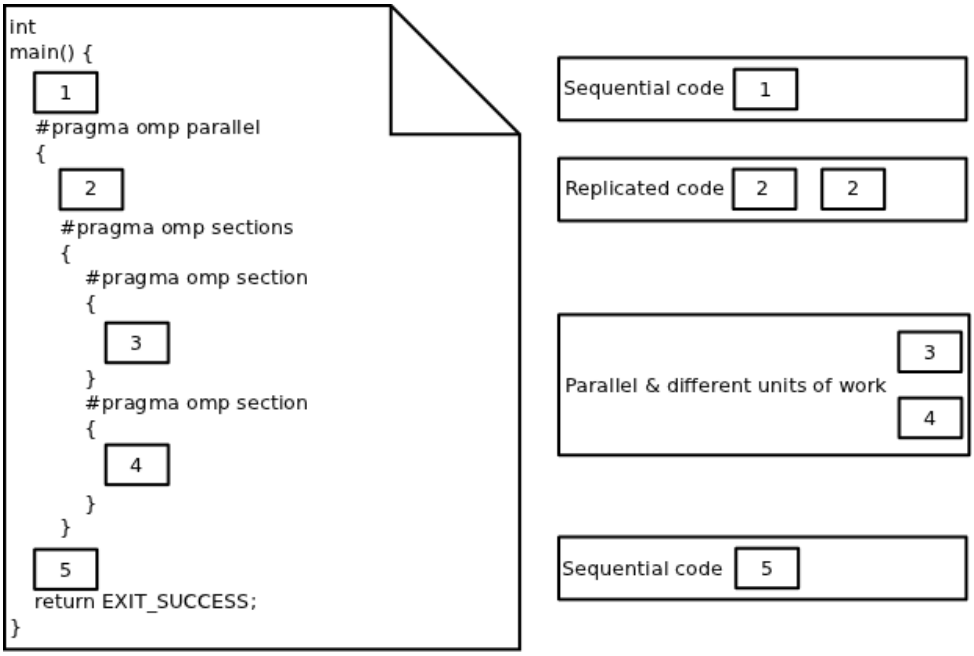

FIGURE 38 – Sections parallèles sur un bi-processeur.

Le nombre de sous-blocs `omp section` fixe le nombre de threads qui travaillent simultanément. Si ce nombre est inférieur au nombre de threads qui exécutent la région parallèle englobante alors ceux qui ne travaillent pas attendent les autres sur la barrière de synchronisation évoquée ci-dessus. Si le nombre de sous-blocs est supérieur au nombre de threads qui exécutent la région parallèle alors les threads disponibles terminent un sous-bloc avant d'en exécuter un autre (sérialisation).

La figure 39 présente la parallélisation d'un algorithme récursif par le biais de sections parallèles. Si le parallélisme imbriqué n'est pas autorisé alors un maximum de deux threads exécutent les sections. Inversement, si le parallélisme imbriqué est autorisé alors il faut prévoir une clause `if` permettant d'arrêter de créer des threads lorsque la taille du tableau à trier devient insuffisante. Si une région parallèle ne contient qu'un seul et unique sous-bloc `omp section` alors l'ensemble des directives peut être fusionné en `omp parallel section`.

L'utilisation de sections parallèles traduit le modèle d'exécution MIMD. L'un des principaux intérêts de cette construction réside en la possibilité de pratiquer le recouvrement communication/calcul, c'est à dire que certains threads communiquent pendant que d'autres calculent, ce qui évite les temps morts et donc

améliore la performance des applications parallèles. Le principal inconvénient est que cette construction est purement statique, c'est à dire fixée dès la compilation.

```
1  void
2  quicksort(double t[], const int& a, const int& b) {
3    const double pivot = t[(a + b) / 2];
4    int i = a, j = b;
5    do {
6      while (t[i] < pivot) {
7        i ++;
8      }
9      while (t [j] > pivot) {
10       j --;
11     }
12     if (i <= j) {
13       std::swap(t[i], t[j]);
14       i ++;
15       j --;
16     }
17   } while (i <= j);
18 #pragma omp parallel sections
19   {
20 #pragma omp section
21     if (a < j) {
22       quicksort(t, a, j);
23     }
24 #pragma omp section
25     if (i < b) {
26       quicksort(t, i , b);
27     }
28   }
29 }
```

FIGURE 39 – Parallélisation d'un *quicksort* à l'aide de sections parallèles.

Le mécanisme des sections parallèles pose problème dans le cas des algorithmes récursifs puisqu'il suppose d'activer le parallélisme imbriqué. Or, dans la plupart des cas, ce mécanisme conduit à une baisse dramatique des performances de l'application pour les raisons suivantes :

- une mauvaise implémentation pour la création et la destruction dynamique de threads (OpenMP est un standard et non une bibliothèque : par conséquent, vous êtes à la merci de votre compilateur et des possibilités de votre machine) ;
- l'overhead induit par la synchronisation d'un trop grand nombre de threads ;
- la difficulté de contrôler le nombre total de threads à un instant donné ;
- etc.

Il est possible de limiter la création de nouveaux threads via la clause additionnelle `if (...)` mais la valeur de coupure doit être fournie par le programmeur, ce qu'il fera quasi systématiquement de manière empirique. Une autre façon de limiter ce nombre consiste à fixer le nombre maximum de threads (initiaux plus dynamiques) via la variable d'environnement `OMP_THREAD_LIMIT` (par exemple `OMP_THREAD_LIMIT=4`). Lorsque cette limite est atteinte, plusieurs blocs `omp_section` peuvent être exécutés séquentiellement par les threads disponibles.

4.1.3 Boucles parallèles (de type `for`)

Une boucle de type `for` ne peut être parallélisée que sous trois conditions :
— les itérations doivent pouvoir être réalisées dans n'importe quel ordre (boucle de type « *forall* ») ;
— son indice doit être un entier (ou implicitement convertible en entier) ou un itérateur de type tableau (`RandomAccessIerator`, C++) ;
— la condition de continuation ne doit faire intervenir que les opérateurs `<`, `<=`, `>` ou `>=` (les opérateurs `==` et `!=` sont exclus) afin que le nombre d'itérations puisse être pré-calculé par l'implémentation OpenMP.

La parallélisation d'une boucle de type `for`, effectuée via la directive `omp for` (Figure 40), signifie que les threads qui exécutent la région parallèle englobante se répartissent les itérations de cette boucle. De fait, cette parallélisation traduit le modèles d'exécution SIMD.

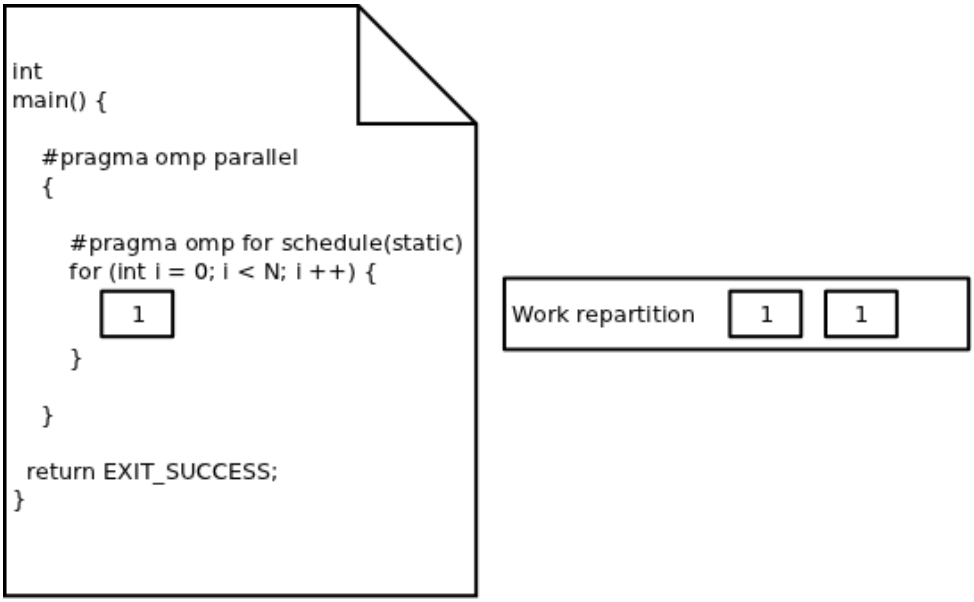

Figure 40 – Parallélisation d'une boucle de type `for` sur un bi-processeur.

Lorsqu'une région parallèle ne contient qu'une seule et unique directive `omp for`, il est possible de fusionner le tout en `omp parallel for`.

Une directive `omp for` est souvent accompagnée de la clause `schedule(type [, size])` qui précise la manière de répartir les itérations entre les différents threads.

Le paramètre optionnel `size` indique le nombre d'itérations consécutives qu'un thread doit traiter : lorsqu'il est omis, la valeur par défaut de ce paramètre est fixée à une itération (sauf dans le cas `static`, voir ci-dessous).

Le paramètre `type` désigne la façon de répartir les itérations. Ses valeurs peuvent être :

`static` : les itérations sont réparties cycliquement par blocs de taille `size`. Lorsque le paramètre `size` est absent, sa valeur est fixée par l'implémentation à N/P où N et P désignent respectivement le nombre d'itérations de la boucle et le nombre de threads disponibles dans la région parallèle. Ce type de répartition concerne les corps de boucles dont la durée de calcul est fixe (homogène) ou quasi fixe (quasi-homogène) ;

`dynamic` : les itérations sont réparties indistinctement par blocs de taille `size` : dès qu'un thread a

terminé son bloc, il commence le prochain bloc disponible. Ce type de répartition concerne les corps de boucles dont la durée de calcul varie fortement d'une itération à l'autre (hétérogène). Comme les blocs d'itérations sont implantés dans une file partagée par l'ensemble des threads exécutant la région parallèle, cette file est protégée par une exclusion mutuelle d'où un *overhead* important lié à la synchronisation. Par conséquent, le programmeur doit tenter de déterminer au mieux la valeur du paramètre `size` permettant de compenser cet *overhead* ;

`guided` : même comportement que le type `dynamic` mais l'implémentation tente d'adapter la valeur du paramètre `size` au cours des itérations (sa valeur décroit exponentiellement jusqu'à 1). Cela peut tout aussi bien réduire drastiquement la durée d'exécution de la boucle que l'aggraver fortement ;

`runtime` : le choix est reporté au moment de l'exécution en positionnant la variable d'environnement `OMP_SCHEDULE`, par exemple : `export OMP_SCHEDULE="guided, 4"`. Ce réglage est commun à toutes les clauses exploitant le type de répartition `runtime` ;

`auto` : laisse le soin à l'implémentation de choisir la meilleure répartition des itérations possible en fonction des seules informations disponibles à la compilation (le paramètre `size` n'est pas mentionné) ;

`ordered` : force l'ordre d'évaluation des itérations de la boucle comme si cette dernière était placée dans une région séquentielle. Ce type de répartition n'est utilisé qu'à des fins de *debugging* (le paramètre `size` n'est pas mentionné).

Par défaut, une barrière de synchronisation implicite est placée à la sortie de la boucle parallèle. Pour des considérations d'optimisation, il est possible de la supprimer via la clause `nowait`.

Considérons l'exemple d'une boucle séquentielle extérieure encageant une boucle parallèle intérieure. La norme actuelle d'OpenMP garantit que le thread ayant traité l'itération j de la boucle parallèle à l'itération i de la boucle séquentielle, continue de le faire à l'itération $i+1$ de cette dernière. Par conséquent, la localité des données au sein du cache privé de niveau 1 de chaque cœur est correctement exploitée : cette caractéristique est appelée « *cache affinity* ».

Enfin, la figure 41 présente la façon de paralléliser efficacement des boucles imbriquées. La clause `collapse(profondeur)` dans laquelle `profondeur` représente un niveau d'imbrication, demande au compilateur de fusionner les espaces d'itération des boucles correspondantes en un seul espace. Les conditions à respecter sont :

— des espaces d'itération n-aires (rectangulaire dans le cas 2D) ;
— des instructions uniquement localisées dans le corps de la boucle la plus profonde.

```
1  const int p = 10;
2  const int q = 20;
3  int m[p][q];
4
5  #pragma omp parallel for schedule(auto) collapse(2)
6  for (int i = 0; i < p; i ++) {
7    for (int j = 0; j < q; j ++) {
8      m[i][j] = i + j;
9    }
10 }
```

Figure 41 – Parallélisation de boucles imbriquées.

4.1.4 Synchronisation des threads

Les directives `omp parallel`, `omp sections` et `omp for` utilisent des barrières de synchronisation implicites qu'il est possible de lever via la clause `nowait`. Il est également possible d'utiliser les mécanismes de synchronisation explicites présentés en figure 42.

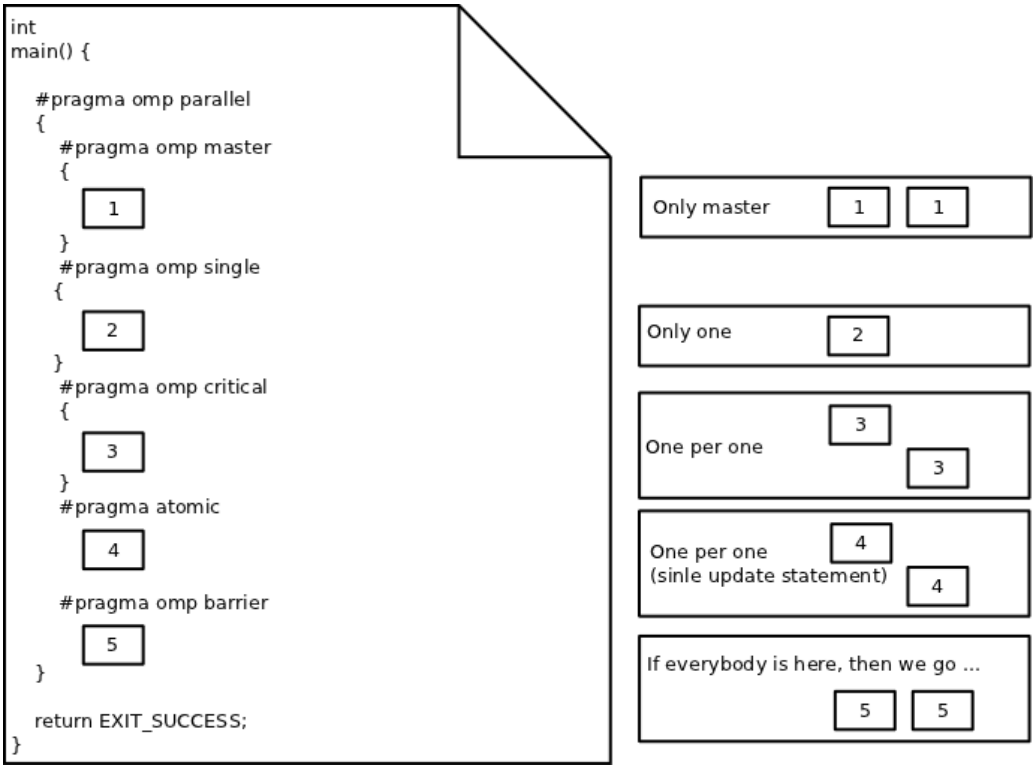

FIGURE 42 – Mécanismes de synchronisation explicites sur un bi-processeur.

Une zone de code délimitée par un bloc `omp master` n'est exécutée que par le seul thread maître. De manière analogue, une zone délimitée par un bloc `omp single` n'est exécutée que par un seul thread, en fait le premier arrivé. La directive `omp single` impose une barrière de synchronisation implicite en sortie pour que les threads qui ne travaillent pas attendent celui qui exécute le bloc. Cette barrière peut être levée via la clause `nowait`. À l'inverse, la directive `omp master` n'impose aucune barrière de synchronisation implicite.

Une zone de code délimitée par un bloc `omp critical` désigne une section critique qui ne peut être exécutée que par un seul thread à la fois. Une barrière de synchronisation explicite est placée à la sortie de ce bloc, barrière qu'il est possible de lever via la clause `nowait`. Il faut se méfier de ce mécanisme de synchronisation car pour garantir l'absence d'interblocage, les implémentations OpenMP utilisent un unique verrou pour l'ensemble des sections critiques de l'application : par conséquent, lorsqu'un thread se trouve dans l'une de ces sections (n'importe laquelle), tous les autres sont mis en attente ...

La directive `omp atomic` est destinée aux opérations d'incrémentation ou de décrémenentation, c'est à dire des expressions exploitant les opérateurs de la forme θ`=` ou θ désigne un opérateur binaire, les opérateurs de pré et post-incrémentation `++` ainsi que les opérateurs de pré et post-décrémentation `--`. Cette directive, qui ne peut être associée qu'à une seule et unique opération, exploite les possibilités de vérouillage/déverouillage du matériel pour la mener à bien. Bien que leur principe soit identique (une barrière de synchronisation implicite est placée en sortie, barrière qu'il est possible de lever via la clause

`nowait`), la directive `omp atomic` doit être systématiquement préférée à la directive `omp critical` à chaque fois que cela est possible.

La directive `omp barrier` représente une barrière de synchronisation explicite.

Bien qu'il ne s'agisse pas à proprement parler d'un mécanisme de synchronisation, la norme OpenMP propose maintenant un moyen permettant de forcer les threads qui exécutent une portion de code parallèle à l'abandonner proprement. Ce mécanisme concerne :
- les régions parallèles ;
- les sections parallèles ;
- les boucles ;
- les groupes de tâches.

et permet de faire l'économie d'un montage complexe basé sur une réduction. La syntaxe de la directive correspondante est :

`#pragma omp cancel` *`construct-type-clause`* `[ [,]` *`if-clause`* `]`

La figure 43 présente une exploitation de ce mécanisme.

```
1  #pragma omp parallel
2    {
3  #pragma omp for schedule(auto)
4      for (size_t i = 0; i < n; i ++) {
5        if (t[i] == 1) {
6          std::cout << omp_get_thread_num() << std::endl;
7  #pragma omp cancel for
8        }
9      }
10   }
```

FIGURE 43 – Mécanisme d'abandon de l'exécution.

Dans cet exemple, le premier thread qui découvre un élément du tableau `t` valant 1 affiche son numéro sur la sortie standard puis abandonne l'exécution de la boucle. Il saute alors à la barrière de synchronisation implicite placée à la sortie de cette boucle. À chaque itération, les threads qui exécutent la boucle vérifient que l'abandon n'a pas encore été activé à un point de vérification placé implicitement en début de boucle. Si tel est le cas, ils abandonnent l'exécution de la boucle et vont se placer en attente sur la barrière de synchronisation correspondante. Dans le cas contraire, ils exécutent l'itération courante.

Il est possible de fixer explicitement un point de vérification via la syntaxe suivante :

`#pragma omp cancellation point` *`construct-type-clause`*

La figure 44 présente une utilisation de cette possibilité pour l'exemple de la figure 43.

4.1.5 Attributs et manipulation des données

Dans une application C++/OpenMP, les attributs de classes sont systématiquement partagés par l'ensemble des threads exécutant une région parallèle. De même, toute variable déclarée à l'extérieur d'une région parallèle est considérée par défaut comme partagée par les threads qui exécutent cette région.

Il est possible de modifier ce statut en ajoutant les clauses suivantes à une directive `omp parallel` :

`private(a, b, c)` : une instance de chaque variable partagée `a`, `b` et `c` est créée dans la mémoire locale de chaque thread. Cependant, ces instances ne sont pas initialisées ;

```
#pragma omp parallel
  {
#pragma omp for schedule(auto)
    for (size_t i = 0; i < n; i ++) {
#pragma omp cancellation point for
      if (t[i] == 1) {
        std::cout << omp_get_thread_num() << std::endl;
#pragma omp cancel for
      }
    }
  }
```

FIGURE 44 – Point de vérification d'abandon explicite.

`firstprivate(a, b, c)` : même comportement que `private` mais les instances locales sont initialisées à partir des valeurs de `a`, `b` et `c` ;

`lastprivate(a, b, c)` : même comportement que `private` mais les variables partagées originales `a`, `b` et `c` sont mises à jour à partir de leurs homologues locales les plus récemment calculées. Cette clause doit être maniée avec beaucoup de prudence et peut concerner, par exemple, le résultat d'une boucle de calcul ;

`default(global)` : où `global` peut prendre comme valeur `private`, `firstprivate`, `shared` ou `none`. Cette clause définit la conduite à tenir pour toutes les variables déclarées à l'extérieur de la région parallèle. La clause `shared` est présentée ci-dessous. La valeur `none` impose de re-déclarer chaque variable déclarée en dehors de la région parallèle avec la clause `shared`. La clause `default` doit être vue comme un outil de contrôle strict du statut de chaque variable mis entre les mains du programmeur ;

`shared(d, e)` : annule l'effet d'une clause `default` avec les arguments `private`, `firstprivate` ou `none` pour les variables `d` et `e` ;

`threadprivate(MaClassee::attribut)` : une instance de l'attribut de classe `MaClassee::attribut` est systématiquement créée dans la mémoire locale de chaque thread pour toutes les régions parallèles rencontrées. Cependant, comme pour `private`, ces instances ne sont pas initialisées ;

`copyin(MaClasse::attribut)` : demande à ce que l'attribut de classe `MaClasse::attribut` ayant fait l'objet d'une clause `threadprivate` soit maintenant initialisé à partir de l'instance de cet attribut possédée par le thread maître ;

`proc_bind(policy)` : qui gère l'affectation des threads (thread affinity) aux différents cœurs d'une machine de type CC-NUMA en fonction de la valeur de la variable d'environnement `OMP_PLACE`, celle-ci définissant la granularité du *cluster* (nœud de calcul, processeurs multi-cœurs de ce nœud ou bien encore cœurs de ce processeur). Le paramètre `policy` peut prendre l'une des valeurs suivantes :

`master` : les threads sont affectés au même emplacement que le thread maître ;

`close` : les threads sont affectés à des emplacement consécutifs depuis celui du thread maître ;

`spread` : les threads sont répartis uniformément sur l'ensemble des emplacements.

Des clauses telles que `firstprivate` ou `copyin` permettent de recopier le contenu de données partagées dans des données locales aux threads. OpenMP propose également des mécanismes permettant d'effectuer le chemin inverse :

`copyprivate(a, b)` : systématiquement associée aux directives `omp master` ou `omp single`, cette clause permet au thread qui exécute le bloc de diffuser, à l'issue, le contenu de ses variables locales `a` et `b` aux autres threads qui exécutent la même région parallèle ;

`flush(a, b)` : demande à ce que la cohérence des caches soient rétablie pour toutes les instances locales des variables partagées `a` et `b`. Cette directive doit être maniée avec prudence et n'existe

que pour des considérations d'optimisation. Elle peut être utilisée sans argument auquel cas elle s'applique à toutes les données locales aux threads ;

`reduction(opérateur : a)` : effectue une réduction sur la variable partagée `a`. Une réduction est une opération associative appliquée à une variable partagée (scalaire ou tableau). Cette opération peut être arithmétique ou logique. Chaque thread calcule un résultat partiel indépendant des autres (variable locale initialisée à l'élément neutre de l'opération). L'ensemble des threads se synchronise ensuite pour mettre à jour la variable partagée concernée en combinant les exemplaires locaux avec la valeur initiale de cette variable.

Le programmeur peut définir ses propres opérateurs de réduction. Pour ce faire, il doit tout d'abord déclarer les caractéristiques de cet opérateur avant de pouvoir l'exploiter dans une clause `reduction`. Cette déclaration est effectuée par le biais d'une directive dont la syntaxe est :

```
#pragma omp declare reduction( reduction-identifier : typename-list : combiner )
                        [ initializer(initialize-clause) ]
```

avec :

`reduction-identifier` : le nom que donne le programmeur à son opérateur ;

`typename-list` : une liste de types sur lesquels peuvent être appliqués l'opérateur, chaque nom étant séparé du précédent par une virgule ;

`combiner` : une expression décrivant comment combiner la copie locale détenue par chaque thread avec le nouvel élément. La copie locale est désignée par le mot-clé `omp_out` tandis que le nouvel élément est désigné par `omp_in` ;

`initialize-clause` : cette clause optionnelle décrit la façon dont la (ou les) copie locale (désignée par le mot-clé `omp_priv`) à chaque thread doit être initialisée. La donnée partagée correspondante est désignée par le mot-clé `omp_orig`. En l'absence de cette clause, chaque donnée locale est initialisée par défaut.

La figure 45 présente la définition *inline* d'une classe `Long` destinée à illustrer notre propos. Cette classe propose un constructeur par défaut/logique, un accesseur ainsi qu'une surcharge de l'opérateur `+` qui retourne la somme de deux instances de la classe.

```
1  class Long {
2  public:
3    Long(const long& value = 0): value_(value) { }
4  public:
5    const long& value() const { return value_; }
6  public:
7    Long operator+(const Long& rhs) const { return Long(value_ + rhs.value_); }
8  private:
9    const long value_;
10 };
```

FIGURE 45 – Définition *inline* de la classe `Long`.

Nous souhaitons effectuer des réductions avec l'opérateur `+` sur le type `Long`. Bien que cet opérateur soit explicitement surchargé dans notre classe, la norme OPENMP nous interdit de le faire figurer dans une clause `reduction` puisque `Long` n'est pas un type fondamental. La seule solution consiste à déclarer explicitement un opérateur de réduction puis à l'exploiter.

La figure 46 présente la déclaration du notre. Ici, l'opérateur de réduction `sumLong` n'est autorisé à travailler que sur le type `Long`. Comme ce dernier surcharge l'opérateur `+`, la réduction est tout simplement définie comme : `omp_out = omp_out + omp_in`. Afin de rester compatible avec les réductions sur

les types fondamentaux, nous initialisons volontairement chaque copie locale avec la valeur initiale de la variable partagée correspondante.

```
1 #pragma omp declare reduction(sumLong : Long : omp_out = omp_out + omp_in) \
2   initializer(omp_priv(omp_orig))
```

FIGURE 46 – Déclaration de notre opérateur de réduction `sumLong` sur le type `Long`.

Ne reste plus qu'à exploiter classiquement notre opérateur, ce que montre la figure 47.

```
1   Long somme;
2 #pragma omp parallel for schedule(auto) reduction(sumLong : somme)
3   for (long i = 0; i < n; i ++) {
4     somme = somme + t[i];
5   }
```

FIGURE 47 – Exploitation de l'opérateur de réduction `sumLong` de la figure 46.

4.2 Vectorisation

En règle générale, la vectorisation est activée via les options d'optimisation transmises au compilateur. Elle peut également être mise en œuvre par des directives OPENMP dont la syntaxe est de l'une des formes suivantes :

```
#pragma omp simd [ clause-list ]
#pragma omp for simd [ clause-list ]
```

La première forme demande à ce que l'ensemble de la boucle `for` qui suit soit simplement vectorisée. Dans le second cas, les itérations de cette boucle sont réparties équitablement entre les threads disponibles et les sous-boucles correspondantes sont vectorisées. Les clauses qui peuvent apparaitre derrière une directive de vectorisation sont :

`safelen(n)` : où `n` est un entier positif indiquant le nombre d'itérations maximum de la boucle pouvant être traitées simultanément. Par défaut, cette longueur est tout simplement celle de l'espace d'itération de la boucle ;

`linear(list [: linear-step ])` : une variable partagée définie à l'extérieur d'une boucle vectorielle ne doit pas voir sa valeur modifiée à l'intérieur de cette boucle, sans quoi le comportement du programme correspondant n'est pas défini. Si le programmeur veut modifier le contenu de cette variable à l'intérieur de sa boucle alors il peut le faire via la clause `linear` ou la clause `reduction` que nous avons déjà rencontrée. La clause `linear` indique tout simplement que la valeur de la variable en question dépend de l'indice de boucle selon la relation :

$$x_i = x_{orig} + i \times linear - step$$

La valeur du pas par défaut est 1 ;

`aligned(list [: alignment ])` : qui indique que les constantes ou variables définies dans la liste possèdent des alignements particuliers en mémoire. La valeur par défaut est celle prévue par l'architecture pour ces données ;

`private` : déjà décrite ;

`lastprivate` : déjà décrite ;

`reduction` : déjà décrite ;

`collapse` : déjà décrite.

La figure 48 présente un exemple de vectorisation.

```
  somme = 0;
#pragma omp simd linear(somme : 2)
  for (size_t i = 0; i < n; i ++) {
    t[i] = somme;
    somme += 2;
  }
```

FIGURE 48 – Exemple de vectorisation

Lorsqu'un corps de boucle invoque une fonction avec des arguments de type scalaire, le compilateur ne peut pas la vectoriser car il faudrait que ces arguments soient des tableaux. La norme propose un mécanisme permettant de traiter ce problème : il suffit de déclarer la fonction concernée comme étant destinée à être invoquée dans une boucle vectorielle. Le compilateur génère alors autant de fonctions particulières que d'appels dans la boucle vectorisée. Cette déclaration est effectuée via la directive suivante placée devant la déclaration (ou la définition) de la fonction :

`#pragma omp declare simd [` *`clause-list`* `]`

Les clauses pouvant apparaître dans cette déclaration sont :

`simdlen(n)` : génère une fonction dont les arguments sont des tableaux de taille `n` ;

`uniform(argument-list)` : indique que tous les paramètres de la fonction mentionnés dans la liste se comportent comme des constantes ;

`inbranch` : indique que la fonction doit être invoquée dans un bloc `if then else`. Par défaut, la fonction peut être invoquée dans ou à l'extérieur d'une alternative ;

`notinbranch` : indique que la fonction ne doit pas être invoquée à l'intérieur d'un bloc `if then else`. Par défaut, la fonction peut être invoquée dans ou à l'extérieur d'une alternative ;

`linear(argument-list [: linear-step ])` : déjà décrite ci-dessus et s'applique aux arguments de la fonction mentionnés dans la liste ;

`aligned(argument-list [: alignment ])` : déjà décrite ci-dessus et s'applique aux arguments de la fonction mentionnés dans la liste.

La figure 49 présente un exemple d'invocation de fonction au sein d'une boucle vectorielle.

```
#pragma omp declare simd uniform(x)
int
square(const int& x) {
  return x * x;
}

...

#pragma omp simd
  for (size_t i = 0; i < n; i ++) {
    t[i] = square(i);
  }
```

FIGURE 49 – Invocation d'une fonction au sein d'une boucle vectorielle.

4.3 Tâches

Les tâches sont un apport majeur dans la norme OpenMP, apport destiné à pallier les faiblesses des versions antérieures en matière de parallélisation d'algorithmes récursifs, d'algorithmes manipulant des structures de données irrégulières ou bien encore de parallélisation de boucles d'un autre type que `for`.

Considérons l'algorithme récursif présenté en figure 39. Avant l'introduction des tâches, sa parallélisation passait par les sections parallèles. Or, le coût de création des threads demeurait suffisamment dissuasif pour ne pas recourir au parallélisme imbriqué. Par conséquent, le programmeur ne pouvait pas exploiter tout le parallélisme potentiel de cet algorithme. La figure 50 présente la parallélisation du même algorithme à l'aide de tâches.

```
1  void
2  quicksort(double t[], const int& a, const int& b) {
3    const double pivot = t[(a + b) / 2];
4    int i = a, j = b;
5    do {
6      while (t[i] < pivot) {
7        i ++;
8      }
9      while (t [j] > pivot) {
10       j --;
11     }
12     if (i <= j) {
13       std::swap(t[i], t[j]);
14       i ++;
15       j --;
16     }
17   } while (i <= j);
18 #pragma omp task shared(a) if (b - a > 100)
19   if (a < j) {
20     quicksort(t, a, j);
21   }
22 #pragma omp task shared(b) if (b - a > 100)
23   if (i < b) {
24     quicksort(t, i , b);
25   }
26 #pragma omp taskwait
27 }
28
29 void
30 parquicksort(double t[], const int& a, const int& b) {
31 #pragma omp parallel
32   {
33 #pragma omp single
34     quicksort(t, a, b);
35   }
36 }
```

Figure 50 – Parallélisation d'un quicksort à l'aide de tâches.

La fonction `parquicksort` constitue le point d'entrée de cet algorithme. Nous y trouvons une région parallèle englobant un bloc `omp single`. L'appel de la fonction `quicksort` sur l'ensemble du tableau à trier est donc effectué par un seul et unique thread. Chaque appel de la fonction `quicksort` a pour effet non pas son exécution mais la création de deux tâches, l'une rappelant la fonction `quicksort` sur la moitié gauche du tableau et l'autre sur la moitié droite. Ces tâches sont placées au fur et à mesure

de leur construction dans une file de tâches. Lorsque la dernière tâche est créée, le bloc `omp single` se termine et tous les threads qui exécutent la région parallèle commence à exécuter les tâches placées dans la file. La barrière de synchronisation `omp taskwait` permet à une tâche d'attendre la fin de l'exécution des tâches qu'elle a engendrées.

L'exemple de la figure 50 illustre la philosophie des tâches en OpenMP. Une tâche est à l'image d'un thread, un bloc d'instructions avec un environnement, c'est à dire des données privées. Lorsqu'un thread exécutant une région parallèle rencontre un constructeur de tâche, il créé une instance de cette tâche puis la place dans une file associée à la région parallèle. Si ce thread n'exécute pas un bloc `omp single` alors l'exécution de la tâche peut démarrer immédiatement ou plus tard. Cette exécution est assurée par l'équipe de threads qui exécute la région parallèle englobante. Une tâche est affectée à un thread unique et seul ce dernier, par défaut, peut l'exécuter. Il peut cependant la mettre en sommeil pour la reprendre ultérieurement. Lorsqu'une clause `if (...)` est associée au constructeur de tâche et que la condition correspondante est fausse alors le thread qui rencontre ce constructeur doit exécuter immédiatement la tâche correspondante.

Le statut des données manipulées par une tâche est par défaut `firstprivate` pour toutes les données locales à la région parallèle, les autres ayant par défaut le statut `shared`. Une tâche peut demander à partager une donnée `data` locale à une région parallèle en faisant suivre son constructeur de la clause `shared(data)`.

La barrière de synchronisation implicite placée à la fin d'une région parallèle garantit que toutes les tâches associées à cette région seront exécutées quant la région se terminera. Cependant, une tâche de cette région (nous dirons une tâche mère) peut donner naissance à d'autre tâches (nous dirons des tâches filles) en leur fournissant des arguments qui correspondent à certaines de ses données locales. L'exécution des tâches pouvant être différée, il est possible que l'exécution de la tâche mère se termine alors que celle des tâches filles n'a pas encore commencé. Par conséquent, si des alias ou des pointeurs ont été fournis aux tâches filles lors de leur création, ces derniers ne référencent plus rien lorsque les tâches filles commencent leur exécution. De fait, il faut absolument que la tâche mère soit exécutée après ses tâches filles. Cet ordonnancement est assuré par la barrière de synchronisation explicite `omp taskwait`.

Ainsi, dans l'exemple de la figure 50, les paramètres de la fonction `quicksort` sont respectivement un tableau et deux alias. La clause `firstprivate` est implicite pour le tableau mais seul le pointeur associé (l'adresse de base) est capturée par la tâche et non pas le contenu. Pour les deux alias, le compilateur exige qu'ils fassent l'objet d'une clause `shared`, ce qui n'est pas un problème puisqu'il s'agit de constantes. Lorsque la tâche mère créée deux tâches filles, elle leur fournit (via des alias) une nouvelle borne de tableau calculée à partir de l'ancienne. Par conséquent, cette nouvelle borne est locale à la tâche mère et celle-ci doit donc suspendre son exécution afin de laisser terminer ses tâches filles. Notons que la directive `omp taskwait` ne concerne que les descendants directs et pas les descendants des descendants.

Il est possible, pour des considérations d'optimisation (et plus précisément d'équilibrage de charge), de demander à ce qu'une tâche puisse commencer son exécution avec un thread, puis, si elle est mise en sommeil, de pouvoir être réveillée et ré-affectée à un autre thread qui serait immédiatement disponible. Cette possibilité, à manier avec beaucoup de prudence, est activée via la clause `untied`. Les tâches peuvent être créées dans n'importe quelle mécanisme de répartition du travail (boucle de type `for`, sections parallèles) au sein d'une région parallèle. Elles permettent également de paralléliser des boucles de type `for` dont la borne supérieure est inconnue et plus généralement n'importe quel type de boucle.

La clause `reduction` n'a pas de sens pour les tâches : il faut donc utiliser la directive `atomic` pour obtenir le résultat correspondant.

Notons enfin que la norme ne précise rien sur la façon dont son gérée les tâches en interne, notamment si la file de tâches associée à une région parallèle est unique ou au contraire distribuée sur l'ensemble des cœurs de la machine. Le problème d'une file unique est que pour un nombre élevé de threads, celle-ci

devient un sérieux point de contention. Ce problème peut être résolu si le scheduler utilisé par l'implémentation exploite un algorithme de work-stealing emprunté à Cilk et repris par Intel Threading Building Blocks.

4.3.1 Dépéndances entre tâches

La clause `depend( dependence-type : list )` permet d'exprimer des dépendances fines entre tâches affectées à une même régions parallèles. Ses paramètres sont :

— *list* est une liste de variables et/ou de constantes pouvant contenir des tableaux ou des tranches de tableaux ;

— *dependence-type* exprime le type de dépendance liant les tâches filles à leur mère. Les valeurs permises sont :

`in` qui indique que la tâche (appelons la $\mathcal{T}$) qui va être créée ne pourra être exécutée tant que toutes les tâches définies précédemment et mentionnant au moins dans leur clause `depend` l'une des variables ou constantes de la clause de $\mathcal{T}$ avec les modes `out` ou `inout`, n'auront pas terminé ;

`out` qui indique que la tâche $\mathcal{T}$ à créer ne pourra être exécutée tant que toutes les tâches définies précédemment et mentionnant au moins une variable ou constante de $\mathcal{T}$ (quel que soit le mode) n'auront pas terminé ;

`inout` : même comportement que le mode `out`.

Considérons le petit exemple de la figure 51 :

— la tâche `A` est définie la première et sa clause `depend` mentionne deux variables partagées `x` et `y` en mode `out`. Par conséquent, le compilateur regarde si d'autres tâches définies précédemment dans la même région parallèle mentionne ces deux variables dans leur clause `depend` quel que soit leur mode. Comme il n'y en a pas, la tâche `A` peut être exécuté immédiatement ;

— la tâche `B`, définie en second position, mentionne la variable `x` avec le mode `in`. Par conséquent, `B` dépend de `A` ;

— la tâche `C`, définie en troisième position, mentionne la variable `y` avec le mode `in`. Par conséquent, `C` dépend de `A` mais pas de `B` : les deux tâches peuvent donc être exécutées simultanément dès que `A` a terminé ;

— la tâche `D`, définie en dernière position, mentionne les deux variables en mode `out`. Par conséquent, `D` dépend à la fois de `B` et de `C` : c'est donc la tâche qui sera exécutée en dernier.

4.3.2 Groupes de tâches

En règle générale, les tâches associées à une région parallèle peuvent être considérée comme un même groupe et toutes se termineront sur la barrière de synchronisation placée à la sortie de cette région. Cependant, il est parfois nécessaire de définir explicitement un groupe de tâches afin de lui faire faire une action particulière. Dans ce cas, comme une tâche isolée, il doit être possible de :

— interrompre toutes les tâches du groupe ;

— garantir que toutes les tâches de ce groupes, initiales comme celles créés dynamiquement, ont achevé leur exécution à un point donné du code.

La notion de groupe de tâches était absente des normes d'OpenMP antérieure à la version 4.0. Il n'existait pas de clause d'abandon et la seule directive permettant de synchroniser des tâches sur un point particulier du code était `omp taskwait`. Or, cette dernière ne synchronise que les tâches définies au niveau de la région parallèle mais pas leur descendantes. Ainsi, dans l'exemple de la figure 52, seules les tâches `T1` et `T2` seront synchronisées sur la barrière `taskwait` mais pas `T3` puisque celle-ci est créée par `T2`.

La nouvelle directive `taskgroup` permet de résoudre les deux problèmes mentionnée ci-dessous, ainsi que le montre la figure 53. Dans ce nouvel exemple, les tâches `T1`, `T2` et `T3` sont considérées comme appartenant au même groupe. Par conséquent, elle achève leur exécution et s'attendent sur la barrière de synchronisation implicite placée à la sortie de la directive `omp taskgroup`.

```
1   // Variable partagée x.
2
3   int x = 0, y = 100;
4
5  #pragma omp parallel
6   {
7
8     // Tâche A.
9
10 #pragma omp task depend (out : x, y)
11    {
12      x = 1;
13      y = 101;
14    }
15
16    // Tâche B.
17
18 #pragma omp task depend(in : x)
19    {
20      x = 2;
21    }
22
23    // Tâche C.
24
25 #pragma omp task depend(in : y)
26    y = 202;
27
28    // Tâche D.
29
30 #pragma omp task depend(out : x, y)
31    {
32      x = 3;
33      y = 303;
34    }
35
36  }
```

FIGURE 51 – Dépendances entre tâches.

```
#pragma omp parallel
  {
#pragma omp task
    {
      // T1.
    }

#pragma omp task
    {
      // T2.

#pragma omp task
      {
        // T3.
      }

    }

  ...
  // Point du code.

#pragma omp taskwait
  ...
  }
```

FIGURE 52 – Utilisation de la directive `omp taskwait`.

```
#pragma omp parallel
  {

#pragma omp taskgroup
    {

#pragma omp task
      {
        // T1.
      }

#pragma omp task
      {
        // T2.

#pragma omp task
        {
          // T3.
        }
      }

    } // omp taskgroup

  // Point du code.
  ...
  }
```

FIGURE 53 – Utilisation de la directive `taskgroup`.

4.4 Pièges à éviter

Sont résumés ici les pièges les plus courants.

4.4.1 *Race condition*

Il s'agit du premier piège dans lequel tombe les débutants mais aussi les programmeurs plus chevronnés (lorsqu'ils utilisent des objets par exemple).

La *race condition* désigne soit l'écriture quasi-simultanée, soit la lecture/écriture quasi-simultanée d'une variable partagée par plusieurs threads. La figure 54 présente un exemple non trivial.

```
1  class MaClasse {
2  public:
3    MaClasse const int& attribut) : attribut_(attribut) { }
4  public:
5    const int& attribut() const { return attribut_; }
6  public:
7    void add(const int& valeur) { attribut_ += valeur; }
8  protected:
9    int attribut_;
10 };
11 ...
12 int
13 main ( ) {
14   MaClasse objet(1);
15 #pragma omp parallel
16   {
17       objet.add(10);
18   }
19   return EXIT_SUCCESS;
20 }
```

FIGURE 54 – Race condition sur l'attribut `MaClasse::attribut_`.

Dans cet exemple, l'instance `objet` de classe `MaClasse` est partagée par tous les threads qui exécutent la région parallèle. Par conséquent, l'attribut `attribut_` de l'instance objet l'est également. Or, chaque thread invoque la méthode `add` qui modifie la valeur de cet attribut. La solution consiste ici soit à :

- encager l'invocation de la méthode `add` dans un bloc `omp critical` : c'est la solution coté utilisateur si la classe n'est pas prévue pour un usage dans un contexte multithreadé ;
- faire précéder l'expression `attribut_ += valeur` de la directive `omp atomic` : c'est la solution coté développeur si la classe est prévue pour être exploitée dans un contexte multithreadé.

4.4.2 *False sharing*

Le *false sharing* désigne l'écriture par plusieurs threads de données stockées sur la même ligne d'un cache partagé. Lorsqu'un thread accède à une donnée de cette ligne en lecture, le processeur sous-jacent la recopie dans son propre cache. Si un autre thread modifie la donnée dans le cache partagé alors toutes les données de la ligne doivent être invalidées dans les caches locaux. Par conséquent, si plusieurs threads modifient les données de la ligne, il se produit un phénomène de ping pong entre le cache partagé et les caches locaux, ce qui peut nuire gravement aux performances de l'application.

La figure 55 présente un exemple trivial. Dans ce dernier, nous déclarons un tableau dont la taille est le nombre de threads par défaut exécutant une région parallèle. Chaque thread récupère ensuite son identifiant pour mettre à jour la case correspondante du tableau.

```
1  int threads ;
2
3  #pragma omp parallel
4  #pragma omp single
5  threads = omp_get_num_threads();
6
7  int valeurs[threads];
8  #pragma omp parallel
9    {
10     const int id = omp_get_thread_num();
11     valeurs[id] = id * 10 + 5;
12   }
```

FIGURE 55 – *False sharing* induit par un tableau.

Il est évident ici que le tableau étant de petite taille, ses éléments partageront une même ligne du cache partagé, provoquant ainsi du *false sharing* lors de chaque écriture. Le false sharing est un problème difficile à éradiquer mais la conduite à tenir pour l'éviter consiste à utiliser un maximum de données locales aux threads.