École Nationale Supérieure
d'Ingénieurs de Caen

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

Architecture parallèle

Niveau	3ème **année**
Parcours	**Électronique et Physique Appliquée**
Unité d'enseignement	2E1AC2 **- Architectures pour le calcul** [**Parallélisme**]
Création	9 **Novembre** 2020
Responsables	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

Table des matières

1 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*).

1.1 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 1).

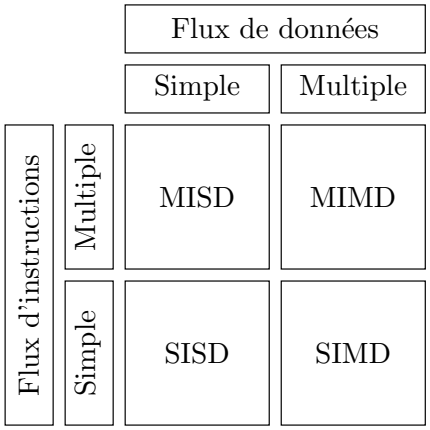

FIGURE 1 – Classification de J. FLYNN (1972).

1.1.1 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 2). Elle ne comporte qu'une seule instance : le modèle de J. VON NEUMAN (1946).

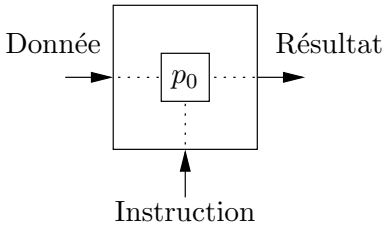

FIGURE 2 – Classe des modèles SISD.

1.1.2 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 3).

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 4) :

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

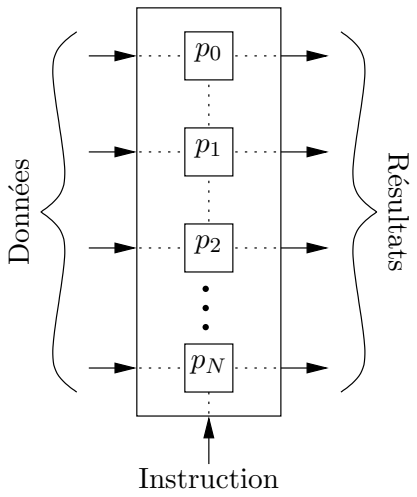

FIGURE 3 – Classe des modèles SIMD.

— 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 5, 6 et 7 présente le cheminement de tout programmeur.

La figure 5 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 6).
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 7). 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).

1.1.3 Classe des modèles MISD

Dans la réalité, cette classe (Figure 8) 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.

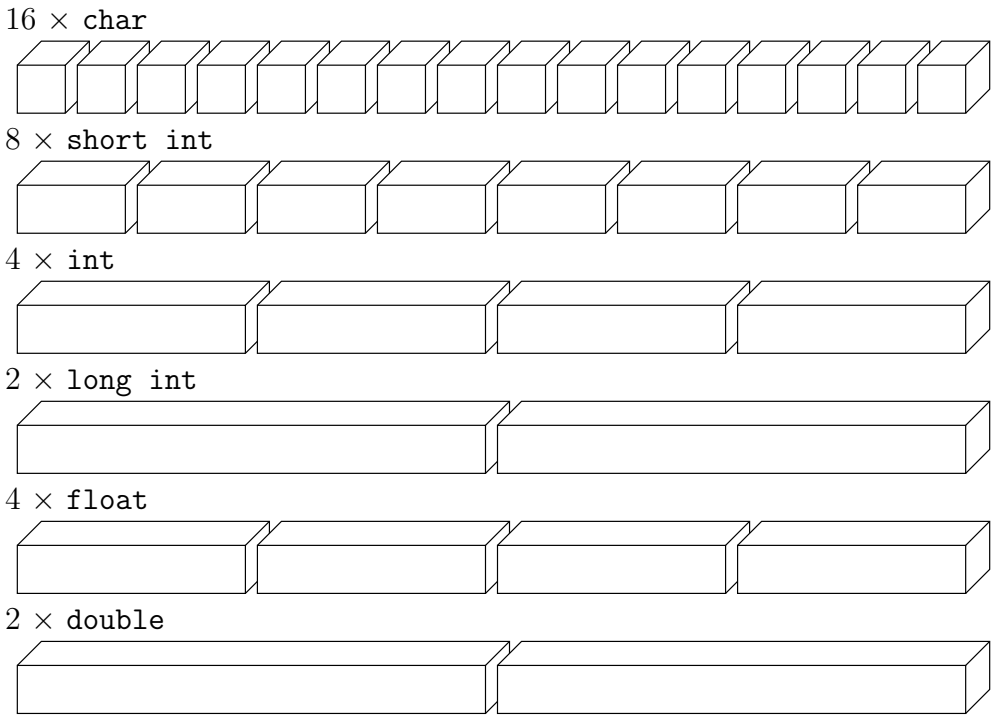

FIGURE 4 – L'un des huit registres 128 *bits* `XMMx` du *Pentium IV*.

```
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 5 – 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 6 – 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 7 – Forme définitive « vectorisée ».

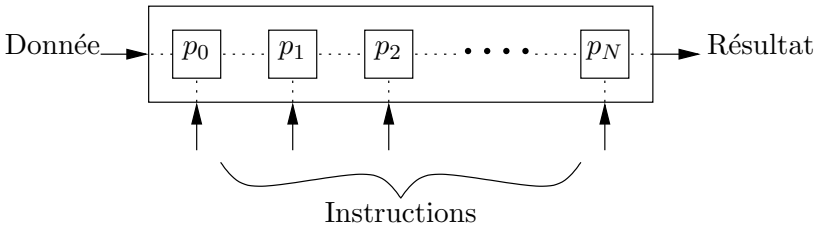

FIGURE 8 – Classe des modèles MISD.

1.1.4 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 9).

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 10 et 11 présente l'utilisation de l'un de ces outils : le standard OPENMP permettant d'écrire des applications multi-threadées.

La figure 10 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 11.

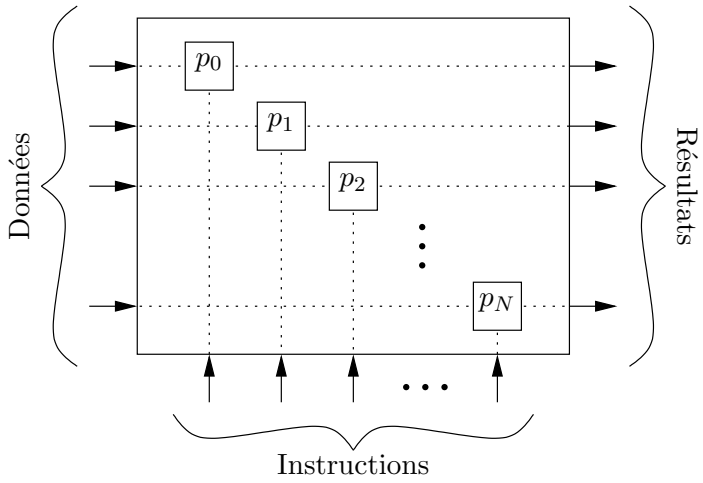

FIGURE 9 – Classe des modèles MIMD.

```
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 10 – Un code séquentiel potentiellement parallélisable.

```
1  int a;
2  double b;
3
4  // Deux fonctions indépendantes.
5  #pragma omp parallel
6  {
7    #pragma omp sections
8    {
9  #pragma omp section // Le premier thread.
10     a = f(5);
11
12 #pragma omp section // Le deuxième thread.
13     b = g(36.25);
14   }
15
16 } // barrière de synchronisation implicite.
17
18 // ... faire qqch avec a et b.
```

FIGURE 11 – Sections parallèles en OPENMP.

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

1.2 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.

1.2.1 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 12 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.

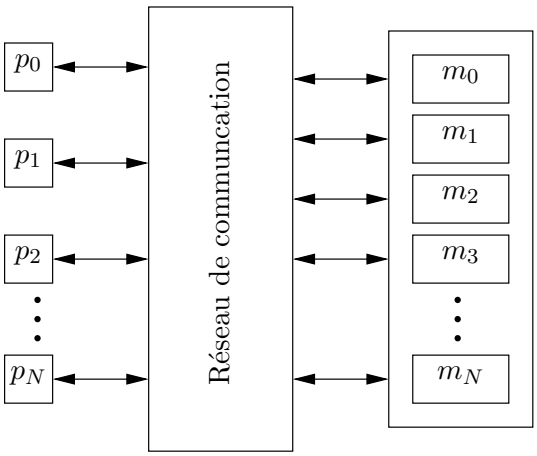

FIGURE 12 – Machine à mémoire partagée.

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 13).

FIGURE 13 – 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 14 présente un *crossbar* 3×3.

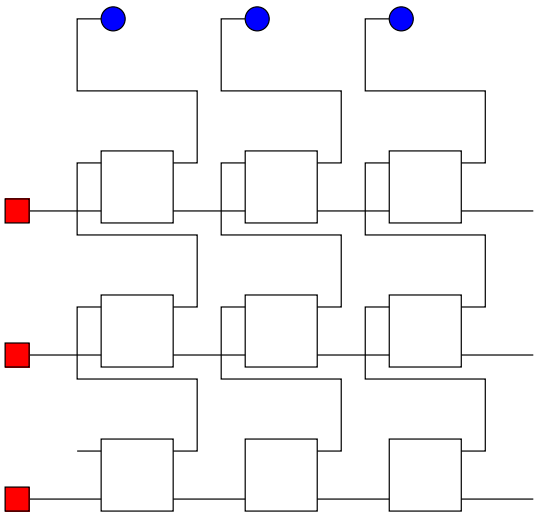

FIGURE 14 – *Crossbar* 3×3

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 15 présente un exemple de réseau 6×6 à trois étages appelé « CLOS$(2, 3, 3)$ ».

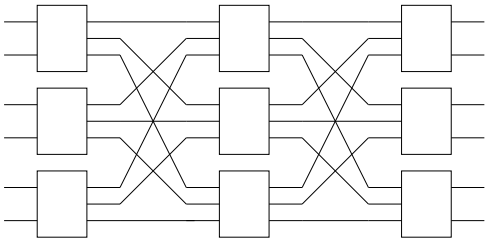

FIGURE 15 – Réseau CLOS$(2, 3, 3)$.

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 :

$$bloc\_num\_a = addr\_a \ / \ 64, \tag{1.1}$$
$$offset\_a = addr\_a \ \% \ 64, \tag{1.2}$$

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.

1.2.2 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 16 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.

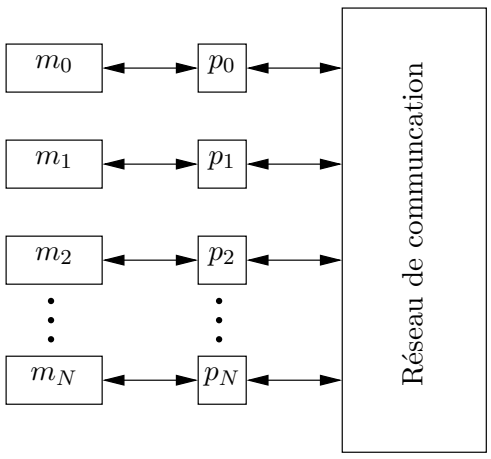

FIGURE 16 – Machine à mémoire distribuée.

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 17) 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.

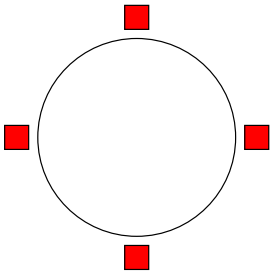

FIGURE 17 – Topologie en anneau.

Un réseau de $n \times n$ processeurs en grille (Figure 18) 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).

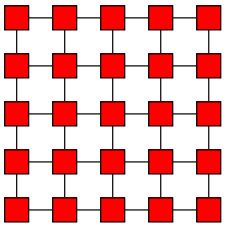

FIGURE 18 – Topologie en grille.

Un réseau de $n = 2^p$ processeurs en arbre (Figure 19) 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).

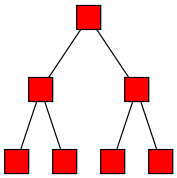

FIGURE 19 – Topologie en arbre.

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 20 présente le processus de construction récursif d'un hyper-cube de degré 4.

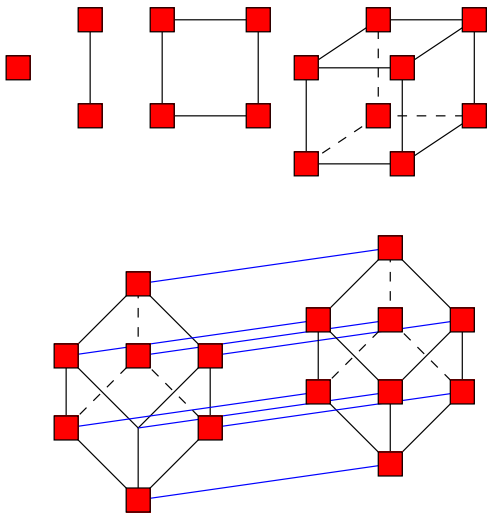

FIGURE 20 – Construction récursive d'un hyper-cube de degré 4.

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.

1.2.3 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 21 en présente l'architecture générale.

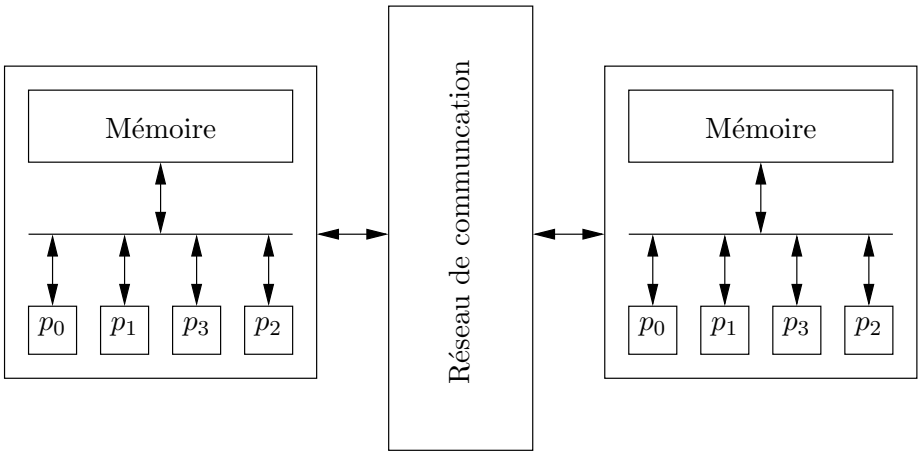

FIGURE 21 – Architecture de type « *SMP cluster* ».

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 21) 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).

1.3 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*).

1.3.1 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 :

$$s(N,P) = \frac{t_1(N)}{t_P(N)}. \tag{1.3}$$

La figure 22 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.

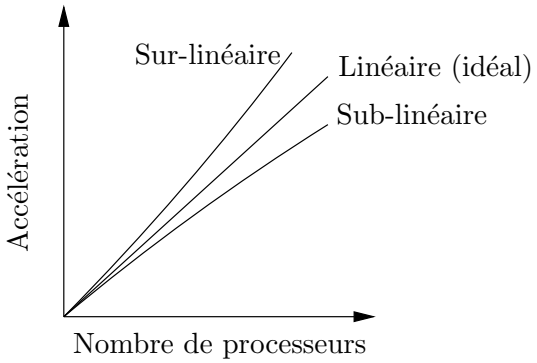

FIGURE 22 – Facteur d'accélération.

1.3.2 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 :

$$e(N,P) = \frac{s(N,P)}{P} = \frac{t_1(N)}{P \times t_P(N)}. \tag{1.4}$$

La figure 23 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.

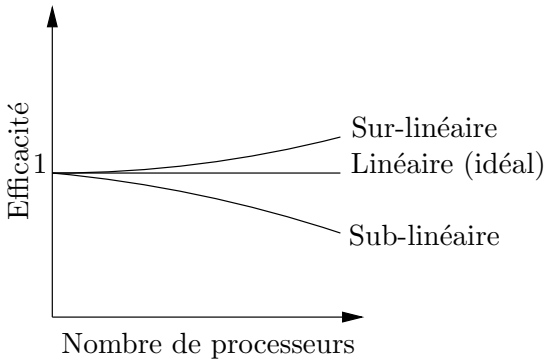

FIGURE 23 – Facteur d'efficacité.

1.3.3 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.

1.3.4 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 :

$$t_1(N) = t_{\text{séq}}(N) + t_{\text{par}}(N), \tag{1.5}$$

sur une machine mono-processeur et sous la forme :

$$t_P(N) = t_{\text{séq}}(N) + \frac{t_{\text{par}}(N)}{P}, \tag{1.6}$$

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 :

$$\begin{aligned} t_{\text{séq}}(N) &= f(N) \times t_1(N), & (1.7)\\ t_{\text{par}}(N) &= \big\{1 - f(N)\big\} \times t_1(N). & (1.8)\\ & & (1.9) \end{aligned}$$

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

$$s(N,P) = \frac{t_{\text{séq}}(N) + t_{\text{par}}(N)}{t_{\text{séq}}(N) + \frac{t_{\text{par}}(N)}{P}} \leq \frac{t_1(N)}{t_{\text{séq}}(N)}. \tag{1.10}$$

Nous pouvons alors écrire :

$$s(N,P) = \frac{1}{f(N) + \frac{1-f(N)}{P}} \leq \frac{1}{f(N)}, \tag{1.11}$$

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.

1.3.5 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 :

$$S(N,P) = \frac{t_1(N)}{t_P(N)} = \frac{t_{\text{séq}}(N) + P \times t_{\text{par}}(N)}{t_{\text{séq}}(N) + t_{\text{par}}(N)}. \tag{1.12}$$

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

$$S(N,P) = t_{\text{séq}}(N) + P \times t_{\text{par}}(N) \geq P \times t_{\text{par}}(N), \tag{1.13}$$

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 :

$$S(N,P) = \frac{t_1(N)}{t_P(N)} = \frac{N \times t_{max}(1)}{N \times t_{max}(P)} = \frac{t_{max}(1)}{t_{max}(P)}. \tag{1.14}$$