



Date de publication :  
**10 août 2017**

# Introduction au parallélisme et aux architectures parallèles

Cet article est issu de : **Sciences fondamentales | Mathématiques**

par **Franck CAPPELLO, Daniel ETIEMBLE**

## Mots-clés

parallélisme de données et de  
contrôle | extensions SIMD |  
classification de Flynn |  
mémoires partagées et  
distribuées | modèles  
d'exécution | modèles de  
programmation | OpenMP |  
MPI | pThreads | loi d'Amdhal |  
modèle Roofline

## Keywords

data and control parallelism |  
SIMD extensions | Flynn's  
taxonomy | shared and  
distributed memories |  
execution models |  
programming models |  
OpenMP | MPI | pThreads |  
Amdhal's law | Roofline model

**Résumé** Le parallélisme est dorénavant utilisé dans la majorité des architectures, des systèmes embarqués aux superordinateurs. Les monoprocesseurs sont remplacés par des processeurs multicœurs. Cet article décrit la notion de parallélisme et ses différents types. Il présente les grandes classes d'architectures parallèles avec leurs ressources et organisations mémoire, en distinguant les architectures homogènes et hétérogènes. Les principes des techniques de programmation sont introduits avec les extensions parallèles des langages de programmation couramment utilisés et les modèles de programmation visant à rapprocher la programmation parallèle de la programmation séquentielle, en incluant les spécificités des architectures. Enfin, les modèles et métriques d'évaluation des performances sont examinés.

**Abstract** Since the early 2000s, parallelism has found use in most computer architectures, from embedded systems to supercomputers. Multi-core processors have replaced uniprocessors. This article describes parallelism and its different types. It presents the main classes of parallel architectures with their resources and memory organizations, in both homogeneous and heterogeneous architectures. The basic parallel programming techniques are introduced with the parallel extensions of commonly used programming languages, and the programming models designed to close the gap with sequential programming, while allowing for the specific features of parallel architectures. Finally, performance evaluation is presented with metrics and performance models.

## Pour toute question :

Service Relation clientèle  
Techniques de l'Ingénieur  
Immeuble Pleyad 1  
39, boulevard Ornano  
93288 Saint-Denis Cedex

## Par mail :

infos.clients@teching.com

## Par téléphone :

00 33 (0)1 53 35 20 20

Document téléchargé le : **23/12/2022**

Pour le compte : **7200035676 - insa rouen normandie // 195.220.135.37**

# Introduction au parallélisme et aux architectures parallèles

par **Franck CAPPELLO**

Docteur en Informatique de l'université Paris Sud  
IEEE Fellow

et **Daniel ETIEMBLE**

Ingénieur de l'INSA de Lyon  
Professeur émérite à l'université Paris Sud

**Note de l'éditeur :** Cet article est la version actualisée de l'article [H 1 088] intitulé *Introduction au parallélisme et aux architectures parallèles*, de Franck CAPPELLO et Jean-Paul SANSONNET, paru dans nos éditions en 1999.

<b>1. Motivations pour le parallélisme .....</b>	<b>H 1 088v2 – 2</b>
1.1 Besoins des applications .....	— 2
1.2 Mur de la chaleur .....	— 3
<b>2. Qu'est-ce que le parallélisme ? .....</b>	<b>— 4</b>
2.1 Approche intuitive du parallélisme .....	— 4
2.2 Définition formelle .....	— 4
<b>3. Sources du parallélisme et opérations fondamentales .....</b>	<b>— 5</b>
3.1 Parallélisme de données .....	— 5
3.2 Parallélisme de contrôle .....	— 5
3.3 Opérations fondamentales du parallélisme .....	— 6
3.4 Consistance mémoire .....	— 7
<b>4. Parallélisme dans les monoprocesseurs .....</b>	<b>— 8</b>
<b>5. Classification des architectures parallèles .....</b>	<b>— 9</b>
5.1 Classification de Flynn .....	— 9
5.2 Classification selon le modèle mémoire .....	— 10
5.3 Classification suivant le grain de calcul .....	— 11
5.4 Architectures parallèles homogènes ou hétérogènes .....	— 12
5.5 Organisation du système d'exploitation dans les architectures parallèles .....	— 14
<b>6. Ressources des architectures parallèles .....</b>	<b>— 14</b>
<b>7. Modèles d'exécution .....</b>	<b>— 15</b>
<b>8. Programmation des architectures parallèles .....</b>	<b>— 15</b>
8.1 Extensions parallèles des langages séquentiels .....	— 16
8.2 Modèles de programmation .....	— 19
<b>9. Lois et métriques de performances des architectures parallèles .....</b>	<b>— 21</b>
9.1 Performances .....	— 21
9.2 Lois de performance .....	— 21
9.3 Modèle « Roofline » .....	— 22
9.4 Métriques et benchmarks .....	— 23
<b>10. Remarques pour conclure .....</b>	<b>— 23</b>
<b>11. Glossaire .....</b>	<b>— 23</b>
<b>Pour en savoir plus .....</b>	<b>Doc. H 1 088v2</b>

**L**a notion de parallélisme, qui consiste à utiliser plusieurs processeurs ou opérateurs matériels pour exécuter un ou plusieurs programmes, est ancienne. Les multiprocesseurs datent des années 1960. De cette période jusqu'à la fin des années 1990, des architectures parallèles ont été utilisées pour les applications nécessitant des besoins de calcul que les

monoprocesseurs étaient incapables de fournir. Étaient concernés les mainframes et serveurs d'une part, et les machines vectorielles puis parallèles utilisées pour le calcul scientifique hautes performances d'autre part. Les années 1980 ont vu l'apparition de différentes sociétés proposant des machines parallèles, sociétés qui ont assez rapidement disparu. La raison essentielle est liée aux progressions exponentielles des performances des microprocesseurs, utilisés dans les PC et les serveurs multiprocesseurs. L'utilisation massive du parallélisme se limitait aux très grandes applications de simulation numérique avec les architectures massivement parallèles. Le début des années 2000, avec les limitations des monoprocesseurs et le « mur de la chaleur », a complètement changé la situation (voir [H 1 058]). Les processeurs multicœurs sont présents en 2016 dans les architectures matérielles pour tous les types de composants : appareils mobiles (smartphones, tablettes), systèmes embarqués, télévisions, PC portables et PC de bureau, et jusqu'aux machines parallèles et superordinateurs pour la très haute performance.

Dans cet article, nous introduisons la notion de parallélisme, présentons les différents types de parallélisme et les différentes formes d'architectures parallèles. Alors que la programmation des machines parallèles a été longtemps réservée à des spécialistes, tout programmeur doit maintenant maîtriser les notions essentielles de la programmation parallèle pour tirer parti des possibilités des architectures. Nous présentons les extensions parallèles des langages de programmation couramment utilisés, les modèles de programmation développés qui visent à « rapprocher » la programmation parallèle des techniques de la programmation séquentielle tout en prenant en compte les spécificités des architectures parallèles. Enfin, l'intérêt des architectures parallèles réside dans les performances qu'elles permettent d'atteindre. Pour optimiser ces performances et/ou réduire la consommation énergétique, il est nécessaire de modéliser d'une part le parallélisme existant dans une application et d'autre part les architectures parallèles. Nous examinons donc les métriques utilisées pour évaluer ou prévoir les performances et les grandes lois qui les gouvernent.

## 1. Motivations pour le parallélisme

L'exploitation du parallélisme dans l'architecture des ordinateurs est liée à la conjonction de trois éléments : les besoins des applications, les limites des architectures séquentielles et l'existence de parallélisme dans les applications.

### 1.1 Besoins des applications

La notion de parallélisme est souvent attachée à celle de la performance d'exécution des applications. Ce dernier terme recouvre différentes notions suivant les besoins des applications. En effet, quel que soit le domaine d'application, le parallélisme peut être exploité pour répondre à deux besoins : la puissance de traitement et/ou la disponibilité.

La puissance de traitement recouvre deux grandes notions : le temps de traitement et le débit de traitement. Le premier terme est le temps nécessaire pour l'exécution d'un traitement. Le second représente le nombre de traitements exécutables par unité de temps.

Ces deux notions peuvent être indépendantes. Réduire le temps de traitement est plus difficile qu'augmenter le débit de traitement. Dans le premier cas, il s'agit de lutter contre le temps qui, en

dernier ressort, est fixé par les possibilités technologiques. Alors que, dans le deuxième cas, si plusieurs traitements sont indépendants, l'augmentation du nombre de ressources suffit pour exécuter plus de traitements en même temps.

La puissance de traitement dépend aussi de la capacité et de l'organisation de la mémoire d'un ordinateur. Certaines applications requièrent des ensembles de données dont la taille est supérieure à la capacité d'adressage d'un ordinateur séquentiel. Multiplier les ressources qui possèdent chacune leur mémoire permet d'accroître la taille de la mémoire totale adressable. Certaines organisations d'architectures parallèles permettent donc d'adresser plus de mémoire que des architectures séquentielles.

La majorité des applications requérant de hautes performances appartiennent au « **supercomputing** » ou au « **commercial computing** ». Le premier domaine concerne les applications du traitement numérique (applications scientifiques ou en ingénierie) alors que le deuxième concerne principalement les applications avec données massives (*Big data*, *Cloud*, *data centers*). Ces deux domaines recouvrent principalement quatre types d'applications : la simulation numérique, l'analyse extensive de grands volumes d'informations, les serveurs de ressources et les applications à contraintes (temps réel, service continu).

La figure 1, extraite de [1], illustre les besoins en performance de traitement et en capacité mémoire pour un ensemble d'applications de simulation numérique. Ces applications ont des besoins en puissance de calcul et en capacité mémoire.

En fait, les applications se décomposent en deux catégories :

- les applications limitées par le débit mémoire ;
- les applications limitées par la puissance de calcul.

C'est pourquoi les applications sont placées sur le graphique de la figure 1 en fonction de leur besoin en mémoire (en peta-octets, soit  $10^{15}$  octets) et du débit mémoire nécessaire (octet/flop). Le modèle « *roofline* » présenté au chapitre 9.3 utilise une décomposition du même type en octet/Flop et performance de calcul (GFlop/s) pour déterminer les types d'optimisation utilisables en fonction des limites des applications (mémoire ou calcul).

Le tableau 1, publié en 2013 [2], présente les besoins estimés pour des applications dites « exascale » en 2020.

Le besoin de hautes performances n'est pas limité aux super-ordinateurs, mais concerne l'ensemble de la gamme des ordinateurs. L'article [H 1 058] présente les besoins en puissance de calcul pour les applications des *smartphones*, tablettes et PC. Pour ne donner ici qu'un seul exemple, le tableau 2 présente l'évolution des besoins (calcul et débit mémoire) de standards successifs de la téléphonie mobile.

## 1.2 Mur de la chaleur

De la naissance des microprocesseurs (1974) au début des années 2000, il y a eu une distinction nette entre les architectures monoprocesseurs (microprocesseurs) et les architectures parallèles. Ces dernières, des multiprocesseurs symétriques simples (voir 5.2.2) utilisés dans les serveurs aux machines massivement parallèles, étaient réservées aux applications dépassant les possibilités des monoprocesseurs. La progression exponentielle des performances des monoprocesseurs, de l'ordre de 50 à 60 % par an dans cette période, n'a pas diminué le besoin de machines parallèles car la taille des données des très grosses applications a également progressé de manière exponentielle : augmentation de la taille des modèles pour une meilleure qualité et précision des simulations.

À partir du début des années 2000, les limitations des monoprocesseurs et le « mur de la chaleur » ne permettent plus de continuer l'augmentation des fréquences d'horloge et ont provoqué un tournant dans l'évolution des architectures de microprocesseurs (voir [H 1 058] pour plus de détails). Les multiprocesseurs en une seule puce, appelés multicœurs, sont devenus la brique processeur de base. Ordinateurs de bureau, portables, tablettes et

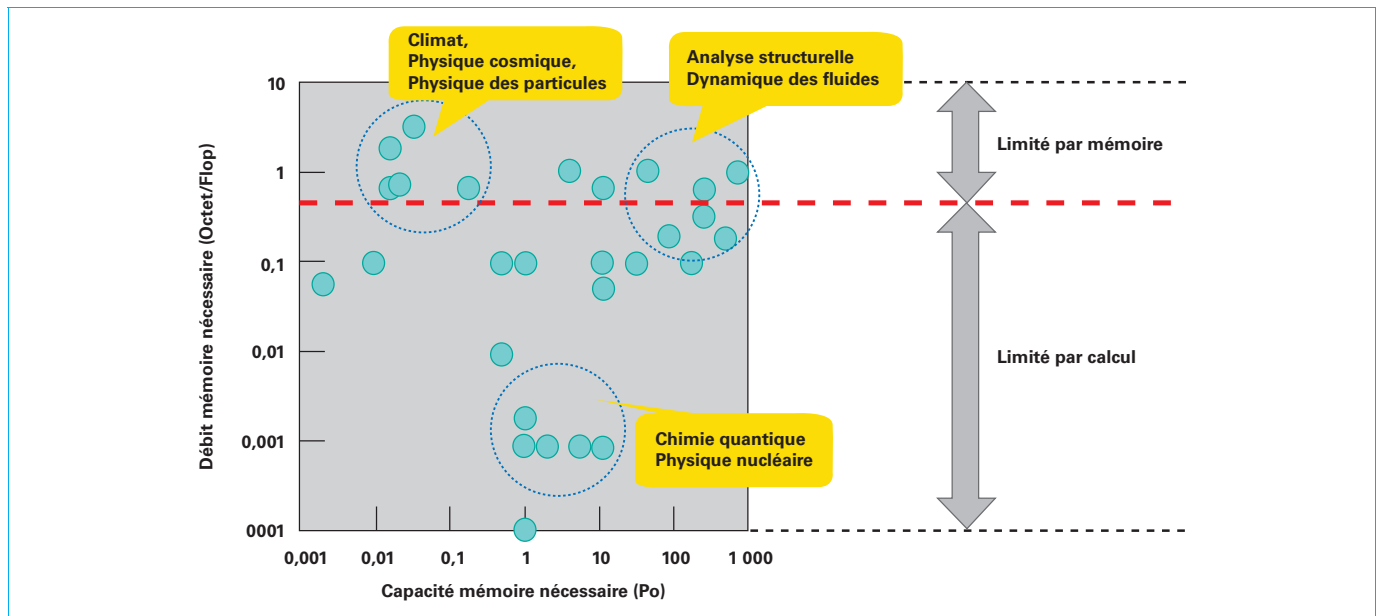


Figure 1 – Besoins des applications hautes performances

Tableau 1 – Besoins des applications « Exascale » en 2020 [2]

Applications	Débit mémoire (Octet/Flop)	Nb EFlops ( $10^{18}$ Flops)	Temps calcul estimé (heures)	Besoins mémoire (To)
Tremblements de terre	8	520	24	14
Tsunami	2,14	1 000	8	2 900
Climat/Météo	4	720	6	175
Dynamique des fluides	5,47	1	0,5	14
Simulation de désastre climatique	2 à 8	25 000	3 000	98
Turbine numérique	2,33	140	20	165

Tableau 2 – Besoins des standards de téléphonie mobile

Standard	2G GSM	2.5G GPRS	3G UMTS	3.5G HSPA	4G LTE
Débit	0,01 Mb/s	0,1 Mb/s	1 Mb/s	10 Mb/s	100 Mb/s
Calcul	5 MOPS	50 MOPS	500 MOPS	5 GOPS	50 GOPS
Puissance	100 mW	200 mW	300 mW	400 mW	500 mW
Technologie	130 nm	90 nm	65 nm	45 nm	32 nm

*smartphones* et les architectures parallèles utilisent des processeurs multicœurs. La programmation de leurs applications est parallèle. En 2016, seuls les systèmes embarqués ou enfouis bas de gamme et l'Internet des objets (IoT) utilisent des monoprocesseurs ou des microcontrôleurs.

Avant d'aller plus loin, il faut préciser ce qu'est le parallélisme.

## 2. Qu'est-ce que le parallélisme ?

### 2.1 Approche intuitive du parallélisme

C'est le parallélisme présent dans une application qui permet d'exécuter simultanément, par des ressources matérielles différentes, plusieurs parties de cette application. Les notions que nous allons introduire sont générales : elles concernent à la fois le parallélisme exploité dans les monoprocesseurs et celui exploité dans les architectures parallèles.

**Exemple** : pour introduire la notion de parallélisme, nous allons examiner la boucle suivante :

```
Pour i de 1 à n
faire A[i] ← B[i] + C[i]
FinPour
```

Le corps de cette boucle ne comporte qu'une seule opération. Si cette boucle présente du parallélisme, il faut le chercher entre les itérations. Voici les trois premières itérations :

```
A[1] ← B[1] + C[1] (a)
A[2] ← B[2] + C[2] (b)
A[3] ← B[3] + C[3] (c)
```

La sémantique introduite par le programmeur indique les résultats attendus en mémoire après l'exécution de la boucle. Quel que soit l'ordre d'exécution (a, b, c ou c, b, a, ou encore b, c, a, etc.), les résultats en mémoire sont identiques. La sémantique du programme ne dépend pas de l'ordre d'exécution des itérations de cette boucle. En particulier, l'exécution simultanée de ces trois itérations respecte la sémantique. Les itérations de cette boucle peuvent donc être exécutées en parallèle.

### 2.2 Définition formelle

Pour étudier la présence de parallélisme dans une application, nous avons besoin d'outils plus formels.

Bernstein [3] a introduit en 1966 un ensemble de conditions permettant d'établir la possibilité d'exécuter plusieurs programmes (processus) en parallèle. Supposons deux programmes : P1 et P2.

Supposons que chacun utilise des variables en entrée et produise des résultats en sortie. Nous parlerons des variables d'entrée de P1 et P2 (respectivement E1 et E2) et des variables de sortie de P1 et P2 (respectivement S1 et S2).

Selon Bernstein, les programmes P1 et P2 sont exécutables en parallèle (notation :  $P1 \parallel P2$ ) si et seulement si les conditions suivantes sont respectées :

$$\{E1 \cap S2 = \emptyset, E2 \cap S1 = \emptyset, S2 \cap S1 = \emptyset\}.$$

Plus généralement, un ensemble de programmes P1, P2... Pk peuvent être exécutés en parallèle si et seulement si les conditions de Bernstein sont satisfaites, c'est-à-dire si  $Pi \parallel Pj$  pour tout couple (i, j) avec  $i \neq j$ .

L'exemple intuitif et les conditions de Bernstein introduisent la notion de dépendance entre deux ou plusieurs programmes (ou opérations). Pour que deux programmes ou deux opérations puissent être exécutés en parallèle, il faut :

- 1) qu'ils (elles) soient indépendants (tes) ;
- 2) que l'on puisse détecter cette indépendance ;
- 3) qu'il existe suffisamment de ressources pour les exécuter simultanément.

En pratique, il existe trois limites au parallélisme : les dépendances de données, les dépendances de contrôle et les dépendances de ressources.

**Exemple** : la figure 2 présente trois exemples très simples de ces dépendances entre opérations sur des variables de type scalaire (les mêmes peuvent exister entre des programmes).

Pour chaque type de dépendance, nous présentons un programme, un graphe représentant les opérations du programme (sommets du graphe) et les dépendances (arcs) qui lient ces opérations. Une légende indique le type de dépendance en fonction du dessin des arcs.

Le premier programme présente les trois types de dépendance de données. Si l'on examine l'état mémoire supposé après l'exécution du programme (la sémantique du programme), il est aisé de vérifier que seul l'ordre I1, I2, I3 respecte la sémantique.

Dans le deuxième programme, les opérations I1 et I4 sont a priori indépendantes. Cependant, selon la valeur de A, l'opération I3 peut être exécutée introduisant une dépendance de données entre I3 et I4. Il existe une dépendance de contrôle entre I2 et I3 (l'exécution de la condition doit précéder l'exécution du corps de la conditionnelle). Comme il existe aussi une dépendance de données entre I1 et I2, I1 et I4 ne peuvent pas être exécutés en parallèle.

Le troisième programme présente le cas simple des dépendances de ressources. En l'absence de ressources en nombre suffisant, certains calculs devront être « séquentialisés » ; c'est-à-dire exécutés les uns après les autres alors qu'il n'existe ni dépendance de données, ni dépendance de contrôle.

Pour le second programme, si les variables sont des vecteurs et les itérations sont réparties entre les différents processeurs, il est possible de paralléliser I2-I3 avec une barrière de synchronisation avant I4.

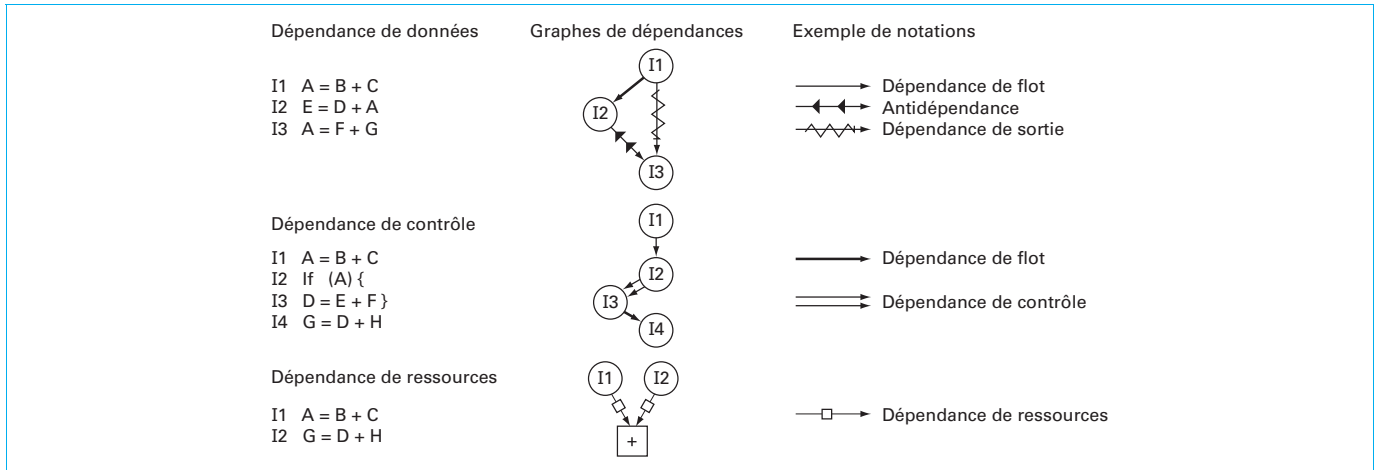


Figure 2 – Trois types de dépendance limitant le parallélisme

### 3. Sources du parallélisme et opérations fondamentales

#### 3.1 Parallélisme de données

Dans le parallélisme de données, la même opération est réalisée simultanément par plusieurs processeurs sur des données différentes. Cette définition recouvre deux notions : la présence d'un parallélisme issu des données et la manière d'exploiter ce parallélisme.

**Exemple :** prenons un exemple simple d'algèbre linéaire et plus particulièrement de calcul matriciel. L'addition de deux matrices consiste, pour tous les éléments de mêmes indices des deux matrices opérantes à les additionner et à ranger le résultat dans l'élément de même indice de la matrice résultat. Voici la boucle correspondante pour les matrices opérantes B et C et la matrice résultat A :

```
Pour i de 1 à n
  Pour j de 1 à n
    A[i][j] ← B[i][j] + C[i][j]
  FinPour
FinPour
```

Comme pour la boucle étudiée au paragraphe 2.1, les itérations de cette boucle sont indépendantes. Il y a  $n^2$  itérations avec une opération par itération. Le potentiel de parallélisme exploitable dans cette boucle est donc de  $n^2$  opérations simultanées.

L'ampleur du potentiel de parallélisme exploitable dépend directement de la taille des structures de données manipulées. L'exploitation de ce parallélisme est fondamentale car les structures de données manipulées dans les applications numériques, les traitements de base de données, le traitement du signal et de l'image sont généralement très grandes : matrices de plusieurs milliers d'éléments de côté, base de données avec des millions d'entrées, des millions de pixels par image. Les données sont de loin la source de parallélisme qui offre le plus de potentiel.

La définition indique aussi une manière d'exploiter ce parallélisme. Il s'agit d'utiliser de nombreux processeurs simultanément et de leur faire exécuter la même opération. Généralement, le nombre de processeurs est beaucoup plus petit que le parallélisme

potentiel et chaque processeur devra donc traiter plusieurs données. Nous verrons au § 4 que ce parallélisme de données est utilisé dans les monoprocesseurs avec les instructions SIMD et les GPU avec le modèle d'exécution SIMT. L'utilisation la plus efficace du parallélisme de données combine son utilisation dans les monoprocesseurs maintenant appelés cœurs (instructions SIMD) et son utilisation entre les différents cœurs pour les architectures multicœurs ou clusters de multicœurs (voir 5.2.3). Les GPU utilisent massivement le parallélisme de données. (Voir [H 1 058] pour l'utilisation des GPU comme accélérateurs).

#### 3.2 Parallélisme de contrôle

Dans le parallélisme de contrôle, des opérations différentes sont réalisées simultanément. Ce parallélisme peut provenir de l'existence dans le programme de fonctions indépendantes. Il peut aussi provenir d'opérations indépendantes dans une suite d'opérations. Ce parallélisme ne dépend donc pas des données mais de la structure du programme à exécuter. C'est l'absence de dépendances entre différentes parties du programme (quelle que soit leur taille : fonctions, boucles, opérations) qui est la source du parallélisme de contrôle.

**Exemple :** voici un exemple très simple de programme pour un serveur :

```
faire toujours
  détecter (demande_client)
  si (demande_client = vrai)
    lancer (traitement_client)
  fin si
finfaire
```

Voici le programme correspondant pour le traitement client :

```
début
...
ouvrir (fichier_client)
faire (traitement_demandé)
fermer (fichier_client)
...
fin
```



Supposons que la fonction `lancer ( )` termine son exécution immédiatement après avoir lancé le programme `traitement_client` ; c'est-à-dire sans attendre que celui-ci se termine. Dans ce cas, le programme du serveur continue à s'exécuter alors que le programme `traitement_client` est en cours d'exécution. S'il y a suffisamment de ressources, ces deux programmes seront exécutés simultanément. Si le serveur reçoit une nouvelle demande `client`, il lancera l'exécution du nouveau `traitement_client` de la même manière. Il est donc possible, à un instant donné, que plusieurs `traitement_client` s'exécutent en même temps (avec des niveaux d'avancement différents) en plus du programme serveur. Comme ces programmes peuvent réaliser des opérations différentes, il ne s'agit pas de parallélisme de données mais bien de parallélisme de contrôle. L'exploitation de ce parallélisme suppose que les processeurs fonctionnant simultanément soient capables de dérouler leur propre programme (puisque les traitements peuvent être tous différents).

Il existe une autre forme d'exploitation du parallélisme de contrôle. Elle est appropriée à un cas particulier de parallélisme de contrôle pour lequel il existe une dépendance entre les parties (fonctions, boucles, opérations) du programme qui peuvent être exécutées en parallèle.

**Exemple :** soit une application vidéo pour laquelle une scène est filmée (opération F) en permanence par une caméra numérique qui fournit un flux d'images numérisées ( $i_0, i_1, \dots$ ). On applique à chaque image un filtrage numérique (opération N) qui, à son tour, fournit une image numérisée ( $j$ ). Cette image est ensuite compressée (opération C qui fournit l'image  $k$ ) puis elle est stockée (opération S). L'algorithme de cette application est le suivant :

```
x = 0
tant que la scène est filmée faire
    ix = F ( )
    jx = N(ix)
    kx = C(jx)
    S(kx)
    x = x + 1
finfaire
```

Les opérations F, N, C et S sont toutes liées par des dépendances de données. Pourtant, ces opérations sont exécutables en parallèle. Il est impossible d'exécuter ces opérations simultanément pour la même image. L'exécution simultanée de ces opérations réside dans le traitement par chacune d'elles d'une image différente. Autrement dit, lorsque F fournit l'image  $i_n + 3$ , N traite l'image  $i_n + 2$ , C traite l'image  $i_n + 1$  et S stocke l'image  $i_n$ . Lorsque ces opérations sont terminées, toutes propagent leur résultat à l'opération suivante et traitent une nouvelle image. Une chaîne est constituée et fonctionne tant que la scène est filmée.

Cette forme d'exploitation du parallélisme de contrôle s'appelle le **pipeline**. Elle repose, comme nous venons de le voir, sur l'organisation des données à traiter sous la forme d'un flux d'informations et sur l'application de plusieurs traitements consécutifs sur chaque élément de ce flux.

### 3.3 Opérations fondamentales du parallélisme

Dans les applications, il existe des données de type scalaire et des structures de données appelées **collections** (typiquement les tableaux) regroupant plusieurs données. Lors de l'exécution, les données scalaires sont typiquement calculées par un seul processeur et les collections sont traitées en parallèle par plusieurs processeurs. Si une donnée scalaire est nécessaire pour le traitement parallèle d'une collection, la donnée doit être diffusée à tous

les processeurs devant participer au calcul sur la collection. De même, un calcul scalaire peut nécessiter le résultat d'un calcul sur une collection. Il est donc nécessaire de pouvoir étendre une donnée scalaire pour permettre le calcul d'une collection et aussi de pouvoir réduire une collection pour permettre un calcul scalaire. Les changements de dimensions (scalaire  $\rightarrow$  collection, collection  $\rightarrow$  scalaire) sont réalisés par des **opérations spatiales** dites de diffusion et de réduction. Ces opérations sont dites spatiales car elles n'intègrent pas de composante temporelle.

Il existe aussi des opérations spécifiques du parallélisme liées à la coordination temporelle des traitements parallèles. La coordination temporelle peut être impliquée par une dépendance de ressources, c'est-à-dire une situation pour laquelle il existe un nombre de ressources inférieur au nombre de demandes simultanées d'accès à ces ressources. Dans ce cas, le mécanisme utilisé s'appelle un **verrou**. La coordination temporelle peut aussi être nécessaire, pour respecter la sémantique d'un programme. Deux cas de figure se présentent : un ordre d'accès particulier à une ressource doit être respecté par les traitements parallèles ou une progression simultanée des traitements parallèles doit être garantie. Le premier cas de figure est traité par un **sémaphore** ; le deuxième par une **barrière**. Ces trois opérations (verrou, sémaphore et barrière) forment la base des opérations dites de **synchronisation**.

Dans le modèle « mémoire partagée », les différents processeurs exécutent des **threads** qui communiquent via des variables globales. L'activité de ces différents **threads** doit être synchronisée selon deux types de synchronisation :

- l'exclusion mutuelle où un seul **thread** est autorisé à accéder à une variable globale ou à une section critique ;
- la synchronisation d'événements qui peut se faire point par point, par groupe ou synchronisation globale (barrière de synchronisation).

Dans les microprocesseurs, la réalisation de l'exclusion mutuelle fait appel à des **primitives atomiques** dont le rôle est de réaliser plusieurs instructions de manière inséparable : par exemple, lire une case mémoire dans un registre, comparer avec un autre registre et écrire cette même case mémoire à partir d'un troisième registre si le résultat de la comparaison est positif. Ces trois instructions sont atomiques parce qu'aucun autre microprocesseur, processus ou **thread** ne peut réaliser d'instruction avec cette case mémoire pendant la primitive atomique.

Il existe trois primitives de base pour implanter l'exclusion mutuelle :

- **Test and Set** : La figure 3 présente la primitive « test and set (T&S) ». La variable « verrou » est un booléen de valeur 1 ou 0. Lorsque verrou = 1, T&S renvoie 1. Lorsque verrou = 0, T&S renvoie 0. Dans les deux cas, verrou = 1 en sortie de T&S ;
- **Compare and Swap** : C&S compare de manière atomique le contenu d'une case mémoire avec une certaine valeur et, en cas d'égalité, écrit en mémoire une autre valeur, selon le code de la figure 4 ;
- **Fetch and Add** : F&A, de manière atomique, lit une variable, ajoute une valeur et réécrit la nouvelle valeur dans la variable selon le code de la figure 5.

Les barrières de synchronisation permettent une synchronisation globale d'un certain nombre de processus. Elles permettent de garantir qu'un certain nombre de processus ont atteint un certain point dans l'exécution parallèle. Il existe différentes manières d'implanter ces barrières. À titre d'exemple, la figure 6 montre le code avec la primitive **Fetch and Add** permettant de lancer l'exécution de N processus et d'implanter une barrière de synchronisation en fin d'exécution de tous les processus.

Les jeux d'instructions des processeurs ont des instructions utilisables ou spécialement définies pour implanter les primitives de synchronisation et les barrières.

```

int TestAndSet(int *verrou) {
    int Valinit;

    Valinit = *verrou; // Les deux instructions en rouge doivent être atomiques
    *verrou = 1;

    return Valinit; }

volatile int *verrou;

void Critical() {
    while (TestAndSet(*verrou) == 1);

    code section critique // un seul processus peut être dans cette section

    *verrou = 0; // libération du verrou en fin de section critique
}

```

Figure 3 – Code de la primitive Test and set et utilisation pour créer une section critique

```

int compareAndSwap(int *loc, int attendu, int range) {
    int val = *loc; // Les deux instructions en rouge doivent être atomiques
    if (val == attendu) *loc = range;
    return val; }

```

Figure 4 – Code de la primitive Compare and Swap

```

int FetchAndAdd(*loc, int inc) {
    int val = *loc; // Les deux instructions en rouge doivent être atomiques
    *loc = val + inc
    return val; }

```

Figure 5 – Code de la primitive Fetch and Add

```

Main()
{
    int X=0,Y=N ; // X et Y sont des variables partagées
    // Lancement de N processus Pi.
    while F&A(Y,0) != 0 ; //barrière de synchronisation

    Process I {
        int i
        i=F&A(X,1) // Numérotation successive des processus de 1 à N
        Code du processus i
        F&A(Y,-1) // Chaque Pi se terminant décrémente Y (de N à 1).
        // Suppose que tous les processus aient incrémenté
        // X avant de commencer à décrémente Y
    }
}

```

Figure 6 – Utilisation du Fetch and Add pour lancer des processus et implanter une barrière de synchronisation

Par **exemple**, le jeu d'instructions  $\times 86$  (IA-32 et Intel64 [4]) permet de garantir l'atomicité par verrouillage de bus à l'aide du préfixe **LOCK**. L'instruction **XCHG** est atomique pour la permutation d'un registre et d'une case mémoire. Des instructions spéciales implémentent les barrières de synchronisation, respectivement **LFENCE** (sur les lectures), **SFENCE** (sur les écritures) et **MFENCE** (sur lectures et écritures). Le jeu d'instructions ARM dispose d'instructions d'échange atomique (**SWP** et **SWPB**), des instructions **LDREX** et **STREX** qui permettent la lecture-modification-écriture de manière atomique et d'une barrière de synchronisation (**DMB**).

Le lecteur intéressé pourra consulter [5].

Les bibliothèques logicielles associées aux systèmes d'exploitation ou compilateurs (GCC, CLang, Windows, etc.) fournissent des fonctions de synchronisation pour la programmation parallèle et

concurrente. Ces fonctions sont aussi disponibles dans la bibliothèque standard de C++2011 [6].

### 3.4 Consistance mémoire

Dans le modèle mémoire partagée (voir § 5.2.1), les processeurs lisent et écrivent dans la mémoire commune. Pour simplifier la présentation, on ne considérera pas la hiérarchie de caches. La consistance mémoire correspond à la cohérence de l'état de la mémoire telle qu'elle est vue à un instant donné. La lecture d'une variable doit retourner la dernière valeur écrite dans l'ordre du programme.

La **consistance séquentielle** est le modèle le plus simple. Sa définition par Lamport (1979) est la suivante : « un multiprocesseur est séquentiellement consistant si le résultat de toute exécution est le



même que si toutes les opérations de tous les processeurs s'exécutent dans un certain ordre séquentiel, et que les opérations de chaque processeur individuel apparaissent dans cette séquence dans l'ordre spécifié par le programme ». Il y a donc deux contraintes :

- séquentialisation de tous les accès mémoire ;
- accès mémoire dans l'ordre du programme pour chaque processeur individuel.

Ces contraintes impliquent que les opérations mémoire soient atomiques, c'est-à-dire ne puissent pas être interrompues.

Soit deux processeurs P1 et P2 et deux variables A et B initialement à 0.

Le tableau 3 donne un exemple de programme et les résultats possibles avec la consistance séquentielle.

La consistance séquentielle est très restrictive. Elle se heurte aux caractéristiques des microprocesseurs visant à améliorer la performance :

- mécanismes pour diminuer la latence mémoire (caches, tampons d'écriture, lectures non bloquantes) ;
- l'interconnexion mémoire peut modifier l'ordre des écritures envoyées par les processeurs ;
- les optimisations du compilateur peuvent réordonner les opérations mémoire.

Un certain nombre de modèles de **consistance relâchée** ont donc été définis. Ces modèles relâchent certaines contraintes d'ordre d'opération mémoire (Écriture avant Lecture : E → L ; Écriture avant Écriture : E → E ; Lecture avant Écriture : L → E et Lecture avant Lecture : L → L), de la consistance séquentielle. On peut citer notamment :

- consistance processeur et ordre de rangement total. La contrainte E → L est relâchée : un processeur peut relire une donnée qu'il vient d'écrire dans son cache avant que l'écriture soit terminée (les écritures sont rangées dans un tampon d'écriture) ;
- ordre de rangement partiel. Les contraintes E → L et E → E sont relâchées : l'ordre des écritures peut être modifié quand elles portent sur des données différentes ;
- ordre faible et consistance à la libération. Les quatre types de réordonnement mémoire (L → L, L → E, E → L, E → E) sont possibles. Le processeur dispose d'instructions de synchronisation. Les opérations mémoire avant la synchronisation doivent se terminer avant le début de la synchronisation. Les opérations mémoire après la synchronisation ne peuvent commencer avant la fin de la synchronisation.

Chaque jeu d'instructions définit un modèle de consistance mémoire. Par exemple, [7] définit le modèle de consistance mémoire

**Tableau 3 – Exemple de programme et d'exécution obéissant à la consistance séquentielle**

P1	P2
(1a) A = 1 ;	(2a) print B ;
(1b) B = 2 ;	(2b) print A ;
<i>Suite d'instructions SC</i>	<i>Résultat</i>
2a → 2b → 1a → 1b	A = 0, B = 0
1a → 2a → 1b → 2b	A = 1, B = 0
1a → 2a → 2b → 1b	A = 1, B = 0
1a → 1b → 2a → 1b	A = 1, B = 2

pour le jeu d'instructions Intel 64. Le lecteur intéressé par une étude approfondie des modèles de consistance mémoire peut consulter [8].

## 4. Parallélisme dans les monoprocesseurs

Les monoprocesseurs utilisent un parallélisme compatible avec une programmation séquentielle, c'est-à-dire que le flux d'exécution des instructions est strictement séquentiel. Ce parallélisme est appelé « **parallélisme d'instructions** » car il correspond aux instructions du programme séquentiel qui peuvent s'exécuter en parallèle, sans dépendances de données, de contrôle ou de ressources.

Le temps d'exécution d'un programme est donné par l'équation suivante :

$$T_{ex} = NI * CPI * T_c = \frac{NI}{IPC * F}$$

où :

- NI est le nombre d'instructions à exécuter ;
- CPI est le nombre de cycles d'horloge par instruction. Il comprend les cycles d'exécution des instructions et les cycles supplémentaires d'attente des données mémoire ;
- IPC le nombre d'instructions exécutées par cycle ;
- T<sub>c</sub> est le temps de cycle d'horloge et F la fréquence d'horloge.

L'augmentation de la fréquence d'horloge F que permettent les nœuds technologiques CMOS successifs a longtemps été le moyen le plus simple d'augmenter les performances. Avec le « mur de la chaleur », les fréquences d'horloge des processeurs dépassent rarement 4 GHz.

Au terme IPC correspond le parallélisme d'instructions. Il correspond à l'utilisation des pipelines (voir [H 1 004]), à l'exécution superscalaire des instructions dans l'ordre [H 1 010] et non ordonnée [H 1 011] ou à l'exécution VLIW [H 1 012]. Alors que le pipeline simple a pour limite IPC = 1, les processeurs superscalaires ont permis rapidement d'avoir IPC supérieur à 1, mais ces processeurs ont atteint très vite des limites.

Par **exemple**, les cœurs des processeurs Intel en 2015 ne peuvent exécuter plus de 4,35 pseudo-instructions RISC (μops) résultant de la traduction des instructions IA-32 ou Intel64 par cycle d'horloge.

Le parallélisme d'instructions est exploité par le matériel dans les superscalaires ou via le compilateur dans l'approche VLIW, dans les deux cas à partir d'un programme séquentiel.

Les monoprocesseurs utilisent également des techniques pour diminuer NI :

- les instructions SIMD sont apparues dans les jeux d'instructions au début des années 1990. L'acronyme SIMD a été introduit par Flynn en 1966 pour les machines parallèles (voir 5.1). Pour les monoprocesseurs, le principe des instructions SIMD est illustré par la figure 7 : elles effectuent la même opération sur des sous-ensembles des opérandes contenus dans un registre ou une case mémoire. On peut considérer un opérande comme un vecteur des différents sous-ensembles. Pour les extensions Intel, les registres ont 128 bits (SSE2/3/4), 256 bits (AVX), 512 bits (AVX2) et d'autres versions verront le jour. Les données dans un registre SIMD peuvent être 8/16/32 bits pour les entiers, 32 et 64 bits pour les flottants. Les extensions SIMD permettent de diminuer le nombre d'instructions exécutées (voir [H 1 200] pour une étude des extensions SIMD) dans un programme séquentiel ;
- les processeurs graphiques (GPU) utilisent un modèle d'exécution SIMT (*single instruction multiple threads*) illustré par

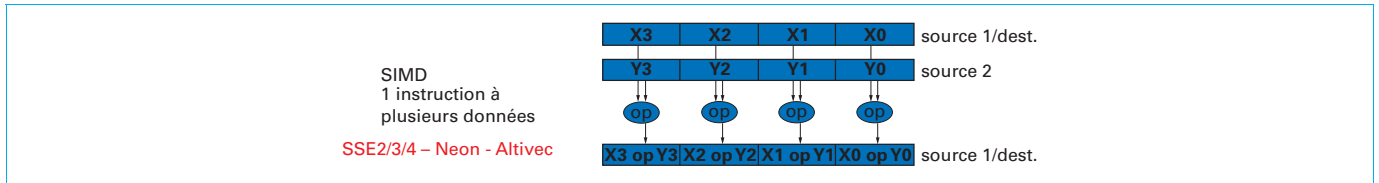


Figure 7 – Principe des instructions SIMD

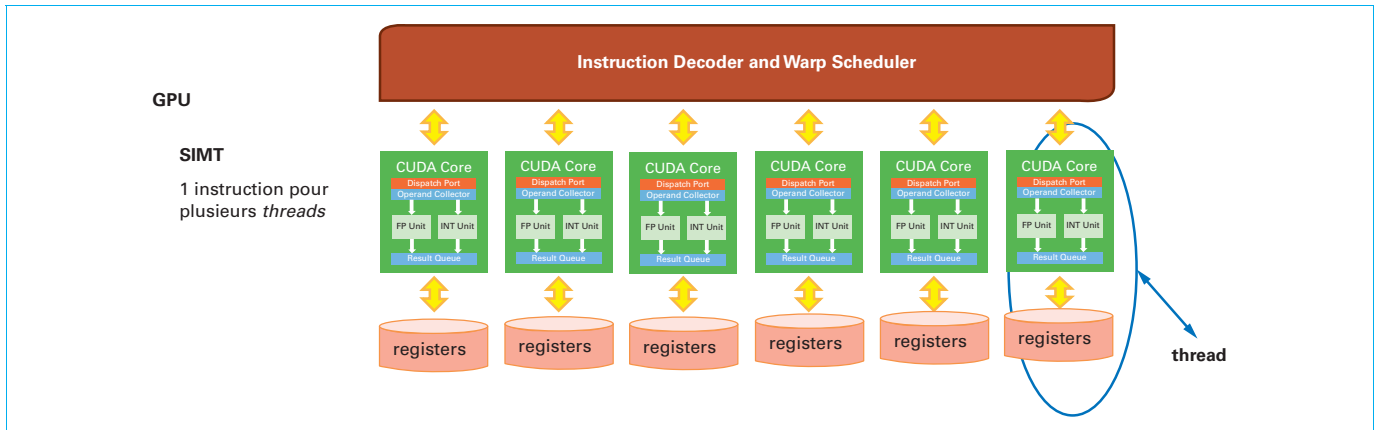


Figure 8 – Modèle d'exécution SIMT

la figure 8. Une instruction exécute la même opération sur plusieurs *threads*. Les GPU se sont développés comme accélérateurs pour les programmes exhibant un fort parallélisme de données. Là encore, le flux d'exécution des instructions SIMT est séquentiel.

L'autre possibilité pour diminuer NI est de répartir NI sur plusieurs cœurs : si NIt est l'ensemble des instructions à exécuter pour l'ensemble du programme, et que l'on dispose de  $n$  processeurs (ou cœurs), on répartit NIt instructions sur les différents cœurs. Dans le cas idéal, on a pour chaque cœur

$$NIc = NI/n$$

C'est le principe des architectures parallèles, exploitant un parallélisme nécessitant une programmation parallèle.

## 5. Classification des architectures parallèles

Pour une architecture utilisant plusieurs processeurs, la diversité des ressources et leur nombre important conduit à une grande variété d'agencements possibles de ces ressources. Pour distinguer des familles d'architectures parallèles, on utilise des classifications.

### 5.1 Classification de Flynn

La classification la plus utilisée est connue historiquement sous le nom de classification de Flynn [9]. Elle classe les architectures suivant les relations entre les unités de traitement et les unités de contrôle. Les unités de traitement calculent chacune un flux de données ou « *Data stream* ». Les unités de contrôle déroulent chacune

Tableau 4 – Classification des architectures parallèles selon Flynn

Flux	1 flux d'instruction (1 programme)	Plusieurs flux d'instructions
1 flux de données	SISD	MISD ( <i>pipeline</i> )
Plusieurs flux de données	SIMD	MIMD

un programme (un flux d'instructions ou « *Instruction stream* »). Le tableau 4 présente cette classification.

Le modèle SISD (*Single Instruction stream Single Data stream*) correspond au modèle classique de Von Neuman (architecture séquentielle) pour lequel une seule unité de traitement reçoit un seul flux d'instructions pour traiter un seul flux de données. Le monoprocesseur exécutant des instructions scalaires correspond au modèle SISD du point de vue du programme. Il utilise cependant le plus souvent le parallélisme *pipeline* ou de type superscalaire.

Dans une architecture SIMD (*Single Instruction stream Multiple Data stream*), une seule unité de contrôle gère le séquençement du programme pour plusieurs unités de traitement. Un schéma simplifié est présenté dans la figure 9.

Les processeurs élémentaires (PE) reçoivent les mêmes instructions en même temps et calculent sur un flot de données différent. Des exemples de telles architectures sont la CM2 de Thinking Machine et MasPar dans les années 80. Les architectures parallèles SIMD ont disparu pour plusieurs raisons :

- le contrôleur et les processeurs élémentaires sont spécifiques et donc plus coûteux que les microprocesseurs standards ;
- une fréquence d'horloge de 12,5 MHz (MasPar) permet de diffuser une instruction à des dizaines de PE. C'est évidemment impossible avec des fréquences d'horloge de plusieurs GHz ;

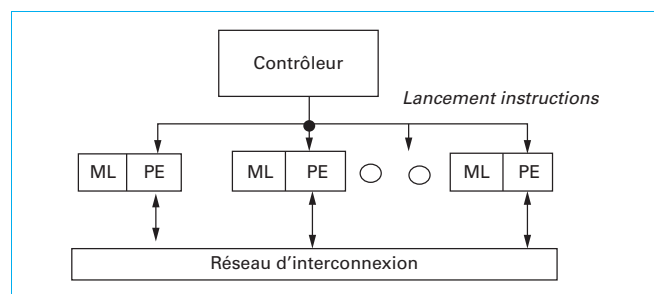


Figure 9 – Schéma de principe des architectures SIMD

- avec l'approche SIMD, les instructions conditionnelles du type « si condition alors séquence\_vrai sinon séquence\_faux » conduisent à l'exécution successive des séquences vrai et faux selon le résultat de la comparaison sur chacune des données ;
- le modèle SIMD peut être émulé sur des architectures MIMD avec le modèle d'exécution SPMD (Single Program Multiple Data Stream).

Le modèle d'exécution SIMD a trouvé une nouvelle vie dans les extensions SIMD des microprocesseurs.

Il n'y a pas eu d'architectures matérielles MISD réellement implantées. Le MISD (Multiple Instruction stream Single Data stream) où un seul flux de données reçoit plusieurs traitements simultanément, correspond cependant assez bien au mode *pipeline* d'exploitation du parallélisme de contrôle présenté en 3.2. Il est implanté sur des architectures MIMD.

L'organisation MIMD (Multiple Instruction stream Multiple Data stream) est la plus utilisée. Plusieurs unités de contrôle gèrent chacune une ou plusieurs unités de traitement. Ce modèle permet donc d'exécuter un programme différent sur tous les processeurs. Les applications peuvent donc être conçues comme un ensemble de tâches différentes coopérantes (applications parallèles et/ou distribuées). Le modèle MIMD correspond aussi à l'organisation de la grande majorité des architectures parallèles. La tendance à l'utilisation massive de composants standards pour la réalisation des machines parallèles se concrétise par l'adoption des microprocesseurs pour réaliser les parties calcul et contrôle de ces machines. Dans les microprocesseurs, on trouve une unité de contrôle et une ou plusieurs unités de traitement. En répliquant les microprocesseurs, on réplique donc, dans la même proportion, les unités de contrôle et les unités de traitement. Lors de l'exécution d'un programme parallèle, plusieurs flux d'instructions différents peuvent donc être déroulés pour traiter plusieurs flux de données.

Pour le traitement numérique, écrire des programmes différents est très difficile et souvent inutile. Aussi, on préfère exécuter le même programme sur tous les processeurs. Le programme traite des données différentes sur chaque processeur. Ce mode d'exécution est nommé SPMD (Single Program over Multiple Data stream).

## 5.2 Classification selon le modèle mémoire

La figure 10 présente la hiérarchie mémoire d'un monoprocesseur des années 2000 : processeur, caches L1 et L2 et mémoire principale.

Avec plusieurs processeurs, il y a différentes manières d'organiser la mémoire.

### 5.2.1 Mémoire physiquement centralisée ou répartie

La figure 11 illustre la différence entre les deux organisations mémoire.

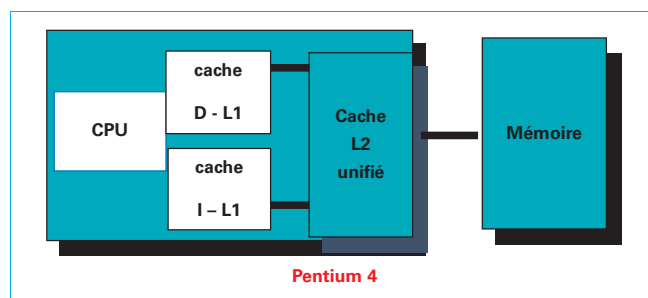


Figure 10 – Hiérarchie mémoire d'un monoprocesseur

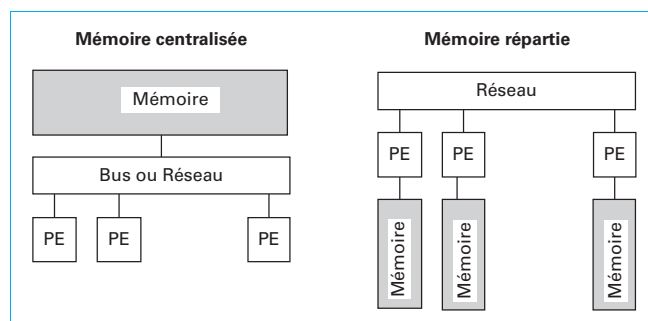


Figure 11 – Mémoire centralisée (multiprocesseur) et mémoire répartie (multi-ordinateur)

La partie gauche correspond à une mémoire physiquement centralisée. Cette organisation correspond aux architectures appelées **multiprocesseurs**. Les différents processeurs (PE) accèdent à une seule mémoire et ont donc le même espace d'adressage. Si la connexion entre processeurs et mémoire se fait via un bus, les temps d'accès mémoire sont uniformes (si l'on ne considère pas les effets de cache). Si la connexion a lieu via un réseau, on a des temps d'accès non uniformes.

La partie droite correspond à une mémoire répartie, constituée de l'ensemble des mémoires locales à chaque processeur. Cette organisation correspond aux architectures appelées **multi-ordinateurs** ou **clusters d'ordinateurs** : des ordinateurs individuels sont connectés via un réseau. Chaque processeur peut accéder à sa mémoire locale privée via les accès mémoire monoprocesseur. L'accès d'un processeur à la mémoire locale d'un autre processeur implique un passage de messages via le réseau d'interconnexion. Les accès mémoire sont non uniformes car ils dépendent de la nature locale ou non de l'accès.

### 5.2.2 Mémoire physiquement répartie et logiquement centralisée

Le **multiprocesseur symétrique** (SMP), c'est-à-dire le multiprocesseur à bus, présente l'avantage d'un espace d'adressage unique et l'inconvénient de ne supporter qu'un nombre limité de processeurs (de l'ordre de 8). Interconnecter plusieurs SMP avec un réseau assurant la cohérence des caches permet d'avoir une mémoire physiquement répartie et logiquement partagée (*Cache Coherent, Non Uniform Memory Access* : CC-NUMA), dont la version la plus simple est présentée en figure 12.

### 5.2.3 Multicœurs et clusters de multicœurs

Les organisations mémoire présentées précédemment supposaient des PE monoprocesseurs. Depuis le milieu des années 2000, les multicœurs sont devenus la norme. Un multicœur est un multiprocesseur dans une seule puce, avec des hiérarchies de

cache complexes. La figure 13 résume l'évolution des multicœurs d'IBM entre 2010 et 2014. Le Power6 en 2010 a deux cœurs avec des caches L2 privés. Le Power8 en 2014 a huit cœurs, des caches L2 privés, des caches L3 physiquement distribués mais logiquement partagés. Il y a de nombreuses organisations possibles de la hiérarchie des caches.

De plus, pour disposer de plus de cœurs que le nombre disponible dans un multicœur, la solution est d'utiliser un cluster, comme le montre un exemple AMD de clusters de multicœurs (figure 14). La hiérarchisation est alors puce multicœur, socket et ensemble de sockets.

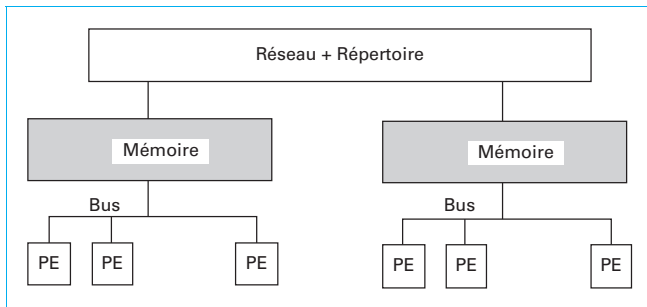


Figure 12 – Organisation CC-NUMA

Des niveaux supplémentaires de hiérarchie permettent de constituer des super-ordinateurs, comme le montre la figure 15 tirée de [10] :

- multicœur ;
- socket ;
- carte ;
- cabinet.

Des accélérateurs, comme un GPU, peuvent être associés à un certain niveau de la hiérarchie.

L'augmentation du nombre de cœurs par puce avec les années permet de diminuer le nombre de niveaux à nombre de cœurs constant, ou d'augmenter le nombre de cœurs à nombre de niveaux constant.

### 5.3 Classification suivant le grain de calcul

La recherche de la performance et la maîtrise de la consommation énergétique conduisent à explorer un vaste espace de configurations possibles pour les systèmes et les machines parallèles. Un des axes de cet espace est celui du grain de calcul. Il existe de nombreux doublets possibles (nombre de processeurs/cœurs, performance par processeur). La situation peut être différente selon la classe d'applications visée.

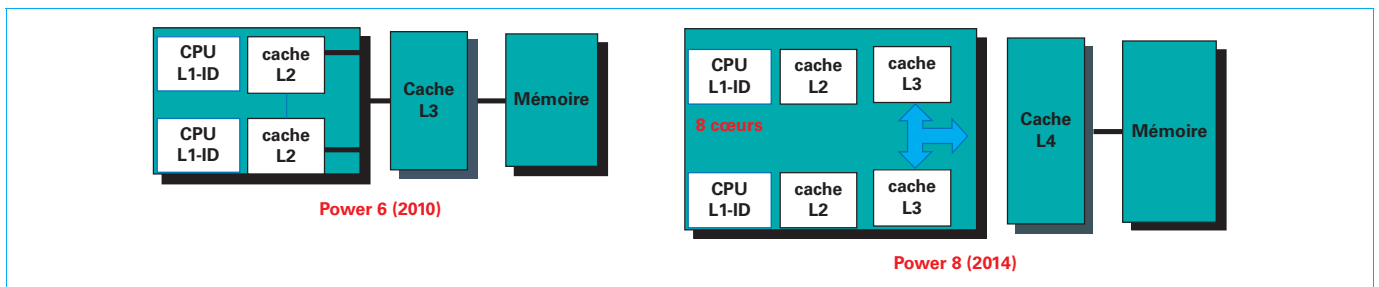


Figure 13 – Évolution des multicœurs d'IBM entre 2010 et 2014

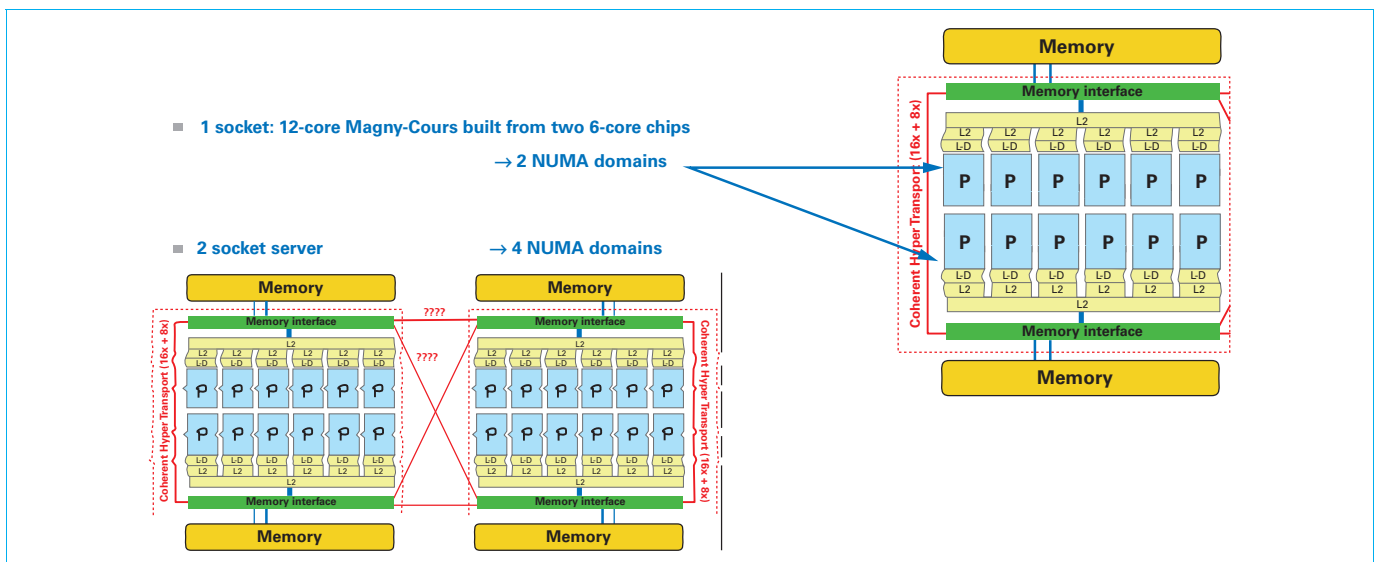


Figure 14 – Exemple de clusters de multicœurs

Dans des appareils nomades comme les *smartphones* et les tablettes, les contraintes sont la performance pour les différentes fonctionnalités, la durée de vie des batteries et l'encombrement (taille de l'appareil). Ces appareils contiennent multicœurs, GPU, capteurs, afficheurs, etc. La puissance de calcul nécessaire dans un *smartphone* varie dans le temps en fonction de son utilisation. Le choix des multicœurs peut répondre aux objectifs contradictoires : la puissance de calcul lorsqu'elle est nécessaire, le fonctionnement en veille sinon. La figure 16 montre comment le SoC utilisé dans le Galaxy S6 répartit la charge sur deux 4-cœurs en fonction de la charge de travail du *smartphone*, les cœurs A57 étant plus performants et consommant plus que les cœurs A53.

Les machines parallèles et les super-ordinateurs utilisent des processeurs multicœurs ou des processeurs many-cores (64 cœurs

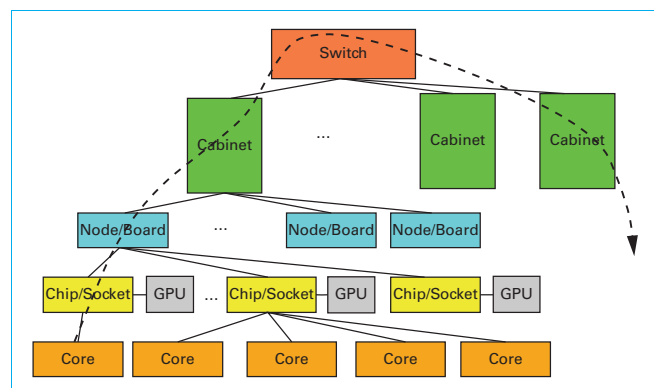


Figure 15 – Hiérarchisation dans un super-ordinateur

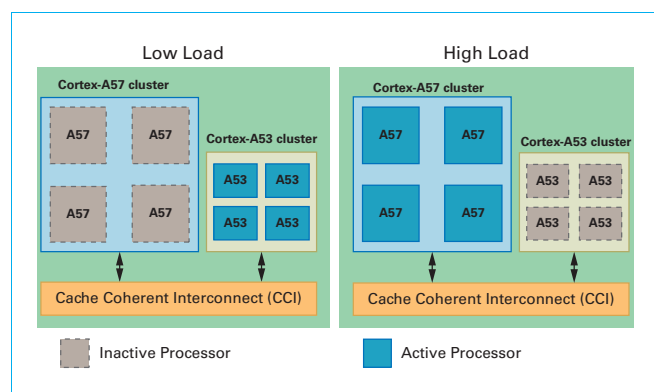


Figure 16 – Répartition de la charge de travail dans le circuit Exynos 7420 (Galaxy S6)

et plus) comme le Xeon Phi, avec éventuellement des accélérateurs (GPU). Dans les années 1980, la question a été posée : grand nombre de processeurs peu puissants ou petit nombre de processeurs puissants.

Par exemple, la Connection machine 2 de Thinking Machine (1987) possédait 65 536 processeurs calculant seulement sur des données 1 bit. Le Cray X-MP (1988) avait 8 processeurs vectoriels (voir l'annexe de [H 1 200] pour une brève introduction aux architectures vectorielles).

Pour les super-ordinateurs des années 90, où le critère performance prédominait sur le critère énergétique, l'utilisation des processeurs les plus puissants s'est imposée pour au moins deux raisons :

- les processeurs utilisés sont les microprocesseurs haut de gamme utilisés dans les PC haut de gamme et les serveurs, et non des processeurs spécifiques ;
- le temps de traitement comprend les temps de calcul, communication et synchronisation. Les temps de communication et synchronisation dépendent du rapport calcul/communication de l'application et des caractéristiques du réseau d'interconnexion. À puissance de calcul globale donnée, utiliser un minimum de processeurs diminue le nombre et la durée des communications et des synchronisations entre processeurs et la complexité des réseaux d'interconnexion.

La situation est toujours la même du point de vue de la puissance de calcul maximale. Les super-ordinateurs du TOP 500 ont un grand nombre de processeurs très puissants. Le tableau 5 donne quelques caractéristiques des 5 premiers en juin 2016.

Dans le domaine de la haute performance pour l'embarqué, la question du Nombre/Puissance des processeurs redevient d'actualité. L'architecture *many-core* Kalray [11] choisit l'option d'un grand nombre de processeurs moins puissants : elle utilise trois niveaux de hiérarchie (figure 17) exploitant trois niveaux de parallélisme. La fréquence utilisée permet de limiter la consommation.

Le tableau 6 présenté par Kalray compare les performances d'un 4-cœur d'Intel, d'un GPU et du MPPA sur un benchmark très facile à paralléliser. S'il n'est pas question de généraliser ce résultat sur un seul benchmark très favorable au MPPA, on voit que l'objectif est d'obtenir un meilleur compromis Performance \* Énergie avec une performance moindre qu'avec un GPU.

## 5.4 Architectures parallèles homogènes ou hétérogènes

La distinction entre architectures parallèles homogènes, utilisant le même type de processeur, et architectures parallèles hétérogènes, utilisant plusieurs types de processeur, est souvent utilisée. Cette classification est peu pertinente. En effet, un PC monoprocesseur contient au moins un CPU (1 seul cœur), un GPU, un contrôleur mémoire, c'est-à-dire plusieurs processeurs avec des tâches

Tableau 5 – Les 5 premiers superordinateurs du TOP 500 en juin 2016

Nom		Processeur	Accélérateur	Nb de cœurs	Réseau
TaihuLight	Sunway	SW26010		10 649 600	Sunway
Tianhe-2	MilkyWay2	Xeon E5 2692	Xeon Phi	3 120 000	TH Express-2
Titan	Cray XK7	Opteron 6274	NVIDIA K20x	560 640	Cray Gemini Interconnect
Sequoia	BlueGeneQ	Power BQC		1 572 864	Custom IBM
K computer		Sparc64 VIIIfx		705 024	Tofu Interconnect Fujitsu



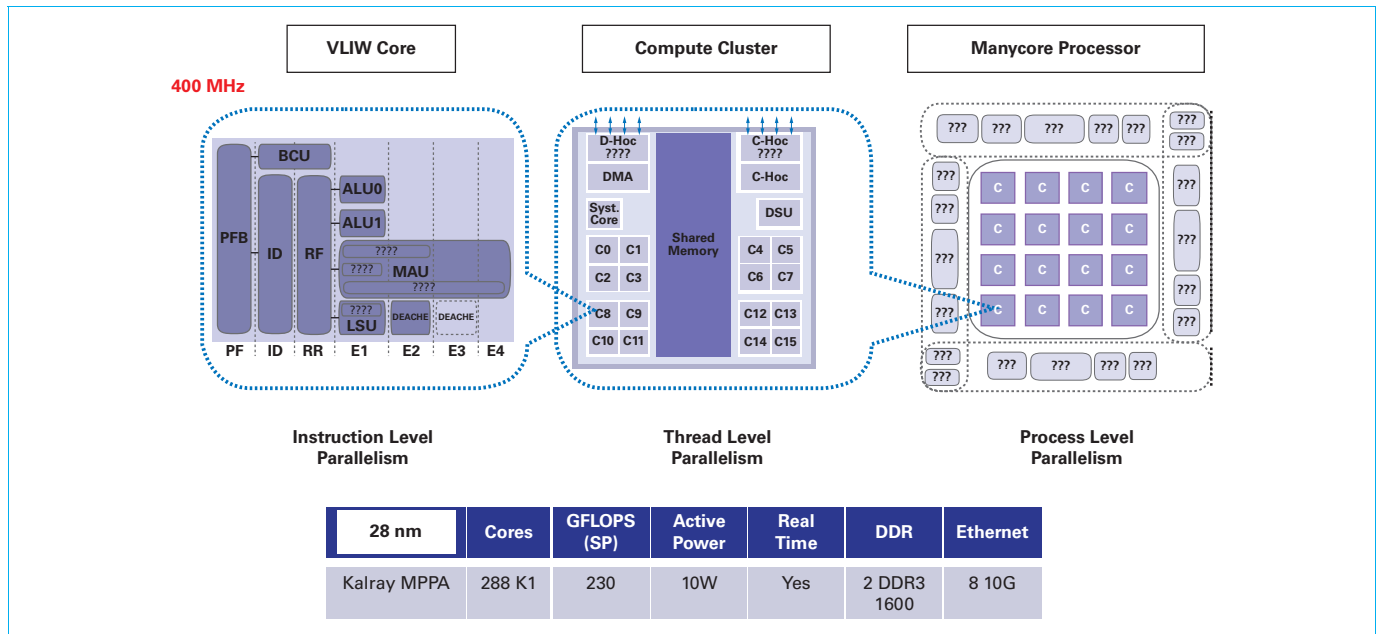


Figure 17 – Architecture MPPA 256 de Kalray [11]

Tableau 6 – Comparaison sur le benchmark « Monte Carlo Option Pricing »

Processeur	Temps (s)	Temps normalisé/ GPU	Énergie (Joule)	Énergie normalisée/ GPU
i7-3830 (4-cœur)	13,86	5,85	1802,2	3,39
Tesla C2075 (GPU)	2,37	1	531,7	1
MPPA-256	5,75	2,43	86,3	0,16

spécifiques et un tel PC n'a jamais été considéré comme une architecture parallèle. Il semble plus pertinent d'utiliser la notion de nœud, comprenant un certain nombre de processeurs, identiques ou différents, et de distinguer les architectures parallèles à un seul nœud et celles constituées de plusieurs nœuds, le nombre pouvant être très grand. À l'intérieur d'un nœud ou entre plusieurs nœuds se pose le problème de l'interconnexion des processeurs et nœuds.

#### 5.4.1 Architectures parallèles à un seul nœud

Sans entrer dans les détails, la différence essentielle entre un PC monoprocesseur et un PC avec un processeur multicœur est que le CPU à un cœur a été remplacé par un CPU multicœur. Les problèmes de hiérarchie mémoire abordés en 5.2.3 sont traités à l'intérieur de la puce du processeur. On peut donc considérer l'ensemble CPU-GPU et contrôleurs mémoire et E/S comme un « nœud de calcul » où les interconnexions sont réalisées par bus. Les appareils nomades utilisent des puces systèmes (SoC) comprenant plusieurs processeurs : multicœurs, GPU, DSP, accélérateurs matériels spécialisés qui sont interconnectés par des systèmes de bus ou des réseaux sur puce. L'exemple de la figure 16 montre que plusieurs types de multicœurs peuvent être utilisés.

Le cas des systèmes embarqués est plus complexe compte tenu de la très grande diversité de ces systèmes. Dans un grand nombre de cas, on est dans la même situation que pour les systèmes nomades. Les plates-formes de ST Microelectronics pour les boîtes de connexion TV (*set-up boxes*) en sont un bon exemple (voir [H 8 000]).

Un nœud contient donc une architecture parallèle hétérogène, avec un système d'interconnexion adapté à la classe d'application visée.

#### 5.4.2 Architectures parallèles à plusieurs nœuds

Utiliser plusieurs nœuds implique une hiérarchisation des niveaux de parallélisme, du parallélisme intra-nœud au parallélisme entre nœuds. Plusieurs nœuds peuvent être dans la même puce, dans des puces différentes ou même dans des ordinateurs différents.

L'architecture MPPA de Kalray [11] présentée dans la figure 17 est un exemple de puce « many-core ». Elle est constituée d'un ensemble de clusters de calcul (partie centrale de la figure). On peut considérer qu'il y a deux niveaux de nœuds : le cœur VLIW et le cluster de ces cœurs autour de la mémoire partagée. Pour cette architecture, le réseau d'interconnexion est une grille 2D.

Les machines parallèles et massivement parallèles vont utiliser des nœuds avec processeurs multicœurs ou many-core, avec ou sans accélérateur matériel associé. Pour les machines avec des dizaines de processeurs, on a généralement des clusters de processeurs multicœurs, selon le schéma présenté en figure 14. Pour les machines massivement parallèles comme celles présentées dans le tableau 5, des réseaux d'interconnexion spécialisés sont utilisés.

L'interconnexion de nœuds peut se faire au niveau d'un ordinateur, selon le modèle multi-ordinateur de la figure 11. C'est le cas des clusters de PC connectés à des réseaux comme Gigabit



Ethernet ou Infiniband et utilisant MPI. La présentation des différents réseaux d'interconnexion et de leurs propriétés sort du cadre de cette introduction au parallélisme.

## 5.5 Organisation du système d'exploitation dans les architectures parallèles

Les quatre fonctions fondamentales que doit remplir un système d'exploitation sont : la gestion des processus, la gestion de la mémoire, la gestion des E/S et la communication interprocessus. Dans une architecture classique, l'organisation du système et les fonctionnalités qu'il assure sont prévues pour fonctionner sur une architecture monoprocesseur. Le système d'exploitation d'une architecture parallèle doit gérer au moins deux points supplémentaires :

- a) assurer la gestion globale des ressources et l'administration des ressources parallèles ;
- b) assurer son propre fonctionnement « parallèle ».

La gestion des fonctions fondamentales doit faire appel à des mécanismes supplémentaires (par rapport à un système classique) pour la répartition de la charge sur les différentes ressources, l'organisation de l'espace d'adressage dans la mémoire multi-bancs, l'organisation des systèmes de fichiers lorsque les disques sont distribués et les communications rapides entre des processus distants. Il existe de nombreuses solutions que nous ne pouvons détailler dans le cadre de cet article.

L'un des objectifs principaux pour le système est d'offrir une image unique à la fois pour les utilisateurs et pour les ressources. Plusieurs organisations de système ont été proposées et fonctionnent pour les architectures parallèles. Il en existe principalement trois :

- 1) les agrégats de systèmes ;
- 2) les systèmes maître-esclaves ;
- 3) les systèmes symétriques.

Les agrégats de systèmes fonctionnent dans les architectures MIMD à mémoire distribuée. Souvent, dans ces architectures, chaque processeur est associé à une mémoire, un disque et une interface réseau. L'ensemble constitue un nœud. Le principe des agrégats de systèmes est de faire fonctionner sur chaque nœud un système d'exploitation complet. Sans mécanisme supplémentaire, l'ensemble des systèmes ne peut pas donner une image unique. Des mécanismes de gestion globale des processus, de gestion de la mémoire et des entrées/sorties sont donc ajoutés par-dessus le système, comme dans GLUNIX ou CONDOR, ou dans le système de chaque nœud pour donner une cohérence à l'ensemble.

Dans les systèmes de type maître-esclaves, un seul processeur exécute le système d'exploitation. Toutes les opérations de l'utilisateur et toutes les activités des ressources qui requièrent l'intervention du système d'exploitation sont exécutées sur ce processeur. L'image du système est donc unique par principe. Les autres processeurs de l'architecture parallèle reçoivent les ordres de traitement parallèle.

Dans les architectures à mémoire distribuée, les nœuds différents du maître exécutent une version minimale du système.

Dans les systèmes symétriques qui concernent principalement les architectures à mémoire partagée, tous les processeurs sont susceptibles d'exécuter l'intégralité ou une partie du système d'exploitation. Il n'y a pas en principe de site privilégié pour l'exécution du système. La manière la plus simple de réaliser un tel système consiste à considérer le système entier comme une section critique exécutable seulement par un processeur à la fois. Dans des versions plus sophistiquées, certaines parties du système peuvent être exécutées en parallèle. Cependant, la conception d'un tel système reste très complexe. Pour ces systèmes aussi,

l'image du système du point de vue des utilisateurs et des ressources est unique.

Au-delà de ces caractéristiques communes pour toutes les architectures parallèles existent des spécificités propres à chaque classe de système parallèle.

Par exemple, Android, utilisé dans les tablettes et les *smartphones*, doit prendre en compte la limitation de la consommation énergétique et l'existence d'interfaces particulières avec le monde extérieur (gyroscope, capteurs, absence de véritable clavier, etc.). Ces particularisations favorisent les systèmes d'exploitation modulaires, notamment les différentes variantes de Linux. Android cité ci-dessus est dérivé de Linux. À l'autre extrémité du spectre, Linux équipait 97 % des superordinateurs du Top 500 en 2014

## 6. Ressources des architectures parallèles

Jusqu'à la fin du 20<sup>ème</sup> siècle, il était relativement facile de présenter l'aspect matériel des architectures parallèles. Celles-ci concernaient les serveurs, utilisant des multiprocesseurs et les super-ordinateurs, généralement constitués de clusters de clusters de multiprocesseurs. Pour les super-ordinateurs, les caractéristiques des processeurs, des hiérarchies mémoire et des réseaux d'interconnexion étaient les éléments principaux à détailler.

Avec le « mur de la chaleur » (voir [H 1 058]) qui a conduit au « virage vers le parallélisme » et la naissance des multicœurs, puis des GPUs et des many-cores, présenter les architectures matérielles parallèles revient à présenter toute la gamme des architectures, des systèmes mobiles (*smartphones* et tablettes) et embarqués aux super-ordinateurs et serveurs des *data-centers* en passant par les PC de bureau et PC portables. Dans tous les cas, les performances nécessaires aux applications s'accompagnent de contraintes de limitation de la consommation énergétique et de dissipation de chaleur. Pour les appareils mobiles, il s'agit de prolonger la durée de vie des piles ou batteries. Pour les PC de bureau, il faut que la puissance dissipée soit compatible avec un refroidissement par air. Pour les super-ordinateurs et *data-centers*, il s'agit de limiter la consommation électrique et limiter le coût des systèmes de refroidissement.

Comme pour les architectures monoprocesseurs, les ressources des architectures parallèles comprennent les processeurs, les mémoires et les réseaux d'interconnexion. Il est impossible dans le cadre de cet article de fournir des exemples détaillés couvrant l'ensemble des domaines utilisant des architectures parallèles :

- *smartphones* et tablettes ;
- différents types de systèmes embarqués ;
- PC de bureau et PC portables ;
- serveurs ;
- super-ordinateurs.

Le lecteur intéressé trouvera dans les articles [H 1 058], *Évolution de l'architecture des ordinateurs* et [H 8 000], *Introduction aux systèmes embarqués, enfouis et mobiles* de nombreux exemples de processeurs utilisés dans les architectures parallèles pour ces différentes classes.

Nous nous contentons de donner quelques caractéristiques des processeurs qui sont au cœur de ces architectures. S'il existe différents fabricants de processeurs, deux grandes familles dominent le marché des multicœurs :

- Intel, et à degré moindre AMD, implantent l'architecture IA-32 et Intel 64 ;
- différents fabricants implantent sous licence les jeux d'instruction ARM dans les versions 32 bits et 64 bits.

La quasi-totalité des processeurs Intel disponibles en 2016 est constituée de multicœurs. La gamme se décline en processeurs « nomades », « bureau », « embarqués » et « serveurs ». Ils se différencient notamment par le nombre de cœurs, la fréquence d'utilisation, la puissance dissipée, la taille du cache L2, etc. Le tableau 7 donne quelques caractéristiques. Le lecteur intéressé trouvera tous les détails sur l'URL d'Intel : <http://ark.intel.com/fr>. Les informations sur les processeurs ARM peuvent être trouvées à l'URL : <http://www.arm.com/products/processors/>.

Dans presque toute la gamme d'applications, le multicœur s'accompagne d'un processeur graphique (GPU). Les systèmes embarqués peuvent utiliser en plus des processeurs de traitement du signal (DSP), des FPGA, des accélérateurs spécialisés, etc. Pour les super-ordinateurs, GPU et/ou coprocesseurs comme le Xeon Phi sont utilisés.

## 7. Modèles d'exécution

La réplication des ressources et leur agencement ne suffisent pas à faire fonctionner une architecture parallèle. L'ensemble doit être contrôlé selon un modèle d'exécution. Le modèle d'exécution est aussi une représentation intermédiaire entre le programme qu'exécute l'utilisateur dans un langage de programmation et les programmes qui vont s'exécuter sur les unités de traitement.

Il existe plusieurs modèles d'exécution dont trois ont été ou sont plus particulièrement utilisés : le modèle multi-séquentiel, le modèle flot de données (*dataflow*) [12] [13] et le modèle systolique [14] [15]. Nous ne présenterons que le modèle multi-séquentiel, le plus simple et le plus utilisé.

Ce modèle, comme le modèle MIMD, correspond à l'architecture des microprocesseurs actuels. Dans le principe, il est extrêmement simple. Il s'agit de faire fonctionner chaque microprocesseur d'une machine parallèle comme un processeur de machine séquentielle exécutant son propre programme. Sur chaque processeur ou cœur, il existe donc un compteur de programme géré localement, à partir duquel le programme du processeur est déroulé. La coopération entre les processeurs, qui vont fonctionner à leur propre rythme et de façon asynchrone, passe alors par l'échange d'informations et une gestion de leur coordination. Le principe général suivi pour établir le programme de chaque processeur consiste, en simplifiant à l'extrême, à compiler le programme initial comme pour une exécution séquentielle puis à répartir sur les processeurs les itérations des boucles qui sont exécutées en parallèle. Chaque processeur se voit alors attribuer des bornes de boucles différentes. Des opérations de coordination sont introduites aux points du programme où des contraintes temporelles doivent être respectées. La figure 18 présente un exemple de transformation d'un programme utilisateur en programmes parallèles.

## 8. Programmation des architectures parallèles

L'exécution parallèle de plusieurs programmes suppose leur génération par un compilateur ou leur écriture par un programmeur. Idéalement, le programmeur ne devrait pas se soucier du parallélisme et laisser à un compilateur le soin de paralléliser automatiquement son programme. Cela est particulièrement vrai si l'on désire exécuter sur une machine parallèle des programmes écrits

**Tableau 7 – Gamme de processeurs Intel en 2016 (Extrait)**

Nom	Utilisation	Nb de cœurs	Puissance dissipée
Xeon E7	Serveur	14/16/18	115 W
Xeon D	Serveur	4/6/8	35 à 45 W
	Embarqué	4/6/8	35 à 45 W
6 <sup>ème</sup> génération core i7	Nomade	2/4	15 à 45 W
	Bureau	4	35 à 91 W
	Embarqué	2/4	15 à 65 W
6 <sup>ème</sup> génération core i5	Nomade	2/4	15 à 45 W
	Bureau	4	35 à 91 W
	Embarqué	4	15 à 65 W
6 <sup>ème</sup> génération core i3	Nomade	2	15 à 35 W
	Bureau	2	15 à 51 W
	Embarqué	2	15 à 51 W
6 <sup>ème</sup> génération Core m7	Nomade	2	4,5 W
Atom	Nomade	1/2/4	3,5 à 8,5 W
	Embarqué	4	5 W
	Serveur	2/4/8	6 à 20 W

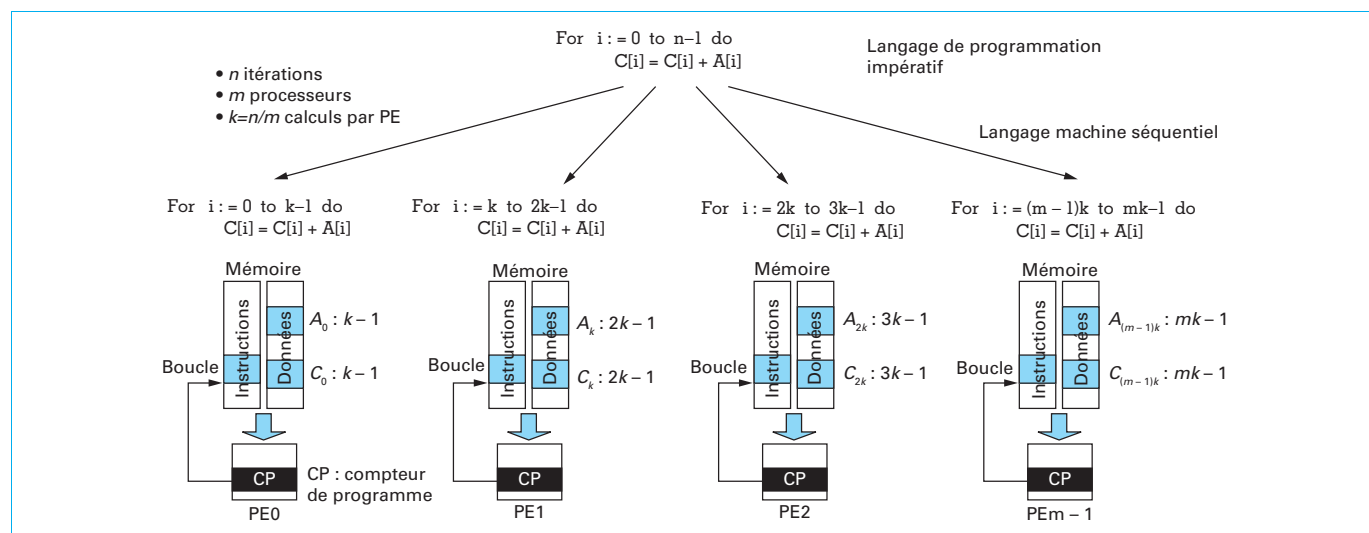


Figure 18 – Principe d'exécution d'une boucle parallèle avec le modèle multi-séquentiel

pour des machines séquentielles. Cependant, la parallélisation automatique se heurte au problème difficile de l'extraction du parallélisme. Un autre problème, plus profond encore, provient du fait qu'il s'agit de paralléliser des programmes issus d'algorithmes pour lesquels aucun effort d'exploitation du parallélisme n'a été fait. Les algorithmes utilisés peuvent même être intrinsèquement séquentiels. Si ces programmes n'ont pas été pensés « parallèles », on comprend qu'il soit difficile d'atteindre un très haut degré de parallélisme avec cette approche. Ces constats sont à la base du développement et du maintien de modèles de programmation parallèles qui impliquent que le programmeur exprime le parallélisme de l'application. Il existe un large choix de modèles de programmation parallèle. En pratique, la sélection d'un modèle de programmation est faite à partir de plusieurs critères dont voici les principaux :

- le type de parallélisme à exploiter ;
- le modèle mémoire de la machine parallèle visée ;
- la portabilité désirée ;
- la facilité de programmation ;
- l'évolution de la structure de l'application pendant son exécution.

## 8.1 Extensions parallèles des langages séquentiels

Un des moyens de réduire le plus possible le fossé entre la programmation séquentielle et les contraintes de la programmation parallèle a consisté à définir des extensions parallèles des langages séquentiels de programmation. L'objectif est de supprimer totalement, ou de réduire l'effort de programmation pour la distribution des données et la synchronisation. Une première approche consiste à faire évoluer un langage séquentiel vers un langage parallèle (HPF). La seconde définit des directives « au-dessus » d'un langage séquentiel classique (C, C++, Fortran) : c'est le cas pour OpenMP, MPI et les pthreads. Cette façon d'écrire des programmes parallèles répond au souci de récupération de très nombreux programmes déjà écrits pour les machines séquentielles. En minimisant leur transformation, on facilite leur parallélisation.

Ces extensions peuvent être considérées comme des modèles de programmation de bas niveau par rapport à ceux présentés en 8.2.

### 8.1.1 High Parallel Fortran (HPF)

HPF est un bon exemple d'évolution d'un langage du séquentiel vers le parallèle. Nous le présentons comme exemple de langage ayant été proposé, mais n'ayant pas été adopté comme standard en dépit des efforts d'IBM, bien que des variantes comme Cray Chapel [16] continuent d'être explorées. HPF exploite le parallélisme de données qui est exprimé au moyen de directives de compilation, de types nouveaux ou de constructeurs.

**Exemple :** Soit une boucle séquentielle écrite en Fortran 77 [17] :

```
DO I=1,50
  X[I] = 2 * X[I]
ENDDO
```

Les itérations de cette boucle ne présentent pas de dépendance. Cette boucle présente donc du parallélisme de données.

Pour exprimer simplement le parallélisme dans cette boucle, Fortran 90 [18] permet d'écrire des expressions dans lesquelles un nom de tableau représente tous les éléments du tableau.

**Exemple : en Fortran 90,** la boucle précédente peut s'écrire :

```
X = X * 2
```

À partir du type de X, le compilateur détermine qu'il s'agit d'un tableau et que cette expression signifie « appliquer l'opération \* 2 à tous les éléments du tableau ». Le nombre d'éléments est connu par le compilateur : c'est la valeur indiquée dans la déclaration du tableau X.

Ce type d'expression permet d'exprimer le parallélisme et d'économiser la construction en boucle avec une indexation des différents éléments du tableau X. Fortran 90 comprend d'autres extensions pour l'expression du parallélisme de données : une nouvelle syntaxe pour les sections de tableau, le traitement parallèle des conditionnelles, l'utilisation de tableaux comme arguments de fonctions, des fonctions spécifiques (appelées intrinsèques) comme la somme des éléments d'un tableau.

**Exemple**

```
y = SUM(X)
```

Le langage HPF (*High Performance Fortran*) [19], [20] se veut le successeur de Fortran 90. HPF ajoute à Fortran 90 essentiellement trois extensions : des directives de compilation pour le placement des données, des constructeurs parallèles permettant l'expression du parallélisme de données lorsque cela n'est pas possible avec le simple typage et des fonctions intrinsèques.

Les **directives de compilation** permettent de préciser l'organisation des données par rapport à l'organisation de la mémoire dans le calculateur parallèle. L'objectif est de réduire le plus possible les communications (qui sont toujours coûteuses) et d'équilibrer la charge de calcul sur les processeurs. Ces directives permettent donc d'indiquer le nombre de processeurs (*HPF\$ PROCESSORS*), la distribution des éléments d'un tableau sur les processeurs (*HPF\$ DISTRIBUTE*) et l'alignement des tableaux entre eux (*HPF\$ ALIGN*).

**FORALL** est un **constructeur parallèle** de HPF qui permet de construire une boucle parallèle à la manière d'une boucle séquentielle, c'est-à-dire en utilisant un indice d'itération. Le constructeur, utilisé à la place d'un **DO**, indique que les itérations de la boucle sont indépendantes.

#### Exemple :

Imaginons que nous voulions calculer chaque élément d'un vecteur à partir de la somme de ses deux voisins. Voici la boucle correspondante en HPF pour un vecteur de  $m$  éléments :

```
FORALL(I=2 : m-1)
  V(I) = V(I-1) + V(I+1)
END FORALL
```

La sémantique du **FORALL** ne précise pas d'ordre pour l'affectation de chaque élément du vecteur résultat. Pour que le résultat soit déterministe, il faut que chaque élément à la gauche de l'expression (du signe « = ») ne soit affecté qu'une seule fois.

Les **fonctions intrinsèques** de HPF comprennent des fonctions de réduction, de tri, de manipulation, des fonctions de calcul souvent utilisées, etc.

Les multiplications de vecteurs et de matrices font partie de ces fonctions.

#### Exemple

```
REAL, DIMENSION (N, N) :: A, B, C
C = MATMUL (A, B)
```

HPF permet donc d'exprimer le parallélisme de données dans un programme. Il permet aussi d'éviter au programmeur de gérer les communications entre les processeurs et la synchronisation. C'est le compilateur qui se chargera d'introduire, dans le programme exécuté par les processeurs, les communications et les synchronisations nécessaires pour respecter la sémantique du programme HPF.

Comme déjà mentionné, HPF n'a pas connu un grand succès. La raison essentielle est le travail demandé à un compilateur est trop important, d'autant plus que les machines sont devenues de plus en plus hétérogènes (clusters) avec la cohabitation mémoire partagée – mémoire distribuée. Le parallélisme de données ayant inspiré HPF se retrouve cependant dans le modèle PGAS (voir 8.2.2).

### 8.1.2 OpenMP

OpenMP est un standard d'interface de programmation (API) pour la programmation à mémoire partagée.

Le modèle à mémoire partagée permet d'exprimer le parallélisme de données comme le parallélisme de contrôle. La mémoire

étant partagée, le programmeur n'a pas besoin de spécifier la distribution des données sur les processeurs. En revanche, c'est à lui de spécifier le parallélisme et de gérer la synchronisation des processeurs. Dans certains cas, tout cela peut être géré implicitement.

Le programmeur doit utiliser des constructeurs spécifiques ou des directives de programmation pour indiquer les régions parallèles de son programme. Dans OpenMP, c'est la directive **!\$OMP PARALLEL** qui indique les régions parallèles.

**Exemple :** soit un exemple de programme séquentiel

```
DO I=1,N
  x = w * (I-0.5)
  sum = sum + x
ENDDO
```

Voici sa transformation en OpenMP :

```
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP REDUCTION(+:sum)
DO I=1,N
  x = w * (I-0.5)
  sum = sum + x
ENDDO
```

L'objectif est d'exécuter chaque itération de la boucle sur un processeur différent. La directive **PARALLEL DO** indique que cette boucle doit être exécutée en parallèle.  $x$  est une variable intermédiaire du calcul.  $x$  n'a pas besoin d'être partagé par tous les processeurs. La directive **PRIVATE (x)** indique que  $x$  sera privé pour chaque processeur (aucun processeur ne peut accéder à un autre  $x$ ).

En revanche, la variable  $w$  doit être vue par tous les processeurs. Cela est indiqué par la directive **SHARED (w)**. La directive **REDUCTION (+:sum)** indique que **sum** doit être calculé à partir d'une fonction de réduction.

L'exécution du programme commence avec un seul processus qui exécute de façon séquentielle les opérations qui précèdent la boucle. Lorsque la directive **PARALLEL** est rencontrée, un ensemble de *threads* est lancé, chacun avec ses propres variables d'environnement. Pour chaque *thread*, il y a une variable privée ( $x$ ), une variable de réduction (**sum**) et une variable partagée ( $w$ ). La région parallèle se termine avec **ENDDO**.

Ce modèle d'exécution correspond au modèle classique **FORK/JOIN**. Mais dans le cas d'OpenMP, le **lancement des threads** et leur **terminaison** sont invisibles au programmeur.

OpenMP comporte deux types de **synchronisation** : implicite et explicite. Par exemple, une synchronisation implicite existe au début et à la fin d'une région parallèle (directive **PARALLEL**). Les synchronisations explicites sont gérées par le programmeur pour contrôler l'ordre d'exécution ou l'accès à une variable partagée. Les directives de synchronisation explicites sont principalement : **BARRIER** pour forcer une synchronisation globale des processeurs, **CRITICAL** pour l'exclusion mutuelle, **ATOMIC** pour la mise à jour d'une valeur. Il existe d'autres directives pour gérer les verrous.

Depuis la version 1 publiée en 1997, OpenMP a évolué, pour améliorer ses performances et prendre en compte l'évolution des architectures parallèles, notamment les multicœurs. Dans la version 4 publiée en 2013, des fonctionnalités ont été améliorées et d'autres ajoutées. Sans prétendre à l'exhaustivité, on peut citer :

- les directives **SIMD** *#pragma omp simd [clauses]*. Elles permettent l'utilisation des extensions SIMD des jeux d'instructions ;
- l'utilisateur peut définir de nouvelles opérations de réduction *#pragma declare reduction [clauses]* en plus des opérations de réduction disponibles ;



- l'augmentation du contrôle du placement des *threads* sur les processeurs (affinité des *threads*) pour améliorer les performances avec les architectures NUMA (*OMP\_PROC\_BIND*) ;
- un support pour les accélérateurs, notamment GPU ;
- la possibilité d'annuler une construction (*parallel*, *section*, *do* ou *subgroup*) si une condition est réalisée par la directive *!\$omp cancellation construct [if (exp)]* ;
- plus de directives pour l'ordonnancement des tâches.

Les caractéristiques détaillées d'OpenMP peuvent être trouvées sur le site de référence [21] et dans de nombreux cours en ligne accessibles sur le Web.

### 8.1.3 MPI

Le modèle de programmation à passage de messages a été conçu pour les architectures parallèles ne possédant pas de mémoire partagée. L'absence de mémoire partagée engendre deux conséquences importantes pour le programmeur. D'abord, il doit gérer la répartition des données sur les différents processeurs. Cela signifie que le découpage et le placement des données d'un ensemble (d'un tableau par exemple) doivent être gérés par le programmeur. Ensuite, l'échange d'informations entre les processeurs nécessite la communication d'un message entre un émetteur et un récepteur. C'est au programmeur d'identifier les émetteurs et les récepteurs, et de placer des fonctions de communication dans son programme. Le programmeur doit aussi gérer la coordination des processus, c'est-à-dire contrôler l'exécution et gérer les sections critiques.

MPI (*Message Passing Interface*) est le représentant type de ce modèle.

MPI est une bibliothèque de fonctions de communication utilisable avec les langages C et Fortran. MPI comprend des fonctions de communication point à point, de communication collective et de gestion de l'environnement d'exécution.

D'autres fonctions sont disponibles.

Un programme utilisant des fonctions MPI correspond au modèle d'exécution SPMD (*Single Program Multiple Data*). Cela signifie qu'un seul programme doit être écrit qui sera exécuté par tous les processeurs, chaque processeur ayant des données différentes (SPMD). Dans un même programme, il faut donc écrire les parties correspondantes aux émetteurs et celles correspondantes aux récepteurs des communications.

Les fonctions de communication point à point permettent d'échanger des informations entre un émetteur et un récepteur.

#### Exemple

```
MPI_Init(...);
MPI_Comm_rank(-, &my_rank);
MPI_Comm_size(-, &p);
if(my_rank != 0){
    dest = 0;
    MPI_Send(my_rank, longueur, MPI_INT, dest, ...);
}
else { /* my_rank == 0 */
    for (source = 1; source < p; source++)
        MPI_Recv(rank, max_size, MPI_INT, source, ...);
}
```

Ce programme sera exécuté sous la forme de plusieurs processus (typiquement un processus par processeur). Toutes les fonctions MPI sont précédées par `MPI_`.

La fonction `MPI_Init` permet d'initialiser l'environnement d'exécution pour chaque processus. La fonction `MPI_Comm_rank` retourne le rang du processus.

La fonction `MPI_Comm_size` retourne le nombre de processus en parallèle pour ce programme. Dans ce programme, nous avons défini deux familles de processus : celui de rang 0 et les autres. Le processus de rang 0 reçoit à tour de rôle le rang des autres processus. C'est le seul à exécuter la branche « else » de la conditionnelle. Tous les autres envoient avec la fonction `MPI_Send` leur rang en utilisant comme destinataire le processus 0. Celui-ci reçoit par la fonction `MPI_Recv` le rang des autres dans l'ordre indiqué par la succession des valeurs de « source ».

Les fonctions `MPI_Send` et `MPI_Recv` sont bloquantes. Cela signifie que la fonction `MPI_Send` ne retourne que lorsqu'il est possible de modifier le tampon d'émission depuis l'émetteur sans que cela ne corrompe le message envoyé. La fonction `MPI_Recv` suspend l'exécution du processus récepteur jusqu'à ce que le message ait été placé dans un tampon de l'application. MPI possède des variantes non bloquantes de ces fonctions. La fonction `MPI_Isend` peut retourner alors qu'une modification dans le tampon d'émission par le processus émetteur corromprait le message à émettre. Pour prévenir cette situation, l'émetteur peut appeler une fonction pour tester régulièrement l'état de la transmission. Cela permet à l'émetteur de continuer à calculer alors que le message est en cours de transmission. Il existe une fonction réciproque pour la réception : `MPI_Irecv`. De même, pour prévenir l'utilisation prématurée du tampon de réception, une fonction peut être appelée régulièrement pour tester l'état de la transmission.

Les fonctions `MPI_Bcast`, `MPI_Reduce` et `MPI_Barrier` sont des fonctions de communication collectives. La fonction `MPI_Bcast` permet à un seul processus d'envoyer la même donnée (le même message) à tous les autres processus. La fonction `MPI_Reduce` permet de combiner les données indiquées par tous les processus qui l'exécutent en leur appliquant l'opération spécifiée en argument. La fonction `MPI_Barrier` fournit un mécanisme pour synchroniser tous les processus.

Aux versions 1 successives depuis 1994 du standard MPI, la version 2 (1997) a ajouté les accès mémoire distants (RMA pour *Remote Memory Access*), aussi appelés « *one-sided communications* ». Cette fonctionnalité permet des copies de mémoire à mémoire. Après avoir défini les paramètres de la zone mémoire d'origine et de la zone mémoire cible, les fonctions `MPI_GET` ( ), `MPI_PUT` ( ) et `MPI_ACCUMULATE` ( ) permettent à un processus local de respectivement lire, écrire et mettre à jour des données situées dans la mémoire locale d'un processus distant. Les transferts sont terminés par une synchronisation définie par l'utilisateur.

Le standard MPI est accessible sur le site de référence [22]. De nombreux cours disponibles sur le Web détaillent les caractéristiques de MPI, dont [23]. La version 3 est disponible depuis 2012. Elle introduit notamment un modèle MPI mémoire partagée (SHM) : voir [24], [25] et la liste de références de cet article.

Si OpenMP est défini pour les architectures à mémoire partagée et MPI pour les architectures à mémoire distribuée, une programmation mixte MPI + OpenMP peut être la solution la plus performante sous certaines conditions [26]. Dans un cluster de multicœurs (SMP), la programmation hybride semble naturelle : OpenMP dans les nœuds SMP et MPI entre les nœuds, ou MPI-3.0 dans les nœuds et MPI entre les nœuds. Le lecteur intéressé trouvera dans [27] une discussion sur la programmation hybride.

### 8.1.4 PThreads et modèle multi-flots (*multi-thread*)

Le modèle multithreads [28] est très proche du modèle à mémoire partagée. Dans les versions originales de ce modèle, l'échange d'informations repose sur la mémoire partagée. Des environnements de programmation permettent le passage de messages sous la forme d'appels de procédures distantes.

Ce modèle est approprié pour la programmation du côté serveur des applications client-serveur. Dans ces applications, un serveur doit répondre aux requêtes de multiples clients simultanément. Une autre contrainte est que le nombre de clients varie dans le temps : les clients se connectent au serveur et se déconnectent dynamiquement.

La programmation du serveur met généralement en œuvre plusieurs processus : un ou plusieurs processus interface, un programme principal et un ou plusieurs processus de traitement. Sur une machine parallèle, typiquement, un processus de traitement est lancé pour chaque utilisateur et, dans certains cas, même pour les différentes requêtes d'un utilisateur.

Le modèle multiflots repose sur plusieurs constatations que l'on peut faire sur la plupart des systèmes d'exploitation existants. D'abord, les temps de création de processus et de commutation de contexte sont très importants ( $\times 100 \mu s$ ). Ensuite, les processus ne partagent pas de données communes alors que cela est nécessaire dans beaucoup d'applications. Enfin, le programmeur n'a pas de moyens efficaces pour gérer l'ordonnement des processus.

Le modèle multiflots aussi appelé à processus légers apporte une réponse à chacun de ces problèmes. Son principe est de gérer à l'intérieur d'un même processus plusieurs flots d'exécution.

**Exemple :** le programme suivant présente un exemple d'utilisation des Pthreads.

```
int n, num_thread ;
pthread_mutex_t reduction ;
float res, w ;
pthread_t *tid ;
main(...){
    for(i = 0 ; i < num_thread ; i++)
        pthread_create(&tid[i], ..., Pworkers, ...);
    for(i = 0 ; i < num_thread ; i++)
        pthread_join(tid[i], ...);
}
Exemple
Pworkers( ){
    float sum, x ;
    myid = pthread_self( )...;
    for( i = myid ; i < n ; i += num_threads){
        x = w * (i - 0.5)
        sum = sum + x
    }
    pthread_mutex_lock(&reduction);
    res = res + sum ;
    pthread_mutex_unlock(&reduction);
}
```

Comme pour OpenMP, l'intégralité de l'exécution a lieu au sein d'un même processus. Les différents flots d'exécution partagent les variables globales, les descripteurs de fichiers, les signaux du processus dans lequel ils s'exécutent. Les flots d'exécution sont créés avec la fonction *pthread\_create*. Cette fonction prend comme argument le programme que doit exécuter le flot d'exécution et retourne l'identifiant du flot. Lorsque tous les flots sont créés, ils sont ordonnancés par l'environnement d'exécution. Les flots d'exécution peuvent tous accéder aux variables *res* et *w*. Par contre, les

variables *sum* et *x* sont des variables locales à chaque flot. Elles ne sont pas visibles par les autres flots d'exécution. Après s'être identifié (*pthread\_self*), chaque flot d'exécution exécute indépendamment la boucle. En revanche, l'écriture dans la variable partagée *res* doit être effectuée de manière atomique. Chaque flot utilise donc les fonctions d'accès à un verrou (la variable partagée *reduction*) pour entrer et sortir de la section critique. À la fin de l'exécution, tous les flots retournent et la fonction *main* termine lorsque tous les flots sont terminés.

En pratique, les flots d'exécution sont placés sur les processus légers des systèmes qui le permettent (Mach [29], Solaris [30], Chorus [31], etc.) ou s'exécutent au sein d'un processus « lourd » UNIX. Dans les deux cas, la création, la destruction et l'ordonnement des flots d'exécution au sein du processus hébergeant sont beaucoup plus rapides puisque ces opérations n'affectent pas le contexte du processus au niveau du système d'exploitation. Dans la norme POSIX, l'utilisateur peut aussi spécifier l'ordonnement des flots avec la fonction *pthread\_setschedparam*. Dans les systèmes Mach, Solaris, Chorus et Linux, les flots d'exécution d'un même processus peuvent être exécutés sur des processeurs différents si l'architecture est à mémoire partagée.

## 8.2 Modèles de programmation

Les différents types de parallélisme (parallélisme de données, de concurrence, multi-flots) peuvent être mis en œuvre à travers des modèles d'abstraction plus élevée appelés modèles de programmation parallèle, qui définissent la manière dont le programmeur va écrire son programme parallèle. Les modèles de programmation peuvent être classés selon différents critères.

Une première classification distingue les modèles selon qu'ils mettent l'accent sur les fonctions exécutées (squelettes algorithmiques) ou sur les données (PGAS – *Partitioned Global Address Space* – ou AGAS – *Active Global Address Space*). Une seconde classification distingue les modèles basés sur la performance (BSP – *Bulk Synchronous Parallelism*) et ceux basés sur la sémantique (UPC – *Unified Parallel C*). Les modèles utilisés peuvent mélanger des caractéristiques des éléments des deux classifications. Nous présentons brièvement un modèle de chaque classe.

### 8.2.1 Squelettes algorithmiques

Les squelettes algorithmiques, introduits en 1989 [32] correspondent à un modèle basé sur des fonctions couramment trouvées dans les programmes parallèles. Trois types de squelettes sont utilisés :

- les squelettes de données parallèles. *Map* distribue les données sur les processeurs. *Reduce* est l'opérateur de réduction de données de différents processeurs. *Permute* et *Shift* sont des opérateurs de communication des données entre processeurs ;
- les squelettes de tâches parallèles. *Pipeline* organise le fonctionnement pipeline entre différentes tâches. *Farm* divise les tâches d'un maître entre plusieurs esclaves et récupère les résultats ;
- les squelettes de contrôle. *For* : les squelettes itératifs travaillent comme la boucle *for* des programmes séquentiels. *If* contrôle d'autres squelettes selon le résultat de la condition.

Le programme parallèle est constitué par l'assemblage des différents squelettes. L'orchestration et la synchronisation des activités parallèles sont implicitement définies par les schémas des squelettes.

### 8.2.2 PGAS

Le modèle « Espace d'adresse global partagé » connu sous le nom PGAS (*Partitioned Global Address Space*) [33] est un modèle basé sur l'organisation des données. Il définit un espace



d'adressage mémoire global (commun à tous les processeurs) qui est logiquement partitionné de manière à ce que chaque portion soit locale à un processeur ou à une *thread*. De cette manière, il peut exploiter la localité des références pour chaque processeur. L'objectif est de combiner les avantages d'une programmation SPMD pour les systèmes à mémoire distribuée avec les accès mémoire des systèmes à mémoire partagée. Ce modèle est la base d'un certain nombre de modèles comme UPC (*Unified Parallel C*), Coarray Fortran, Fortress, Global Arrays, SHMEM, etc.

Une variante (APGAS pour *Asynchronous Partitioned Global Address Space*) permet des créations locales et distantes de tâches, avec des synchronisations non bloquantes.

8.2.3 BSP

BSP [34] [35] est un modèle de programmation qui vise à permettre l'évaluation de la performance du programme sur une architecture. Deux ressources ont un impact majeur sur les performances : le couple processeur-mémoire et le réseau de communication. L'écriture du programme doit identifier clairement les parties du programme qui vont utiliser les deux ressources : la partie calcul a un temps d'exécution  $\sum t_{calcul}$  et la partie communication a un temps d'exécution  $\sum t_{comm}$ . Le temps d'exécution global est la somme de ces deux temps :

$$T = \sum t_{calcul} + \sum t_{comm}$$

BSP a été défini pour rendre compte d'une exécution fondamentalement synchrone d'un traitement parallèle. Un programme BSP est modélisé par une séquence de **super-étapes**. Une super-étape consiste en trois phases consécutives : une phase de calcul, une phase de communication globale et une barrière de synchronisation.

La phase de calcul peut utiliser des données provenant de la phase de communication de la super-étape précédente et produit des données pouvant être communiquées lors de la phase de communication globale de la super-étape courante. La barrière de synchronisation garantit à tous les processeurs que tous les calculs sont terminés et que toutes les communications sont arrivées à destination avant le passage à la super-étape suivante.

Le temps d'exécution d'une super-étape est exprimé à partir de paramètres :

- *w* est le maximum des temps pris par les processeurs pour le calcul des opérations de la super-étape ;
- *g* est l'inverse du débit de communications du réseau pour chaque processeur ;
- *h* est le maximum du nombre de messages émis ou reçus par les processeurs ;
- *l* est la durée de la barrière de synchronisation.

$$T_{super-étape} = W + gh + l$$

Le temps de calcul total est la somme des temps d'exécution des super-étapes.

LogP [36] est un autre modèle basé sur les mêmes principes, mais avec une définition plus détaillée des performances de communication.

8.2.4 Le modèle asynchrone « futures et promesses »

Les futures (*futures*) et promesses (*promises*) ont été définies dans les années 70 et sont utilisées depuis le début des années 2000 dans de nombreux langages de programmation, les plus connus étant Scheme, Java, Python, C++ depuis la norme 2011 et implantées dans des bibliothèques comme Boost. De nombreuses

références sont disponibles sur le Web pour les différents langages qui les utilisent.

Le principe est illustré par la figure 19. Un *thread* client a besoin d'un calcul : sans attendre le résultat, il crée une future (objet encapsulant la valeur qui sera disponible plus tard) et une promesse (objet destiné à résoudre la future, via une fonction utilisant des arguments). Le *thread* continue son exécution et demande le résultat à la future quand il en a besoin. Il n'est bloqué en attente du résultat que si la promesse n'a pas encore écrit le résultat dans la future. Il y a possibilité de créer des combinaisons séquentielles et des combinaisons parallèles de futures.

Soit à calculer  $y = f3(f1(a), f2(b))$ . Le tableau 8 donne les codes C++ séquentiel et avec futures pour calculer cette expression. La figure 20 donne les chronogrammes correspondants.

Les modèles de programmation par *threads* restent entravés par les mécanismes de synchronisation, notamment pour les machines massivement parallèles. La programmation parallèle asynchrone, basée sur les « futures et promesses » permet de relâcher les contraintes de synchronisation. L'outil HPX [37], développé par l'équipe Stellar de l'université d'État de Louisiane, est un bon exemple de cette approche. C'est un *runtime* multi-plateformes encapsulé dans une bibliothèque C++.

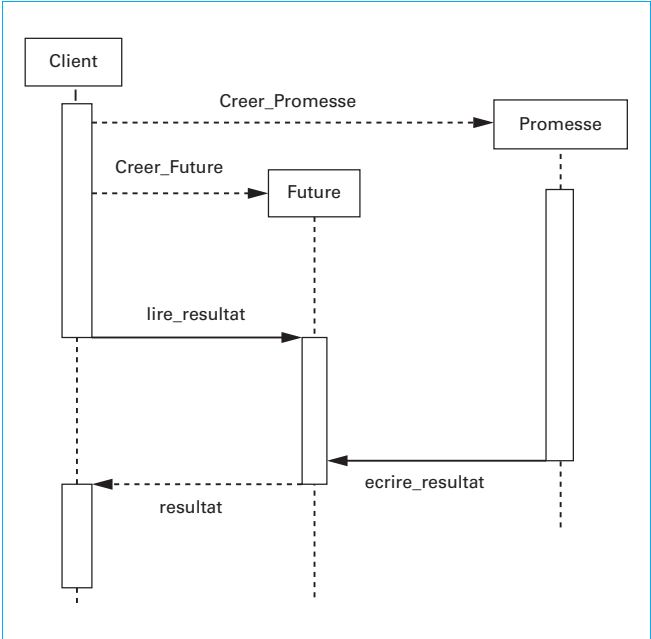


Figure 19 – Schéma comportemental des futures et des promesses

Tableau 8 – Exemple de code C++ séquentiel et avec futures	
Code séquentiel	Code avec futures
int a, b ;	int a, b ;
int y1 = f1(a) ;	Future <int> y1 = async (f1,a)
int y2 = f2 (b)	Future <int> y2 = async (f2,b)
y = f3(y1,y2)	y = f3(y1.get() , y2.get()) ;

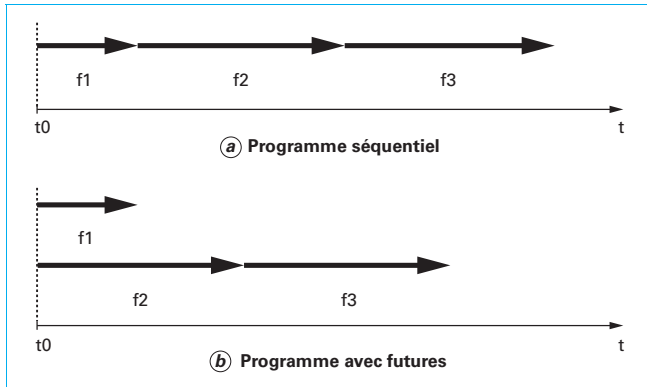


Figure 20 - Chronogramme des programmes exemple

## 9. Lois et métriques de performances des architectures parallèles

### 9.1 Performances

Ce paragraphe introduit différentes mesures de la performance : performance crête, accélération.

La performance la plus simple à calculer mais aussi la moins utile pour prédire les performances d'une architecture parallèle est la **performance crête**. Elle est simplement obtenue par la multiplication de la performance crête individuelle de toutes les unités de traitement par leur nombre dans l'architecture.

Une définition très utilisée de la performance crête est : la performance dont le constructeur vous garantit que vous ne pourrez pas la dépasser !

Une information plus pertinente est le **temps de calcul d'une application** (ou temps d'exécution). On peut en dériver le débit de calcul sur des nombres flottants ou des nombres entiers. La formule suivante donne le débit de traitement en fonction du temps de calcul :

$$P = \text{nombre opérations} / \text{temps du traitement}$$

Le dividende représente le nombre d'opérations que l'application nécessite pour son calcul.

**Exemple** : le nombre d'opérations flottantes pour le produit de matrices carrées est  $2n^3$  si  $n$  est la taille du côté de la matrice. Avec un côté de 1 000 éléments, il y a donc  $2 \times 10^9$  calculs à réaliser. Si le temps de traitement est de 1 seconde, le débit de calcul est de  $2 \times 10^9$  opérations flottantes par seconde ; soit 2 gigaFlops.

Dans des programmes plus compliqués, cette information est peu pertinente, car le temps d'exécution des opérations flottantes varie beaucoup, des plus simples comme l'addition et la multiplication, à d'autres comme la division ou la racine carrée. Il faut alors, pour chaque architecture, prendre en compte les temps d'exécution individuels de chaque opération flottante.

Le temps d'exécution sert de base à la comparaison d'architectures pour le traitement d'une même application.

Avec la formule suivante :

$$A = t_1 / t_2$$

on calcule l'accélération que permet l'architecture 2 (qui a pris le temps  $t_2$  pour calculer l'application) par rapport à l'architecture 1 (temps  $t_1$ ).

Une des caractéristiques recherchées dans les architectures parallèles est leur **extensibilité**. L'extensibilité rend compte de la possibilité d'augmenter le nombre de processeurs dans l'architecture. Étendre une architecture a un intérêt seulement si cela se traduit par de meilleures performances. Deux paramètres mesurent le gain de performance obtenu par l'augmentation du matériel (plus de processeurs) : l'accélération (le *speedup*) et l'augmentation du débit (le *scaleup*). L'**accélération** en fonction du nombre de processeurs indique le rapport de performance entre deux architectures identiques traitant le même problème, mais avec un nombre de processeurs différents. L'accélération est calculée avec la formule précédente.  $t_2$  représente alors le temps de l'architecture avec  $x$  processeurs et  $t_1$  le temps avec  $y$  processeurs. On suppose  $x > y$ . L'accélération devrait augmenter linéairement avec le nombre de processeurs. En pratique, c'est rarement le cas.

L'**augmentation du débit** mesure le rapport de la quantité d'informations traitées dans le même temps par deux architectures similaires, l'une ayant plus de ressources que la seconde. La formule suivante mesure le « *scaleup* » :

$$S = Q_2 / Q_1$$

$Q_2$  est la quantité de données traitées par l'architecture 2 et  $Q_1$  est la quantité de données traitées par l'architecture 1 dans le même temps.  $S > 1$  peut être obtenu en augmentant la performance individuelle des processeurs ou en augmentant le nombre de processeurs. Dans ce cas aussi, l'augmentation du nombre de processeurs devrait accroître proportionnellement le *scaleup*. En pratique, même si cela est moins difficile que pour le cas du *speedup*, la présence de communications entre les processeurs et leur coordination réduit le *scaleup*.

### 9.2 Lois de performance

Comme d'autres disciplines, l'architecture des ordinateurs est gouvernée par des lois. Les lois connues découlent directement des propriétés des programmes. Dans le cadre des architectures parallèles, la première loi formulée fut la loi d'Amdahl. Elle indique la limite de l'accélération atteignable par une architecture parallèle (quel que soit son type) pour un programme de taille donnée. Supposons un programme X. Ce programme est composé d'une partie P exécutable en parallèle et d'une partie S ne pouvant pas être exécutée en parallèle. Soit  $p$  le temps d'exécution séquentiel de la partie P et  $s$  le temps d'exécution de la partie S. Soit  $n$  le nombre de processeurs utilisés pour l'exécution parallèle de P. Alors dans le meilleur des cas, l'accélération est égale à :

$$A = (s + p) / (s + p/n)$$

Selon cette formule, on remarque que quel que soit le nombre de processeurs, l'accélération est toujours limitée par  $s$ . Si  $s$  représente seulement 10 % du temps d'exécution totale ( $s + p$ ), l'accélération sera bornée par  $(s + p) / s$ , soit 10. Même avec une infinité de processeurs, l'accélération ne pourra pas dépasser 10. On déduit deux règles de cette loi. Premièrement, lors de l'écriture d'un programme parallèle, il faut limiter autant que possible la partie séquentielle. Deuxièmement, un ordinateur parallèle doit être un excellent ordinateur séquentiel pour traiter le plus rapidement possible la partie séquentielle.

La loi d'Amdahl est une mauvaise nouvelle pour le parallélisme. Elle indique que la performance ne dépend pas seulement du nombre de ressources mises en parallèle et de plus, elle désigne la partie séquentielle comme le facteur limitant. La limite est aussi très sévère car il est très difficile de paralléliser 90 % d'un programme.

La loi de Gustavson modère les conclusions de la loi d'Amdahl. Gustavson fait remarquer que la partie parallèle est composée de

boucles qui traitent les données. Si la quantité de données à traiter progresse, alors la contribution de la partie parallèle dans le temps d'exécution va augmenter (cela pour un même nombre de processeurs). Par conséquent, la contribution de la partie séquentielle va diminuer. Comme généralement, l'utilisateur du parallélisme utilise plus de processeurs parce qu'il veut traiter plus de données, l'approche de Gustavson est cohérente avec l'utilisation du parallélisme. Plus rigoureusement, soit un programme X avec une partie séquentielle S et une partie parallèle P. Avec  $n$  processeurs, le temps de calcul selon la loi d'Amdahl est :

$$t = s + p/n$$

En supposant que la partie parallèle augmente linéairement avec le nombre de processeurs  $p = a n$ , le temps d'exécution avec  $n$  processeurs devient :

$$t = s + a$$

D'où l'accélération :

$$A = (s + a n) / (s + a)$$

Et  $A \rightarrow n$  lorsque  $a \rightarrow \infty$ .

Notons que  $a$  représente la taille de la partie parallèle attribuée à chaque processeur. Plus cette taille est importante, plus l'accélération tend vers le nombre de processeurs. Cette taille dépend directement de la quantité d'informations à traiter. Comme les structures de données comportent généralement beaucoup plus d'éléments qu'il n'y a de processeurs dans une machine parallèle (plusieurs milliards d'éléments contre de l'ordre du million de processeurs ou cœurs), chaque processeur a une quantité importante de données à traiter. L'augmentation de cette quantité provoque généralement une augmentation proportionnelle du temps de traitement pour chaque processeur. La loi de Gustavson correspond donc bien à une réalité. La conclusion de cette loi est que le parallélisme peut dépasser la limite établie par la loi d'Amdahl à condition d'augmenter la quantité d'informations à traiter.

Une autre façon d'appréhender la notion d'accélération dans les architectures parallèles est de considérer les limitations qui réduisent la performance, au fur et à mesure que le nombre de processeurs en parallèle augmente. La figure 21 présente la forme générale de l'évolution de l'accélération avec le nombre de processeurs (ou cœurs).

La courbe est séparée en trois parties. Dans la première partie, l'accélération peut être inférieure à 1 à cause du temps nécessaire

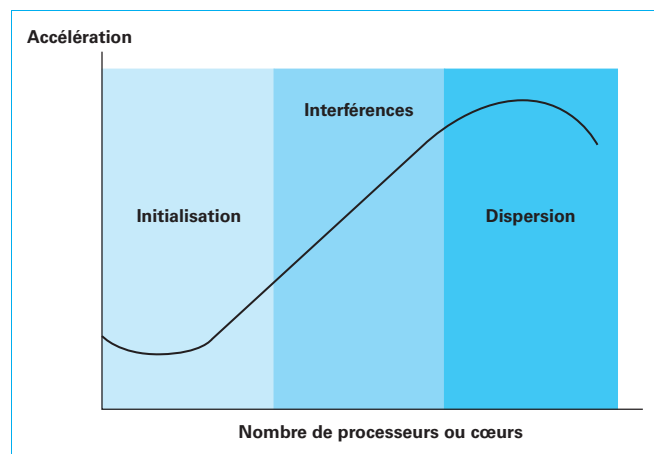


Figure 21 – Évolution générale de l'accélération avec le nombre de processeurs en parallèle

à l'initialisation d'une exécution parallèle. Si ce temps domine le temps d'exécution de la partie parallélisable, le gain obtenu sur l'exécution parallèle devient négligeable devant le surcoût d'initialisation. Dans la deuxième partie de la courbe, l'accélération est supérieure à 1 mais reste inférieure au nombre de processus. Cela est dû aux interférences entre les processus s'exécutant en parallèle, notamment lors des accès aux ressources partagées. Chaque nouveau processus ralentit alors un peu plus les autres. Dans la troisième partie, la charge moyenne de traitement attribuée à chaque processus diminue. La charge attribuée à chaque processus peut être différente et conduire à une variance entre les charges qui excède la charge moyenne. Dans ce cas, c'est le temps d'exécution du processus le plus long (celui qui a la plus grosse charge) qui détermine l'accélération. L'amélioration de l'accélération devient alors très faible.

### 9.3 Modèle « Roofline »

Le modèle « Roofline » [38] est un modèle visuel qui prend en compte les deux caractéristiques essentielles d'une architecture parallèle, c'est-à-dire la puissance de calcul crête et le débit mémoire maximal. Plus précisément, la puissance de calcul, en GFlops/s est exprimée en fonction de l'intensité opérationnelle en GFlops/Goctet. La puissance de calcul est donnée par la formule suivante :

$$\text{Puissance calcul (GFlops/s)} = \text{Min (GFlops/s max, Débit max * Intensité opérationnelle)}$$

La figure 22 montre le modèle Roofline avec deux processeurs différents. Les courbes présentent deux parties :

- pour de faibles intensités opérationnelles, c'est le débit mémoire qui limite la performance de calcul max. Les instructions de calcul attendent les données mémoire. Cette situation correspond à la partie inclinée des courbes ;
- pour de fortes intensités opérationnelles, c'est la puissance de calcul qui limite la performance. La puissance de calcul maximale sur l'application atteint un plateau (partie horizontale des courbes).

En fonction des performances mémoire, une application peut être limitée en mémoire (*memory bound*) ou limitée en calcul (*compute bound*). Dans la figure 23, pour une intensité opérationnelle de 2, le processeur 1 est limité en calcul et le processeur 2 est limité par la mémoire.

L'intérêt de ce modèle est de donner des indications sur les possibilités d'optimisation d'une application ou de l'architecture. Si l'application est limitée par l'accès aux données, ce sont les techniques d'optimisation des accès mémoire qu'il faut mettre en œuvre : limitation des accès mémoire, techniques de prefetching, etc. Si l'application est limitée par le calcul, ce sont les techniques

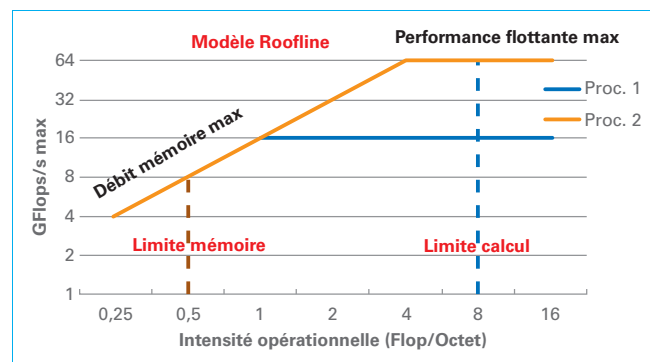


Figure 22 – Le modèle Roofline

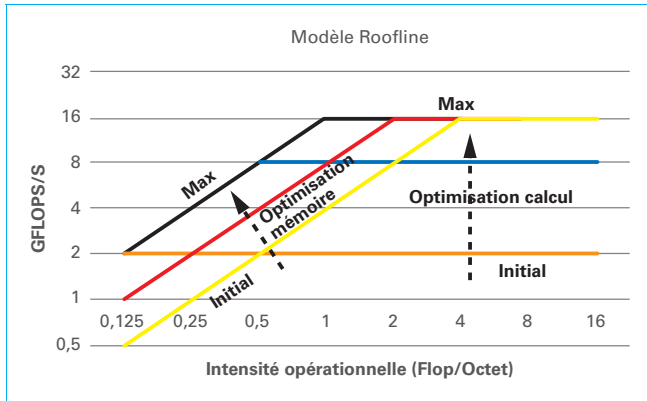


Figure 23 – Types d'optimisation pour un processeur donné en fonction de l'intensité opérationnelle

d'optimisation de code qu'il faut utiliser : SIMD, parallélisation, etc. C'est ce qu'illustre la figure 23.

## 9.4 Métriques et benchmarks

La mesure des performances des architectures parallèles est une discipline en soi. Pour comparer les performances de différentes machines ou pour mesurer l'apport de l'augmentation du nombre de processeurs, on mesure le temps d'exécution d'ensembles de programmes standards (benchmarks). Il existe différents types de benchmarks : les benchmarks synthétiques, des applications existantes, des noyaux d'applications, etc.

Les benchmarks synthétiques regroupent un ensemble de petits programmes représentant chacun une partie critique d'applications plus importantes. Ainsi, les « boucles de Livermore » regroupent un ensemble de boucles rencontrées dans les applications exécutées au Lawrence Livermore National Laboratory.

Les benchmarks réalisés à partir d'un ensemble d'applications complètes sont plus appropriés pour rendre compte des performances de chaque composante d'une architecture (processeur, mémoire, réseau, E/S). L'ensemble le plus connu dans le monde des microprocesseurs s'appelle SPEC (*System Performance Evaluation Cooperative*). Dans le domaine des architectures parallèles, les benchmarks reconnus sont les NAS (*Numerical Aerospace Simulator facility*) [39]. Parmi les applications incluses dans les NAS, on trouve : une décomposition LU, une FFT, un gradient conjugué, un tri entier, une méthode multigrille, etc.

On mesure aussi les performances des architectures parallèles à partir de fonctions de bibliothèques mathématiques. Les BLAS (*Basic Linear Algebra Subroutines*) [40] et Lapack (*Linear algebra package*) [41] sont utilisés pour connaître les capacités d'une architecture pour le traitement matriciel.

Souvent, les performances d'une architecture varient avec la taille de l'ensemble de données traitées par un benchmark ou une application. En effet, la grande majorité des applications parallèles possède un rapport entre le nombre d'opérations et le nombre de communications qui croît avec la taille de l'ensemble de données à traiter. Si l'on prend le cas simple d'un produit de matrices avec une distribution par blocs des matrices opérantes et résultat sur les processeurs, le volume de calcul que doit réaliser chaque processeur croît de façon quadratique avec la taille des matrices alors que le nombre de communications ne croît que linéairement. Les communications étant généralement le plus important facteur de ralentissement, l'augmentation du ratio calculs/communications (dû à l'accroissement de la taille de l'ensemble de données) se traduit par une augmentation de la performance globale de l'architecture sur l'application.

Le TOP 500, qui fournit la liste des 500 superordinateurs les plus puissants, donne la performance maximale (en TFlops/s) atteinte sur Linpack (algèbre linéaire) avec l'ensemble de données le plus grand possible, la puissance dissipée et le nombre de cœurs.

## 10. Remarques pour conclure

Le mur de la chaleur a complètement modifié le panorama des architectures matérielles et de la programmation. Avec les mono-processeurs, les architectures parallèles se limitaient aux serveurs et aux machines parallèles massivement parallèles. Les processeurs devenant multicœurs puis many-core, les architectures parallèles sont devenues majoritaires et sont présentes dans l'ensemble du spectre : *smartphones*, tablettes, PC, systèmes embarqués, serveurs, super-ordinateurs. Seuls les systèmes embarqués bas de gamme nécessitant peu de puissance de calcul ou à très faible consommation continuent d'utiliser des monoprocesseurs.

Le problème essentiel est au niveau de la programmation. En 2010, David Patterson publiait un article dans IEEE Spectrum [42] au titre évocateur : « Le problème avec les multicœurs : les fabricants de circuits conçoivent des microprocesseurs que la plupart des programmeurs ne peuvent utiliser ». Les modèles de programmation bas niveau (OpenMP, MPI, pthreads) sont matures, mais nécessitent de bonnes connaissances de la programmation parallèle. Différents modèles de programmation de plus haut niveau existent avec leurs avantages et leurs inconvénients. La possibilité de tirer parti de toutes les possibilités des architectures matérielles parallèles dépend d'un certain nombre de facteurs parmi lesquels :

- l'amélioration des modèles de programmation existants pour « paralléliser » les vieux programmes conçus pour les machines séquentielles ;
- l'amélioration de la formation des programmeurs pour qu'ils pensent « parallèle » lorsqu'ils développent une nouvelle application.

## 11. Glossaire

CC-NUMA : *Cache Coherent Non Uniform Memory Access*

Organisation mémoire à accès non uniforme avec cohérence des caches

DSP : *Digital Signal Processor*

Processeur de traitement du signal

FPGA : *Field Programmable Gate Array*

Réseau de portes à configuration programmable

GPU : *Graphics Processing Unit*

Processeur graphique

MIMD : *Multiple Instruction streams Multiple Data streams*

Plusieurs flux d'instructions exécutant plusieurs flux de données

MISD : *Multiple Instruction streams Single Data stream*

Plusieurs flux d'instructions exécutant un flux de données

MPI : *Message Passing Interface*

Bibliothèque de fonctions pour la programmation par passage de messages

NUMA : *Non Uniform Memory Access*

Organisation mémoire à accès non uniforme

**SIMD** : *Single Instruction streams Multiple Data streams*

Un flux d'instructions exécutant plusieurs flux de données

**SIMT** : *Single Instruction streams Multiple Threads*

Un flux d'instructions exécutant sur des données de plusieurs threads

**SISD** : *Single Instruction stream Single Data streams*

Un flux d'instructions exécutant un flux de données

**SMP** : *Symmetric Multiprocesseur*

Multiprocesseur symétrique

**SoC** : *System-on-Chip*

Système sur puce

**SPMD** : *Single Program Multiple Data stream*

Le même programme s'exécute sur plusieurs flux de données

---



# Introduction au parallélisme et aux architectures parallèles

par **Franck CAPPELLO**

Docteur en Informatique de l'université Paris Sud  
IEEE Fellow

et **Daniel ETIEMBLE**

Ingénieur de l'INSA de Lyon  
Professeur émérite à l'université Paris Sud

## Sources bibliographiques

- [1] KOBAYASHI (H.). – *Feasibility Study of a Future HPC System for Memory-Intensive Applications : Final Report*. Proceedings of the joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University, pp 3-16 (2014).
- [2] KOBAYASHI (H.). – *Feasibility Study of a Future HPC System for Memory-Intensive Applications : Final Report*. in SuperComputing, NEC Booth, [http://jpn.nec.com/hpc/info/pdf/SC13\\_NEC-Tohoku\\_Prof.Kobayashi.pdf](http://jpn.nec.com/hpc/info/pdf/SC13_NEC-Tohoku_Prof.Kobayashi.pdf) (2013).
- [3] BERNSTEIN (A.J.). – *Analysis of Programs for Parallel Processing*. IEEE Transactions on Electronic Computers. EC-15 (5): 757-763 (October 1966).
- [4] Intel® 64 and IA-32 Architectures Software Developer Manuals, <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.
- [5] ARM Synchronization Primitives, [http://info-center.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A\\_arm\\_synchronization\\_primitives.pdf](http://info-center.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf).
- [6] Atomic operations library, <http://en.cppreference.com/w/cpp/atomic>.
- [7] Intel 64 Architecture Memory Ordering White Paper, [http://www.cs.cmu.edu/~410-f10/doc/Intel\\_Reordering\\_318147.pdf](http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf).
- [8] ADVE (S.V.) et GHARACHORLOO (K.). – *Shared Memory Consistency Models : A Tutorial*. <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf>.
- [9] FLYNN (M.J.). – *Some Computer Organizations and Their Effectiveness*. IEEE Trans. Comput. C-21 (9) : 948-960 (September 1972).
- [10] DONGARRA (J.). – *Lecture CS 594, – Scientific Computing for Engineers, HPC Start to Finish*. <http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2015/lect01-overview.pdf>.
- [11] DUPONT DE DINECHIN (B.) et al. – *A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications*. IEEE High Performance Extreme Computing Conference (HPEC '13), [http://iee-hpec.org/2013/index\\_html\\_files/44.pdf](http://iee-hpec.org/2013/index_html_files/44.pdf) (2013).
- [12] DENNIS et MISUNAS. – *A Preliminary Architecture for a Basic Data Flow Processor*. ISCA (1974).
- [13] GURD et al. – *The Manchester prototype dataflow computer*. CACM (1985).
- [14] LANG (H.W.). – *The Instruction Systolic Array, a Parallel Architecture for VLSI Integration*. The VLSI Journal 4, 65-74 (1986).
- [15] KUNG (S.Y.). – *VLSI Array Processors*. Prentice-Hall, Inc. (1988).
- [16] Chapel : Productive Language Programming – Cray, [chapel.cray.com](http://chapel.cray.com).
- [17] Fortran 77 standard : [http://www.fortran.com/fortran/F77\\_std/rjcnf.html](http://www.fortran.com/fortran/F77_std/rjcnf.html).
- [18] Fortran 90 standard : [ftp://ftp.nag.co.uk/sc22wg5/N001-N1100/N692.pdf](http://ftp.nag.co.uk/sc22wg5/N001-N1100/N692.pdf).
- [19] High Performance Fortran Forum, « *High Performance Fortran Language Specification, version 0.2* ». Center for Research on Parallel Computation, Rice University, Technical Report CRPC-TR92225, Houston, TX September (1992).
- [20] KOELBEL (C.H.) et al. – *The High Performance Fortran Handbook*. The MIT Press (1997).
- [21] OpenMP, <http://openmp.org/wp/>.
- [22] Message Passing Interface Forum, <https://www.mpi-forum.org/>.
- [23] CHERGUI (J.) et al. – *Message Passing Interface*. [http://www.lac.inpe.br/~stephan/CAP-372/IDRIS\\_MPI\\_cours\\_couleurs.pdf](http://www.lac.inpe.br/~stephan/CAP-372/IDRIS_MPI_cours_couleurs.pdf).
- [24] MPI version 3, <https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>.
- [25] BRINSKY (M.) et LUBIN (M.). – *An Introduction to MPI-3 Shared Memory Programming*. [https://software.intel.com/sites/default/files/managed/eb/54/An\\_Introduction\\_to\\_MPI-3.pdf](https://software.intel.com/sites/default/files/managed/eb/54/An_Introduction_to_MPI-3.pdf).
- [26] CAPPELLO (F.) et ETIEMBLE (D.). – *MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks*. Proc. Supercomputing'00, Dallas, TX, <http://www.sc2000.org/techpaper/papers/pap.pap214.pdf> (2000).
- [27] RABENSEIFNER (R.), HAGER (G.) et JOST (G.). – *Hybrid MPI and OpenMP Parallel Programming, MPI + OpenMP and other models on clusters of SMP nodes*. [http://openmp.org/sc13/HybridPP\\_Slides.pdf](http://openmp.org/sc13/HybridPP_Slides.pdf).
- [28] LEWIS (B.) et BERG. (D.J.). – *Multithreaded Programming with Pthreads*. California : Prentice Hall (1998).
- [29] Mach. <https://directory.fsf.org/wiki/Gnumach>.
- [30] Solaris. <https://www.oracle.com/solaris/solaris11/index.html>.
- [31] Chorus. <http://docs.oracle.com/cd/E19048-01/chorus5/index.html>.
- [32] COL (M.). – *Algorithmic Skeletons : structured management of parallel computation*. MIT Press, Cambridge, MA, USA (1989).
- [33] STITT (T.). – *An Introduction to the Partitioned Global Address Space (PGAS) Programming Model*. <http://cnx.org/contents/gtg1Azd1@7/An-Introduction-to-the-Partiti>.
- [34] VALIANT (L.G.). – *A bridging model for parallel computation*. Communications of the ACM, Volume 33 Issue 8 (Aug. 1990).
- [35] VALIANT (L.G.). – *A bridging model for multi-core computing*. Journal of Computer and System Sciences, 77(1), 154-166 (2011).
- [36] CULLER (D.) et al. – *LogP : towards a realistic model of parallel computation*. PPOPP '93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, Volume 28 Issue 7. Pages 1-12 (July 1993).
- [37] KAISER (H.) et al. – *HPX – A Task Based Programming Model in a Global Address Space*. PGAS 2014 : The 8<sup>th</sup> International Conference on Partitioned Global Address Space Programming Models. <http://stellar.cct.lsu.edu/pubs/pgas14.pdf> (2014).
- [38] WILLIAMS (S.), WATERMAN (A.) et PATERSON (D.). – *Roofline : An Insightful Visual Performance Model for Multicore Architectures*. Communications of the ACM, pp 65-76 (April 2009).
- [39] NAS benchmarks : <http://www.nas.nasa.gov/publications/npb.html>.
- [40] BLAS : <http://www.netlib.org/blas/>.
- [41] Lapack : <http://www.netlib.org/lapack/>.
- [42] PATTERSON (D.). – *The Trouble With Multi-core : Chipmakers are busy designing microprocessors that most programmers can't handle*. IEEE Spectrum, <http://spectrum.ieee.org/computing/software/the-trouble-with-multicore> (June 2010).



## À lire également dans nos bases

ANCEAU (F.) et ETIEMBLE (D.). – *Introduction à l'architecture des ordinateurs*. [H 1 000] (2010).

ANDRADE (G.B.). – *Calcul généraliste sur carte graphique – Du rendu au calcul massivement parallèle*. [TE 5 990] (2016).

ETIEMBLE (D.) et ANCEAU (F.). – *Hiérarchies mémoire : la mémoire virtuelle*. [H 1 003] (2014).

ETIEMBLE (D.) et ANCEAU (F.). – *Hiérarchies mémoire : les caches*. [H 1 002] (2012).

ETIEMBLE (D.) et ANCEAU (F.). – *Processeurs superscalaires multi-pipelines*. [H 1 010] (2015).

ETIEMBLE (D.) et ANCEAU (F.). – *Processeurs : exécution pipeline des instructions*. [H 1 004] (2013).

ETIEMBLE (D.) et LACASSAGNE (L.). – *Les extensions SIMD des jeux d'instructions*. [H 1 200] (2015).

ETIEMBLE (D.). – *Évolution de l'architecture des ordinateurs*. [H 1 058] (2016).

ETIEMBLE (D.). – *Introduction aux systèmes embarqués, enfouis et mobiles*. [H 8 000] (2016).

ETIEMBLE (D.). – *Les jeux d'instructions des processeurs*. [H 1 199] (2016).

ETIEMBLE (D.). – *Processeurs VLIW*. [H 1 012] (2015).

KRAKOWIAK (S.). – *Systèmes d'exploitation : principes et fonctions*. [H 1 510v2] (2015).

LEMOINE (F.). – *La programmation des systèmes parallèles hétérogènes*. [H 3 160] (2016).

SENTIEYS (O.) et TISSERAND (A.). – *Architectures reconfigurables FPGA*. [H 1 196] (2012).

## Sites Internet

Intel  
<http://ark.intel.com/fr>

ARM  
<http://www.arm.com/products/processors/>

---

# Gagnez du temps et sécurisez vos projets en utilisant une source actualisée et fiable



RÉDIGÉE ET VALIDÉE  
PAR DES EXPERTS




MISE À JOUR  
PERMANENTE



100 % COMPATIBLE  
SUR TOUS SUPPORTS  
NUMÉRIQUES



SERVICES INCLUS  
DANS CHAQUE OFFRE

- + de 340 000 utilisateurs chaque mois
- + de 10 000 articles de référence et fiches pratiques
- Des Quiz interactifs pour valider la compréhension 

## SERVICES ET OUTILS PRATIQUES



### Questions aux experts\*

Les meilleurs experts techniques et scientifiques vous répondent



### Articles Découverte

La possibilité de consulter des articles en dehors de votre offre



### Dictionnaire technique multilingue

45 000 termes en français, anglais, espagnol et allemand



### Archives

Technologies anciennes et versions antérieures des articles



### Info parution

Recevez par email toutes les nouveautés de vos ressources documentaires

\*Questions aux experts est un service réservé aux entreprises, non proposé dans les offres écoles, universités ou pour tout autre organisme de formation.

## Les offres Techniques de l'Ingénieur



### INNOVATION

- Éco-conception et innovation responsable
- Nanosciences et nanotechnologies
- Innovations technologiques
- Management et ingénierie de l'innovation
- Smart city – Ville intelligente



### MATÉRIAUX

- Bois et papiers
- Verres et céramiques
- Textiles
- Corrosion – Vieillessement
- Études et propriétés des métaux
- Mise en forme des métaux et fonderie
- Matériaux fonctionnels. Matériaux biosourcés
- Traitements des métaux
- Élaboration et recyclage des métaux
- Plastiques et composites



### MÉCANIQUE

- Frottement, usure et lubrification
- Fonctions et composants mécaniques
- Travail des matériaux – Assemblage
- Machines hydrauliques, aérodynamiques et thermiques
- Fabrication additive – Impression 3D



### ENVIRONNEMENT – SÉCURITÉ

- Sécurité et gestion des risques
- Environnement
- Génie écologique
- Technologies de l'eau
- Bruit et vibrations
- Métier : Responsable risque chimique
- Métier : Responsable environnement



### ÉNERGIES

- Hydrogène
- Ressources énergétiques et stockage
- Froid industriel
- Physique énergétique
- Thermique industrielle
- Génie nucléaire
- Conversion de l'énergie électrique
- Réseaux électriques et applications



### GÉNIE INDUSTRIEL

- Industrie du futur
- Management industriel
- Conception et production
- Logistique
- Métier : Responsable qualité
- Emballages
- Maintenance
- Traçabilité
- Métier : Responsable bureau d'étude / conception



### ÉLECTRONIQUE – PHOTONIQUE

- Electronique
- Technologies radars et applications
- Optique – Photonique



### TECHNOLOGIES DE L'INFORMATION

- Sécurité des systèmes d'information
- Réseaux Télécommunications
- Le traitement du signal et ses applications
- Technologies logicielles – Architectures des systèmes
- Sécurité des systèmes d'information



### AUTOMATIQUE – ROBOTIQUE

- Automatique et ingénierie système
- Robotique



### INGÉNIERIE DES TRANSPORTS

- Véhicule et mobilité du futur
- Systèmes aéronautiques et spatiaux
- Systèmes ferroviaires
- Transport fluvial et maritime



### MESURES – ANALYSES

- Instrumentation et méthodes de mesure
- Mesures et tests électroniques
- Mesures mécaniques et dimensionnelles
- Qualité et sécurité au laboratoire
- Mesures physiques
- Techniques d'analyse
- Contrôle non destructif



### PROCÉDÉS CHIMIE – BIO – AGRO

- Formulation
- Bioprocédés et bioproductions
- Chimie verte
- Opérations unitaires. Génie de la réaction chimique
- Agroalimentaire



### SCIENCES FONDAMENTALES

- Mathématiques
- Physique Chimie
- Constantes physico-chimiques
- Caractérisation et propriétés de la matière



### BIOMÉDICAL – PHARMA

- Technologies biomédicales
- Médicaments et produits pharmaceutiques



### CONSTRUCTION ET TRAVAUX PUBLICS

- Droit et organisation générale de la construction
- La construction responsable
- Les superstructures du bâtiment
- Le second œuvre et l'équipement du bâtiment
- Vieillessement, pathologies et réhabilitation du bâtiment
- Travaux publics et infrastructures
- Mécanique des sols et géotechnique
- Préparer la construction
- L'enveloppe du bâtiment
- Le second œuvre et les lots techniques