Programmation parallèle

Juvigny Xavier

Septembre 2021

Contents

1 Introduction

1.1 Qu'est-ce la programmation parallèle

La programmation parallèle consiste à concevoir et mettre en œuvre des algorithmes permettant de traiter plusieurs données simultanément dans le but de réduire de restitution par rapport à un algorithme séquentiel effectuant le même traitement sur ces mêmes données, l'algorithme employé ou l'ordre des opérations effectuées pouvant différer.

Le besoin pour certaines applications de répondre en temps réel ou contraint sur des données de plus en plus complexes ou volumineuses, ou le souhait del'industriel de vouloir réduire le coût d'exécution de l'application, associés à des limites de l'électronique motive l'utilisation de la programmation parallèle.

1.2 Quelques cas d'applications nécessitant un traitement intensif des données

Simulation numérique De tout temps, la simulation numérique a été consommatrice de mémoire vive et de temps CPU.

En exemple, on veut calculer le bruit généré par de petites turbulences au niveau du bec de sécurité d'une aile d'avion

Le phénomène physique en jeu sont des petites turbulences qui demandent un modèle préciset un maillage très fin pour bien les capturer et les modéliser, sur une partie d'une aile d'avion à l'échelle réelle.

Typiquement, une telle simulation demande un maillage contenant sept milliards de sommets avec cinq inconnues à résoudre par sommet. La mémoire totale nécessaire pour la simulation est de 7 To et il faut vingt-trois jours sur un calculateur actuel pour simuler

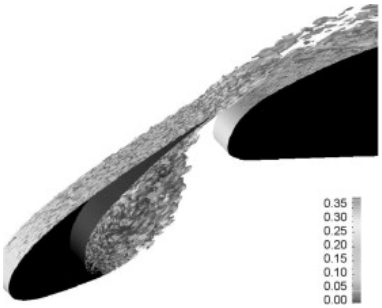


Figure 1: Simulation de petites turbulences autour d'un bec d'aile d'avion.

en séquentiel le problème physique sur une échelle de temps simulée de $\frac{1}{100}^{e}$ de secondes, sans parler du stockage mémoire prohibitif !.

Contrôle-Commande Le contrôle-commande permet à l'aide de capteurs permettant de mesurer des paramètres physiques de contrôler et commander à l'aide d'un logiciel un processus physique, par exemple contrôler le processus de fission se développant dans un réacteur nucléaire.

Ces logiciels sont contraints au temps réel et le nombre de capteurs et par conséquence de paramètres augmente avec le progrès de la technologie.

Par exemple, le contrôle-commande d'un moteur à combution est devenu de plus en plus complexe au fur et à mesure des besoins en rendement, consommation et pollution. En effet, en plus de fournir une puissance mécanique en fonction de l'appui sur la pédale de l'accélérateur, il est nécessaire d'avoir une optimisation de la consommation du véhicule en contrôlant les différents paramètres de la combustion : pression air, température, mélange, allumage, etc. L'ensemble de ces paramètres moteur est géré par un calculateur spécifique. Actuellement, un véhicule possède de nombreux calculateurs dédiés à des fonctions très diverses : freinage ABS, gestion moteur, éclairage, climatisation, etc. Ces différents calculateurs communiquent par un bus de terrain afin de partagerles informations et de gérer le véhicule de façon cohérente.

Le contrôle-commande de cette application est faite par l'intermédiaire de **sept capteurs** (pédale accélérateur, température air, pression air, température eau, rotation vilebrequin et deux capteurs de pollution) et de **quatre actionneurs** (injection essence, allumage, admission air, ré-injection gaz échappement ou brûlés). Le calculateur doit donc en temps réel optimiser le rendement moteur en fonction des sept capteurs et des données reçues des autres calculateurs qui calcule également en parallèle avec leurs propres capteurs et les données qu'ils ont reçues. Si un seul calculateur devait gérer l'ensemble des capteurs, le calcul d'optimisation serait trop coûteux pour pouvoir le faire en temps réel. Le calcul parallèle distribué mis en place par l'ingénierie automobile à l'aide de calculateurs multiples a permis un contrôle temps réel du rendement moteur.

Deep learning Le deep learning (apprentissage profond) est un ensemble de méthodes d'apprentissage automatique fondées sur l'apprentissage de modèles de données. Le deep learning s'applique dans des domaines variés comme internet (reconnaissance d'images, traduction automatique, etc.), la médecine (détection cellules cancéreuses, détection de drogues, etc.), la sécurité et la défense (détection faciale, surveillance video, …), l'autonomie des machines (détection pédestre, reconnaissance des panneaux de signalisation, …).

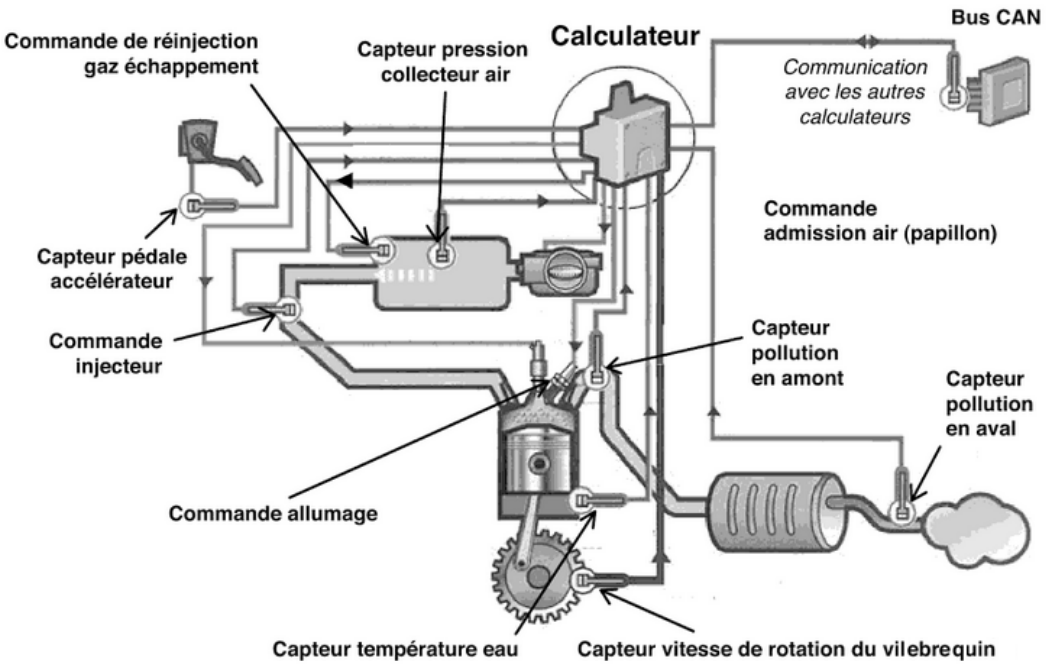


Figure 2: Représentation schématique de l'application de la commande d'un moteur à combustion.

La plupart des algorithmes de deep learning modélisent un réseau neuronal. Sur un ordinateur séquentiel l'apprentissage dure plus d'une année. La parallélisation sur GPGPU permet de ramener cet apprentissage à moins de un mois.

Ainsi, Alphago est le premier jeu de go sur ordinateur à avoir battu en Mars 2016 le champion du monde Fan Hui. Basé sur une parallélisation massive sur un cluster de plusieur nœuds de calcul contenant 1202 processeurs et 176 GPGPUs, son apprentissage n'a pris que trois semaines sur seulement cinquante GPGPUs (contre plusieurs années sur un ordinateur séquentiel). Néanmoins, Alphago a eu dans un premier temps un apprentissage supervisé.

En Octobre 2017, son successeur, Alphago zéro, dont l'apprentissage s'est fait complètement à l'aide d'un apprentissage profond, a battu Alphago cent parties à zéro !

Traitement de l'image De nos jours, les performances de l'électronique embarquée augmentent regulièrement allant de pair avec une miniaturisation de plus en plus poussée. Il est devenu de ce fait envisageable d'utiliser des capteurs optiques pour la navigation des véhicules autonomes (drônes, voiture autonome, etc.), en calculant des flux optiques entre deux images d'une séquence.

L'estimation temps réel du flux optique est défini en fonction de la fréquence des images, généralement 25 Hz, voire plus. La résolution des images allant en augmentant, aujourd'hui un système de calcul de flux optique se doit de savoir calculer un flux optique à partir de deux images de 1920×1080 pixels chacune en moins de $\frac{1}{30}$ de secondes soit plus de quatre millions de pixel en un trentième de seconde.

Ainsi, l'ONERA a mis au point un algorithme massivement parallèle sur GPGPU, eFOLKI qui permet de calculer le flux optique enrtre deux images de 1920×1080 pixels chacune en 13ms. Il est donc capable de calculer le flux d'une vidéo à soixante dix images par secondes.

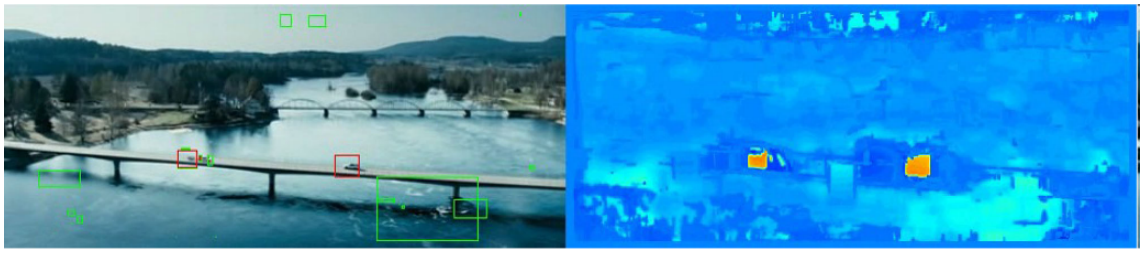


Figure 3: Détection d'objets en déplacement sur une vidéo aérienne. Sur la gauche, la vidéo d'origine avec des rectangles de détection pour deux niveaux de seuil (vert : faible déplacement, rouge : fort déplacement). Sur la droite, la norme du flux utilisé pour la détection (remerciements à Aurélien Plyer pour ses explications sur eFOLKI).

1.3 La limitation électronique

Pendant de nombreuses années, des architectures toujours plus complexes et performantes, des gravures toujours plus fines qui permettaient des fréquences toujours plus élevées ont permis aux processeurs de calculer toujours plus vite sans que l'on ait à changer de paradigme de programmation.

La première difficulté vint avec la mémoire vive dont l'accès aux données ne parvint pas à suivre la fréquence toujours plus haute des processeurs. Qu'importe, on rajouta alors une petite mémoire ultra rapide qui permettait d'accélérer l'accès aux données pour peu qu'on puisse suivant l'algorithme employé lire ou écrire successivement dans une petite région de la mémoire vive. Ces mémoires caches n'ont fait que grossir depuis au fur et à mesure de la montée en puissance des processeurs.

Durant toutes ces années, les industriels de l'informatique se forcèrent à suivre une "loi" empirique énoncée par **Gordon Moore**, un chercheur de l'entreprise Intel, en 1965 qui stipule que :

> Le nombre de transistors dans les microprocesseurs double tous les vingt-quatre mois.

Depuis quelques années, une seconde difficulté est apparue : la finesse des gravures et la hausse des fréquences induit de plus en plus de chaleur thermique difficile à dissiper. De plus, la finesse des gravures commencent à se heurter à un "mur quantique" matérialisé par l'effet tunnel.

D'autres alternatives au silicium contenu dans les processeurs graphiques sont à l'étude : ordinateur à calcul quantique, ordinateurs neuromorphiques (composés de neurones), transistors en graphène, etc. mais aucun n'a encore dépassé le stade de l'expérimentation.

Actuellement, pour augmenter la puissance de calcul de leurs processeurs, les constructeurs ont donc opté pour augmenter le nombre de cœurs de calcul sur un même processeur pouvant effectuer des opérations arithmétiques et logiques simultanément. Ainsi, de nos jours, un processeur "grand public" contiendra de 2 à 8 cœurs tandis que les processeurs vendus pour les calculateurs auront jusqu'à 72 cœurs (le processeur intel Xeon Phi capable de gérer sans pénalité 288 threads).

Simultanément, poussés par l'industrie vidéo ludique, les fabricants de cartes graphiques ont accrus la puissance et la souplesse de programmation des cartes graphiques. De GPU (Graphic Process Unit), la carte graphique edt devenue GPGPU (Global Purpose Graphic Process Unit). L'intérêt des GPGPUs sur les CPUs est de fournir plusieurs milliers d'unités de calcul. Les GPGPUs sont d'architectures plus simples que les CPUs (un GPGPU ne

serait pas capable de gérer un système d'exploitation) pouvant effectuer des calculs en parallèle. Très vite, les ingénieurs ont utilisé les GPGPUs comme coprocesseur pour accélérer les calculs et les traitements des données.

Ainsi, dans l'état de l'art actuel, tout programmeur voulant exploiter la puissance des CPUs et des GPGPUs se doit de pouvoir traiter plusieurs données en parallèle sur une machine donnée. On parle alors de **Programmation parallèle en mémoire partagée**.

Cependant la vitesse d'exécution des programmes avec la montée en puissance des processeurs devient de plus en plus tributaire de l'accès à la mémoire vive. Si la vitesse d'un programme est limitée par l'accès à la mémoire, on parle alors de programme *memory bound*, et la vitesse du programme est limitée par la vitesse de traitement du processeur, on parle de programme *cpu bound*.

Malgré des mémoires caches et des mémoires rapides (comme la mémoire cache mais gérée entièrement par le programmeur) toujours plus grandes (Le Xeon Phi d'Intel possède 16Go de mémoire rapide sur son chipset), permettant si elles sont bien exploitées d'accélérer l'accès à la mémoire, on se retrouve rapidement limité par l'accès mémoire d'autant que le nombre de cœur de calcul est grand.

Il faut donc pouvoir utiliser des unités de calcul ayant chacun sa propre mémoire vive pour encore avoir espoir d'accélérer la vitesse de traitement des données, chaque unité étant alors reliée par un réseau éthernet rapide : On parle alors de cluster et de **programmation parallèle à mémoire distribuée**.

2 Architecture des calculateurs parallèles

Comme nous avons pu voir dans la section précédente, les calculateurs parallèles se déclinent sous plusieurs formes ayant chacun leurs caractéristiques propres. Nous allons classer ces différentes architectures selon les contraintes qu'elles imposent sur la programmation.

2.1 Taxonomie de Flynn

La taxonomie de Flynn est une classification des architectures d'ordinateur, proposée par Michael Flynn en 1966.Les quatre catégories définies par Flynn sont classées selon le type d'organisation d'accès aux données et aux instructions.

Ces quatre catégories sont :

1. **SISD** (**S**imple **I**nstruction **S**imple **D**ata) : L'ordinateur n'exécute qu'une seule instruction à la fois qu'il applique que sur une donnée unique. C'est une architecture séquentielle excluant tout parallélisme, dont la programmation n'impose rien de particuliers.
2. **SIMD** (**S**imple **I**nstruction **M**ultiple **D**ata) : L'ordinateur n'exécute qu'une seule instruction à la fois mais celle ci est appliquée simultanément à plusieurs données simultanément. On peut voir cette architecture comme une unité de calcul vectorielle : au lieu de traiter des données scalaires, on traite des données vectorielles. Les unités SSE ou AVX des processeurs Intel peuvent être vues comme des unités SIMD. Les GPGPUs de NVIDIA possèdent des unités de calcul regroupées en "warp", chaque warp fonctionnant comme une unité SIMD.
   Ces unités de calcul SIMD demandent un soin particulier dans la mise en œuvre des algorithmes. Les instructions demandant des sauts conditionnels (if, boucles conditionnelles, etc.) peuvent être très pénalisantes sur ce type de machines, dû au système de masques utilisé pour ce type d'instructions. Par exemple, considérons le bout de code suivant :

```
if ( a[i]>= 0 )
  b[i] = c[i];
else
  b[i] = -d[i];
```

Une machine SIMD traitera simultanément plusieurs indices i simultanément, mais en appliquant exactement la même instruction élémentaire. Pour l'exemple ci-dessus, elle ne pourra pas appliquer les deux embranchements simultanément et devra utiliser un système de masque. Elle exécutera donc un code équivalent au code précédent comme suit :

```
msk[i] = a[i]>=0;// = 1 si vrai, 0 sinon
b[i] = msk[i] * c[i] - (1-msk[i]) * d[i];
```

On voit donc sur cet exemple que la machine SIMD est obligée d'exécuter les deux embranchements contrairement à une unité de calcul classique qui évalue l'un ou l'autre embranchement pour chaque i.
Pour les boucles conditionnelles, le problème est encore plus crucial. Considérons la boucle suivante qui effectue des itérations simultanées sur plusieurs suites de Mandelbrot $z_{n+1}[i] = z_n^2[i] + c[i]$ jusqu'à détecter une divergence lorsque $|z_n[i]| \geq 2$ pour divers i avec $c[i]$ fixé pour chaque i.

```
const long nIterMax = 65000;
z[i] = 0;
iter[i] = 0;
while ( (abs (z[i] ) < 2) && (iter[i] < nIterMax) )
{
  z[i] = z[i]*z[i]+c[i];
  iter[i] += 1;
}
```

Cette suite peut très bien diverger rapidement pour un `c[i]` donné et converger pour une valeur de `c[i]` proche ! Ainsi, certaines unités de calcul devront faire un grand nombre d'itérations pendant que d'autres en font très peu. Que se passe t'il dans le cas d'unités de calcul SIMD ?
Il est clair que toutes les UCs (Unités de Calcul) doivent itérer tant que l'une d'elles calcule une suite qui n'a pas encore divergé, du fait que toutes les UCs doivent exécuter la même instruction simultanément. Et un système de masque permettra aux suites ayant divergé de stagner sur la même valeur. Le code exécuté par des UCs SIMD ressemblera alors au pseudo code C++ suivant :

```
const long nIterMax = 65000;
z[i] = 0;
iter[i] = 0;
msk[i] = true;
while ( find (msk,true) ) // Tant qu'un des masques d'une UC est vrai
{
  z[i] = msk[i]*(z[i]*z[i]+c[i]) + (1-msk[i])*z[i];
  iter[i] += msk[i];
  msk[i] = (abs (z[i]) < 2) && (iter[i] < nIterMax);
}
```

Ainsi, chaque UC devra itérer autant de fois que l'UC itérant le plus, consommant inutilement de la puissance de calcul.
C'est pour ces raisons que les machines massivement SIMD telle que la connection machine dans les années 1980 (qu'on peut voir à la fin du film Jurrasic Park) ont

été peu à peu abandonnées au début des années 1990. Néanmoins l'architecture SIMD est reapparue sous forme de petites unités de calcul avec les GPGPUs et les registres vectoriels d'Intel.

3. **MISD** (**M**ultiple **I**nstruction **S**imple **D**ata) : Dans ce type de calculateur, une seule donnée est lue à la fois mais plusieurs instructions opèrent simultanément un traitement dessus. Cela peut sembler paradoxal mais recouvre en fait un type très ordinaire de micro-parallélisme dans les microprocesseurs modernes : les architectures pipelines associées aux registres vectoriels.

```
for ( int i = 1; i <= 4; ++i )
  r[i] = a[i] + b[i];
```

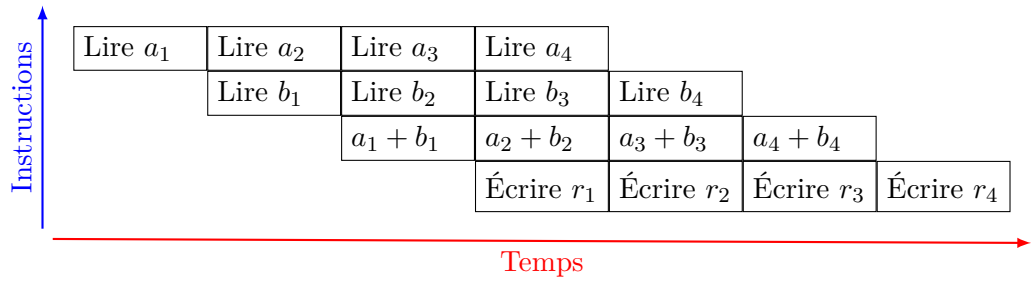


Figure 4: Principe d'une architecture pipeline

Sur la figure (4), on remplit les registres vectoriels a et b petit à petit, et dans le même instant, on effectue une addition sur les parties de a et b déjà remplies.

4. **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata)
Cette architecture permet de lire/écrire plusieurs données et d'effectuer des traitements différents sur ces données simultanément. Les processeurs multi cœurs des machines actuelles sont des unités MIMD.

2.2 Mémoire partagée et mémoire distribuée

La mémoire vive est un composant limitant l'efficacité du traitement des données dans un calculateur.

Comme on peut le voir sur la figure (6), l'évolution de l'efficacité des mémoires vives est bien plus lente que celle des unités de calculs. C'est pour cela que les calculateurs actuels offrent deux types de mémoire:

- **La mémoire partagée** : Les différentes unités de calcul de calcul partagent une même mémoire vive centrale au travers d'un bus mémoire. Cependant, la mémoire vive est d'un accès lent par rapport au traitement dont sont capables les processeurs actuels. Plusieurs techniques existent pour accélérer l'accès aux données :
  - *La mémoire vive entrelacée* : Une mémoire entrelacée à n voies de largeur w est constituée de n blocs mémoires. Le bloc numéroté i, avec $i \in [0, n-1]$ contient toutes les cellules dont les adresses sont égales à $w \times (n \times a + i) + k$ avec $k \in [0, w-1]$ et $a \in \mathbb{N}$. De cette manière, deux mots de largeur w rangés consécutivement sont stockés dans deux blocs différents. Cette organisation permet de réduire le temps d'accès à la mémoire lors de la lecture ou de l'écriture par groupe de mots.
  Supposons qu'une unité de calcul de cycle δt dont la mémoire principale possède un temps d'accès de δta après réception d'une requête, veuille lire un ensemble

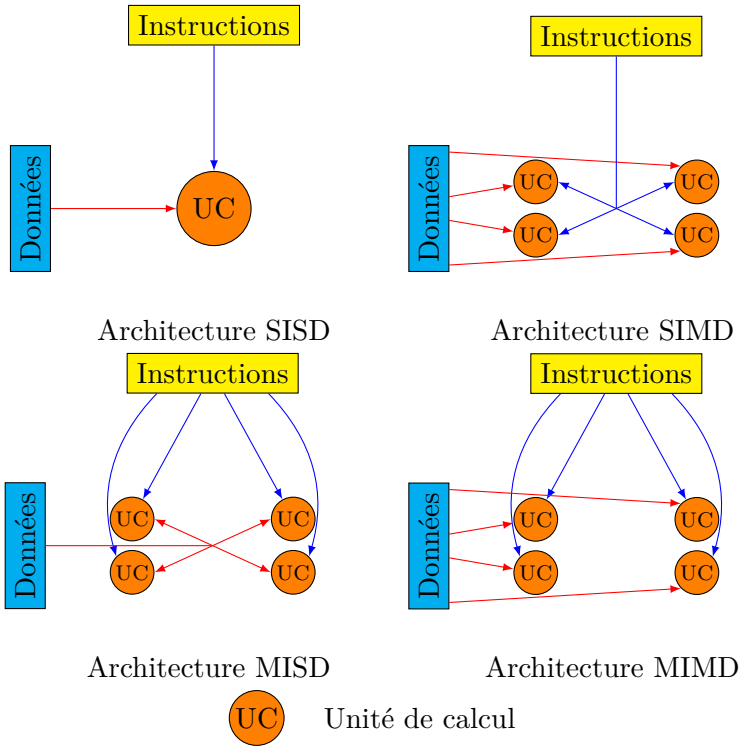


Figure 5: Taxonomie de Flynn

de N valeurs de taille w par valeur, rangées de façon contigüe dans un tableau. Dans une mémoire non entrelacée, cette opération demande un temps $\Delta t_1 = N.\delta ta$

Dans une mémoire entrelacée à n voies, telle que $\delta ta \leq n.\delta t$, la première requête est transmise au bloc i de la mémoire. Au cycle d'horloge suivant, la seconde requête est transmise au bloc $i+1$, et ainsi de suite. Chacune des n premières requêtes est ainsi transmise à un bloc différent. Ces blocs fonctionnent soit en parallèle (il possible de faire des requêtes simultanées sur des blocs différents) soit en pipeline (une requête par cycle d'horloge sur des blocs différents). À l'instant δta après la première requête, la première donnée est disponible. La deuxième donnée est disponible un cycle δt plus tard et ainsi de suite comme illustré sur la figure (8).

Les données suivantes pourront être récupérées à chaque cycle. Il faut donc un temps $\Delta t_2 = \delta ta + N.\delta t$ pour lire N valeurs et attendre un temps δta sans accéder à aucune donnée.

Ce type de mémoire, dont le coût dépend du nombre de voies et coûte bien plus chère que la mémoire non entrelacée est principalement utilisée sur les GPGPUs et est associée avec de la mémoire cache.

– *La hiérarchie de mémoire cache* : La mémoire cache sur les unités séquentielles permettent, lorsqu'elles sont bien employées, c'est à dire lorsque la mise en œuvre de l'algorithme satisfait à un accès local temporel et spatial des variables (on essaie de reutiliser successivement le plus possible un même petit ensemble de données) d'accélérer de façon significative le traitement des données par une unité de calcul. Néanmoins, sur un calculateur parallèle, la mémoire cache obéit également à d'autres contraintes du fait du partage de certains niveaux

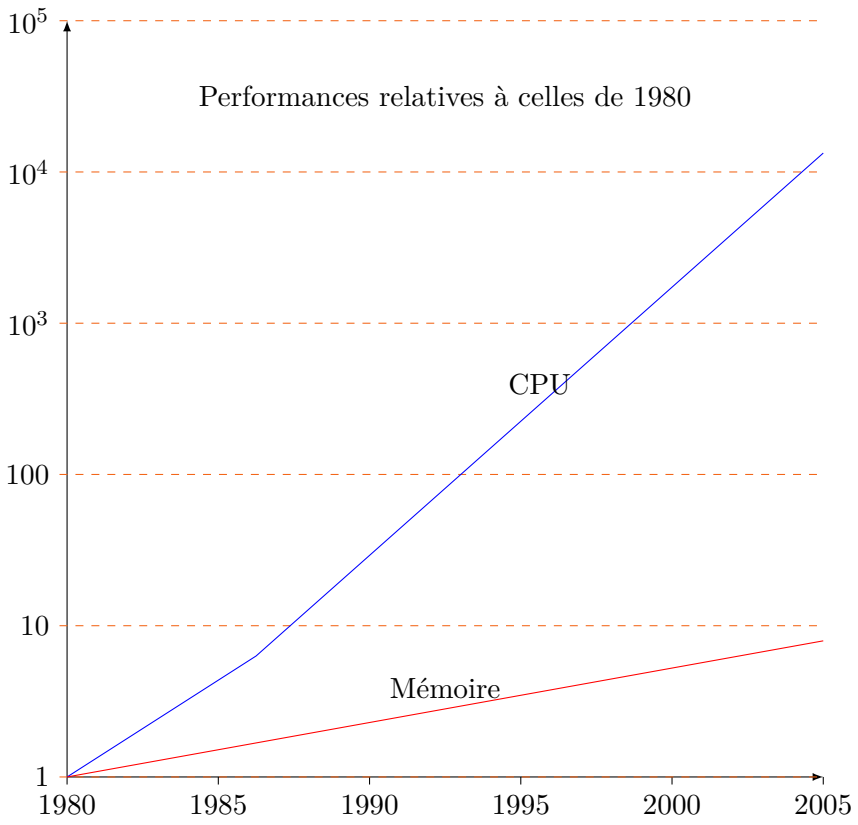


Figure 6: Comparaison de l'évolution des unités de calcul et de la mémoire vive

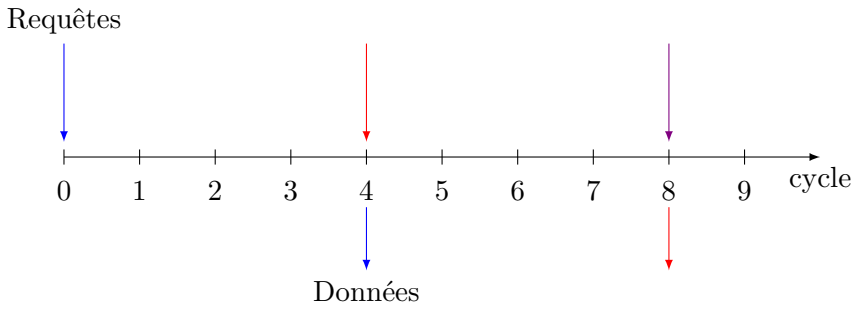


Figure 7: Accès à une mémoire non entrelacée

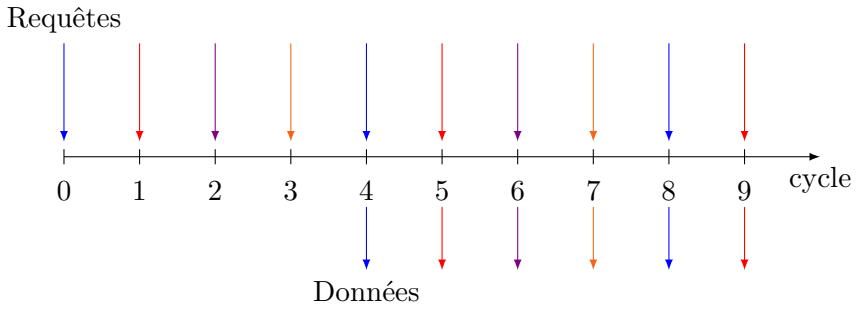


Figure 8: Accès à une mémoire entrelacée à quatre voies

de cache et de la mémoire vive entre les différentes unités de calcul.

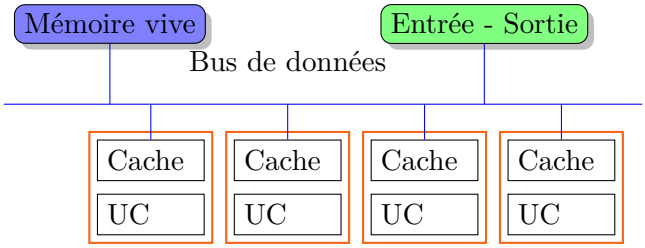


Figure 9: Organisation de la mémoire cache dans un calculateur multi–processeur

Dans le cas d'une machine multiprocesseurs (voir figure (9), il faut assurer la cohérence des données entre mémoires caches :

* Seule cette mémoire cache contient la donnée. Elle est donc valide et aucune synchronisation n'est nécessaire avec la mémoire vive;
* La donnée est partagée avec d'autres caches. Il faut donc vérifier à chaque accès si elle n'a pas été modifiée par d'autres processeurs et la marquer dans ce cas comme invalide.
* La valeur a été modifiée dans le cache et la valeur en mémoire vive n'est plus à jour. Il faudra donc mettre à jour la mémoire vive si d'autres processeurs veulent lire la valeur;
* La valeur dans la mémoire cache est invalide. La prochaine lecture de cette valeur déclenchera une lecture en mémoire vive.

Dans le cas d'une machine multi–cœurs, plusieurs unités de calcul partagent les mêmes niveaux de cache (voir la figure (10)) afin d'éviter de passer par la

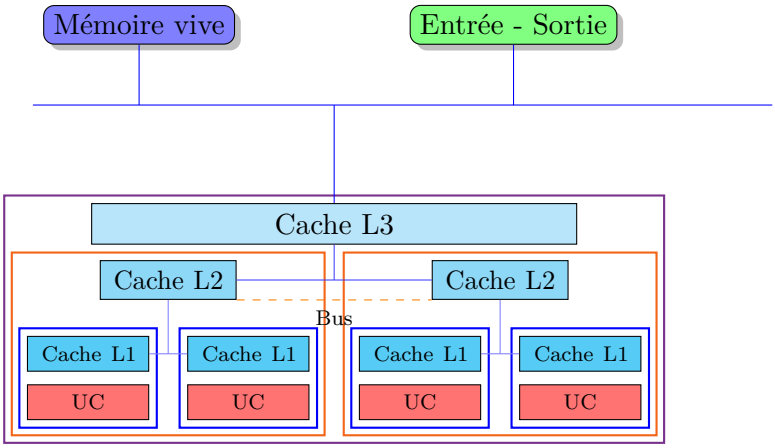


Figure 10: Organisation de la mémoire cache dans un calculateur multi–cœur

mémoire vive pour s'assurer de la cohérence des données en mémoire cache. De plus, afin d'éviter de remonter d'un niveau de cache, les multi–cœurs récents possèdent des bus reliant les mémoires caches d'un même niveau. Lors de l'optimisation d'un code sur une machine multi–cœurs, il est important d'avoir cette organisation des mémoires caches en tête afin d'organiser les threads et les données de sorte à ce qu'un minimum de données transitent par les bus reliant les mémoires caches. Cela impose donc les mêmes contraintes qu'une machine NUMA (Non Uniform Memory Access) que nous allons voir plus loin.

- **La mémoire distribuée**
  L'efficacité d'une machine multi–cœurs dépend de l'organisation de sa mémoire qui, pour être efficace demande une architecture complexe et onéreuse dont le coût est sur-linéaire au nombre de cœurs. Afin d'obtenir un calculateur performant sans être prohibitif, chaque unité de calcul (en général multi–cœurs) possède sa propre mémoire vive. Un ensemble unité de calcul multi cœur et mémoire vive s'appelle un **nœud de calcul** sur les machines distribuées.
  Ces différentes unités de calcul échangent des données au travers d'un bus spécialisé ou plus souvent au travers d'un réseau de type Gigabit. On parle alors d'un calculateur *à mémoire distribuée*. Ce type de calculateur a un coût de fabrication linéaire en fonction du nombre d'unités de calcul sur le réseau du calculateur.
  La programmation de ces calculateurs demande une mise en œuvre spécifique et des algorithmes spécifiques permettant d'exploiter efficacement la capacité de calcul de la machine.
  Ce type de calculateur permet d'exploiter simultanément plusieurs dizaines de milliers d'unités de calcul. Cependant, la lenteur relative des réseaux ou des bus ainsi que le "degré de parallélisme" de l'algorithme employé limitent la rapidité de traitement des données
  Les réseaux connectant les différentes unités de calcul offrent des topologies variées. Il peut être important d'avoir en tête la topologie du calculateur employé. En effet, les données échangées via le réseau transiteront selon la topologie du réseau via d'autres unités de calcul, ce qui ralentira les échanges de données en fonction du nombre d'unités de calcul intermédiaires par lesquelles les données transiteront. Pour calculer le nombre maximal de nœuds intermédiaires, on définit pour un réseau la distance entre deux nœuds et son diamètre :
  On appelle **distance entre deux nœuds** sur un réseau donné le minimum de nœuds

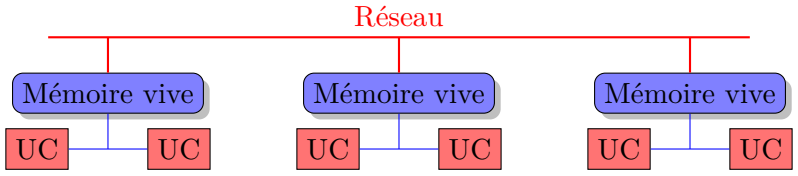


Figure 11: Architecture calculateur à mémoire distribuée

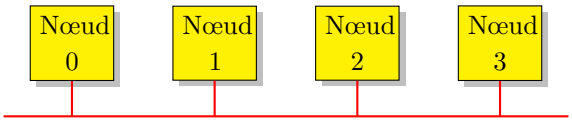


Figure 12: Réseau Linéaire

intermédiaires par lesquels doivent transiter des données echangées entre les deux nœuds.
On appelle **diamétre d'un réseau** la distance maximale entre deux nœuds du réseau.
Pour la commodité de l'exposé, et en adéquation avec la vue programmeur fournie par les bibliothèques de calcul parallèle distribuée, on numérote chacun des N nœuds de calcul de façon unique à l'aide d'un nombre entre 0 et $N-1$.

- **Topologie linéaire**
  Les N nœuds de calcul sont reliés de sorte qu'on peut trouver une numérotation des nœuds de calcul tel qu'un nœud i est seulement lié directement au nœud $i-1$ (si $i > 0$) et au nœud $i+1$ (si $i < N-1$).
  Le diamètre de ce réseau est $N-1$.
- **Topologie anneau**
  La topologie de ce réseau est similaire à celui d'un réseau linéaire. Seule différence notable, le nœud 0 est relié au nœud $N-1$.
  Le réseau étant bidirectionnel, le diamètre de ce réseau est $\frac{N}{2}$ si le nombre de nœud N est pair, $\frac{N-1}{2}$ sinon.
- **Topologie grille**
  Le réseau connectant les $N = W \times H$ nœuds est topologiquement équivalent à une grille de $W \times H$ nœuds.
  On voit, en considérant la communication du premier nœud avec le dernier nœud que le diamètre d'un réseau sur grille est égal à $W + H - 2$.
- **Topologie hyper cube**
  La topologie hyper cube est une topologie telle que la connection des nœuds de calcul est équivalente à celle reliant les sommets d'un hyper cube de dimension n.

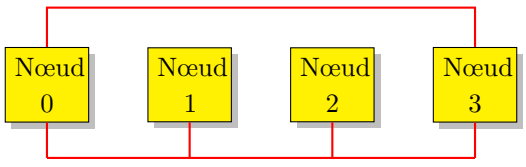


Figure 13: Réseau en anneau

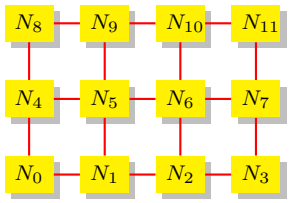


Figure 14: Réseau sur grille

La construction d'un hypercube de dimension n se fait de façon itérative à partir de l'hypercube de dimension 1 (l'hypercube de dimension 0 étant un cas à part, constitué d'un et un seul nœud de calcul) :

* L'hypercube $\mathcal{C}_1$ de dimension 1 est constitué de deux nœuds numérotés 0 et 1 reliés par une connexion directe (voir figure 15b);
* Pour passer d'un hypercube de dimension $n-1$ à un hypercube de dimension n ($n > 1$), on fabrique une copie du réseau hypercube de dimension $n-1$ en rajoutant (en binaire) un 0 à gauche dans la numérotation des nœuds d'origine et un 1 à gauche dans la numéroation des nœuds copiés. On relie ensuite par une connexion directe chaque nœud d'origine avec son équivalent dans la copie.

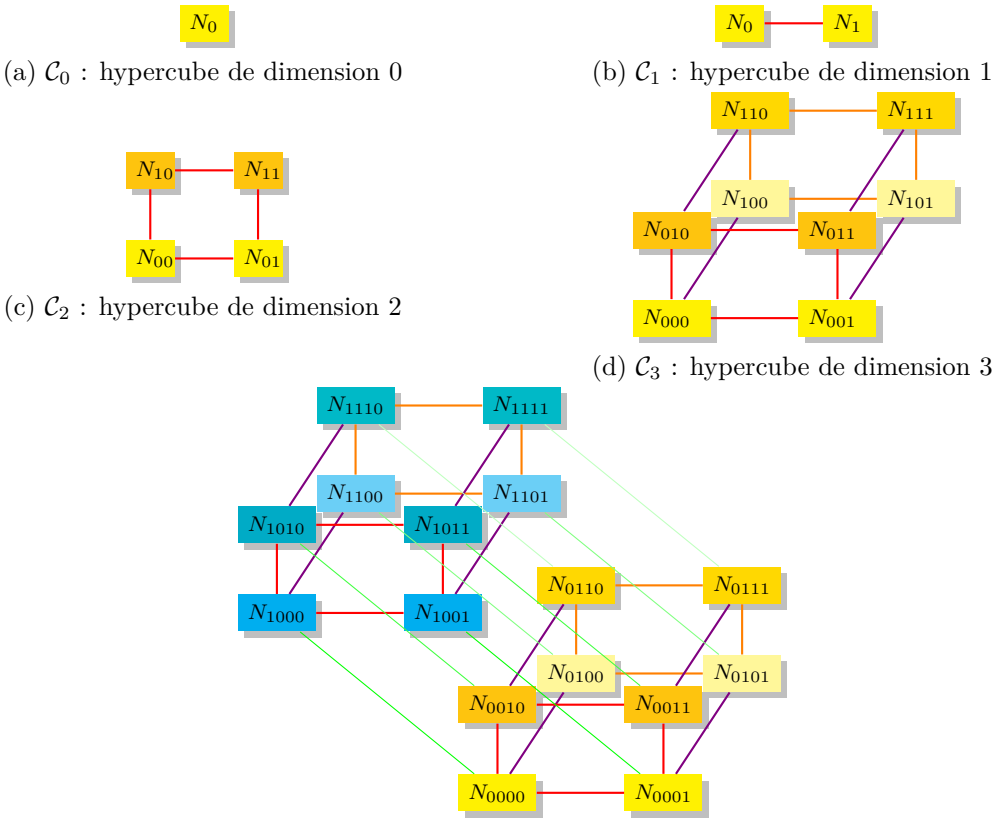
Par exemple, pour construire un hypercube $\mathcal{C}_2$ de dimension 2, on démarre de l'hypercube $\mathcal{C}_1$ possédant deux sommets 0 et 1 qu'onon copie en rajoutant un bit à gauche de la numérotation binaire, valant 0 pour le graphe original et 1 pour la copie. On obtient ainsi quatres nœuds numérotés $00, 01, 10$ et 11 reliés comme illustré dans la figure 15c.
Pour les dimensions supérieurs, on reitère la procédure pour obtenir des hypercubes de dimension 3, 4 ou supérieur, comme illustré sur les figures 15d et 15e.
La numérotation obtenue lors de la construction des hypercubes se nomme le *code Gray* (sur n bits, n étant la dimension de l'hypercube). Cette numérotation possède de nombreuses propriétés intéressantes, dont la propriété suivante :

Propriétés 1 *La distance entre deux nœuds i et j dans un hypercube de dimension n est le nombre de bits différents dans leur numérotation binaire.*

Par exemple, la distance entre les nœuds (numérotés en binaire) 011001 et 110011 est de 3 (trois bits de différents entre les deux numéros).
On en déduit directement le diamètre du réseau hypercube :

Propriétés 2 *Un réseau hypercube de dimension n (contenant 2^n nœuds de calcul) a un diamètre de n.*

Les réseaux hypercubes sont des réseaux proposant des connections très riches, et permettant d'obtenir de bonnes performances en terme d'échanges de donnée et de résistance aux pannes réseaux sans pour autant avoir un coût trop élevé (comme avoir un graphe complet, ou cluster, c'est à dire que chaque nœud est



(a) $\mathcal{C}_0$: hypercube de dimension 0

(b) $\mathcal{C}_1$: hypercube de dimension 1

(c) $\mathcal{C}_2$: hypercube de dimension 2

(d) $\mathcal{C}_3$: hypercube de dimension 3

(e) $\mathcal{C}_4$: hypercube de dimension 4

Figure 15: Réseau hyper cube de dimensions 0 à 4. Les nœuds sont numérotés en binaire selon la numérotation de Gray.

relié à tous les autres).

Aujourd'hui, le nombre d'unités pouvant être mis sur ce type de calculateur est limité par la consommation électrique ! On a estimé que sur ce type de machine (excluant les GPGPUs), atteindre l'hexaflops demandait une consommation électrique équivalente à l'électricité fournie par un réacteur nucléaire !

C'est une des principales raisons du succès actuel des GPGPUs dont le rapport vitesse de calcul/consommation électrique est bien inférieur à celui d'un CPU classique. Les fabricants de CPUs et de GPGPUs (Intel et NVIDIA associé avec ARM principalement) se livrent aujourd'hui à une course sur la consommation électrique de leurs unités de calcul, d'autant plus que c'est un point clef pour les systèmes embarqués (drônes, voitures, etc) et nomades (téléphones mobiles, tablettes, etc.).

2.3 Architecture NUMA

Nous avons vu que sur un calculateur parallèle à mémoire partagée, le nombre de cœurs de calcul était limité par la complexité et le coût de fabrication. Actuellement, le Knight Landings d'Intel est le chipset contenant le plus grand nombre de cœurs de calcul, c'est à dire soixante douze cœurs, accompagnés d'une mémoire rapide entrelacée de 16 Go.

D'autre part, si les architectures parallèles à mémoire distribuée permettent d'atteindre plusieurs centaines de milliers de cœurs de calcul, elles demandent une mise en œuvre spécifique.

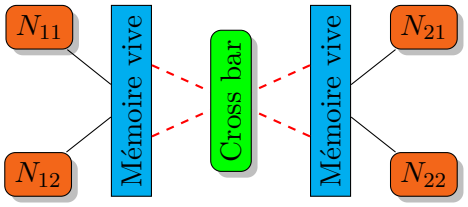


Figure 16: Calculateur à architecture NUMA

L'idée est donc de connecter les différentes mémoires vives à l'aide d'un bus de données appelé crossbar (par exemple le Intel QuickPath Interconnect (IQI) est un crossbar) qui permet au programmeur de voir ces différentes mémoires vives comme une seule mémoire vive unifiée.

Pour atteindre des performances honorables, ce type d'architecture demande une programmation particulière. En effet, le crossbar effectue des transferts mémoires entre les différents nœuds mémoires bien plus lentement qu'un simple accès à la mémoire vive. Ainsi, si une unité de calcul d'un nœud donné veut accèder à des données se trouvant sur un autre nœud de calcul, il subira une importante pénalité dûe au temps de transfert au travers de ce crossbar.

Pour éviter ces temps de pénalités importants, il faut donc travailler le plus possible sur des données qui sont locales au nœud. Cependant, comment savoir si une donnée se trouve localement sur le nœud ou non ?

Tout dépend en fait de la politique d'exécution choisit ! En effet, sur ce type de machine, à l'aide de variables d'environnement (dépendant du compilateur ou du système employé) on peut choisir différentes politiques de gestion de la mémoire, comme par exemple :

- Politique "tableaux entrelacés" : Lors de l'allocation d'un tableau, on distribue chaque page mémoire (une page mémoire faisant typiquement 64ko de mémoire vive) de façon cyclique sur chaque nœud de calcul : Ainsi la première page mémoire occupée par le tableau sera sur le nœud zéro, la deuxième sur le nœud un, etc...
- Politique "Politique du premier accès" (First Touch policy) : Lors de l'allocation, le tableau n'est que virtuellement alloué (comme sur tous les systèmes), et son allocation se fait page mémoire par page méoire en fonction des accès. L'idée ici est d'allouer la page mémoire sur le nœud dont une unité de calcul est la première à accéder en lecture ou écriture à des données situées sur cette page.

La stratégie employée pour utiliser des données locales à un nœud dépendra donc fortement de la politique d'exécution choisie !