Parallélisme

Notes de cours – Octobre 2004

Cours de Bastien Chopard
Prise de notes et mise en page Gregory Loichot

Formalités

Forme de l'examen

La note est constituée à 50% par la note d'oral et à 50% de la moyenne des travaux pratiques.

Horaires de cours

Cours : Jeudi de 8h15 à 10h, salle Duf-259.
Exercices : Mardi de 12h15 à 14h, salle des PCs.

Chapitre 1

Architecture parallèle et modèle de programmation

1.1 Définitions et but du parallélisme

Il est très naturel de partager un gros travail parmi plusieurs personnes.

Exemple
Construire la muraille de Chine (impossible à faire avec une seule personne). Donc il faut plusieurs personnes qu'il faut, en plus, gérer.

Il faut donc non seulement gérer ces personnes, mais aussi les coordonner et les empêcher de se contrarier mutuellement.

Exemple
Mettre la pierre du bas avant celle du haut (sic !), rassembler les différents tronçons, …

En informatique, le parallélisme a rapidement été considéré comme une option naturelle pour calculer.

Exemple
L'ENIAC (premier ordinateur construit en 1945) avait déjà des unités de calcul multiples et activables en même temps.

Exemple
Von Neumann, dans les années 40, proposa le concept d'automate cellulaire comme un modèle de calcul. C'est un modèle massivement parallèle.

Mais, à l'époque, il n'y avait pas la technologie ni le savoir-faire suffisant pour réaliser ce parallélisme et pour l'exploiter. Il fallait donc trouver plus simple et à Von Neumann de proposer l'architecture séquentielle (par opposition à « parallèle ») qui est toujours à la base des machines actuelles. Cela a donné un modèle clair tant pour le hardware (HW) que pour le software (SW). Les programmeurs pouvaient tirer parti de la machine efficacement. C'est l'une des raisons du succès de l'architecture de Von Neumann.

Pour le parallélisme, il n'y a pas de convergence aussi claire des modèles de programmation et des architectures. Cela ralenti le développement en plus du fait que c'est plus compliqué à écrire.

Il y a un besoin constant pour plus de performances et le séquentiel n'évolue pas toujours assez vite pour satisfaire tous les besoins donc il faut du parallélisme.

Définition *Ordinateur parallèle*

Un ordinateur parallèle est une machine composée de plusieurs processeurs qui coopèrent à la solution d'un même problème.

Définition *Système distribué*

Un système distribué (ou réparti) est un système composé de plusieurs processeurs impliqués dans la résolution d'un ou plusieurs problèmes.

Ces deux définitions sont proches et la frontière entre systèmes distribués et machines parallèles est un peu floue.

En pratique on peut faire les distinctions suivantes :

Parallélisme	Systèmes distribués
Objectif de traiter des problèmes plus gros plus vite.	Résulte souvent de la nécessité de répartir spacialement des services ou des composants software sur des ordinateurs distants.
Mise en commun organisée de ressources de calcul.	Résulte aussi du fait que certaines applications définissent naturellement des entités en concurrence.
On va souvent essayer d'exploiter des régularités de l'architecture.	Reflète le modèle client – serveur, producteur – consommateur, la mise en réseau de plus en plus de ressources.
On a un couplage fort entre processeurs, une granularité fine du découpage du problème (au niveau des variables du programme : les processeurs travaillent sur des variables différentes du programme).	Il n'y a pas d'enjeu de performance : couplage faible, granularité grosse (découpage au niveau de l'application : distribution de morceaux d'application à chaque processeur).

Fig. I, Parallélisme vs Systèmes distribués

1.2 Architecture parallèle

Quel(s) est(sont) le(s) hardware(s) à disposition pour faire du parallélisme ?
Un premier choix est le SIMD (Single Instruction flow, Multiple Data flow). Les processeurs exécutent en même temps la même instruction mais sur des données différentes.

Exemple
Additionner deux matrices : on répète le `add` pour tous les éléments.

Définition *Machine SIMD*

C'est une machine dans laquelle tous les processeurs sont synchrones. Un seul processeur (le master ou le frontal) exécute le programme et envoie la même instruction à tous les processeurs de calcul. On parle de Processing Element (PE) pour désigner les processeurs de calcul d'une architecture parallèle.

Les SIMD se sont développés au cours des années 80 – 90. Ils ont démontré le potentiel du parallélisme à la communauté scientifique vers 1989 (avec une machine à 65'536 PE).

Le succès du SIMD a été sa « simplicité » par rapport à un modèle plus général, d'où ses performances sur les problèmes scientifiques qui ont des données régulières.

Au début des années 90, des architectures plus flexibles ont vu le jour : le MIMD (Multiple Instruction flow, Multiple Data flow) où chaque PE exécute son propre programme sur ses propres données (la coordination n'est plus assurée par la synchronisation). Les PE sont autonomes et asynchrones. Cette architecture est plus difficile à programmer efficacement et il est plus difficile de construire du hardware fiable et performant. Au fil des années, le MIMD à remplacé progressivement le SIMD (par besoin d'une souplesse pour traiter des problèmes irréguliers). Aujourd'hui les SIMD sont utilisés dans les machines dédiées (traitement du signal, …).

Un autre aspect capital d'une architecture parallèle est sa mémoire : est-elle partagée ou distribuée ?

1.2.1 Mémoire partagée et distribuée

Définition *Mémoire partagée*

Tous les PE ont accès aux mêmes données, l'espace mémoire est global et uniforme. Par analogie, le tableau noir d'une classe est la mémoire partagée que les élèves et le professeur utilisent pour communiquer.

Avec cette méthode, il y a un problème au niveau hardware : les PE ne peuvent pas tous avoir accès à la mémoire en même temps.

Définition *Mémoire distribuée*

Chaque processeur a sa propre mémoire et ses propres données. L'espace mémoire est fragmenté.

La troisième composante architecturale est le réseau d'interconnexion qui relie les processeurs entres eux (dans le cas de la mémoire distribuée) ou qui relie les processeurs avec les bancs mémoire (dans le cas de la mémoire partagée).
Le réseau d'interconnexion doit permettre l'échange d'informations entre les processeurs. Ses performances sont critiques.

1.2.2 Topologie des réseaux d'interconnexion : Mémoire distribuée et partagée

Figure II : C'est un exemple de topologie statique structurée en une grille bidimensionnelle.

Dans ce cas, on connaît cette structure (contrairement aux applications réseaux distribuées) et sa fiabilité donc on peut l'exploiter à son maximum.

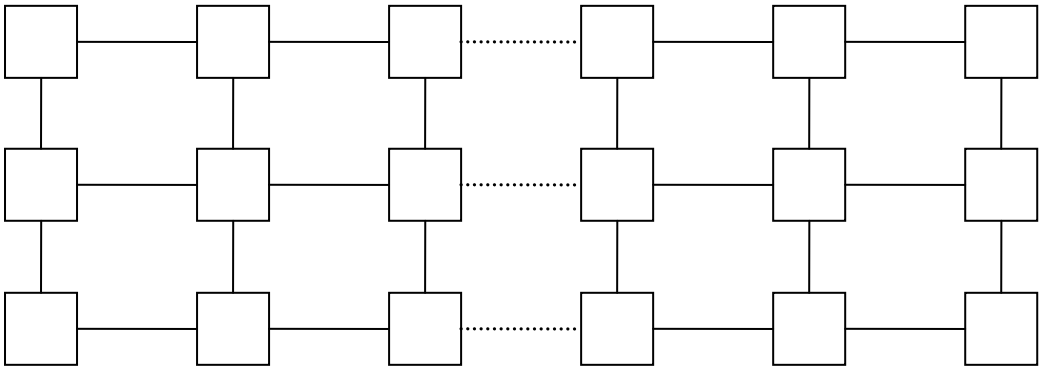
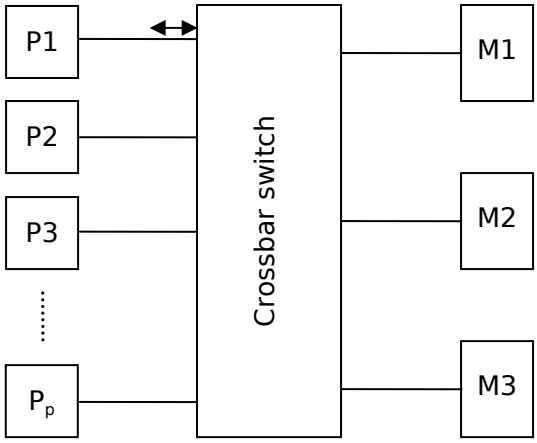


Fig. II – Topologie classique – Chaque carré symbolise un nœud (PE+routeur).

Exemple
Cas de la mémoire partagée.



Pour une machine bi-cpu standard, le switch est remplacé par un bus.

Sur la Fig. III, on dénote par P_i les Processing Elements.

Les bancs mémoire peuvent être en nombre quelconque. Ils sont notés par M_i.

Le Crossbar switch est un switch haute performance afin de minimiser le temps de latence.

Fig. III – Topologie à mémoire partagée

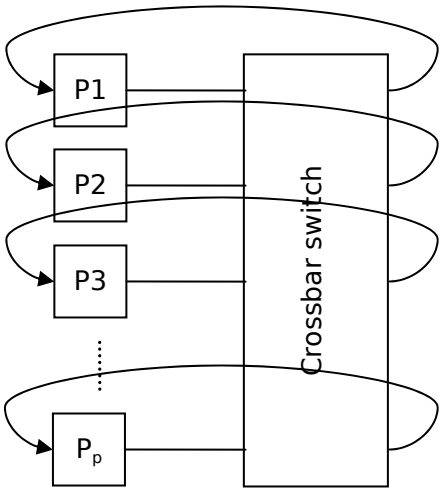


Fig. IV – Topologie à mémoire distribuée avec un réseau d'interconnexion Crossbar

1.2.3 Architecture : SIMD, MIMD

Schéma SIMD

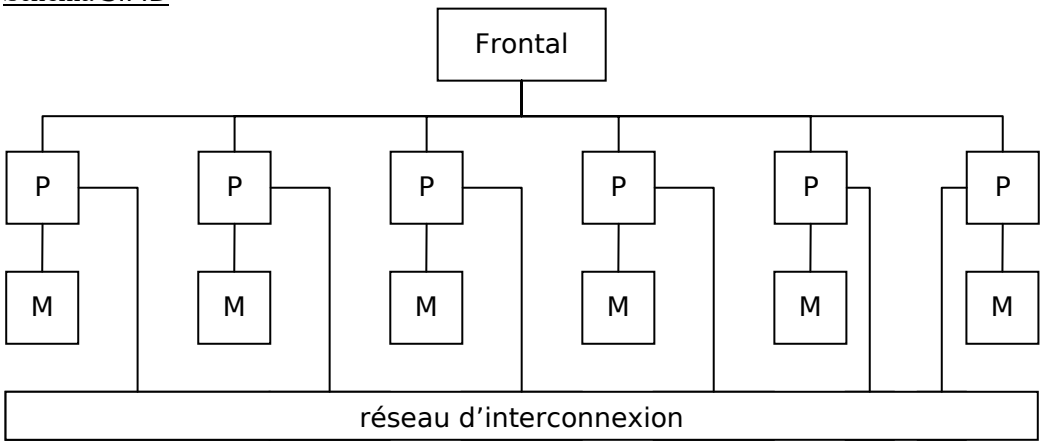


Fig. V – SIMD à mémoire distribuée

La Fig. V représente le cas de l'architecture SIMD à mémoire distribuée. Le réseau d'interconnexion est un graphe régulier (chaque cpu a le même nombre de voisins).

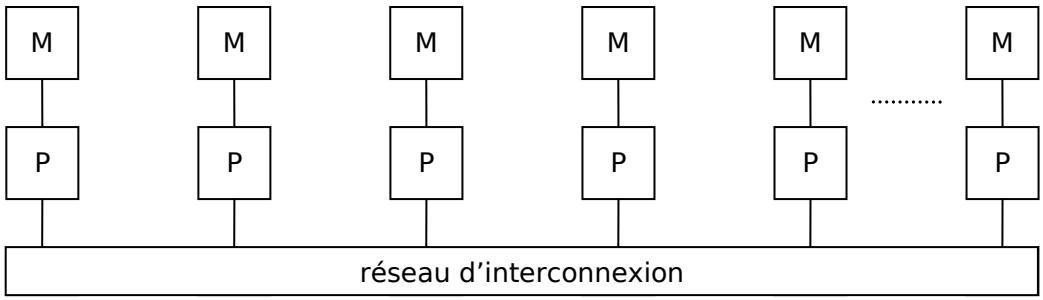


Fig. VI – MIMD à mémoire distribuée

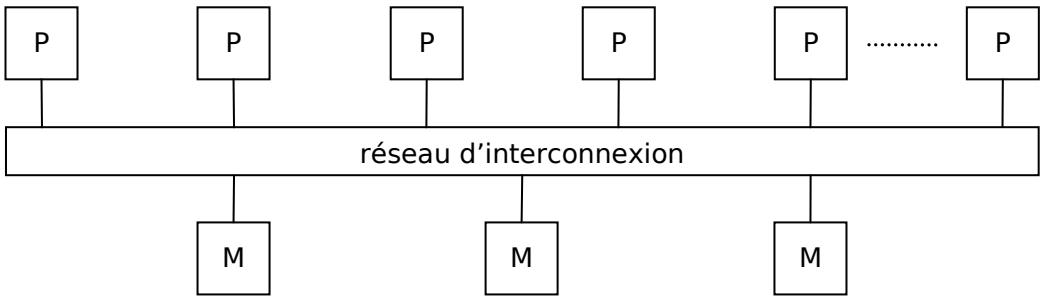


Fig. VII – MIMD à mémoire partagée

Définition *Multiprocesseur*

Un multiprocesseur est une architecture MIMD à mémoire partagée.

Définition *Multicomputer*

Un multicomputer est une architecture MIMD à mémoire distribuée.

Définition *SMP*

(Symetric Multi Processors). Est un multiprocesseur où tous les processeurs sont équivalants.

Définition *MPP*

(Massively Parallel Processor). Est un multi ordinateur comportant beaucoup de nœuds et ayant un réseau performant.

Définition *Beowulf*

C'est le multi ordinateur « du pauvre » construit avec les moyens du marché (PC + interconnexion bon marché). Très bon rapport prix / performance, logiciel libre, … Les Beowulf furent introduits par la NASA.

1.2.4 D'autres visions de l'architecture MIMD

1.2.4.1 NUMA (Non Uniforme Memory Access)

Les processeurs ont un accès privilégié à leur propre mémoire, mais peuvent aussi accéder à celle des autres processeurs. On se rapproche un peu d'un modèle à mémoire partagée sans perdre la scalabilité (possibilité d'avoir beaucoup de processeurs).
Il faut savoir qu'une architecture MIMD à mémoire partagée fonctionne moins bien avec beaucoup de processeurs car il y aura d'autant plus de conflits d'accès mémoire que le nombre de processeurs sera grand.

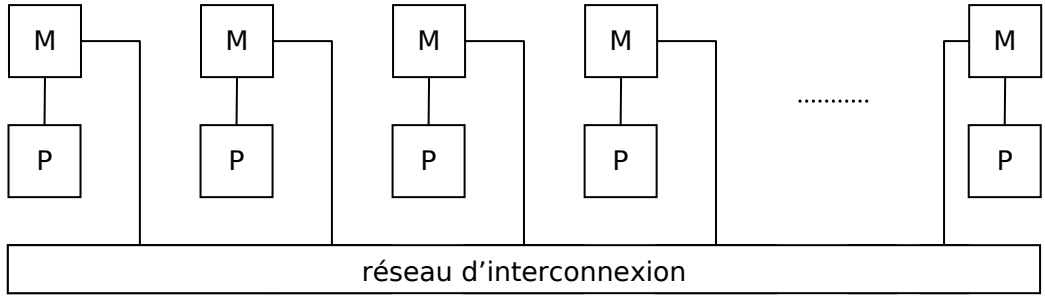


Fig. VIII – Architecture NUMA

1.2.4.2 COMA (Cache Only Memory Access)

Ici les données n'ont pas d'emplacement fixe mais migrent, selon les besoins, vers le cache du PE qui en a besoin.

Rappel La mémoire cache est une mémoire associative (adressable par son contenu).

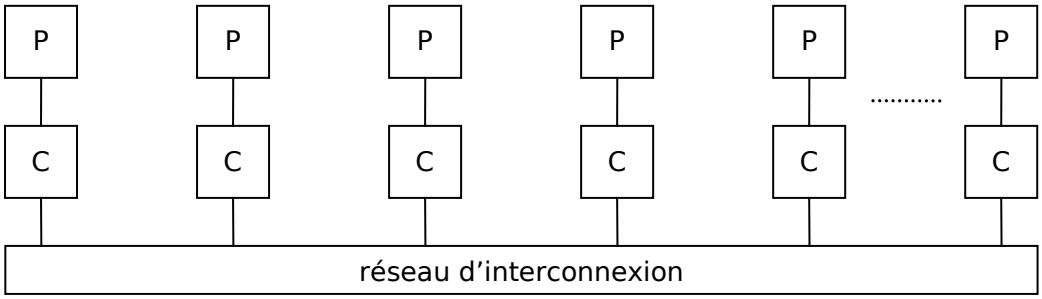


Fig. IX – Architecture COMA – « C » représente la mémoire Cache.

1.2.4.3 Architecture hybride

Un nœud est un multiprocesseur et tous ces nœuds sont reliés de façon à avoir une mémoire distribuée.

Une question se pose : « Doit-on laisser l'OS gérer le parallélisme dans le nœud ou faut-il le gérer explicitement ? »

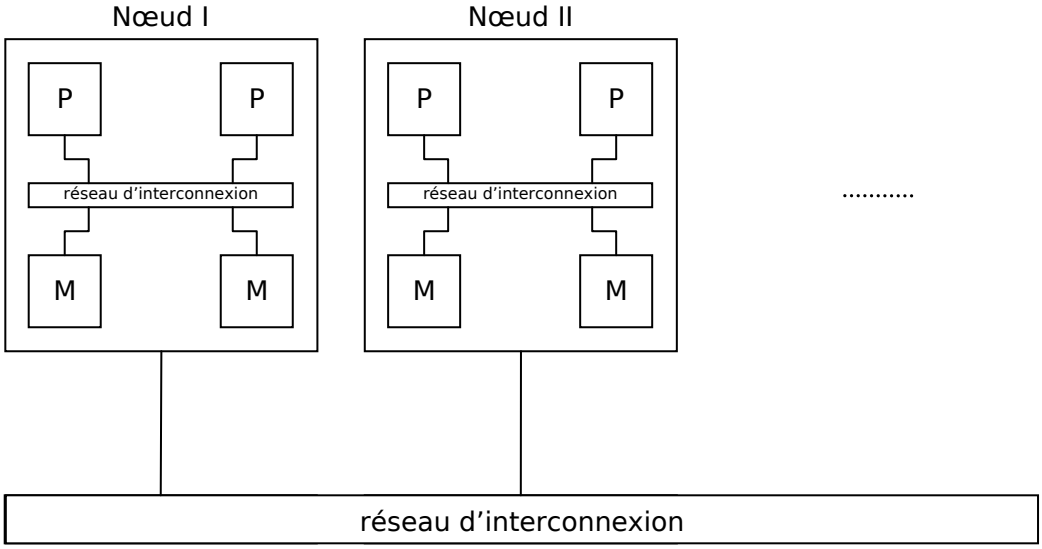


Fig. X – Architecture hybride.

Remarque Le parallélisme est implanté à tous les niveaux : inter processeurs, mais aussi à l'intérieur des processeurs modernes.

Définition *Processeur superscalaire*

Un processeur superscalaire (comme tous le processeurs actuels) peut exécuter plusieurs instructions à la fois (de 2 à 4). Le programmeur n'a que peu d'accès à ce niveau de parallélisme.

1.3 Exploiter le parallélisme

On va illustrer plusieurs modes de collaboration à l'aide d'un problème de la vie de tous les jours.

La famille Dupont (Madame, Monsieur et les deux enfants qui se nomment Pierre et Jean) doit confectionner un grand nombre de tartines pour un anniversaire. Comment faire contribuer chacun à la tâche ?

1.3.1 1ère stratégie : stratégie séquentielle

Ici on suppose que Mme Dupont fait tout le travail :

1. Couper le pain.
2. Beurrer les tranches,
3. Mettre la confiture.
4. Disposer les tartines sur un plat.

Avantage

- Mme Dupont fait sans doute cela très bien et très vite (processeur séquentiel haut de gamme) et ne veut pas s'encombrer de mains inutiles qui demanderaient une organisation coûteuse.

Désavantage

- S'il y a trop de tartines à faire, Mme Dupont seule pourrait ne pas y arriver dans le temps voulu.

Le parallélisme nécessite une gestion et n'est pas forcément payant si on a une solution séquentielle rapide et un problème pas trop gros.

1.3.2 2ème stratégie : le pipeline

On peut utiliser le principe du travail à la chaîne pour augmenter la production de tartines et faire travailler tout le monde.

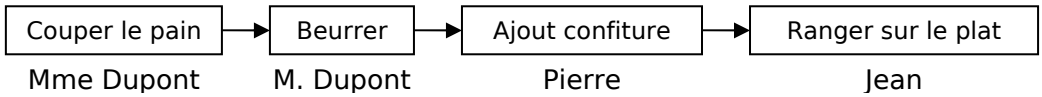


Fig. XI – Stratégie de pipeline

C'est une organisation naturelle.

Avantages

- Performance : en séquentiel, le temps pour confectionner une tartine est égal à la somme des temps de chacune des quatre étapes (car chacune se fait l'une après l'autre).
  Mais en pipeline, une fois la première tranche chez M. Dupont, Mme. Dupont peut déjà couper la prochaine tranche et ainsi de suite…
  Après remplissage du pipeline, il sort de cette chaîne une tartine dans le temps d'une étape, c'est-à-dire qu'il sort quatre fois plus de tartines par unité de temps (accélération d'un facteur quatre).
- Chacun a une tâche spécialisée et peut donc la faire rapidement.
- Les communications se réduisent à se passer les tranches (les données).
- Le modèle est simple et propice à une automatisation.

Désavantages

- Ce n'est pas garanti que le problème global se découpe en quatre morceaux de temps égaux. Si une étape est plus longue que les autres, les autres doivent attendre. Il y a une adéquation entre le problème à résoudre et la façon dont on va le résoudre.
- Ici il n'y a pas de place pour une cinquième personne. Si on doit aller plus vite qu'un facteur quatre, la seule solution serait de créer une deuxième chaîne de quatre personnes.
  Il faut donc une adéquation entre le nombre de tâches et le nombre de processeurs.
- Ce mode est efficace seulement si le nombre de tartines est largement supérieur au nombre d'étages du pipeline (*overhead* au chargement et au déchargement du pipeline).

1.3.3 3ème stratégie : SIMD

Mode militariste ou « chef d'orchestre ».
Madame Dupont est la chef des opérations et indique à chacun ce qu'il doit faire à chaque instant.

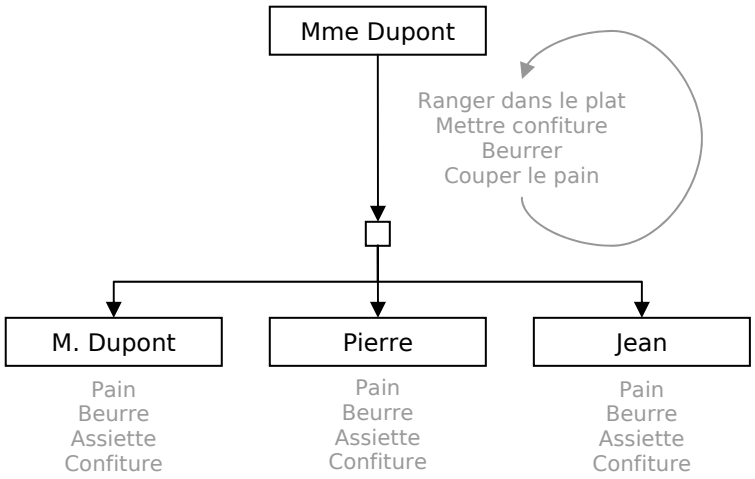


Fig. XII – Stratégie SIMD

Avantages

- Dans le temps mis par une personne pour faire une tartine, on produit autant de tartines qu'il y a de travailleurs.
- Ici, on a un gain d'un facteur trois, mais on a aussi de la place pour autant de travailleurs qu'il y a de tartines à faire.
- Chaque tâche du problème (couper, beurrer, …) peut avoir une durée différente sans perturber la production car chacun en est un même stade au même moment.
- Finalement, on peut facilement produire des tartines avec des pains différents et/ou du fromage au lieu de la confiture.

Désavantages

- On perd un processeur (Mme. Dupont) en tant que travailleur.
- Rigidité du modèle (synchronisme au niveau des instructions) pas toujours compatible avec tous les problèmes.

1.3.4 4ème stratégie : MIMD

On a une souplesse totale d'organisation du travail comme on le veut. Le modèle le plus naturel de collaboration serait :

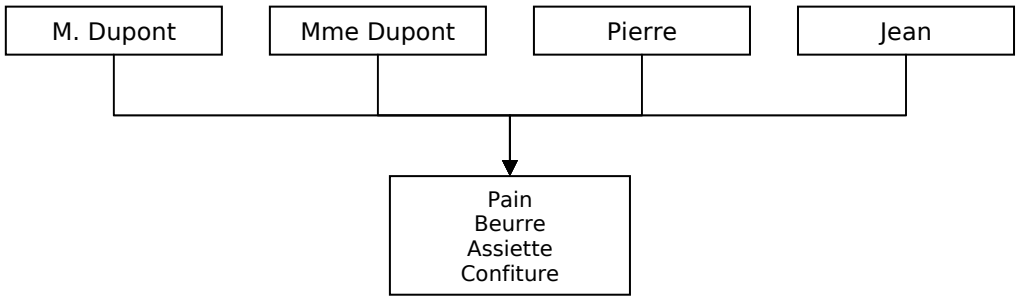


Fig. XIII – Stratégie MIMD à mémoire partagée

Ici chacun fait la même chose mais pas au même moment.

On parle d'une exécution SPMD (Single Program, Multiple Data). C'est un principe plus général que le SIMD. Ici nous avons un modèle à mémoire partagée (Fig. XIII), mais on peut envisager une version à mémoire distribuée :

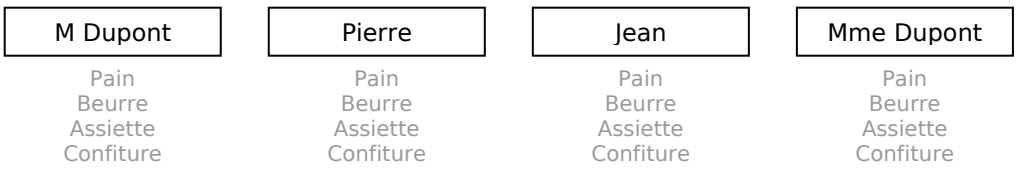


Fig. XIV – Stratégie MIMD à mémoire distribuée

L'approche à mémoire partagée implique des compétitions pour les ressources communes (p.ex. plusieurs personnes voulant la plaque de beurre en même temps). Cela cause donc une perte de performances car tant que la ressource demandée n'est pas disponible on ne fait rien.

Il peut aussi y avoir des situations de *deadlock* (inter blocages).

Exemple
M.Dupont a le couteau et ne le lâche pas tant qu'il n'a pas le beurre.
Mme Dupont a le beurre et ne le lâche pas tant qu'elle n'a pas le couteau.
On est donc dans une situation de *deadlock*.

Exemple
Le premier peut bloquer le deuxième s'il s'arrête avant de tourner car il créera une longue file d'attente.

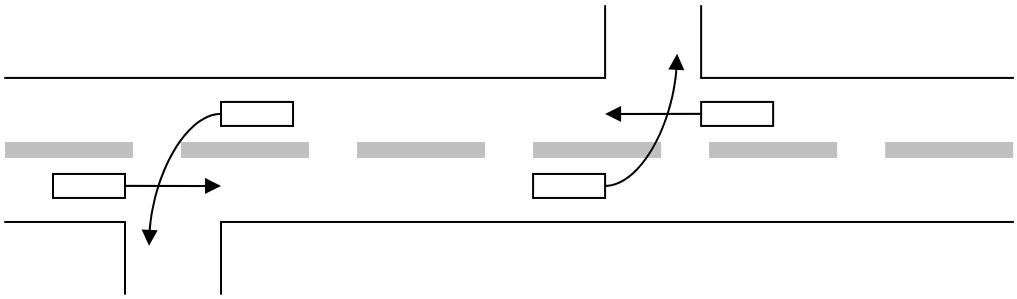


Fig. XV – Situation routière de deadlock

La version à mémoire partagée est naturelle mais va congestionner rapidement s'il y a trop de travailleurs et pas de duplication des ressources. Il y a aussi un problème d'accès physique à la table (mémoire). Nous sommes donc en présence d'une implémentation non scalable (c'est-à-dire non extensible : on ne peut pas agrandir le système tout en augmentant proportionnellement ses performances).

Par contre le modèle à mémoire distribuée, bien que plus artificiel, est plus performant et plus scalable.

Le cas d'un MIMD où chaque processeur exécute un programme différent et où l'ensemble des processeurs résout un même problème n'est pas évident et est très rare.

On distingue deux façons d'exploiter le parallélisme :

1. Parallélisme de contrôle : Il y a plusieurs tâches à effectuer et chacune est associée à un processeur différent.
2. Parallélisme de données : Un même ensemble de tâches est répété sur plusieurs données.

D'expérience on constate que le potentiel du parallélisme de contrôle est faible (2-8 max) alors que le potentiel du parallélisme de données est presque illimité. Même en MIMD on travaille, la plus part du temps, en SPMD.

1.4 Modèle de programmation à mémoire distribuée (échange de messages)

Dans ce modèle, la mémoire est fragmentée et que le programme n'a pas une vision complète de l'espace mémoire.

Les processeurs communiquent par envois explicites de messages contenant des données calculées par certains et utiles à d'autres. Les messages contiennent en général des données.

A la base, un système d'échange de messages demande :

- Un schéma d'adressage qui repère les processeurs les uns par rapport aux autres.
- Deux primitives de base (`SEND`/`RECEIVE`).

Maintenant les échanges de messages sont disponibles au travers de bibliothèques (MPI, PVM) et d'un langage de programmation classique tel que Fortran 90, C, C++, Java, ...

Il y eu d'autres approches avec des langages dédiés comme OCCAM (développé sur une série de machines parallèles), cependant ce langage n'était pas très intuitif.

Définition *PVM : Parallel Virtual Machine*

Libraire d'échange de messages qui ne suppose rien sur l'homogénéité des architectures connectées et permet une gestion dynamique des processus : créer un nouveau processus sur une nouvelle machine, ou en détruire un. Les machines ne sont pas adressées par des nombres continus entre *0* et *n-1* (où *n* est le nombre de processeurs).

Par opposition, MPI est plus simple à utiliser. Certaines des contraintes de PVM disparaissent car on a des hypothèses plus fortes d'homogénéité sur le hardware et on suppose que la configuration est statique (et stable).

1.4.1 Exemple de type de programme par échanges de messages

Un calcul itératif sur un domaine spatial qui implique la connaissance des sites voisins :

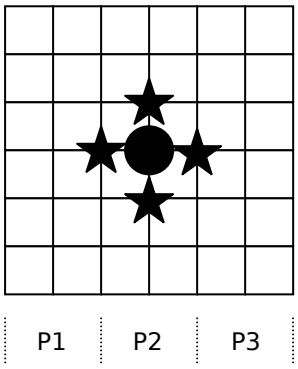


Fig. XVI – Domaine de calcul sous forme de grille

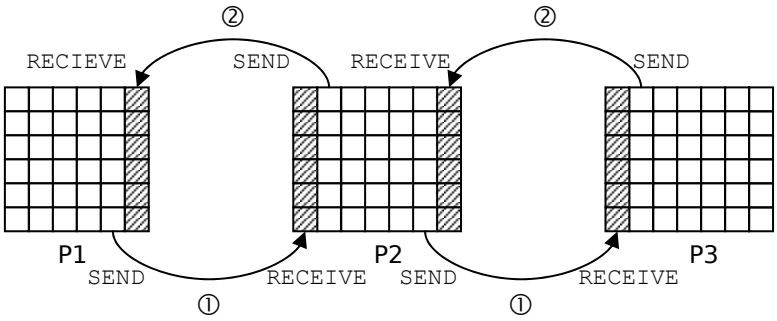
Chaque point est mis à jour en fonction de la valeur courante de ses voisins. Un tel problème se prête bien à un modèle à échanges de messages.

- Il faut d'abord distribuer le domaine complet sur les p processeurs. Chaque processeur contient un sous domaine. C'est sur ce sous domaine que chaque processeur va travailler.
  Chaque processeur a donc moins de travail, on espère donc aller plus vite dans la résolution du problème.

- Les points au bord du sous domaine doivent connaître les valeurs des points dans le sous domaine adjacent afin d'être mis à jour.
  Cela implique la mise en place d'un système de communication.

- Chaque processeur, à chaque itération, envoie ses données de bord aux processeurs voisins, et attend les valeurs de ses voisins.

Exemple



① `send(my_rank + 1, buffer1); receive(my_rank – 1, buffer2);`
② `send(my_rank - 1, buffer1); receive(my_rank + 1, buffer2);`

L'opération qui consiste à distribuer les données dans chaque processeur s'appelle le partitionnement ou la décomposition de domaine. C'est une étape importante pour assurer des performances. Toute décomposition n'implique pas la même quantité de communication. De plus, si les données sont régulières (demande le même travail) il faut les répartir de façon équilibrée.

Dans les décompositions standard, on trouve :

1. Le découpage en tranches :

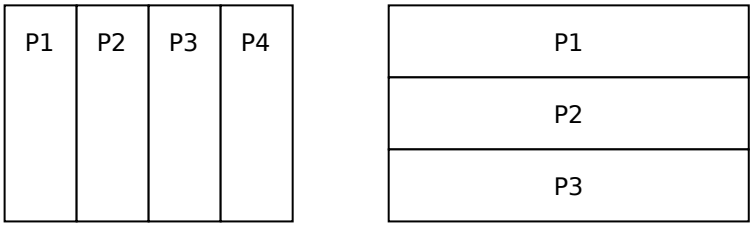


Fig. XVII – Les deux types de découpages en tranches.

2. Le découpage en blocs :

P1	P2	P3	P4
P5	P6	P7	P8

Fig. XVIII – Découpage en blocs

3. Le découpage cyclique :

P1	P2	P1	P2	P3	P1	P2	P1	P2

Fig. XIX – Découpage cyclique 1D

P1	P2	P1	P2
P3	P4	P3	P4
P1	P2	P1	P2
P3	P4	P3	P4

Fig. XIX – Découpage cyclique 2D

Remarque : Le découpage de la *fig. 19* implique beaucoup de communications pour le itératif de la Fig. XVI.

1.4.2 Avantages du modèle à envois de messages

On a un contrôle complet de la mise en œuvre du parallélisme (peu d'interventions du système d'exploitation).
C'est un modèle bien défini et clair, nécessitant un matériel assez simple. Il en résulte une grande efficacité d'exécution pour un bon rapport qualité / prix. Ce modèle permet la scalabilité (utilisation de beaucoup de processeurs ; > 1000 processeurs). De plus le modèle à échanges de messages assure une coordination naturelle des processeurs (auto synchronisation)

- Chaque ressource est privée et les processeurs ne sont pas en compétition pour y accéder.
- La causalité respectée : on ne peut recevoir un message avant qu'il n'est été envoyé.
- Partitionnement des données du problème sur les processeurs qui en ont le plus besoin.

Ce sont des questions que l'on doit se poser dans le cas de la programmation de machines à mémoire partagée.

1.4.3 Désavantages du modèle à envois de messages

- La mémoire est fragmentée et on n'a pas de vision complète des données. Ceci implique un autre modèle de pensée que le séquentiel.

- Gestion explicite du parallélisme, partage des données, flux d'échange. Cela implique de la programmation supplémentaire, donc plus de risques d'erreurs.

- Gérer correctement tous les envois de messages : ne pas attendre un message que ne sera jamais envoyé ; il y a alors un risque de *deadlock*.

- Modèle plus contraignant pour le programmeur (même s'il l'est moins pour le hardware) et le portage d'un code séquentiel sur une machine parallèle engendre une re-conception profonde.

1.5 Communications collectives

Une biblio d'échanges de messages offre plus que des `SEND/RECEIVE`. Il y a aussi des communications plus élaborées à disposition : ce sont des communications qui impliquent un ensemble de processeurs (voire la totalité). On parle de communications collectives et elles reflètent des besoins algorithmiques identifiés.

NB Si le réseau d'interconnexion le permet, ces communications peuvent être efficacement implémentées.
Mais sur des beowulf, on s'attend plutôt à des implémentations naïves.

Exemple
Le broadcast (diffusion) est une communication collective où une donnée dans un processeur racine et envoyée à tous les autres (typiquement un broadcast peut se faire en $log_2(p)$ étapes sur p processeurs. Mais il est à craindre qu'il soit implémenté en p étapes).
Si $p = 1024$, on compare 1024 à $log_2(1024) = 10$.

Définition *La réduction*
Tous les processeurs envoient à un seul processeur leurs données. Ces dernières sont combinées par une opération arithmétique ou logique.

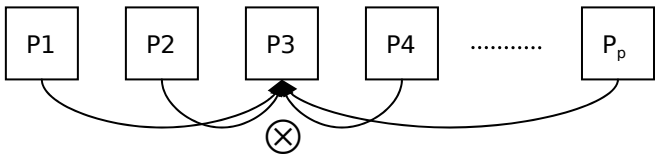


Fig. XX – La réduction – ⊗ est une opération associative quelconque

1.5.1 Primitives de communication

broadcast, reduce , ...(communications collectives)

1.5.1.1 Communication point à point

Les communications point à point (*one-to-one*) : nom générique des `SEND/RECEIVE`.
Ceci se généralise à un échange du type

$$P_i \rightarrow P_{f(i)}$$

où *f(i)* est une permutation de l'ensemble des indices des processeurs.

1.5.1.2 Communication one-to-all

One-to-all personnalisé : ou `SCATTER` en MPI.

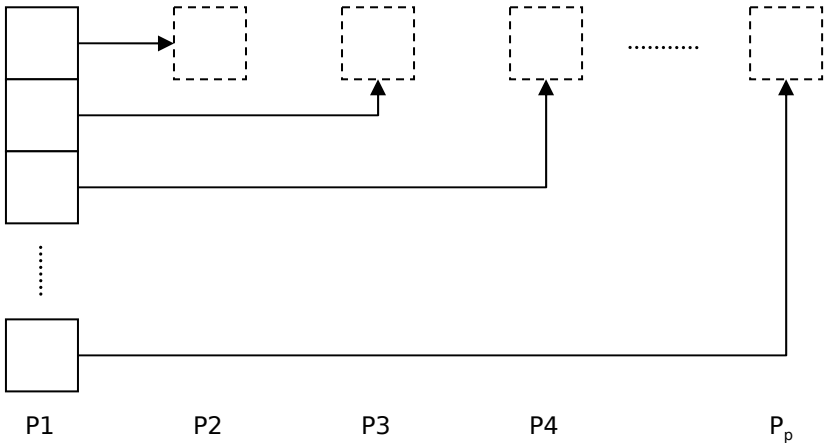


Fig. XXI – Mouvement de données en communication one-to-all.

Le contraire se nomme `GATHER` qui, par opposition au `REDUCE`, ne combine pas les messages mais les stocke.

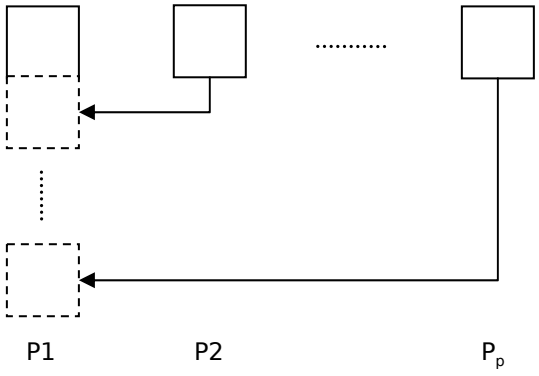


Fig. XXII – Mouvement de données avec GATHER.

1.5.1.3 Communication all-to-all (échange total)

Chaque processeur envoie un message à tous les autres. Cela revient à avoir chaque processeur faisant un broadcast.

1.5.1.4 Communication all-to-all personnalisé

Chaque processeur envoie un message différent à tous les autres (`multiple GATHER`).

Une implémentation efficace et optimale de ces primitives est un problème d'algorithmique et la solution dépend en général de la topologie du réseau d'interconnexion.

1.5.2 Les communications dites « parallel-prefix » ou « scan »

Se sont des communications collectives qui combinent les échanges avec des calculs.

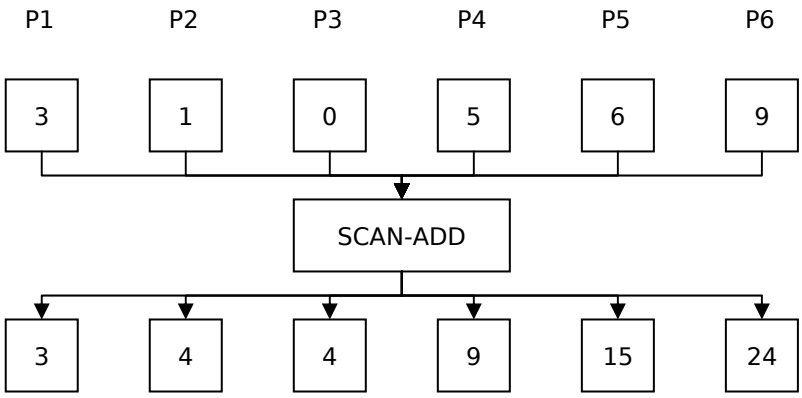


Fig. XXIII – Exemple de SCAN-ADD

Les processeurs P_i reçoivent la somme (ou une autre opération) des valeurs contenues dans les processeurs P_j où $j \leq i$.

1.5.3 Schéma d'abstraction d'un système d'échange de messages

C'est un modèle d'implémentation mais qui ne correspond pas forcément à la façon dont MPI, ou un autre système, est effectivement implémenté.

Pour un `SEND`, le processeur écrit dans le *buffer* de communication *out.* Il y a autant de slot dans ce buffer que d'adresses de destination. De même, un `RECEIVE` consiste à lire le *buffer in* à l'adresse correspondant à l'expéditeur attendu. En cas de "wildcard communication" on peut passer tout le *buffer in* en revue.
Chaque position de ces buffers est une pile fifo. Ils sont soit de la mémoire allouée spécialement par MPI, soit les propres buffers de l'utilisateur (attention alors de ne pas les réutiliser trop vite).

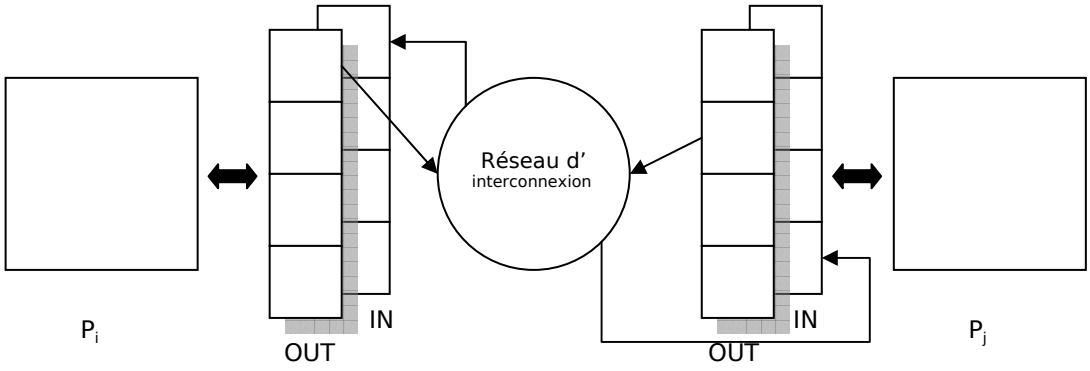


Fig. XXIV – Schéma d'abstraction – IN et OUT sont les buffers de communication.

Pour faire coexister les différentes primitives de communication (broadcast, point-à-point, ...) on peut imaginer qu'il y ait des buffers de communications associés à chaque primitive.

La structure présentée sur le schéma permet de dissocier la partie processeur de la partie réseau. On a un processus spécial MPI qui se charge de transmettre les messages du *buffer out* de P_i dans le *buffer in* de P_j, en concurrence avec le programme utilisateur.

1.6 Le modèle de programmation à mémoire partagée

1.6.1 Présentation du modèle

On a un espace mémoire unique et uniforme, accessible également par tous les processeurs

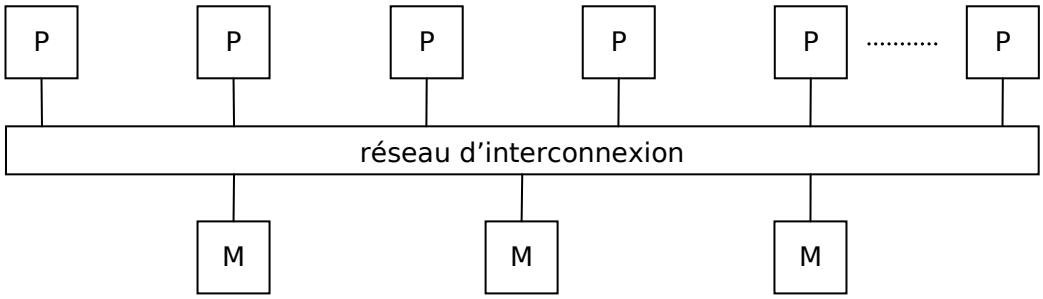


Fig. XXV – MIMD à mémoire partagée

NB Il peut y avoir des conflits d'accès au même banc mémoire ou des congestions sur le réseau.

Le parallélisme dans une telle architecture se met en oeuvre avec le concept de microtasking ou multithreading.
Un processeur racine a un masterthread et peut créer des threads parallèles sur chacun des processeurs du système.

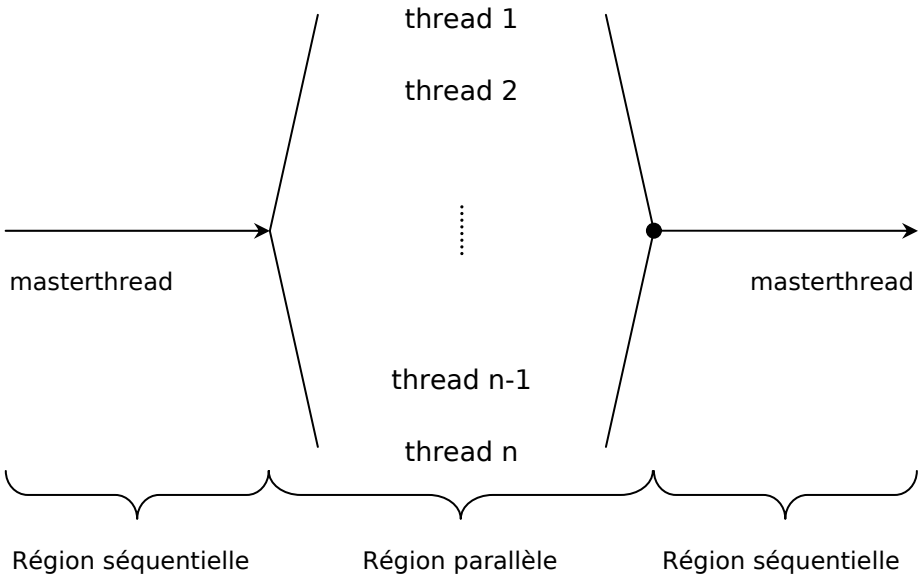


Fig. XXVI – Modèle de programmation à mémoire partagée.

Région séquentielle Qu'un seul processus sur un seul processeur.

Région parallèle Plusieurs threads qui partagent l'état masterthread et qui s'exécutent sur les processeurs de la machine.

On utilise des threads (processus légers dont la création/destruction est rapide) car chaque thread réalise du parallélisme à grain fin (au niveau des données).
Une version macrotasking avec création du processus UNIX (`fork()`) est acceptable seulement par du parallélisme à gros grain.

En pratique, dans un multiprocesseur, les threads parallèles sont mis en oeuvre par OpenMP qui est un standard pour la programmation des mémoires partagées. On programme soit en C, C++, Fortran 90 et on appelle des primitives OpenMP pour gérer les threads. Mais OpenMP offre davantage de souplesse et donne un niveau d'abstraction plus élevé. Le parallélisme est indiqué au compilateur par des directives de compilation.

Exemple
Syntaxe Fortran

```
1.  !$omp parallel do
2.    do i=1,n
3.          z(i) = a * x(i) + y(i)
4.    endo
5.  !$omp end parallel do
```

Ceci crée plusieurs threads en parallèle qui se repartissent les indices de la boucle.

Encore plus simple :

```
1.  !$omp parallel
2.    print *, comp_get_thread_name();
3.  !$omp end parallel
```

En C, les directives s'appellent des pragmas : `#pragma...`

Le nombre de threads qui sont effectivement crées dépend de l'implémentation. OpenMP permet de forcer ce nombre ou de travailler en "threads dynamiques" dont le nombre est choisi à l'exécution par l'OS.

1.6.2 Coordination et synchronisation

Les threads parallèles interagissent entre eux en lisant/écrivant des variables partagées (ou communes). On a donc le risque d'accéder aux mêmes variables de façon incohérente. Il faut introduire de nouvelles primitives de coordination pour gérer cette compétition possible.

Exemple
On a deux threads qui font respectivement $A = A + B$ et $B = B + A$
Les valeurs finales de A et B dépendent crucialement de quel thread passe en premier (*race condition*). Ceci n'est jamais prévisible et on a une exécution non déterministe au niveau de l'avancement de chaque thread.

1.6.2.1 Contrôle d'accès

Garantir que plusieurs processeurs n'accèdent pas en même temps une même variable par des opérations d'écriture.

Exemple
`Global_sum = Global_sum + local_sum` dans une région parallèle est source d'incohérence si `Global_sum` est une variable partagée et `local_sum` une variable privée.

Scénario dangereux :

- P_1 lit `Global_sum` et pendant qu'il lui ajoute `local_sum`, P_2 écrit son propre résultat dans `Global_sum`.
  Quand P_1 écrit son résultat dans `Global_sum`, la contribution de P_2 est perdue.

Comment résoudre le problème ? Il faut assurer qu'un seul processeur à la fois puisse exécuter `Global_sum = Global_sum + local_sum`.
Il faut donc des primitives qui permettent le contrôle de l'accès : ce sont les primitives `lock` et `unlock`.

Donc il faut écrire :

```
1.  Lock(Global_sum);
2.  Glocal_sum = Global_sum + 1ocal_sum;
3.  Unlock(Global_sum);
```

Le premier processeur à atteindre le `lock` va forcer les autres à attendre sur le `lock`, jusqu'à ce que le premier processeur exécute le `unlock`. A ce moment, un deuxième processeur passe le `lock` et, à nouveau, force les suivants à attendre.

Définition *Section critique*
On défini une section critique comme une région du programme où un seul processeur à la fois peut entrer.

On parle aussi d'exclusion mutuelle : un processeur qui passe exclut l'accès aux autres.
Dans OpenMP, l'usage explicite de `lock` et `unlock` est possible, mais on préférera en pratique utiliser la directive/pragma :

```
1.  !$omp critical;
2.  Glocal_sum = Global_sum + 1ocal_sum;
3.  …
4.  !$omp end critical;
```

1.6.2.2 Contrôle de séquence

On ne peut pas prévoir quel thread exécute quelle instruction et à quel moment (thread asynchrone → non déterministe).
Parfois, il est nécessaire de resynchroniser les threads pour éviter des incohérences dans l'exécution.

Illustration
Calcul itératif sur un tableau à partager.

```
b(i) = [a(i-1)+a(i+1)]/2
a(i) = b(i) (retour à ligne 1)
```

En OpenMP :

```
1.  !$omp parallel, private me, p, i1, i2, i;
2.  p = get_number_of_thread();
3.  me = get_thread_id();
4.  i1 = (n/p)*me;
5.  i2 = (n/p)*(me+1);
6.  do k times
7.    do = i1, i2
8.          b(i) = 0.5*[a(i-1)+a(i+1)]/2;
9.    end do
10.   do I = i1, i2
11.         a(i) = b(i);
12.   end do
13. end do
14. !$ end parallel.
```

Si on laisse le programme comme ça, on peut avoir des threads qui sont déjà dans la deuxième boucle et qui modifient `a` avec le calcul précédent. Mais d'autres threads pourraient être encore dans la première boucle et accéder à des valeurs de `a` qui auraient déjà été modifiées. Il faut donc mettre une barrière de synchronisation à la fin de la première boucle :

```
!$omp barrier
```

Tous les processeurs sont bloqués à la barrière jusqu'à ce que le dernier processeur arrive. Ensuite la barrière s'ouvre et tout le monde continue.

1.6.3 Primitives de synchronisation

Les `lock/unlock` et les barrières sont mis en œuvre par des mécanismes hardware plus ou moins sophistiqués selon que l'on ait beaucoup ou peu de processeurs.
L'idée de base est d'emprunter aux systèmes multitâches un sémaphore qui permet de bloquer (ou non) un thread.
Si `S` est un sémaphore (une variable globale) et que `S = 1`, cela permet l'accès à la variable associée à une section critique. Si `S = 0`, l'accès est protégé et il faut attendre.

Une barrière se réalise aussi avec un sémaphore, initialisé à p le nombre de threads à synchroniser. Chaque thread arrivant à la barrière décrémente le sémaphore. Quand ce dernier vaut zéro, la barrière s'ouvre. Pour garantir que le sémaphore est accédé de façon cohérente par les processeurs, il faut du hardware qui permet un accès atomique à ce sémaphore. Le hardware garanti que certaines instructions soient non interruptibles et ne puissent s'exécuter que par un processeur à la fois.

Le plus simple du point de vue hardware est d'avoir un mécanisme qui sérialise les accès à `S` en créant une file d'attente FIFO. Les priorités sont définies par des critères hardware (n° du processeur). Avec cette approche, la simple opération de tester `S` (pour souvent apprendre qu'il faut attendre) nécessite une file et du temps perdu.

La question est de savoir si on peut faire des accès « atomiques en parallèle » pour avoir une primitive non bloquante (qui ne nécessite pas de file d'attente). La primitive `fetch_and_add` a été proposée dans ce but. Elle nécessite un hardware élaboré entre les processeurs et la mémoire.

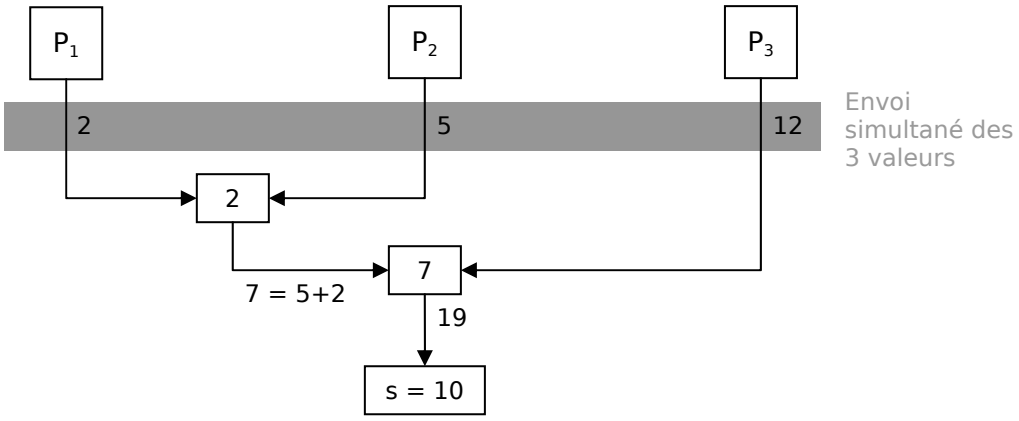


Fig. XXVII – Phase d'écriture du fetch_and_add

On peut décrire ce fonctionnement ainsi :

```
1. fetch_and_add(S, I) //(S = sémaphore global, I = increment
   //local (2, 5, 12 p.ex))
2.    { Tmp = S;
3.    S = S + I; }
4.    Return tmp;
```

`{…}` désigne la partie atomique.

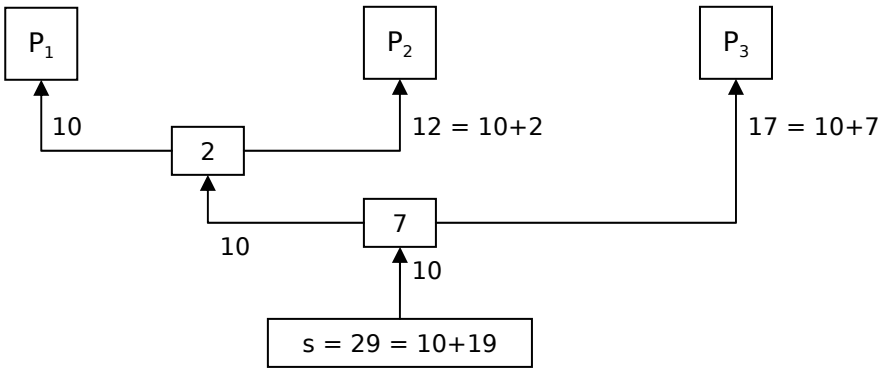


Fig. XXVII – Phase de retour des valeurs du fetch_and_add – Tout se fait en même temps

Dans l'exemple, tout se passe comme si ce `fetch_and_add` avait été exécutée de façon atomique séquentielle avec l'ordre P_1, P_2, P_3. En réalité, l'exécution est simultanée.

1.6.3.1 Exemple d'utilisation

On peut utiliser le `fetch_and_add` pour distribuer automatiquement les itérations d'une boucle sur plusieurs processeurs avec équilibrage de chaque boucle.
Supposons que l'on veuille paralléliser :

```
1.  !omp parallel do
2.    do i = 1, n
3.          compute a(i) ;
4.    enddo
```

Avec le `fetch_and_add`, cela devient :

```
1.  i = 1 //partagé
2.  repeat
3.    j= fetch_and_add(i, 1) ; //j local
4.    if(j > n) exit
5.          compute a(j);
6.  end repeat
```

1.6.3.2 Réalisation d'un sémaphore pour une section critique

Soit S une variable globale initialisée à 1. Un `lock` peut être réalisé ainsi :

```
1.  while(fetch_and_add(S, -1) < 1)
2.    fetch_and_add(S, 1);
3.  end do
4.  section critique
5.  fetch_and_add(S, 1) //unlock
```

Les lignes 1 à 3 correspondent au `lock`.
Le « premier processeur » voit `S = 1` et passe dans la section critique, il laisse `S = 0`. Les autres *p-1* processeurs tournent dans le `while` en faisant varier `S` au pire à *–(p-1)* et au mieux à 0.

C'est seulement quand le premier processeur atteint le `unlock` que `S` a de nouveau la possibilité d'atteindre *1*.

1.6.3.3 Réalisation d'une barrière

`X`, global initialisé à 0.

```
1.  fetch_and_add(X, 1) ;
2.  wait for X = N ;
```

Ces deux lignes de code réalisent une *barrière*. S'il y a plus de threads que de processeurs, il faut avoir un *task switching* afin d'éviter qu'un thread boucle indéfiniment.
Il faudra attendre que les *N* processeurs aient exécuté le `fetch_and_add` pour que le `wait` s'ouvre.

Problème
Comment faire pour réutiliser la même variable `X` pour une prochaine barrière de synchronisation, plus loin dans le code ?

Remettre `X=0` après le `wait` est dangereux. Par exemple :

```
fetch_and_add(X, -1)
```

Le problème est le suivant :

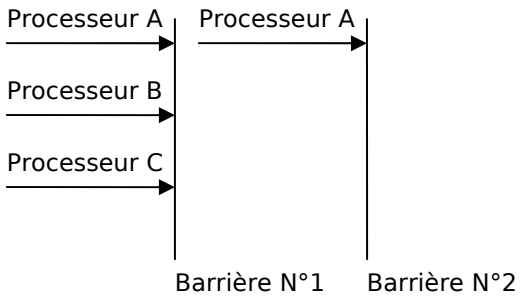


Fig. XXVIII – Situation à deux barrières

Une fois que la première barrière s'ouvre (`X=N`), un processeur peut passer et éventuellement rencontrer la deuxième barrière avant qu'un autre processeur ait pu voir que `X=N`. A ce moment la valeur de `X` sera modifiée et le processeur en retard restera bloqué à la première barrière.

Solution possible
`X=0` pour commencer. `N` est le nombre de processeurs à synchroniser.

```
1.  local_flag = (X < N)
2.  if fetch_and_add(X, 1) = 2N-1 then X = 0
3.  wait for (X < N) != local_flag
```

Première barrière : `X` est initialement nul. `X<N` est vrai, `local_flag` est vrai.

Sur le `fetch_and_add`, `X` augmente de un à chaque passage et à un moment `X` sera égal à `N` (quand tous les processeurs auront atteint à barrière). Mais avant cela, `X<N` est égal à `local_flag` et la barrière est fermée.

A la deuxième barrière, on a `local_flag=faux` mais `local_flag=(X<N)` donc la barrière est fermée en tout cas tant que `X≥N`. Mais `X` repassera à zéro quand exactement `N` processeurs auront fait le `fetch_and_add`. Alors la barrière s'ouvre.

Après deux barrières, on a `X=0` et tout peut remarcher. Les conditions qui ouvrent la barrière sont alternativement `X=N` et `X=2N` (ou `X=0`). Mais principalement ça marche car la première barrière reste ouverte jusqu'à ce que la deuxième s'ouvre à son tour. La deuxième barrière ne s'ouvrira que lorsque les `N` processeurs l'auront atteint et, à ce moment, la première barrière peut bien se fermer.

1.6.3.4 Problème des systèmes multicache

C'est une autre difficulté à résoudre dans les systèmes à mémoire partagée.

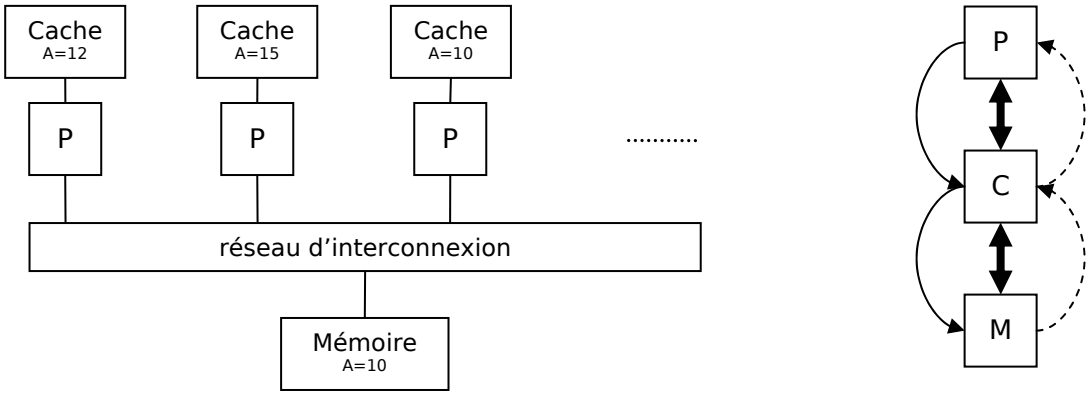


Fig. XXIX – A gauche : système multicache à mémoire partagée. A droite : système classique. – P = Processeur, C = Cache et M = Mémoire centrale.

Le modèle à un processeur est cohérant car dès que l'on efface une donnée du cache, la mémoire centrale est remise à jour avec la valeur du cache.
Dans un système à mémoire partagée, on ne peut pas exclure qu'une même variable existe dans deux caches différents et que les processeurs détenteurs fassent des modifications inconsistantes.
C'est le problème de la cohérence des données dans un système multicache.

On doit donc mettre en place un protocole qui assure la cohérence inter cache (p.ex. protocole MESI).
L'idée est d'associer à chaque donnée un statut :

- `EO` : exclusive owner.
  Le processeur peut en faire ce qu'il veut.
- `C` : Copy
  Le processeur sait que d'autres caches ont aussi une version → opération de lecture.

- `I` : invalid.
  La donnée n'est plus correcte. Faire une update auprès du processeur qui détient la bonne valeur.
- `AO` : atomic owner.
  Comme `EO`, mais en plus aucune copie n'est possible. La ligne du cache est réservée à un seul processeur.

Toute modification de ces statuts doit se faire de manière cohérente en informant les autres processeurs. On peut avoir un modèle de *token* qui circule parmi tous les processeurs pour avertir de chaque changement ou souhait de changement. Ceci se fait en hardware directement.

1.6.3.5 Avantages/Désavantages du modèle à mémoire partagée

Avantages

- Mémoire uniforme.
- D'où une programmation plus proche du modèle séquentiel.
- Portage plus facile du code séquentiel vers le parallèle (grâce à l'abstraction d'OpenMP).
- La duplication des données en mémoire n'est pas nécessaire comme c'est le cas des mémoires distribuées (une seule version du programme principal).

Désavantages

- Simplicité apparente du modèle à cause du modèle de compétition des processeurs.
- Complication du hardware pour garantir la coordination et la cohérence.
- Pas de contrôle complet sur la parallélisation à moins de gérer explicitement les threads et de reproduire le modèle MPI.
  D'ailleurs MPI peut être émulé sur une mémoire partagée avec beaucoup de succès (bonnes performances, souvent meilleures que OpenMP).
- Approche peu scalable. Les performances sont bonnes seulement avec quelques dizaines de processeurs au maximum. C'est lié au fait qu'une mémoire uniforme est incompatible avec des notions de privilèges d'accès à certaines données par certains processeurs. A quoi bon payer pour une grosse machine à mémoire partagée pour l'utiliser en mémoire distribuée.

1.7 Le modèle de programmation « Data Parallel »

1.7.1 Présentation du modèle

Le parallélisme de données exploite le fait que souvent on répète la même opération sur beaucoup de données.
On veut offrir au programmeur un modèle qui reflète cette structure.

Esprit : avoir une abstraction de structures de données parallèles, mais un flot d'instructions séquentielles. Chaque instruction agit sur la totalité de la structure de données, c'est-à-dire

que les éléments sont manipulés en parallèle selon des algorithmes efficaces, transparents à l'utilisateur.

Là-derrière se cache un partitionnement plus ou moins transparent de la structure de données sur les processeurs.

Exemple
$A = B + C$
où A, B et C sont des matrices. Le travail se fait en parallèle sur tous les processeurs.

On peut imaginer qu'il y ait synchronisation après chaque instruction. Le programmeur se concentre sur la logique des transformations et non sur la façon dont elles sont implémentées (*array-based programming* à la MatLab ou Fortran 90).
Il y eu une tentative : HPF (High Performance Fortran). Malheureusement se fut un échec car on a voulu trop en faire et assurer une parallélisation de tout et de rien.

Ce modèle offre un compromis entre la mémoire partagée et distribuée.

Pour le programmeur, on a une vision de données globale (tout un tableau) mais, en hardware, les données sont distribuées et manipulées par échanges de messages entre les proccesseurs.

1.7.1.1 Exemples d'instructions globales sur un tableau

```
cshift(A,dim=1,shift=1)
```

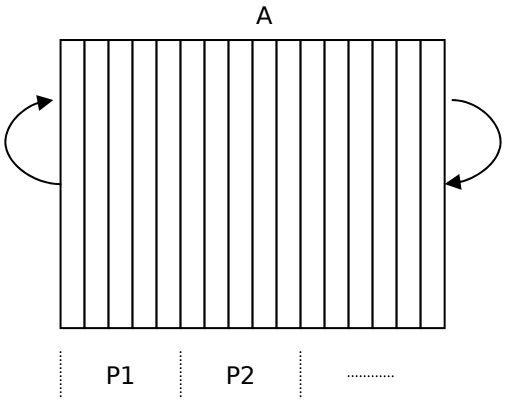


Fig. XXX – Translation d'un élément du tableau vers la gauche.

Par exemple si on veut sommer les valeurs de huit voisins de chaque point de grille on peut écrire :

Soit `a` le tableau originel,

```
1.  b=a+cshift(a, dim=1, shift=1)
2.  b=b+cshift(a, dim=1, shift=-1)
3.  c=b+cshift(b, dim=2, shift=1)
4.  c=c+cshift(c, dim=2, shift=-1)
```

En quatre instructions, on a sommé huit voisins.

Chapitre 2

Performance

2.1 Où gagner des performances ?

Quelles sont les parties d'un système informatique où on peut gagner des performances ?

- Technologie : circuits plus rapides.
- Architecture : parallélisme.
- Algorithmique.

On constate qu'en 20 ans (1970 – 1990) on a un gain de performances de 1000 sur la technologie et l'architecture et de 3'000 sur les algorithme, donc au total un gain de $3 \cdot 10^6$). Les progrès algorithmiques sont donc essentiels.

Aujourd'hui on réussi à casser un code de cryptographie en une année sur 600 machines parallèles. Pour casser le même code, cela aurait pris « l'age de l'univers » il y a une vingtaine d'année. Ceci a été rendu possible grâce à des algorithmes plus évolués.

2.1.1 Technologie

Elle est actuellement basée sur les semi-conducteurs au silicium. Amélioration des performances : augmenter la vitesse de commutation des transistors.

<u>Comment faire ?</u>
Augmenter la fréquence d'exploitation n'est pas simple :

- Problème de chaleur.
- Capacités électriques.
- Fuites…

<u>Solution</u>
Il faut diminuer la finesse du trait pour la gravure du chip.

La loi de réduction stipule que :

> *« Si la finesse diminue d'un facteur S, on peut diminuer la tension d'alimentation du même facteur S et on aura un gain de S^3 dans l'énergie consommée par opération sur le chip. »*

On peut aussi résumer :

$$fréq_{\text{horloge}} \approx \frac{1}{finesseGravure}$$

Mais ce processus a des limites :

- Au-delà d'une certaine taille, des effets quantiques apparaissent (effet tunnel).
- Le nombre d'électrons qui distingue l'état 1 de l'état 0. On ne peut pas aller en dessous de 1.

D'autres facteurs limitatifs

- Chaleur dissipée.
- Interconnexions intra chips.
- Prix.

2.1.1.1 Limites fondamentales de la physique sur les calculs

$E = m \cdot c^2$, constante de Plank, ….
Pour un kilogramme de matière, 5^{50} opérations par secondes et 10^{30} bytes de mémoire sont possibles.

2.1.1.2 Evolution des processeurs Intel

	1971	1990	2002
Processeur	4004	386	Pentium 4
Fréquence	108 kHz	20 MHz	2 GHz
Gravure	10 µm	1 µm	0.13 µm
# de transistors	2'300	885'000	55 millions

Tout croît exponentiellement.

Définition *Loi de Moore*
La puissance augmente d'un facteur deux tous les 18 mois, à coût égal.

Finesse du trait : gain d'environ un facteur deux tous les 6 ans.
Taille des chip : +15% par année.

On sera bientôt aux limites de la technologie du silicium.

2.1.2 Architecture

Mieux structurer et organiser les transformations des signaux pour augmenter le débit de calcul.

- Mémoire cache (c'est un grand progrès architectural).
- RISC (peu d'instructions, mais rapides) – CISC (beaucoup d'instructions, mais lentes).
- Superscalairité (plusieurs instructions par cycle), VILW, pipeline, vectorielle.
- Le parallélisme est la solution architecturale majeurs pour gagner en performance.
- L'accélération des accès mémoire est une autre solution majeure.

« Si on considère le bilan puissance de calcul versus puissance consommée, on voit qu'il est mieux d'avoir beaucoup de processeurs lents qu'un seul processeur rapide. »

R.Feynann – Lecture on computations.

2.2 Travail et Speedup

On veut quantifier le gain du parallélisme et ses limites.

Définition *Le travail*

Le travail est le nombre total d'instructions (ou cycle machine) nécessaire pour calculer un problème.

Supposons que nous ayons un profil d'exécution en parallèle sur $P_{\max}$ processeurs.

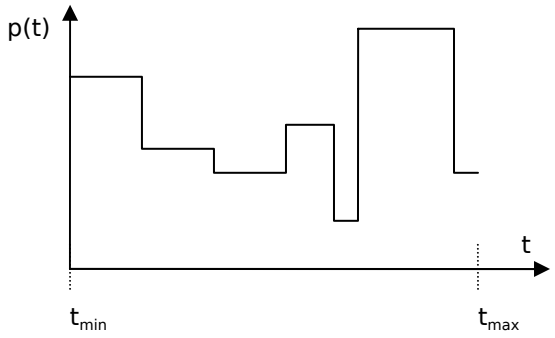


Fig. XXXI – Graphique : Nombre de processeurs utilisés en fonction du temps.

p(t) est le nombre de processeurs utilisés au temps *t*, $1 \le p(t) \le P_{\max}$.
p(t) est le degré de parallélisme.

On définit *R* la puissance d'un processeur (sa vitesse) comme le nombre d'instruction qu'on peut effectuer par seconde.

$$R = \frac{travail}{temps} \text{ (p.ex. } R = 1\left[GFlop/s \right]).$$

Le travail *W* réalisé durant l'exécution avec *p(t)* processeurs est :

$$W = \int_{t_{\min}}^{t_{\max}} p(t) \cdot \underbrace{R \cdot dt}_{\substack{\text{nombre} \\ \text{d'instructions} \\ \text{durant dt}}}$$

Quel temps faut-il en séquentiel pour faire le même travail ?

$$T_{seq} = \frac{W}{R}$$

Le Speedup (gain) est défini alors par :

$$S = \frac{T_{seq}}{T_{par}} = \frac{W/R}{t_{\max} - t_{\min}} = \frac{1}{t_{\max} - t_{\min}} \cdot \int_{t_{\min}}^{t_{\max}} p(t) \cdot dt = \text{degré de parallélisme moyen.}$$

Si on veut un bon Speedup, il faut un bon degré de parallélisme moyen. Si tous les processeurs sont toujours utilisés, on a que $p(t)=P_{\max}$ et $S=P_{\max}$. On espère toujours que $S=P_{\max}$.

Définition *L'efficacité*

Elle est donnée par : E=S/p (Speedup sur le nombre de processeurs). Dans le cas idéal S=p et E =1. E est une mesure, indépendante de p) de la qualité de l'exécution parallèle.

2.2.1 Problèmes

La définition du Speedup demande que T_{seq} soit le temps du meilleur algorithme séquentiel (et non l'algorithme parallèle) connu qui résout le problème sur un processeur. Souvent, on s'aperçoit que W_{par} (travail nécessaire prou résoudre le problème en parallèle) est supérieur à W_{seq} (travail pour résoudre le même problème en séquentiel)) car en parallèle, il y a un *overhead* (W_{par} - W_{seq}) dû à :

- Communication
- Synchronisation.
- Algorithmes.

2.2.2 Loi d'Amdahl

Cas particulier du Speedup = degré moyen de parallélisme.
Le travail *W est donné*. On suppose qu'une fraction α de ce travail ne peut se faire qu'en séquentiel ($p(t)=1$) et que le reste $(1-\alpha)\cdot W$ s'exécute sur p processeurs.

$$T_{seq} = \frac{W}{R},\ T_{par} = \underbrace{\frac{\alpha \cdot W}{R}}_{\substack{\text{partie} \\ \text{séquentielle}}} + \underbrace{\frac{(1-\alpha)\cdot W}{p \cdot R}}_{\substack{\text{partie} \\ \text{parallèle}}}$$

On peut chercher alors le Speedup :

$$S = \frac{T_{seq}}{T_{par}} = \frac{\frac{W}{R}}{\alpha \cdot \frac{W}{R} + \frac{1-\alpha}{p} \cdot \frac{W}{R}} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha} \quad \forall p$$

Exemple
Si $\alpha = 10\%$, alors Amdahl dit que $S \leq 10$.
Ceci donne une vision pessimiste des possibilités du parallélisme.

Plus précisément, si $\alpha=0.1$, $p=1000$, alors $S=9.9$ donc $E=10/1000=0.01$.

2.2.3 Loi de Gustafson

Offre une vision optimiste des possibilités du parallélisme. Les hypothèses sont différentes :

- On se donne T le temps d'exécution en parallèle et on suppose que durant un temps $\beta \cdot T$ l'exécution est purement séquentielle tandis que pendant $(1-\beta)\cdot T$ on est en parallèle sur p processeurs.

Que vaut le Speedup ?

$$T_{par}=T$$

$$T_{seq}= ?$$

$$W = \underbrace{\beta \cdot T \cdot R}_{\substack{\text{1 seul proc} \\ \text{au travail}}} + \underbrace{(1-\beta) \cdot T \cdot p \cdot R}_{\text{travail des p processeurs}}$$

Donc

$$T_{seq} = \frac{W}{R}$$

Finalement

$$S = \frac{\beta \cdot T \cdot R + (1-\beta) \cdot p \cdot R \cdot T}{T \cdot R} = \beta + (1-\beta) \cdot p$$

Maintenant S est proportionnel au nombre de processeurs p.
Si $\beta=0.1$ et $p=1000$, alors $S=900$ et donc $E=900/1000=0.9$

2.2.3 Différence entre les deux lois

La différence entre Amdahl et Gustafson est que dans le premier cas le travail est fixe et on veut le paralléliser. Dans le deuxième cas, on augmente le travail avec le nombre de processeurs.

<u>Illustration</u>
Une femme met neuf mois pour avoir un enfant.
Neuf femmes ne peuvent pas faire un enfant en un mois (idée de Amdahl).
Par contre, neuf femmes peuvent avoir neuf enfants en neuf mois (Gustafson).

Donc le parallélisme sera efficace ($E \cong 1$) si la taille du problème croît avec le nombre de processeurs.

2.2.4 Notion d'overhead (excès de travail)

En général, quand on résout un problème en parallèle, on a plus de travail qu'en séquentiel.

$$W_{par} > W_{seq}$$

W_{seq} = travail du meilleur algorithme séquentiel.

En effet, en parallèle il y a de nouvelles tâches à exécuter :

- Temps de communication inter processeurs qui sont absents en séquentiel.
- Tâche de synchronisation entre les processeurs.
- Souvent, l'algorithme parallèle est différent du meilleur algorithme séquentiel.

Donc, en général, on aura toujours

$$S<P \text{ et } E<1.$$

NB $S \geq 1$ car si l'exécution parallèle est plus lente que l'exécution séquentielle, autant n'utiliser qu'un seul processeur : $T_{seq}=T_{par}$ donc $S=1$.

2.2.5 Speedup super linéaire

Dans certains cas, on observe néanmoins des Speedup supérieurs à p : $S>p$

2.2.5.1 Effet de cache

$$R_{par} > R_{seq}$$

où R_{par} = puissance d'un seul processeur de l'exécution parallèle

Pourquoi cela arrive ?
Le problème séquentiel est trop grand pour rentrer dans la mémoire cache, donc on a un défaut de cache et un temps d'exécution supérieur. Par contre en parallèle, il se peut que les sous domaines passent dans les caches de chaque processeur, donc il n'y a pas de défaut de cache ce qui implique une exécution plus rapide.

2.2.5.2 Algorithme de recherche

On a une liste de N éléments que l'on doit parcourir pour trouver un élément de propriété particulière.

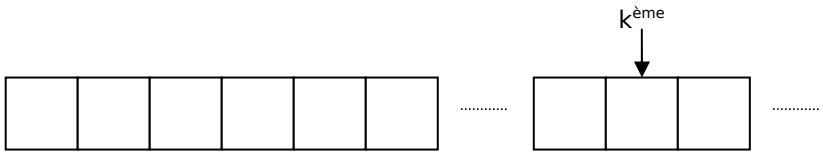


Fig. XXXII – Liste de N éléments

Si l'élément recherché est le k^{ème} élément de la liste, $T_{seq}=k$.
En parallèle, on fractionne la liste en p morceaux et chaque processeur travaille sur N/p données :

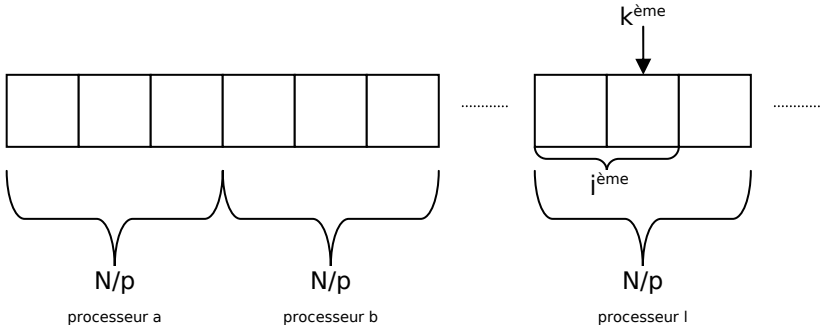


Fig. XXXIII – Liste de N éléments décomposée en p parties

$T_{par} = j$ car l'élément k est en position j dans le processeur l.

$T_{seq} = (l-1)\cdot(N/p) + j$

$$S = \frac{T_{seq}}{T_{par}} = (l-1)\cdot\frac{N}{j\cdot p} + 1$$

Cas très favorable:

- *l-1=p-1* : La donnée cherchée est dans le dernier processeur.
- *j=1* : C'est la première donnée du dernier processeur.
- *S=(p-1)·N/p + 1 ~N*

Si *N>>p*, ce qui est normal, *S>>p* → super linéaire.

Cas très défavorable pour le parallélisme :

- *l=1, j=1*
- *Speedup = 1*

→ Aucun gain.

Exercice

A-t-on un Speedup moyen si on regarde toutes les positions possibles de l'élément cherché ?

NB On peut évidemment modifier le parcours séquentiel pour reproduire le parcours parallèle

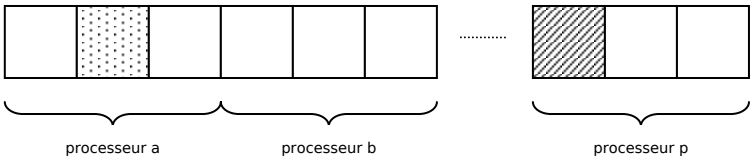


Fig. XXXIV – Liste de N éléments décomposée en p parties

$T_{seq}=p$ et $T_{par}=1$ impliquent que $S=p$

Ce parcours est par contre très peu efficace si l'élément cherché est en position *k=2* (case pointillée). Si l'élément recherché est la case hachurée, alors la recherche parallèle est

beaucoup plus efficace que la recherche séquentielle. Donc les Speedup super linéaires se trouvent fréquemment dans les problèmes de recherche non déterministes.

2.2.6 Remarque sur le Speedup avec des processeurs hétérogènes

On a p processeurs de puissances différentes R_i avec $i=1...p$
Peut-on définir la qualité d'une exécution parallèle. Ici, le T_{seq} est mal définit car on ne sait pas quel processeur prendre.

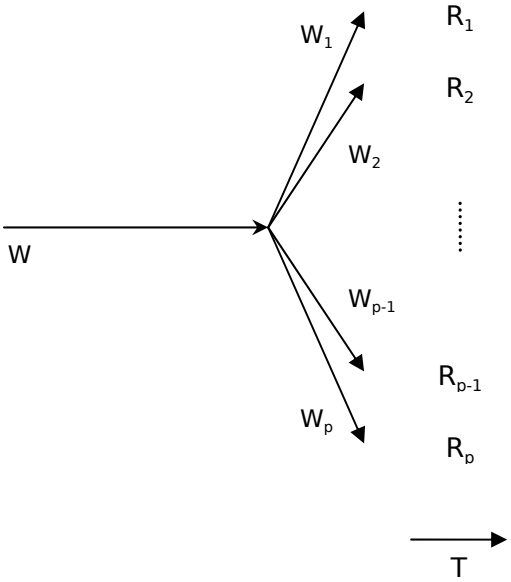


Fig. XXXV – Répartition du travail sur des processeurs de différentes puissances.

Quelles sont les propriétés de l'exécution la plus efficace ?
Il faut choisir W_i tel que T_i (temps d'exécution) soit constant pour tout i. Sinon on peut toujours gagner en déchargeant le processeur le plus lent vers le plus rapide. On veut :

$$T_i = T_{ideal}$$

$$T = \frac{W_i}{R_i} \rightarrow W_i = R_i \cdot T_{ideal}$$

Dans cette dernière équation, on a, par analogie avec l'électronique :

- W_i : le courant électrique.
- T_{ideal} : la tension, la même pour tous les éléments en parallèle. R_i devient l'inverse de la résistance.

$$\sum W_i = W \rightarrow W = \sum R_i \cdot T_{ideal} \rightarrow T_{ideal} = \frac{W}{\sum R_i}$$

On peut définir l'efficacité de l'exécution en comparant T_{par} avec T_{ideal}.

$$E = \frac{T_{ideal}}{T_{par}} = \frac{W}{T_{par} \cdot \sum R_i}$$

Si on suppose que $\Sigma R_i = p \cdot R_{moyen}$ alors

$$E = \frac{W}{T_{par} \cdot p \cdot R_{moyen}} = \frac{T_{seq}}{T_{par} \cdot p} = \frac{S}{p}$$

Donc notre définition de *E* est compatible avec la précédente pour une exécution séquentielle sur un processeur de puissance R_{moyen}. Si $R_{moyen}=R_i$ quelque soit *i*, ça marche aussi.

Définition *L'overhead*

On défini l'overhead ΔW comme $\Delta W = W_{par}-W_{seq}$ d'où

$$\frac{\Delta W}{W_{par}} = 1 - \frac{W_{seq}}{W_{par}} = 1 - \frac{T_{seq} \cdot R}{p \cdot T_{par} \cdot R} = 1 - E$$

Donc ce qui manque à *E* pour faire *1* est directement lié à l'overhead.

2.3 Modèle de performances

Ici, on cherche des expressions algébriques pour $T_{par}(n, p)$ où n est la taille du problème (le nombre de données à traiter) et *p* le nombre de processeurs, et ceci pour différents algorithmes où l'overhead peut être exprimer en détail.

Remarque Le temps de communication inter processeurs peut s'exprimer par la relation :

$$T_{com} = t_s + L \cdot C$$

où t_s est un temps de latence (ou encore de startup time), *L* est la longueur du message (en byte) et *C* est le temps de communication par byte qui est égal à l'inverse de la bande passante (débit) du lien.

2.3.1 Somme de *n* valeurs sur *p* processeurs

Tout d'abord on prend *n=p*. On suppose une architecture à mémoire distribuée, une valeur par processeur. On veut obtenir la somme des *n=p* processeurs dans le dernier processeur. De plus on suppose que le nombre de processeurs est une puissance de 2.

Exemple

Il faut trois étapes de calcul pour huit processeurs. En général il faut $log_2(p)$. Chaque étape consiste en une phase de communication et en une phase d'addition. Tous les processeurs impliqués calculent en même temps et communiquent en même temps.

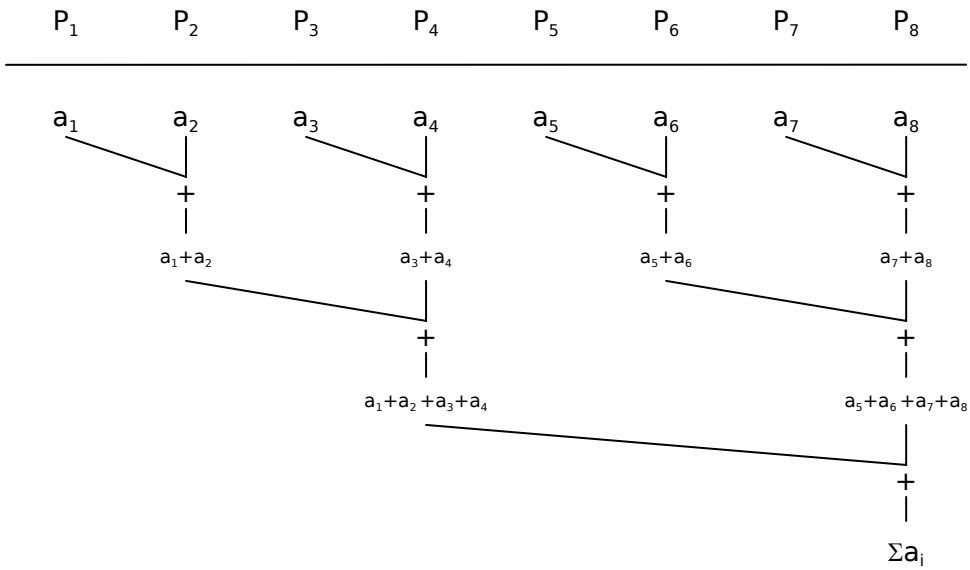


Fig. XXXVI – Somme de huit valeurs sur huit processeurs – méthode 1.

Le temps d'une étape est $C+T$ où C est le temps de communication (t_s+C_l) et T le temps d'addition.

$$T_{par} = (C+T)\cdot \underbrace{\log_2(p)}_{\text{nb d'étapes}}$$

$T_{seq} = (p-1)\cdot T$: on a p valeurs à additionner.

$$S = \frac{T_{seq}}{T_{par}} = \frac{(p-1)\cdot T}{(C+T)\cdot \log_2(p)} = \frac{p-1}{(1+\frac{C}{T})\cdot \log_2(p)} \approx \frac{p}{(1+\frac{C}{T})\cdot \log_2(p)}$$

Ici, on a deux contributions qui font que $S<p$: $C/T\neq 0$ car le temps de communication n'est pas négligeable de plus $log_2(p)\neq 1$. Ce terme reflète le fait que l'algorithme parallèle est différent de l'algorithme séquentiel optimal.

2.3.1.1 Cas ou $n>>p$

On suppose que chaque processeur contient n/p valeurs (p divise n).

On peut utiliser l'algorithme précédent sur chacune des n/p « couches » de valeurs à travers les processeurs. Chaque couche prend un temps $(C+T)\ log_2 p$. On répète cela n/p fois et le dernier processeur doit encore faire la somme des n/p résultats.

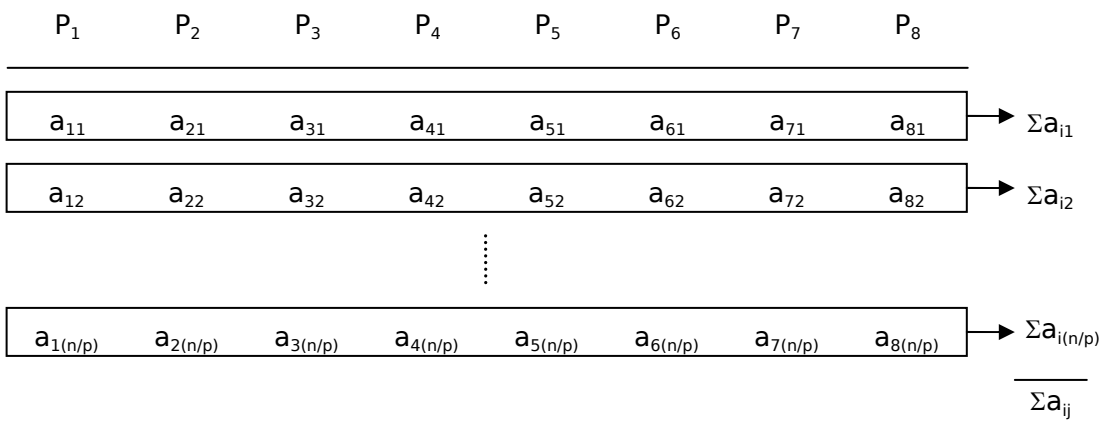


Fig. XXXVII – Somme de huit valeurs sur huit processeurs – méthode 2.

$$T_{par} = \frac{n}{p} \cdot (C+T) \cdot \log_2(p) + (\frac{n}{p} - 1) \cdot T$$

$$T_{seq} = (n-1) \cdot T$$

$$S = \frac{T_{seq}}{T_{par}} = \frac{(n-1) \cdot T}{(\underbrace{\frac{n}{p} - 1}_{\frac{n-p}{p}}) \cdot T + \frac{n}{p} \cdot (C+T) \cdot \log_2(p)} = \frac{p}{\frac{n-p}{n-1} + \frac{n}{n-1} \cdot (1 + \frac{C}{T}) \cdot \log_2(p)}$$

$$\overset{n>>p}{\approx} \frac{p}{1 + (1 + \frac{C}{T}) \cdot \log_2(p)}$$

De nouveau, le Speedup est inférieur à p.

2.3.1.2 Cas ou $n>>p$: meilleur algorithme

Chaque processeur somme ses nombres en local. Cela prend un temps *((n/p) -1)·T* et ensuite on applique l'algorithme pour sommer p valeurs sur p processeurs.

$$T_{par} = (\frac{n}{p} - 1) \cdot T + (C+T) \cdot \log_2(p)$$

$$S = \frac{(n-1) \cdot T}{(\frac{n}{p} - 1) \cdot T + (C+T) \cdot \log_2(p)} = \frac{p}{\frac{n-p}{n-1} + \frac{p}{n-1} \cdot (1 + \frac{C}{T}) \cdot \log_2(p)}$$

$$n>>p \Rightarrow \frac{n-p}{n-1} \cong 1,\ n-1 \cong n \quad \Rightarrow \quad \approx \frac{p}{1+(1+\frac{C}{T})\cdot\frac{p\cdot\log_2(p)}{n}}$$

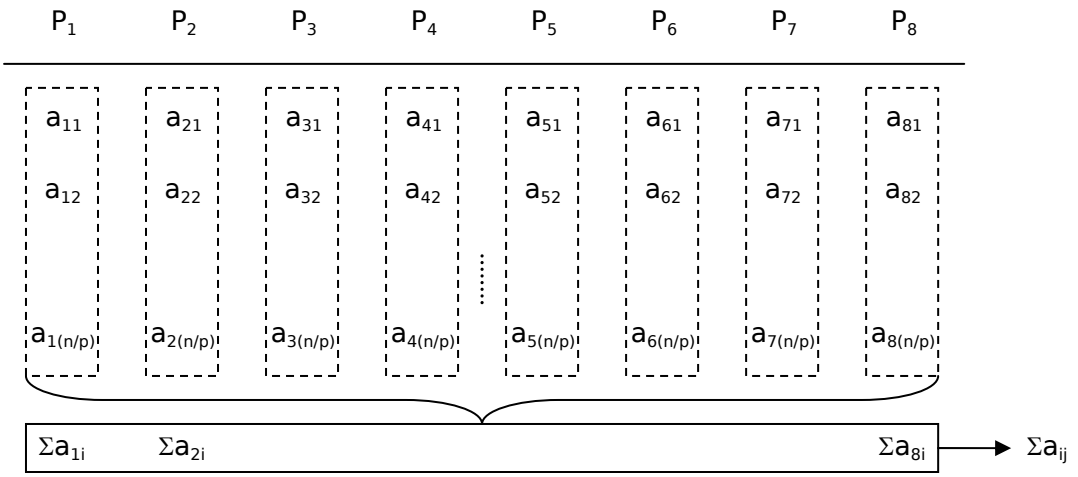
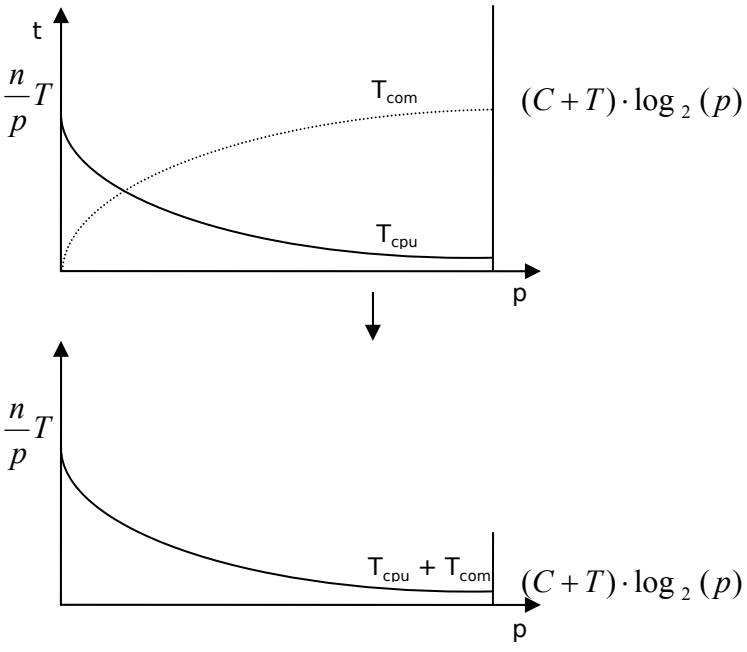


Fig. XXXVIII – Somme de huit valeurs sur huit processeurs – méthode 3.

La taille du problème est encore présente si on choisit n assez grand, on peut rendre le terme d'overhead $(C+T)\cdot log_2(p)/n$ aussi petit qu'on veut et avoir $S \sim p$. Le parallélisme est payant si $n>>p$ (cf. loi de Gustafson).

Le temps T_{par} se compose d'une partie CPU et d'une partie communication.



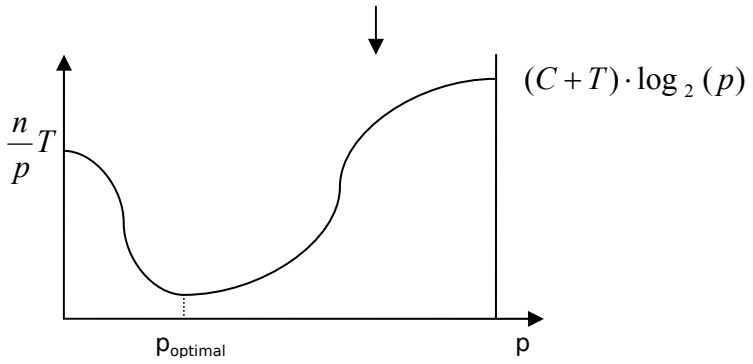


Fig. XXXIX – Graphiques

Sur les graphiques précédents, on défini :

$$T_{cpu} = (\frac{n}{p} - 1)\cdot T = \frac{n}{p}\cdot T \text{ et } T_{com} = (C+T)\cdot\log_2(p)$$

Remarque Si $T_{com}>T_{cpu}$, alors il y a un nombre optimal de processeurs. Si p est trop grand, les performances sont moins bonnes qu'en séquentiel.

Remarque sur la répartition du travail
Que faire si p ne divise pas n ?
$\lfloor n/p \rfloor$ nombres de valeurs par processeurs, et il y a un reste. En général on peut écrire $n=k\cdot p+r$ où $k=\lfloor n/p \rfloor$ et $r=n$ nœuds. Donc on ajoute aux r premiers processeurs une valeur de plus. Ainsi r processeurs ont $\lfloor n/p \rfloor+1=\lceil n/p \rceil$ valeurs et $p-r$ en ont $\lfloor n/p \rfloor$.
Mais ce partage n'est pas forcément le meilleur à cause de la communication qui peut croître avec p.

Exemple
Soit $n=19$ et $p=6$.
Un partage naturel serait : *4 3 3 3 3 3*.
Si l'overhead croit avec p et que la partie cpu est de toute façon limitée par le processeur le plus chargé, il est plus avantageux de distribuer les 19 valeurs sur 5 processeurs : 4 4 4 4 3.

2.3.2 Solution de l'équation de Laplace

On veut résoudre

$$\nabla^2\phi(x,y,z) = 0$$

sur un domaine D inclus dans $\Re^3$.
On discrétise l'espace sur N^3 points de grille et on itère la relation suivante :

$$\phi_{i,j,k}^{(n+1)} = \frac{1}{6}\cdot(\phi_{i-1,j,k}^{(n)} + \phi_{i+1,j,k}^{(n)} + \phi_{i,j+1,k}^{(n)} + \phi_{i,j-1,k}^{(n)} + \phi_{i,j,k+1}^{(n)} + \phi_{i,j,k-1}^{(n)})$$

On veut résoudre cela en parallèle sur p processeurs. Le domaine D est morcelé en p parties. Chaque morceau est un cube contenant l^3 points de grille.

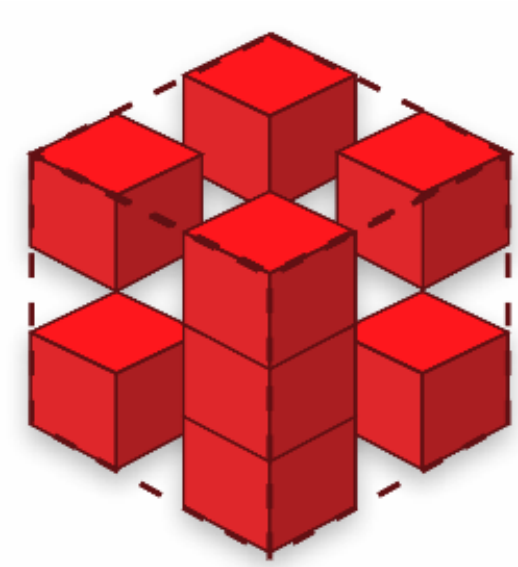


Fig. XXXX – Sous domaine : chaque petit cube rouge représente un sous domaine d'un processeur. Tous les petits cubes rouge sont de dimension l·l·l. L'ensemble des sous domaine crée le grand cube.

On a évidemment $N^3 = p \cdot l^3$.

$$T_{cpu} = 6 \cdot \frac{l^3}{R}$$

où 6 représente le nombre d'opérations par point de grille (cinq `add` et un `mult`). T_{cpu} représente le temps de calcul d'un processeur pour une itération.

$$T_{com} = 6 \cdot l^2 \cdot C$$

où 6 est le nombre de faces du cube, l^2 la taille d'une face et C le temps de communication pour un mot.

$$T_{par} = T_{cpu} + T_{com} = 6 \cdot \frac{l^3}{R} + 6 \cdot l^2 \cdot C$$

Donc si l'overhead est négligeable pour des grands problèmes. Il faut constater que

$$\frac{R \cdot C}{l} = \frac{T_{com}}{T_{cpu}}$$

Plus T_{com}/T_{cpu} est petit, plus on est efficace. Pour réduire T_{com}/T_{cpu} il faut maximiser le rapport surface sur volume du sous domaine (le cube est le meilleur choix ici puisque la sphère est un choix impossible ici (elle possède cependant le meilleur rapport surface/volume)).

Finalement, on obtient un Speedup de :

$$S = \frac{p}{1 + \underbrace{\frac{R \cdot C}{l}}_{\frac{T_{com}}{T_{cpu}}}}$$

Quel est le l qui donne des performances intéressantes ? On va accepter une situation où $T_{com}/T_{cpu} < 1$, c'est-à-dire :

$$\frac{R \cdot C}{l} < 1 \rightarrow l > R \cdot C$$

Supposons que $R=128$ *[Mflops/s]*, $C=10^{-6}$ *[s]* donc $RC=128$. On veut $l>128$. Il faut donc que la mémoire soit assez grande pour accueillir $128 \cdot 128 \cdot 128$ mots de 64 bits ;
Donc *Mémoire* > $128 \cdot 128 \cdot 128 \cdot (8$ *[bytes])* = 16 *[MB]*.
Dans le cas où $R \cdot C = l$, on a $S=p/2$ et l'efficacité est $p/2p = 50\%$.

2.3.2 Scalabilité

Propriété générale des systèmes parallèles. Peut-on augmenter le nombre de processeurs d'une machine parallèle et accroître les performances proportionnellement ?
Cela implique évidemment des contraintes matérielles : mémoire, réseau, entrées-sorties, …

Ici on veut considérer la scalabilité en rapport avec une application donnée. Pour aborder la scalabilité d'application, on défini la notion d'isoefficacité qui décrit comment la taille du problème doit augmenter si le nombre de processeurs augmente tout en gardant l'efficacité constante.

Exemple
On avait (équation de Laplace) :

$$\frac{R \cdot C}{l} < 1 \rightarrow l > R \cdot C \quad \text{où} \quad p \cdot l^3 = N^3 = n \rightarrow l = \sqrt[3]{\frac{n}{p}}$$

Ici, n est la taille du problème (le nombre total de points à traiter).

$$S = \frac{p}{1 + R \cdot C \cdot \sqrt[3]{\frac{p}{n}}}, \quad E = \frac{S}{p} = \frac{1}{1 + R \cdot C \cdot \sqrt[3]{\frac{p}{n}}}$$

On va alors exprimer $n=F(p)$.

$$1 + R \cdot C \cdot \sqrt[3]{\frac{p}{n}} = \frac{1}{E}, \quad R \cdot C \cdot \sqrt[3]{\frac{p}{n}} = \frac{1}{E} - 1 = \frac{1 - E}{E}$$

$$\frac{p}{n} = \frac{1}{(R \cdot C)^3} \cdot \left(\frac{1-E}{E}\right)^3 \rightarrow n = (R \cdot C)^3 \cdot \left(\frac{E}{1-E}\right)^3 \cdot p$$

C'est la fonction d'isoefficacité. Ici, la fonction d'isoefficacité est linéaire : n est proportionnel à p, c'est un cas idéal.

Exemple
Mauvais algorithme de sommation de n valeurs sur p processeurs. On avait :

$$S = \frac{p}{1 + (1 + \frac{C}{T}) \cdot \log_2(p)}$$

Comme n n'apparaît plus dans l'expression de E, il n'y a pas de fonction d'isoefficacité. On ne peut pas augmenter la taille du problème pour augmenter l'efficacité.

Définition *L'isoefficacité*
Une application est scalable sur une architecture parallèle donnée si la fonction d'isoefficacité existe.

L'équation de Laplace est scalable mais le mauvais algorithme de sommation ne l'est pas.

Exemple
Le bon algorithme de sommation.

$$S = \frac{p}{1 + (1 + \frac{C}{T}) \cdot \frac{p}{n} \cdot \log_2(p)}, \quad E = \frac{1}{1 + (1 + \frac{C}{T}) \cdot \frac{p}{n} \cdot \log_2(p)}$$

La fonction d'isoefficacité existe, mais si on veut simplement trouver la dépendance $n \sim p$, il suffit de voir que, pour $E = cte$, il faut que $log(p) \cdot p/n = cte$ donc que $n \sim p \cdot log(p)$.
Ici, la taille du problème augmente un peu plus vite que p. C'est quand même scalable puisque la fonction d'isoefficacité existe.

On parle de

- Scalabilité idéale si $n \sim p$.
- Scalabilité polylogarithmique si $n \sim p \cdot log^k(p)$.
- Scalabilité faible si $n \sim p^k$ où $k > 1$.

2.4 Partitionnement et scheduling

2.4.1 Définitions

Un programme parallèle est constitué d'un ensemble de tâches, certaines de ces tâches peuvent s'exécuter simultanément, et d'autres dans un ordre donné car il peut y avoir des dépendances.

Définition *Une tâche*

Une tâche T est une liste d'instructions réalisant un certain travail et utilisant des données d'entrée dans un ensemble E (ensemble des variables d'input) et qui modifient un ensemble de variables S (variables de sortie).

La taille d'une tâche est une mesure du nombre d'instructions et de la taille des ensembles E et S.

Selon que les tâches soient petites ou grosses, on parlera de granularité fine ou grossière.

Définition *Le partitionnement*

C'est le découpage du problème complet en tâches, avec l'espoir que certaines de ces tâches pourront être résolues en parallèle. Souvent, dans les cas SPMD, le partitionnement revient à découper le domaine de calcul (c'est le E_{tot} qui est fractionné).

On parle de « *domain decomposition* ».

En faisant un partitionnement, on a en principe le choix de la granularité.

Quels sont les enjeux ?
Avec une granularité fine (donc beaucoup de petites tâches), il y a un grand potentiel de parallélisme (augmentation du degré de parallélisme). Mais, en général, plus il y a de tâches, plus il y aura de *l'overhead* de communication ou de coordination. Il y a donc un optimum :

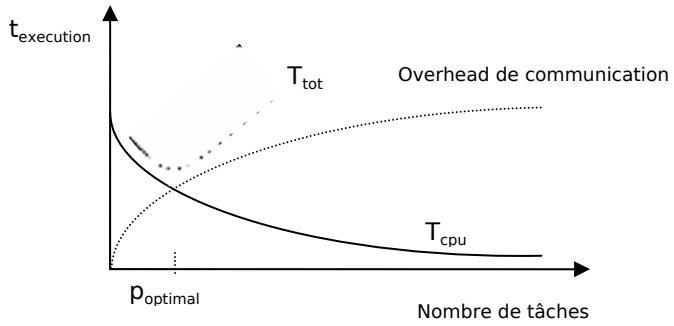


Fig. XXXXI – Détermination de l'optimum – Le nombre de tâches équivaut à (granularité)$^{-1}$ qui équivaut encore au nombre de processeurs.

Le partitionnement n'est, en général, pas une chose facile si on veut garantir une exécution optimale. Heureusement, la nature du problème suggère souvent un bon partitionnement.

Définition *Le placement*

C'est le choix de quel processeur exécutera quelle tâche (le partitionnement étant fait).

Les enjeux sont de garantir un équilibrage de charge optimal, et des communications aussi faibles de possible.

Exemple

On a plusieurs tâches, toutes indépendantes (on peut les exécuter dans l'ordre que l'on veut) et sans communication. Mais elles sont de durées différentes.

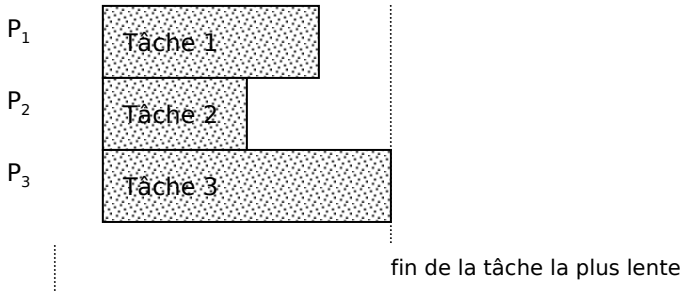


Fig. XXXXII – Répartition de tâches

L'arrangement optimum nécessite des heuristiques et c'est en général un problème (np-) difficile.

Définition *La dépendance*

En général, les différentes tâches issues du partitionnement ne sont pas toutes indépendantes les une des autres.
Deux tâches T_i et T_j sont dépendantes si :

1. $E_i \cap S_j \neq \varnothing$
2. $E_j \cap S_i \neq \varnothing$
3. $S_j \cap S_i \neq \varnothing$

Si des tâches sont indépendantes, on peut les exécuter dans n'importe quel ordre, et en particulier en parallèle.

Si deux tâches sont dépendantes, l'une doit d'exécuter avant l'autre selon les spécifications du programme séquentiel. On dit que $T_i<T_j$ si T_i doit être exécuté avant T_j. Deux tâches T_i et T_j sont consécutives s'il n'existe aucun T_k tel que $T_i<T_k<T_j$.

On peut exprimer ces relations de dépendance par un graphe de précédence (Fig. XXXXIII).

Définition *Le scheduling*

Le scheduling (ordonnancement) est le choix du moment auquel une tâche pourra commencer tout en garantissant que les dépendances ne soient pas violées.

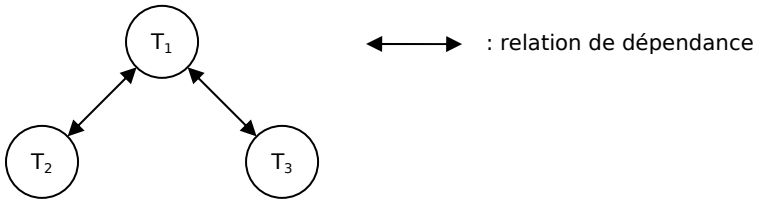


Fig. XXXXIII – Graphe de précédence : T_1 dépend de T_2 et réciproquement. T_1 dépend de T_3 en réciproquement. T_2 et T_3 sont indépendants.

Donc, pour paralléliser une application, il faut la partitionner et ensuite placer et ordonnancer les tâches sur les *p* processeurs. On veut faire ces choix de façon à optimiser les performances (donc diminuer *l'overhead* et diminuer les déséquilibres de charge).

2.4.2 Exemple de placement et ordonnancement avec la méthode des temps au plus tôt et au plus tard

On se donne un graphe de précédence entre plusieurs tâches, ainsi que la durée de chaque tâche.

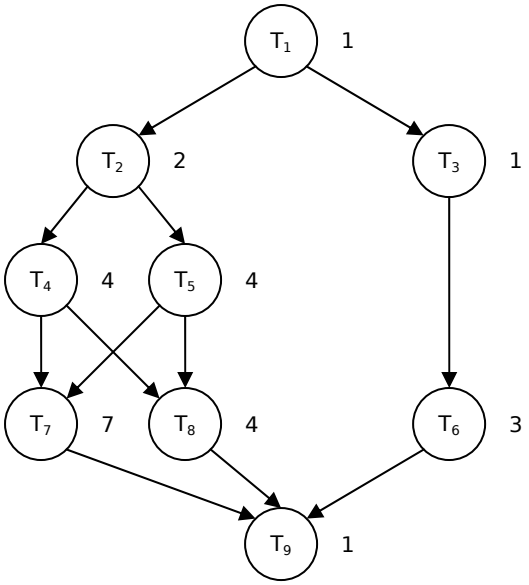


Fig. XXXXIV – Graphe de précédence. Les valeurs à droite des tâches représentent leur durée d'exécution respective.

La méthode d'optimisation que l'on va utiliser détermine une fourchette optimale pour exécuter les tâches, tout en vérifiant les contraintes de dépendance.

Pour remplir le tableau suivant, on prend toujours la contrainte la plus forte quand on a le choix entre plusieurs contraintes.
Les deux colonnes de ce tableau définissent un intervalle de temps pour exécuter chaque tâche et avoir une performance maximale par rapport aux dépendances.

	Tâche	Temps au plus tôt	Temps au plus tard
	1	0 (par définition)	0
Car T_1 aura alors fini →	2	1	1
	3	1	10
	4	3	3
	5	3	3
	6	2	11
	7	7	7
	8	7	10
	9	14*	14**

Les flèches indiquent le sens du remplissage des colonnes.
** On choisi 14 car les contraintes étaient 5, 11 et 14.*
*** On reprend le 14 de la colonne des temps au plus tôt (car on souhaite être optimal).*

Choix d'un intervalle de temps correspondant aux intervalles de temps

temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P_1	T_1	T_2		T_4				T_7							T_9
P_2	-	T_3	-	T_5				T_6			T_8				-

Fig. XXXXV – Répartition des tâches

Si T_6 avait duré plus de 3 [s], il aurait fallu créer un nouveau processeur.

Calcul du Speedup

$$T_{seq} = \sum temps = 27[s]\,,\quad T_{par} = 15[s]\,(\text{temps de fin de } T_9)$$

$$S = \frac{T_{seq}}{T_{par}} = \frac{27}{15} = 1.8$$

$$E = \frac{S}{nbProc} = \frac{1.8}{2} = 0.9$$

P_1 est utilisé à 100% et P_2 est utilisé à 12/15 = 80%. La moyenne des deux donne 90%.

Ici, on voit que deux processeurs suffisent et sont optimums. A cause des dépendances, il n'y a pas plus de parallélisme à exploiter et le Speedup de cette application est de 1.8. Ce que l'on peut encore changer, c'est le partitionnement : par exemple, si T_6 durait 4 [s], on pourrait essayer de la fractionner en sous tâches dans l'espoir de mieux remplir les deux processeurs.

2.4.3 Load balancing (équilibrage de charge)

Un déséquilibrage de charge est souvent la cause de mauvaises performances. Il y a plusieurs stratégies pour rééquilibrer les charges entre les processeurs.

2.4.3.1 Cas statique

Situation où l'on connaît d'avance le temps de chaque tâche. Souvent on doit faire un calcul itératif sur un domaine irrégulier.

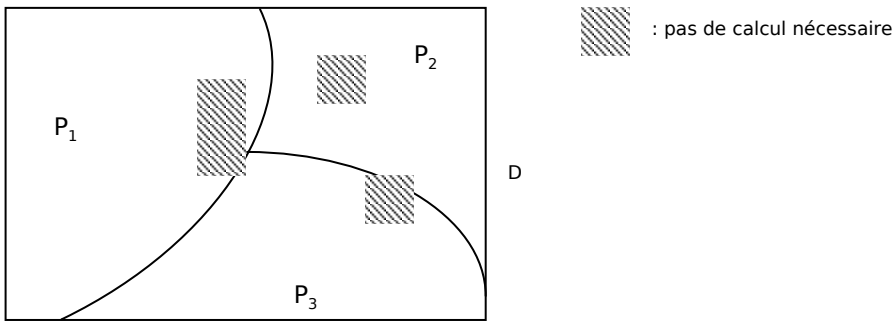


Fig. XXXXVI – Découpage du domaine D

On a fait un découpage qui donne le même nombre de points de gille à tous les processeurs et qui minimise la longueur de la frontière (et donc des communications).

Ce problème correspond au partitionnement de graphes (voir la librairie Metis).

<u>Autre exemple</u>
Domaine régulier, mais des temps de calcul qui dépendent du point de grille. On peut partitionner avec la technique de bissection récursive :

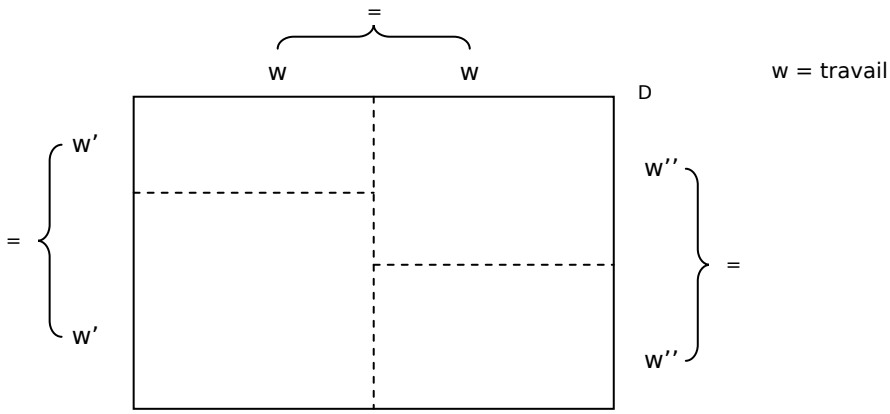


Fig. XXXXVII – Découpage du domaine D avec la technique de la bissection récursive.

Encore un autre exemple
Le temps de calcul est fixe mais dépend de l'espace de façon inconnue et on n'a pas de communication.

P_1	P_2	P_1	P_2	P_1	P_2
P_3	P_4	P_3	P_4	P_3	P_4
P_1	P_2	P_1	P_2	P_1	P_2
P_3	P_4	P_3	P_4	P_3	P_4
P_1	P_2	P_1	P_2	P_1	P_2
P_3	P_4	P_3	P_4	P_3	P_4

Fig. XXXXVIII – Répartition « statistique »

2.4.3.2 Cas dynamique

La charge change en cours d'exécution (soit parce que les données de l'application demandent plus ou moins de travail selon l'itération, soit à cause de perturbations extérieures sur le processeur (jobs d'autres utilisateurs)).

1. Il faut détecter s'il y a ou non un déséquilibrage de charge. A quelle fréquence faut-il mesurer un éventuel déséquilibrage de charge (cela implique un *overhead*) ?

2. Si le déséquilibre est suffisamment grave, il faut échanger des données des processeurs les plus chargés vers les moins chargés. Ceci est coûteux et il faut le faire seulement si le gain escompté est prometteur.

NB Le déséquilibre se mesure, par exemple, à la variance des temps de calcul dans un cas itératif. Il faut aussi s'assurer que ce déséquilibre dure depuis assez longtemps.

C'est des heuristiques avec plusieurs paramètres difficiles à optimiser.

La façon de faire dépend du problème, mais on peut avoir une stratégie d'action locale, globale ou mixte.

Locale Les processeurs décident localement, avec leurs voisins, s'il y a lieu d'échanger des données.
C'est une approche scalable, mais lente à converger vers un équilibre global.

Globale Un processeur maître contrôle les temps d'exécution et le repartitionnement garanti un rééquilibrage efficace mais un processeur est le bottleneck.

Mixte Détection globale du déséquilibre et repartitionnement local.

Exemple
Entièrement local.

On peut avoir un mécanisme pour bouger les frontières. Un processeur trop chargé donne certaines de ses colonnes à son voisin le moins chargé.

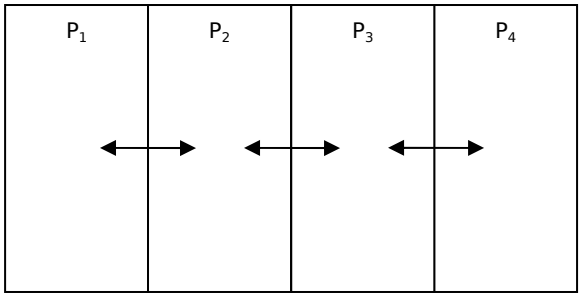


Fig. XXXIX – Echange de données – P_2 est le moins chargé. P_1 et P_3 lui envoient des données car ils sont chargés. Mais P_1 et P_3 ne communiquent pas, donc P_2 peut devenir le plus chargé de tous.

La Fig. XXXIX montre que l'on peut avoir des problèmes de convergence et d'oscillation. Il faut donc les éviter :

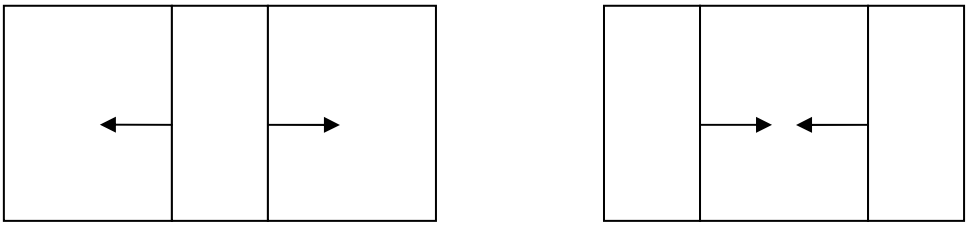


Fig. XXXX – Solution pour palier le problème précédent.

Dans le cas mixte, on peut avoir un processeur qui calcule la répartition optimale et indique à chaque *processing element* la quantité de données à passer à son voisin.

Exemple
Cas totalement global.

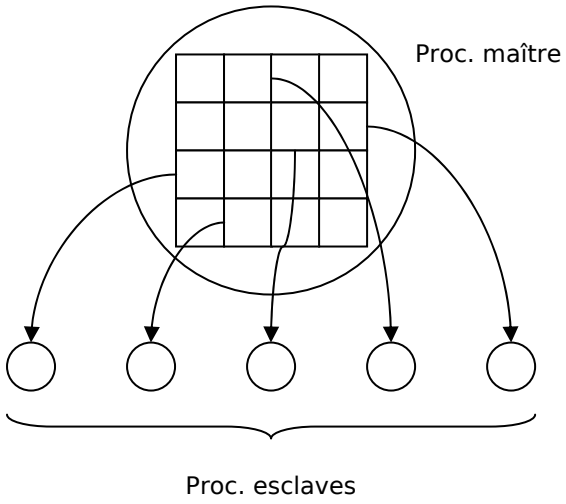


Fig. XXXXXI – Cas totalement global – La grille symbolise le domaine D.

On donne du travail au premier processeur libre. Cette méthode peut devenir lourde en *overhead*.

Dans une version mixte, le processeur maître détient et maintient une liste des processeurs qui, potentiellement, ont du travail à donner. Les processeurs qui ont fini interrogent cette liste et ensuite contactent le processeur en question.
La mise en œuvre est assez complexe car les processeurs doivent continuellement écouter les autres et à la fin il faut arrêter tous les processeurs qui n'ont pas de message en cours.

Chapitre 3

Réseau d'interconnexion

Il y a plusieurs façons d'interconnecter les processeurs entre eux (tant d'une façon physique que logique) pour être efficace dans les échanges de données et aussi pour la formulation de l'algorithme

3.1 L'hypercube

C'est un réseau célèbre d'interconnexion à la base de plusieurs réalisations hardware, mais aussi à la base de la formulation de beaucoup d'algorithmes parallèles.

Définition *L'hypercube*
Un cube d'arête 2 dans un espace de dimension *d*.

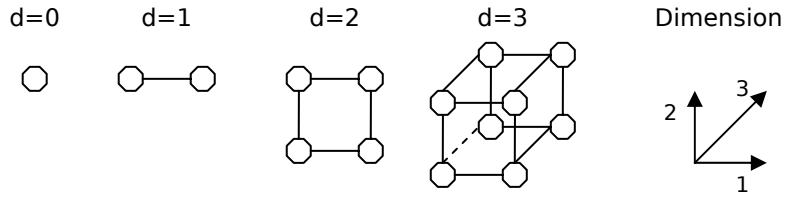


Fig. XXXXII – Hypercube.

Règle de construction
Pour faire un hypercube de dimension *d*, on en prend deux de dimension *d-1* et on connecte les nœuds équivalents. Il y aura donc huit connexions pour un hypercube de dimension 4.

Hypercube de dimension 4

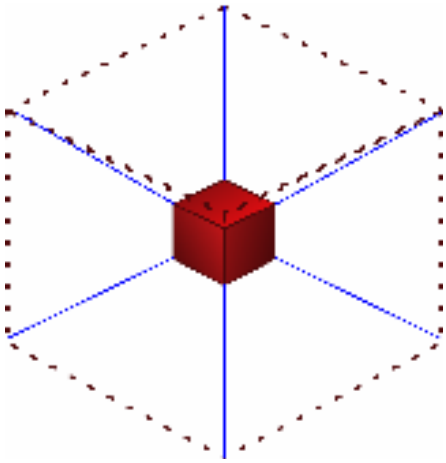


Fig. XXXXIII – Hypercube de dimension 4

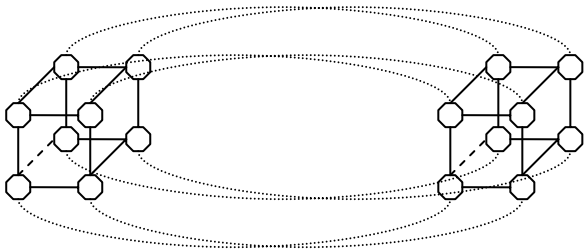


Fig. XXXXIV – Hypercube de dimension 4 – Notation équivalente à Fig. XXXXIV

Les dimensions sont numérotées de façon arbitraire. Cela permet de parler de communications selon une direction ou dimension de l'hypercube. Evidemment, le nombre de nœuds N d'un hypercube est $N=2^d$ où d est la dimension, d'où :

$$d = log_2(N)$$

On voit aussi que chaque nœud reçoit d liens. Cela défini la connectivité (ou le degré) de chaque nœud.

Définition *Le diamètre*

Le diamètre d'un hypercube est la longueur du chemin le plus court qui relie les deux nœuds les plus éloignés.

On voit ici que les points les plus éloignés s'atteignent en parcourant les d arêtes du cube, donc

$$Diamètre = d = \log_2(N)$$

On est toujours à distance raisonnable de tout autre nœud car $log(N)$ croît doucement.

3.1.2 Largeur bissectionnelle

On divise le réseau en deux moitiés équivalentes et on compte le nombre de liens qui relient chaque moitié.

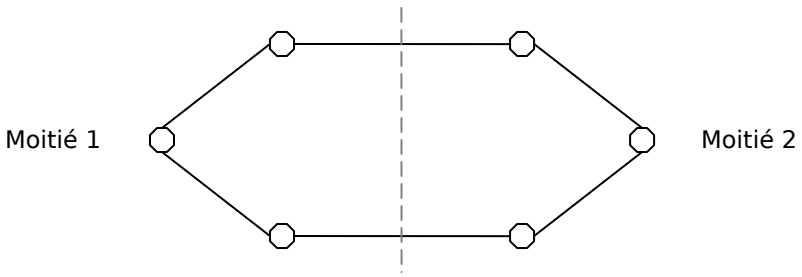


Fig. XXXXV – Anneaux – La largeur bissectionnelle vaut 2 (indépendante de N), d'où une faible capacité à échanger les informations entre les moitiés

Par construction de l'hypercube, chaque moitié contient $N/2$ nœuds et de chacun de ces nœuds part un lien vers le nœud homologue. *Largeur bissectionnelle = N/2*.

3.1.2 Numérotation des nœuds d'un hypercube

On choisit une numérotation binaire avec autant de bits que de dimensions.

- Un nœud est choisi comme racine et prend l'adresse 00…0 (il y a *d* 0).
- Deux nœuds voisins selon la direction *k* ne diffèrent que par leur $k^{ème}$ bit.

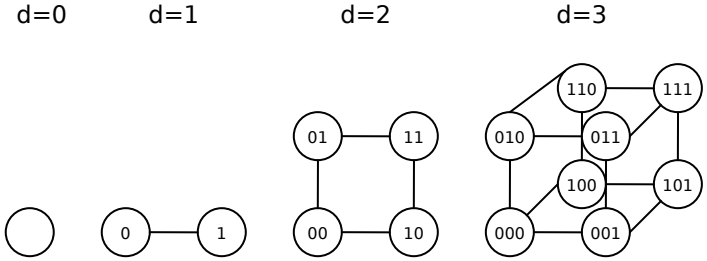


Fig. XXXXV – Numérotation des nœuds dans un hypercube

Cette numérotation est compatible avec les sous hypercubes utilisés pour construire l'hypercube plus grand. Elle donne aussi un algorithme de routage simple. En faisant le XOR des adresses de départ et de destination, on obtient un vecteur indiquant les dimensions à traverser.

On part du nœud 001
vers le nœud 100
XOR 101 ← traverser la dimension 1 et 3 mais pas la 2. L'ordre est quelconque.

3.1.3 Broadcast dans un hypercube

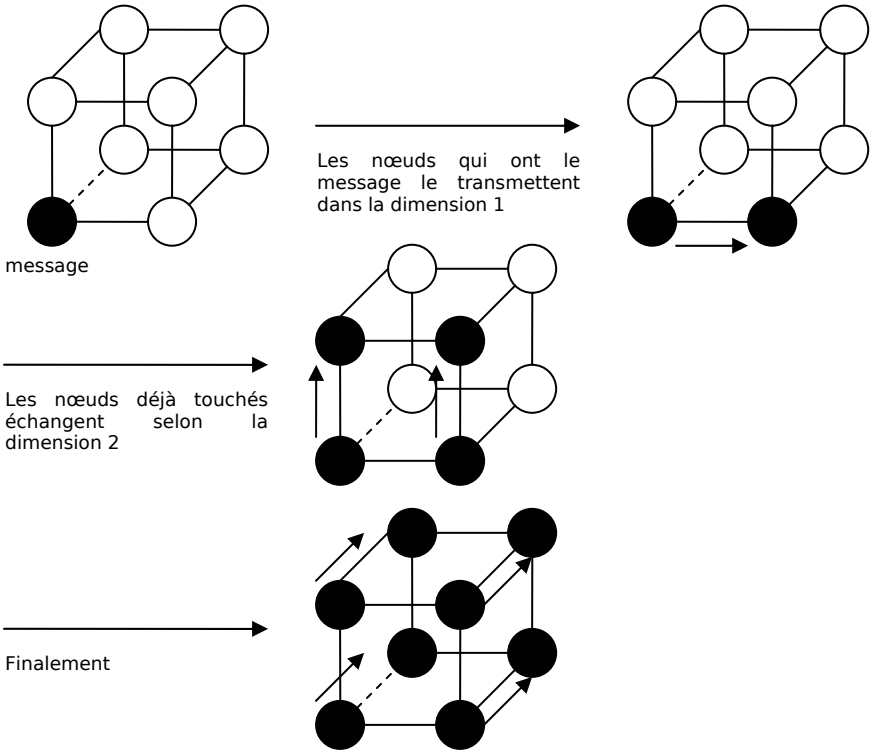
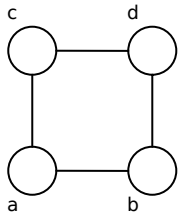


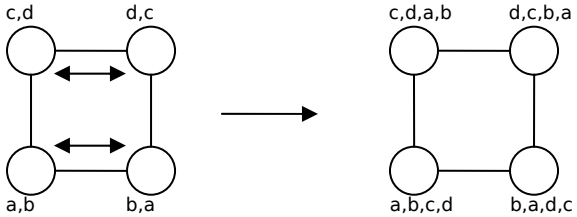
Fig. XXXXVI (page précédente) – Broadcast dans un hypercube

Le temps pour réaliser un broadcast sur un hypercube est donc $d = log_2(N)$

3.1.4 Echange total



On veut que tout le monde connaisse a, b, c, et d. On va faire un échange selon la dimension 1 :



A l'étape *i (i=1, …, d)*, on échange tout ce qu'on a reçu dans la direction *i*. A chaque étape, la taille du message augmente 1, 2, 4, …

$$T_{\text{échange total}} = \sum_{i=1}^{d} 2^{i-1} = 2^d - 1 = N - 1$$

où N croît comme le nombre de nœuds.

3.1.5 Mapping d'une grille sur un hypercube

Peut-on répartir les points d'une grille (tableau) sur les nœuds d'un hypercube de sorte que les points voisins sur la grille (Nord, Sud, Est, Ouest) le soient aussi sur des processeurs voisins par la topologie de l'hypercube.

3.1.5.1 Cas d'une bille

Soit une grille avec 2^d points.

0 — 1 — 2 i-1 — i —i+1— … —2^d-1

Construction itérative : le point *i* de la grille ira sur le processeur $G_d(i)$.

i		*G(i)*	
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Fig. XXXXVII – Technique du code de Gray réfléchi

Avec huit points de grille sur l'hypercube de dimension trois :

i	*G(i)*	
0	0	0
1	0	1
2	1	1
3	1	0
4	1	0
5	1	1
6	0	1
7	0	0

Fig. XXXXVIII – Technique du code de Gray réfléchi

Ce mapping assure de plus que le premier et le dernier point de la grille 1D soient aussi voisins. On a donc une grille périodique.

3.1.5.2 Cas à 2 dimensions

On se donne une grille rectangulaire de taille 2^r *x* 2^s sur un hypercube de dimension $r+s$ (avec 2^{r+s} nœuds). Un point de la grille a des coordonnées *(i, j)* et il sera sur le processeur

$$G_r(i) \,||\, G_s(j)$$

où || représente la concaténation des codes de Gray pour chacune des deux dimensions.

Par exemple, le point *(1,3)* est envoyé sur

$$G_r(1) \,||\, G_s(3)$$

si $r=s=2$ (grille *4x4*)

0110

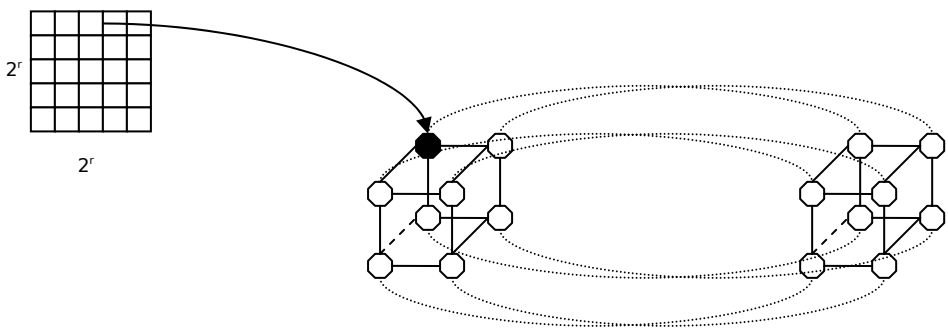


Fig. XXXXIX – Cas à deux dimensions

Cela donne globalement le mapping suivant :

j\i	0	1	2	3
0	0000	0100	1100	1000
1	0001	0101	1101	1001
2	0011	0111	1111	1011
3	0010	0110	1110	1010

On constate que la relation de voisinage est préservée ($i\pm1$, $j\pm1$) à l'exclusion des diagonales qui sont placées sur des processeurs voisins sur l'hypercube.

Ceci est garanti par la construction.

G(i) || *G(j)* est voisin de *G(i±1)* || *G(j)*
et de *G(i)* || *G(j±1)* car *G(i)* ne diffère que d'un bit de *G(i±1)*.

Remarque On peut voir un hypercube indépendamment de son interprétation géométrique, mais simplement à travers une règle de connexion de chaque nœud.

Une adresse d'un nœud en binaire $b_1b_2...b_k$ dans un hypercube de dimension k possède k voisins :

- $\overline{b}_1b_2...b_k$ où $\overline{b}_1 = 1 - b_1$
- $b_1\overline{b}_2...b_k$
- ...
- $b_1b_2...\overline{b}_k$

De façon plus générale, on peut créer une topologie en se donnant des permutations des valeurs des k bits.

Permutation Shuffle-inverse

$$b_1b_2b_3...b_k \rightarrow b_2b_3...b_kb_1$$

Permutation d'échange

$$b_1b_2b_3...b_k \rightarrow b_1b_2...\overline{b}_k$$

3.2 Les réseaux multi-étages

3.2.1 Présentation

Ce sont des réseaux dynamiques (comme un switch où processeurs et/ou mémoires sont connectés) faits de couches de switches très simples.

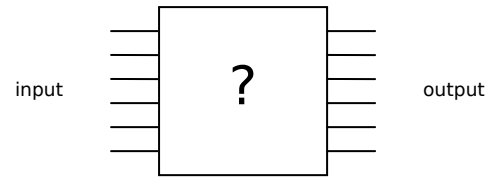


Fig. XXXXX – Réseau multi-étages

On considère maintenant le cas du réseau Oméga. Il est fait à partir de switches 2x2.

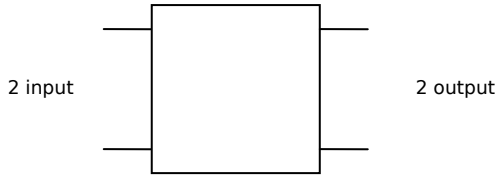


Fig. XXXXXI – Switch 2x2

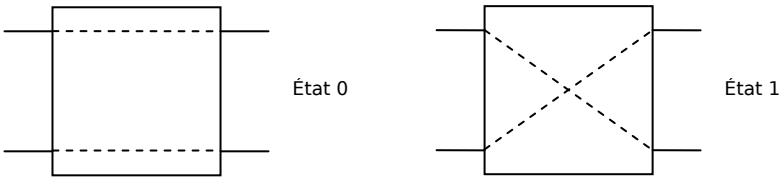


Fig. XXXXXII – Les deux états possibles du switch 2x2

On peut aussi ajouter un état de Broadcast qui est :

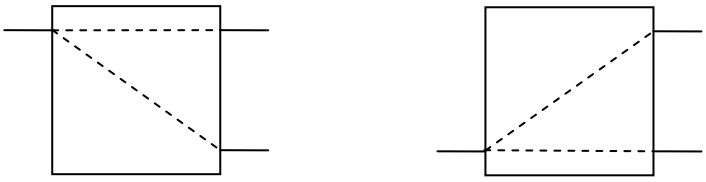
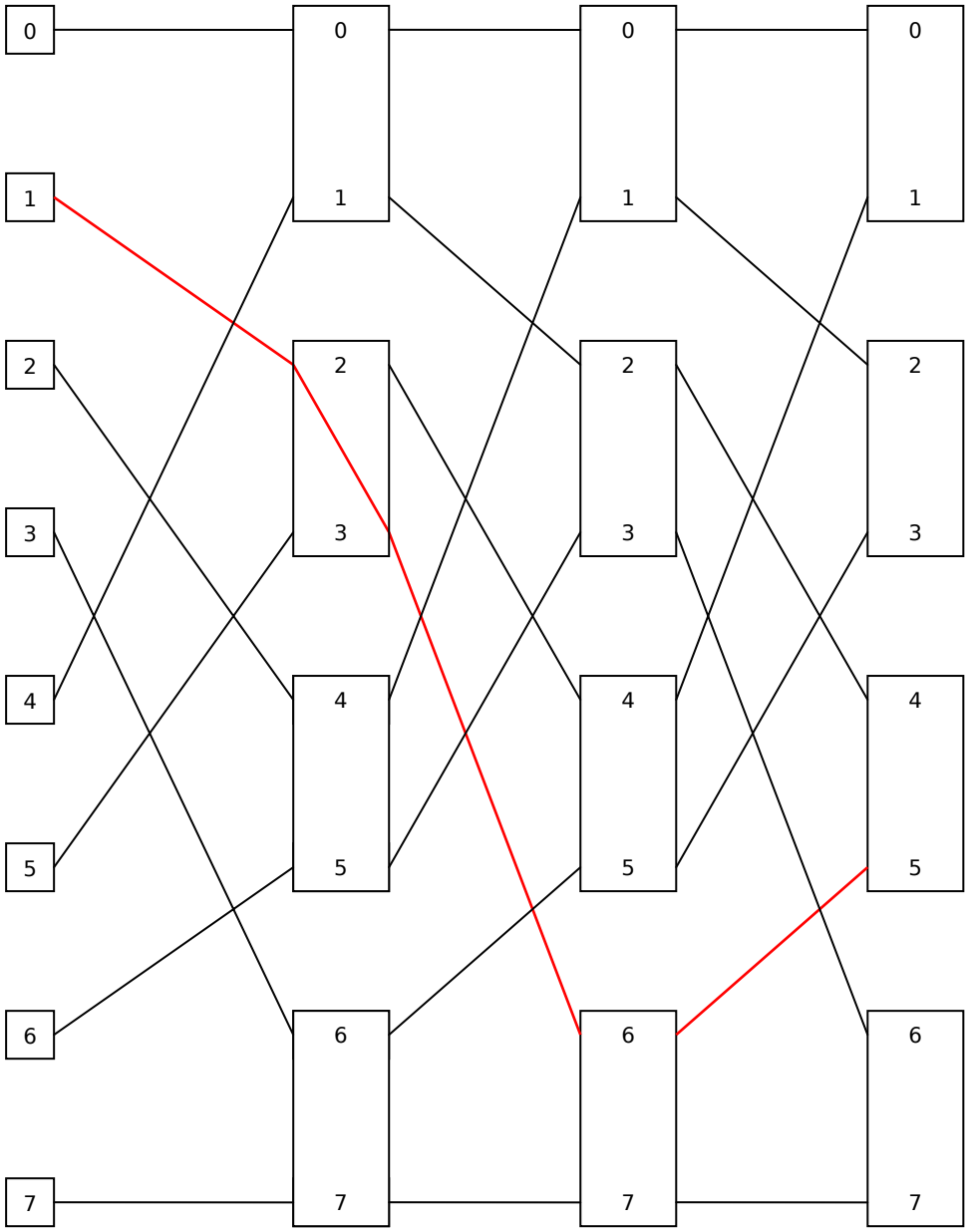


Fig. XXXXXIII – L'état de Broadcast du switch 2x2

Le réseau Oméga considère 2^k entrées/sorties avec k couches de 2^{k-1} switches 2x2.

Exemple
Cas où $k=3$ (8 in-out).



Shuffle inverse (permutation cyclique)

(0) 000→000
(1) 001→010
(2) 010→100
(3) 011→110
(4) 100→001
...
(7) 111→111

Fig. XXXXXIV – Réseau Oméga – En rouge, trajet d'un message de 1 à destination de 5

Lien entre les switches 2x2 et la permutation d'échange.

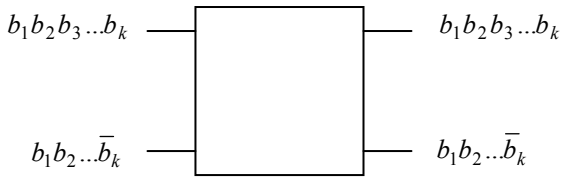


Fig. XXXXXV – Switch 2x2

En mode croisé, le switch réalise la permutation d'échange.

3.2.2 Routage

On cherche un algorithme qui mène d'une entrée $l_1l_2 \ldots l_k$ à une destination $d_1 \ldots d_k$.

Algorithme
A la $k^{ème}$ couche du réseau oméga, on observe le $k^{ème}$ bit le plus significatif de l'adresse de destination et s'il est *0* on sort par la sortie du haut du switch *2x2*, s'il est *1* on sort par la sortie du bas.

Exemple
Trajet du nœud *1* au nœud *5* selon cet algo : $5 = \underset{bas}{1}\ \underset{\text{haut}}{0}\ \underset{\text{bas}}{1}$ (voir trajet rouge de Fig. XXXXXIV)

Explication
Soit $l_1l_2 \ldots l_k$ l'adresse de départ. En passant à la première couche, on se trouve en $l_2l_3 \ldots l_kl_1$ (par shuffle-inverse). Si $d_1=1$ alors on prend la sortie du bas, on se trouve en $l_2l_3 \ldots l_k1$. Si $d_1=0$, alors on prend la sortie du bas donc $l_2l_3 \ldots l_k0$.

A la sortie du premier étage, on est donc en $l_2l_3 \ldots l_kd_1$ ensuite, on entre au $2^{ème}$ étage en $l_3 \ldots l_kd_1l_2$ (shuffle-inverse) et on en sort en $l_3 \ldots l_kd_1d_2$.
Progressivement, en k étapes on « reconstruit » l'adresse de destination par les choix de déplacements.

→ $l_kd_1d_2 \ldots d_{k-1}$
→ $d_1d_2 \ldots d_k$.

NB Un chemin entre *5* et *7* est en conflit avec le *1* à *5* car le même switch et la même connexion sont doublement utilisés.
(plus de détails voir la page 98 du polycopié « *Architecture et technologie des ordinateurs II* » – B.Chopard).

Remerciements

A Harris pour l'aide à la prise en note.