Parallélisme

Bastien Chopard
Département d'Informatique, Université de Genève

Automne 2023

Table des matières

Première partie

Architectures

Chapitre 1

But et définition du parallélisme

1.1 Du parallélisme à l'architecture de von Neumann

En cherchant à construire des ordinateurs, l'homme s'est d'abord inspiré de modèles de calcul naturels, mettant en oeuvre une organisation du travail proche de celle à laquelle il est habitué. Ainsi, pour réaliser une tâche importante dans un temps acceptable, il est naturel de partager le travail parmi plusieurs ouviers, ainsi qu'illustré sur la figure 1.1. On imagine mal en effet la muraille de Chine se construire sans une collaboration massive de nombreuses personnes travaillant simultanément. Mais il n'y a pas que pour les activités manuelles que la participation de plusieurs travailleurs peut être bénéfique. Par exemple, au 19eme siècle, un chercheur avait imaginé une façon d'accélérer les calculs en astronomie en réunissant un grand nombre de personnes, chacune responsable d'une tâche bien précise. Toutes ces personnes, travaillant simultanément selon un programme strict devaient collaborer à la résolution d'un problème unique qui prendrait trop de temps à une personne seule. Dans le même esprit, en 1922, un scientifique anglais, L.V. Richardson avait développé une méthode parallèle pour calculer les prévisions météorologiques dans son pays. Il avait divisé l'Angleterre en 64 000 points selon une grille régulière et utilisé de nombreuses personnes pour calculer l'évolution temporelle de la pression sur ces points. Plus de détails sur ce type d'«ordinateur humain parallèle» sont décrits par David Alan Grier [1].

Comme on le voit, la coopération est certainement une façon naturelle d'aborder de nombreux problèmes et il n'est pas surprenant que les premiers concepteurs d'ordinateurs aient pensé à combiner plusieurs unités de traitement pour résoudre un même problème. Ainsi, en 1945, l'ENIAC, l'un des premiers ordinateurs électroniques fut doté de plusieurs unités arithmétiques activables en même temps.

A la même époque environ, von Neumann étudiait la possibilité de concevoir

1. D. A. Grier, Human computers : the first pioneers of information age. Endeavour vol. 25(1), p. 28–32, (2001)

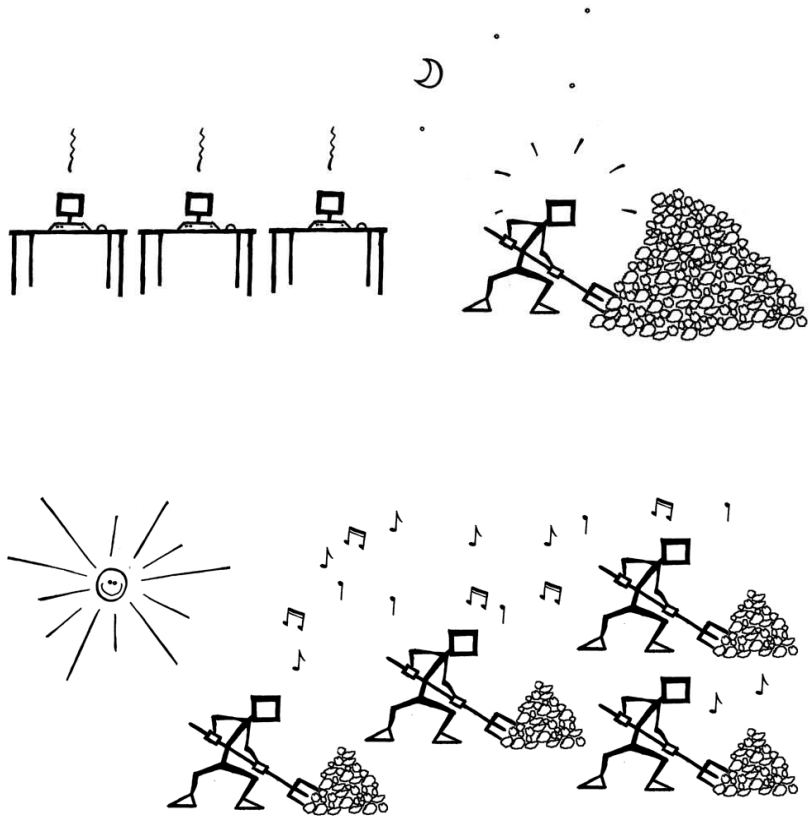


FIGURE 1.1 – Exemple d'un travail partagé entre plusieurs ouvriers (dessin Alexandre Masselot).

une machine imitant le fonctionnement du cerveau. En cherchant à abstraire le modèle de calcul, il proposa le concept d'automate cellulaire qui est en fait le prototype d'une machine parallèle de type SIMD qu'on présentera plus loin.

Pourtant, la réalisation matérielle de telles machines étaient loin d'être simple. La technologie de l'époque n'était guère fiable et la gestion et l'arbitrage d'un ensemble d'unités de traitement plus compliquées que prévu.

De plus, comme on peut aisément l'imaginer en se rapportant à des situations de la vie de tous les jours, la coopération entre plusieurs personnes nécessite une gestion supplémentaire et une coordination des efforts si on souhaite éviter que les travailleurs ne se gênent mutuellement (par exemple les ouvriers de la muraille de Chine qui construisent des sections de mur adjacentes).

On a donc cherché un modèle de calcul différent, plus adapté à des exigences d'automatisation. C'est von Neumann qui en a proposé le principe : une unité centrale de traitement lit séquentiellement une information stockée dans une mémoire. Cette information est constituée d'une liste d'instructions et de données. Chaque instruction est exécutée par l'unité de traitement et correspond à une opération entre plusieurs données. Les emplacements dans la mémoire de ces données sont précisés par l'instruction elle-même.

Ce modèle de calcul à influencé considérablement notre perception de l'informatique au point qu'on n'imagine pas toujours qu'il puisse exister des ordinateurs fonctionnant selon un autre modèle.

Le grand succès de l'architecture de von Neumann réside dans sa simplicité et sa réalisabilité par des circuits électroniques. Elle a permis un développement conjoint de l'architecture matérielle, des logiciels et des applications.

Pourtant, l'architecture de von Neumann, dans sa version la plus simple, présente des limitations importantes de performances. L'évolution technologique montre que la vitesse de traitement de l'unité centrale d'un processeur augmente plus rapidement que la vitesse d'accès aux données de la mémoire. Le processeur doit donc attendre que les données aient transité de la mémoire au CPU. Du fait de la lenteur de ce transfert, la liaison entre la mémoire et le CPU est souvent qualifiée de **goulet d'étranglement de von Neumann** ou, en anglais, le **von Neumann bottleneck**. La raison de cette différence de vitesse entre le CPU et la mémoire provient du fait que le voltage nécessaire pour faire transiter des données est plus important si la distance physique à parcourir est grande (ce qui est le cas d'une mémoire hors chip). Pour limiter l'énergie consommée et dissipée, il faut donc abaisser la fréquence avec laquelle on accède à la mémoire.

Cette limitation de performance s'appelle aussi le *memory wall*. Elle est importante pour des applications intensives en données, c'est-à-dire celles qui requièrent un grand trafic avec la mémoire centrale. Le modèle *roofline* décrit dans la figure 1.2 illustre ce problème. Une application est représentée par son *intensité arithmétique* I, à savoir le nombre d'opérations en virgule flottante réalisées pour chaque byte lu en mémoire. La grandeur I est ainsi mesurée en flop/byte. Plus I est grand, plus une même donnée donne lieux à beaucoup de calcul. La

limite de performance est dans ce cas la vitesse du processeur, R_{max}, mesurée en flop/sec. On dit alors que l'application est *CPU-bound*. Par contre, si I est petit, c'est la bande-passante β entre le CPU et la mémoire qui est le facteur limitatif des applications dites *memory-bound*.

En effet, avec une bande passante de β [byte/sec], et une intensité arithmétique de I [flop/byte], on peut au mieux avoir une performance de βI [flop/sec]. Si cette valeur est inférieur à R_{max}, alors c'est la bande passante qui est limitante. Sinon, si $\beta I > R_{max}$, c'est le processeur qui est limitant.

Une façon plus formelle de dériver ces limites est de considérer le modèle de performance suivant

$$T_{exec} = T_{access} + T_{calcul} = p_{miss} T_{memory} + p_{hit} T_{cache} + \frac{I}{\gamma R_{max}} \tag{1.1}$$

où le temps d'exécution relatif au traitement d'une donnée dépend du temps qu'il faut pour la charger dans les registres et du nombre de fois qu'elle sera réutilisée. Le temps d'accès à la donnée dépend de la probablitité sa présence dans le cache (cache hit) ou non (cache miss), avec évidemment $p_{miss} + p_{hit} = 1$. Ici, $0 < \gamma \leq 1$ est un facteur qui indique si le calcul se fait à la puissance maximum du processeur ou non. Par exemple une division, une racine carrée, etc, ne se feront pas aussi vite qu'une addition.

La puissance effective de l'exécution considérée est donc, en fonction de l'intensité arithmérique I

$$\begin{aligned} R_{eff}(I) &= \frac{I}{T_{exec}} \\ &= \frac{I}{p_{miss} T_{memory} + p_{hit} T_{cache} + \frac{I}{\gamma R_{max}}} \\ &= \frac{I}{\frac{1}{\beta} + \frac{I}{\gamma R_{max}}} \\ &= \frac{R_{max} I}{\frac{R_{max}}{\beta} + \frac{I}{\gamma}} \end{aligned} \tag{1.2}$$

où β est défini comme $\beta^{-1} = p_{miss} T_{memory} + p_{hit} T_{cache}$ et a les dimension d'une bande passante (Byte/seconde).

Dans la limite où I est très grand, la puissance effective tend vers

$$R_{eff} = \gamma R_{max} \leq R_{max}$$

Quand au contraire I/γ est petit par rapport à R_{max}/β, la puissance effective tend vers

$$R_{eff} = \beta I$$

Ces deux limites sont celles indiquées par les pointillés de la figure 1.2.

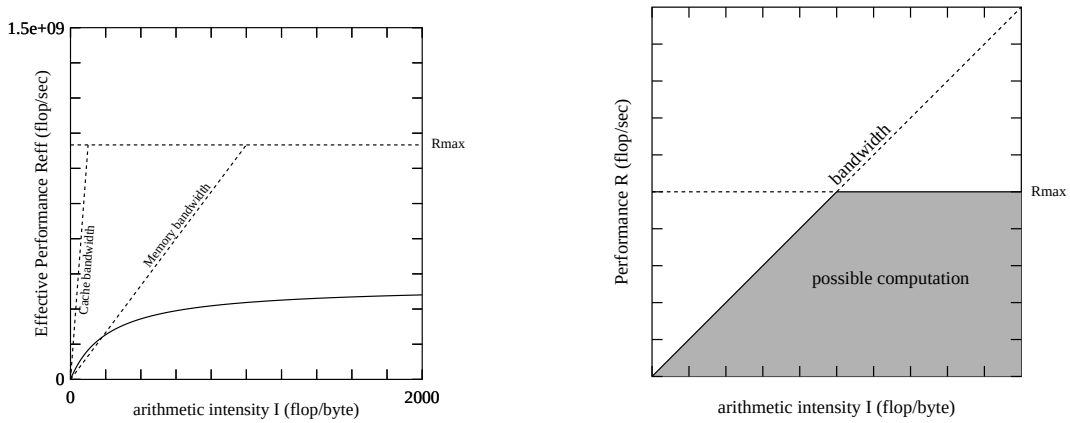


FIGURE 1.2 – Le modèle de performance dit «roofline» indique les limites de performance d'un programme en fonction de *l'intensité arithmétique I* et la puissance maximum du processeur R_{max}. A gauche, le graphique correspond à l'équation (1.2) et à droite il montre les limites absolues.

Que les applications soient CPU-bound ou memory-bound, le parallélisme reste une façon naturelle d'augmenter les performances des ordinateurs en faisant collaborer plusieurs processeurs d'une génération donnée.

Dans les années 70, l'ILLIAC IV (figure 1.3), le premier ordinateur véritablement parallèle fut mis en service, avec une perfomance de 40 Mflop/s. Malheureusement, il coûta quatre fois plus qu'escompté et ne fut réalisé qu'avec un quart des processeurs prévus. Cette machine, trop chère et de surcroît peu fiable a été rapidement remplacée par les processeurs de la génération suivante, les processeurs vectoriels. Cependant, cette tentative à joué un rôle important du point de vue historique pour le développement du parallélisme.

Après les difficultés initiales pour réaliser des sytèmes parallèles, il est intéressant de comprendre pourquoi de telles architectures sont maintenant devenues possibles. La réponse est principalement dans les progrès technologiques qui ont été réalisés depuis les premières tentatives.

La technologie actuelle confère une grande fiabilité aux parties matérielles de l'ordinateur, ce qui est un point essentiel pour faire fonctionner en parallèle plusieurs unités de traitement. Par ailleurs, la partie réseau, ou interconnexion des processeurs est devenue une discipline à part entière. Les améliorations ont été fulgurantes dans cette direction.

Du point de vue financier, on est aussi arrivé à un stade où le rapport coût/performance des système parallèles ou distribués est favorable. La technologie actuelle est maintenant disponible en masse, grâce au marché grandissant de l'informatique, ce qui permet des coûts de production acceptables. A ce propos, il est intéressant de mentionner une loi empirique qui stipule, qu'à un moment

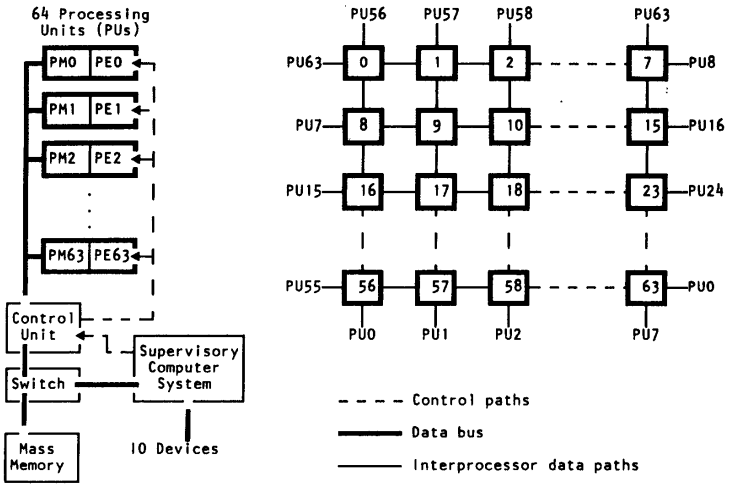


FIGURE 1.3 – *Schéma de l'architecture de l'ILLIAC IV (Almasi & Gotlieb). En complément des unités processeur-mémoires, une mémoire de masse composée d'un disque comprenant 128 têtes travaillant en parallèle, faisait partie du hardware. L'accès très rapide à la mémoire disque permettait à l'ILLIAC de traiter des problèmes généralement trop grands pour s'adapter à la mémoire centrale des autres machines de l'époque.*

donné de l'évolution des architectures, on ne peut pas produire un processeur deux fois plus rapide pour seulement le double du prix. Cela constitue un argument financier en faveur du parallélisme.

Un autre facteur qui a contribué au succès du parallélisme est la convergence des modèles de programmation. Actuellement, il existe des façons standards, qui seront décrites plus loin, de programmer une architecture parallèle (par exemple MPI et OpenMP). Ce n'était pas le cas pendant de nombreuses années et chaque nouvelle architecture impliquait une adaptation laborieuse des programmes.

Le parallélisme consiste non seulement à faire collaborer plusieurs processeurs. Il intervient à tous les niveaux matériels, notamment à l'échelle du processeur (processeurs dit superscalaires, où plusieurs unité d'exécutions peuvent recevoir en parallèle plusieurs instructions en même temps. On parle d'**ILP** (Instruction Level Parallélism) pour décrire ce niveau de parallélisme. Sur les machines multicoeur récentes, on parle de **Thread Level Parallelism** ou TLP pour indiquer que plusieurs threads s'exécutent de façon concurrente.

1.2 Accroissement des besoins de performances

Malgré l'accroissement constant des performances des ordinateurs, le besoin pour des machines plus puissantes est toujours présent. L'ambition des chercheurs pour résoudre des problèmes toujours plus difficiles a été un moteur du dévelop-

pement des ordinateurs. A mesure que de nouvelles technologies apparaissent et que les processeurs deviennent plus rapides, de nouvelles applications émergent avec toujours une plus grande exigence en performance et en mémoire. Ce fait est bien illustré par l'anecdote suivante : vers la fin des années 40, un rapport préparé pour le gouvernement anglais prévoyait que 2, voire 3 ordinateurs seraient suffisant pour couvrir la totalité des besoins de calculs de toute l'Angleterre. A cette époque, on avait donc une vision bien limitée des possibilités d'utilisation des ordinateurs. De même, dans les années 70, on pensait que 64 KB de mémoire centrale serait suffisant pour toutes applications.

Chaque période a eu ses défis numériques. Les applications militaires ont joué un rôle important avec les calculs nécessaires à la conception des bombes atomiques et ensuite les calculs liés aux trajectoires de missiles. Les prévisions météorologiques et les calculs aérodynamiques ont aussi été parmi les domaines qui ont stimulé le développement d'ordinateurs plus rapides. Plus récemment, certains problèmes ont été identifiés comme des "Grand Challenges" pour les ordinateurs. Parmi eux, on peut citer, entre autres, les modèles climatiques, le calcul des courants dans les océans, le traitement du génome humain et les problèmes de reconnaissance de vision et de langage. D'une façon générale, les simulations multi-échelles qui impliquent la co-existence de processus à des échelles d'espace et de temps très différentes sont des défis numériques difficiles. Par exemple, dans le cadre des applications bio-médicales on peut mentioner les projets qui visent à une modélisation complète de la physiologie humaine.

En plus des exemples ci-dessus, des applications nécessitant par exemple du data mining ou de la modélisation financière, se développent avec un besoin grandissant de performance et de taille mémoire.

L'acronyme **HPC** (High Performance Computing) est maintenant courrament utilisé pour décrire ce type de problèmes.

En se basant sur l'évolution historique, on peut conclure qu'il n'y aura jamais assez de ressources pour satisfaire les besoins de performance. Et si maintenant le Teraflop/s (10^{12} opérations en virgule flottante à la seconde) est chose courante dans le monde des super-calculateurs, on a atteint les 1 Petaflop/s (10^{15} opérations en virgule flottante) en 2008. La prochaine étape est d'atteindre l'exaflop/s (10^{18} opérations en virgule flottante)

1.3 Evolution des architectures

D'une manière assez grossière, on peut diviser l'évolution des systèmes informatiques en périodes d'environ 10 ans. Chaque période est caractérisée par un type d'architecture dominant. On peut ainsi identifier les "ères" informatiques suivantes :

1960-1970 : Dominance des systèmes "batch" centralisés. Les utilisateurs soumettent leurs programmes qui seront exécutés par lots. Cette informatique "dinosaure" reste réservée aux institutions de grande taille et au centres de recherche scientifiques.

1970-1980 : C'est l'époque où le time-sharing se développe. Les systèmes informatiques se démocratisent peu à peu. C'est l'âge des "mini-ordinateurs" qui se généralisent dans les entreprises de taille moyenne. Cependant, la gestion de ces ressources nécessite toujours une main d'oeuvre spécialisée.

1980-1990 : C'est la période d'apparition des microprocesseurs, des ordinateurs personnels et stations de travail. Cette approche augmente la productivité des utilisateurs et la souplesse d'utilisation des sytèmes. Au cours des années quatre-vingt, la puissance des processeurs augmente considérablement et des solutions comprenant des stations interconnectées se développent rapidement. Ces réseaux, encore à l'échelle locale sont appelés LAN (pour Local Area Network) remplacent peu à peu les minis.

1990-2000 : C'est l'ère du "network computing," des **WAN** où "Wide Area Network," et des **systèmes distribués**. On voit aussi se développer les machines parallèles dites **SMP** (Symmetric Multi-Processors) qui intègrent plusieurs processeurs dans la même machine pour augmenter, de façon transparente à l'utilisateur, la productivité du système. En même temps, des ordinateurs massivement parallèles (**MPP** pour "Massively Parallel Processors") sont construits, tout d'abord, selon des architectures dédiées et propriétaires pour être ensuite remplacés par des clusters de stations de travail interconnectées de façon de plus en plus efficace.

2000-2010 Avec le tournant du siècle, on assiste à l'apparition du "Web-Computing", ou Meta-Computing illustrée par les approches peer-to-peer (P2P) et **Grid-computing**. Le concept du GRID est d'avoir des ressources de calcul à disposition à partir de n'importe quel point dans le monde, un peu à la manière du réseau électrique qui nous alimente en courant sans avoir besoin de savoir d'où il vient. C'est aussi l'apparition en masse de "l'informatique nomade" ou "ambiante", (en anglais "pervasive computing"), c'est à dire l'informatique présente partout, tout le temps et immédiatement. Des processeurs sophistiqués sont en effet maintenenant présent partout dans notre vie et les objets usuels permettent de plus en plus de traiter de l'information (téléphones mobiles, machines à café, réveils,etc, par exemple). On estime qu'au début des années 2000, un Américain interagit avec 70 processeurs entre le moment où il se réveille et où il prend son petit-déjeuner. Il n'y a pas longtemps, il y avait un taux de 1 processeur pour mille personnes. Ensuite, on est arrivé à 1 processeur par personne et on s'achemine

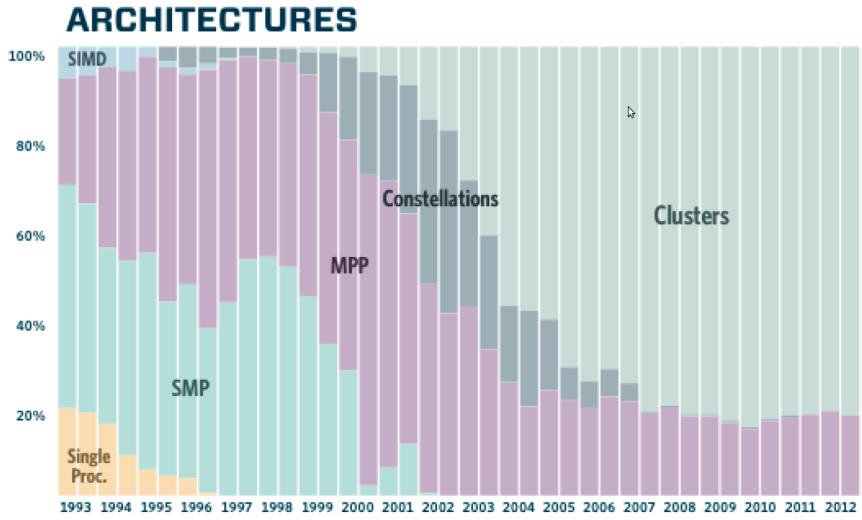


FIGURE 1.4 – Evolution de la répartition des architectures des super-ordinateurs au cours des dernières années (source ? ?)

vers une situation avec beaucoup de processeurs par personne. Les ordinateurs HPC continuent aussi de se développer. Les machines parallèles deviennent des ressources courantes, notamment les réseaux de PC. En plus, des ordinateurs ambitieux sont construits, comme la machine Blue Gene d'IBM qui peut contenir jusqu'à 1 million de processeurs.

2010- Actuellement, la tendence est de construire des processeurs multi-coeurs, avec un grand nombre de coeurs sur le même chip. De plus, les cartes graphiques (GPU) qui se perfectionnent continuellement offrent des possibilités de calcul impressionnante pour de nombreuses applications scientifiques, et plusieurs GPU peuvent s'interconnecter en parallèle pour construire des systèmes encore plus puissants. Une autre tendence des développements actuels est le *Green Computing* qui vise à produire et utiliser des ressources de calcul de façon harmonieuse pour l'environnement.

Les frontières temporelles indiquées ci-dessus ne sont évidemment pas strictes. Ainsi, du côté des super-ordinateurs, les machines vectorielles se sont développées dès le début des années soixante-dix. Elles ont connu leur apogée dans les années quatre-vingt-dix mais continuent d'exister. Le parallélisme, quant à lui s'est développé rapidement dans la deuxième moitié des années 80 et a connu un essor considérable au début des années 90.

La figure 1.4 indique l'évolution de la répartition des architectures des super-ordinateurs au cours des dernières années.

1.4 Systèmes parallèles et répartis

Habituellement, on distingue deux types de systèmes composés de plusieurs processeurs, selon leur degré de couplage et de l'utilisation que l'on en fait :

— Les systèmes **distribués** ou **répartis**.

— Les systèmes **parallèles**

En général, un système réparti est défini comme une architecture composée de plusieurs processeurs interconnectés dont le but est la résolution simultanée de **plusieurs problèmes** plus ou moins reliés, par une mise en commun des ressources.

Par contre, un **ordinateur parallèle** est défini comme une machine composée de plusieurs processeurs fortement couplés qui travaillent en même temps et **coopèrent** à la solution d'un **même** problème.

Malgré les définitions ci-dessus, la distinction entre calcul **distribué** et calcul **parallèle** est souvent assez floue. En règle générale, le calcul distribué implique que les ressources (de calcul ou autres) sont réparties sur plusieurs machines différentes et qu'une machine client envoie, à travers un réseau, une requête à une machine serveur qui en assure le traitement avant de lui renvoyer un résultat. La distribution des ressources peut être due à un éloignement géographique des différents ordinateurs. En général, on fait peu d'hypothèses sur l'homogénéité des architectures des différentes machines et sur la structure du réseau qui les relie. On ne fait pas non plus d'hypothèse forte sur la fiabilité de la connexion ou des machines éloignées. Par contre, les problèmes de sécurité peuvent être importants. En général, le couplage entre les processurs d'une machine ditribuée est faible en ce sens que les échanges de données sont peut fréquents en regard du calcul proprement dit, et que les processurs n'ont pas besoin de se synchroniser tout le temps.

Le calcul parallèle, en revanche, implique une volonté délibérée d'utiliser au mieux un ensemble de processeurs bien identifiés pour résoudre plus rapidement problème donné, ou un problème plus gros. On connait en détail la façon dont les processurs sont reliés et on essaye d'exploiter au mieux la topologie de l'interconnexion. En général, les processeurs sont homogènes, fiable et géographiquement installés l'un à côté de l'autre. Le couplage entre processeurs est fort car ils sont souvent amenés à échanger continuellement des données.

Dans ce qui suit, nous nous intéresserons moins à la spécificité du calcul distribué, qui implique des tâches à gros grain distribuées sur un ensemble de stations de travail et communiquant plus rarement entre elles, qu'au calcul parallèle qui a des exigences plus grandes sur les performances des composants impliqués.

Notons toutefois que les architectures distribuées connaissent un essor considérable avec la mise en réseau systématique des ordinateurs actuels. Le développement considérable que connait maintenant Internet à travers le WWW donne aussi lieu à des nouvelles possibilités de calcul répartis à l'echelle de la planète comme les applications P2P et le GRID (ou Grille, en français).

	Couplage	mise en commun délibérée	granularité	hypothèse sur le système	connaissance mutuelle	problèmes de sécurité
parallélisme	fort	oui	fine	oui	oui	non
réparti	faible	non	grossière	non	non	oui

TABLE 1.1 – Tableau différentiateur entre systèmes parallèles et distribués (ou répartis). La granularité est définie dans la section 1.9.

Le tableau 1.4 résume les différences mentionnées ci-dessus entre les systèmes répartis et parallèles.

1.5 Comment gagner des performances

1.5.1 Technologie, architecture et algorithmes

D'un point de vue pratique, on peut agir de trois façons pour augmenter les performances d'un ordinateur :

(1) Améliorer l'architecture du processeur
(2) Augmenter le vitesse des circuits réalisant les calculs, c'est à dire considérer les progrès sur la technologie du matériel.
(3) Améliorer les algorithmes

Globalement, on estime qu'entre 1970 et 1990, un accroissement des performances d'un facteur 1000 a été obtenu par des améliorations architecturales et technologiques, alors qu'un facteur de 3000 provient d'algorithmes plus ingénieux (ce qui donne une amélioration totale de plus d'un million).

Architecture En améliorant la façon dont les différentes étapes de calculs sont organisées et structurées par les unités de traitement, on peut considérablement augmenter les performances d'un processeur.

Il existe de nombreuses améliorations architecturales notoires. On peut mentionner la mémoire cache, et évidemment le parallélisme qui est le sujet central de ce document.

Technologie Au niveau le plus élémentaire, un processeur réalise des calculs grâce à un processus physique sous-jacent qui permet le traitement de l'information souhaité. Le type de processus utilisé et sa mise en oeuvre définissent la **technologie** employée pour construire un processeur. Actuellement, le processus physique que l'on exploite pour construire les portes logiques de base d'un ordinateur est principalement le mouvement des électrons dans le **silicium**.

On cherche bien sûr la plus grande vitesse de commutation de l'état 0 à l'état 1 des portes logiques, afin d'augmenter les performances du processeur. Mais la chaleur dissipée devient un facteur limitatif important car une haute température détruit le circuit imprimé.

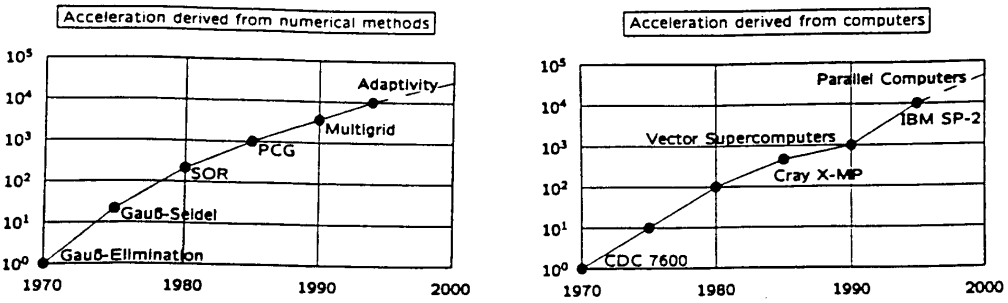


FIGURE 1.5 – *Gain de performance obtenu par des améliorations algorithmiques et des améliorations dues aux ordinateurs (technologie et architecture).*

La façon standard [2] pour augmenter la vitesse d'un circuit tout en limitant les effets néfastes de chaleur est de (1) diminuer l'épaisseur de la photolithogravure, (2) diminuer la tension et (3) augmenter la fréquence. Cependant, actuellement cette approche atteint aussi ses limites. Il faut soit se tourner vers de nouvelles technologie (utiliser les photons plutôt que les éléctrons, ou utiliser les possibilités offertes en nano-technologie,etc).

Concrètement, on observe actuellement que les fabricants de processeurs ont recours à des solutions architecturales pour continuer à satsifaire à la loi de Moore : vu la difficulté d'augmenter la vitesse d'un processeur, ils préfèrent en construire plusieurs sur le même chip, créant ainsi des processeurs multi-coeur. Cependant, l'exploitation de cette puissance potentielle posent des problèmes souvent difficiles : comment un traitement de texte va-t-il pouvoir bénéficier un processeur multi-coeur pour travailler plus rapidement ?

Algorithmes En améliorant les algorithmes de calcul, on peut souvent gagner des facteurs considérables. Un exemple est la factorisation de nombres premiers de 100 chiffres qui, en 1970 était supposée prendre un temps de 10^6 fois l'âge de l'Univers (15×10^9 années) et qui a été réalisée dans les années 90 en 8 mois, avec plus de 600 stations de travail.

La figure 1.5 illustre les augmentations de performance obtenues par le technologie et l'algorithmique, pour le cas de la résolution de systèmes d'équations.

1.5.2 Quelques chiffres

Dans ce paragraphe, nous mentionnons quelques chiffres importants concernant l'état de la technologie des chips et les tendances de l'évolution.

2. M. Bohr, "Silicon Trends and Limits," Communications of the ACM, vol. 41, No. 3, pp. 80–87, 1998

années	1971	1990	2002	2014
Processeur	4004	386	Pentium 4	Core i7
Fréquence	108 KHz	20 MHz	2 GHz	4 GHz
gravure	10 microns	1 microns	0.13 microns	22 nm
transistors	2300	885'000	55 millons	1.4 millard
Puissance				90 W

TABLE 1.2 – *Évolution des performances des processurs d'Intel sur 40 ans.*

La fameuse **loi de Moore**, qui est empirique, indique que, pour un prix fixe, la vitesse des microprocesseurs, mesurée soit en nombre d'instructions par secondes ou en nombre de transistors par chip, double tous les 18 mois. En guise d'illustration, le tableau 1.2 résume l'évolution des processeurs Intel sur 40 ans. En quelques années, les fréquences d'horloge des processeurs ont passé de quelques dizaines de MHz au GHz, soit un facteur de l'ordre de 50 en 5 ans. Actuellement, un processeur à 3 GHz est tout à fait courant. Cependant, depuis quelques années on observe que la fréquence s'est stabilisée, en raison de la chaleur intense qui est produite.

Le nombre d'électrons nécessaires pour distinguer, dans un transistor, l'état 0 de l'état 1, a beaucoup diminué dans ces dernières décénies. De l'ordre de 5×10^6 dans les années soixante, il est maintenant de quelques centaines.

La finesse du trait pour la photolithogravure, aussi appellée **feature size** diminue de façon exponentielle. En 2003, elle était de 90 nano-mètres (nm), 45 nm en 2008 et 10 nm en 2012. En 2018, la technologie à 7 nm se développe. En 2020, certains fabicants annoncent une technologie de 3 nm. Au vu de cette miniaturisation constante il est difficile de définir précisément la limite à partir de laquelle les problèmes physiques fondamentaux vont vraiment se manifester.

À la fin des année 90, le volume de vente des transistors augmentait d'environ 80% par an. En comptant tous les chips vendus en 1998, on estimait que le nombre de transistors produits varie entre 10^{16} et 10^{17}, soit environ autant qu'il y a de fourmis sur Terre. En supposant que ces transistors soient alimentés à une fréquence de 60 MHz, cela implique un puissance de calcul d'environ 10^{24} bop/s (bit operation per second), ce qui serait équivalent à 30 milliards de Teraflop/s sur des nombres de 32 bits.

Parallèlement, le coût de production des nouveaux chips augmente aussi exponnentiellement, en réponse à la technologie de plus en plus sophistiquée qui est utilisée pour leur fabrication.

Finalement, il est intéressant de mentionner quelques chiffres concernant le rapport entre la puissance d'un processeur (nombre d'opérations par seconde qu'il peut calculer) et la puissance électrique consommée pour réaliser un travail donné. On espère évidemment avoir la puissance de calcul maximale pour une consommation minimale. L'ENIAC était capable d'environ 5000 additions par

seconde. Pour calculer la trajectoire d'un missile, il avait besoin de 5.25 MJoule, soit plus que ce que le missile lui même consommait en brûlant son carburant. À ce stade, le calcul coûte plus que l'expérience directe. Les processeurs actuels sont évidemment beaucoup plus rentables de ce point de vue.

Par comparaison, le cerveau humain contient 10^{11} neurones fois 10^4 synapses et la fréquence des excitations est de l'ordre de 10 Hz. Au total cela fait 10^{16} opérations sur des bits par seconde, le tout pour seulement 100 watts. C'est un excellent rendement. Par comparaison, en 2007, un processeur Intel dual-core consommait de l'ordre de 80 watts.

Le célèbre physicien R. Feynman, dans son livre "lecture on computation" s'intéresse à relier de façon fondamentale la puissance de calcul et la puissance consommée. Avec n processeurs de fréquence ν, la puissance de calcul R est typiquement $R = n\nu$. Feynman montre que $n\nu \propto P/\nu$, où la puissance consommée P croît comme

$$P \propto n\nu^2$$

On constate ainsi que la façon la plus rentable d'augmenter le rapport $R/P = 1/\nu$ (puissance de calcul sur puissance consommée) est d'augmenter le nombre de processeurs n (ou, plus généralement, le degré de parallélisme) et de diminuer la fréquence. L'ordinateur idéal aurait donc $n \to \infty$ et $\nu \to 0$.

1.5.3 Limites fondamentales

Les progrès qu'on peut attendre d'une téchnologie donnée sont limités par les lois de la physique. Ci-dessous, nous listons quelques unes de ces limites qu'aucune technologie ne pourra dépasser.

Dans le vide, la vitesse de la lumière est de 30 centimètres par nano-seconde (ns). C'est la limitation ultime pour la vitesse de propagation physique d'une information. Dans le silicium, cette vitesse est beaucoup plus faible. En tenant compte des temps de commutation et du mode de propagation des signaux, on obtient une vitesse effective d'environ 1 cm/ns.

On peut donc estimer qu'il faut 1 ns pour traverser un chip d'un centimètre de diamètre. En supposant qu'en traversant le chip, le signal réalise une opération en virgule flottante, on s'attend au maximum à une performance de 1 Gflop/s si aucun parallélisme n'est utilisé. Cela donne un ordre de grandeur aux limites de puissance d'un chip élémentaire.

Une autre limite fondamentale de la technologie des semiconducteurs est **l'effet tunnel** qui est un phénomème physique qui apparait lorsqu'on atteint des tailles très petites et qu'une description quantique devient nécessaire. Dans l'effet tunnel, des electrons peuvent traverser des régions isolantes qui leur seraient interdites par la physique classique. Pour des épaisseurs inférieures à 2 nm, les couches isolantes de SiO_2 ne sont plus assez efficaces, ce qui pose des problèmes de fiabilité et de fonctionnalité des transistors.

Finalement, une limite à la miniaturisation de la technologie actuelle est le nombre d'électrons nécessaires pour «ouvrir» une porte de transistor. Ce nombre décroît avec la réduction de la taille des transistors. On estime qu'en continuant les améliorations au rythme observé au début des années 2000, il n'y aurait plus que 1 électron en 2010. Evidemment, on ne peut pas descendre en dessous de cette limite car un électron n'est pas divisible. D'où la nécessité, à terme, de développer des transistors (switch) basés sur une autre technologie.

D'autres estimations des limites ultimes de performances ont été faites (S. Lloyd, Nature, août 2000) sur la base des lois fondamentales de la physique, en particulier la théorie de la relativité et la mécanique quantique [3]. Ces limites absolues qui sont indépendantes de la technologie utilisée sont à prendre avec un certain recul. Les deux questions fondamentales sont la vitesse ultime de calcul et la taille maximum de la mémoire qu'un processeur qui occuperait l'espace d'un laptop actuel (cette taille semble raisonnable pour un humain). Avec la relation d'Einstein $E = mc^2$, on conclut que l'energie maximum contenue dans un tel processeur (environ 1 Kg) est de 25 millions de megawatt-heure. Pour obtenir cette energie, le processeur serait obligé de cannibaliser sa propre masse, ce qui ne va pas sans poser des problèmes pratiques. Pour obtenir la vitesse de calcul, on cherche le temps de commutation que cette énergie permet (temps pour passer de l'état 0 à 1 dans des "transistors" adéquats). Pour cela, on utilise la relation $E = h\nu$ qui relie l'énergie et le temps, à travers h, la constante de Plank et ν la fréquece. On obtient alors une puissance de 5^{50} opérations à la seconde. Pour la taille mémoire, on utilise l'argument que l'information et l'entropie sont des grandeurs reliées. On peut montrer que l'entropie maximum de notre laptop correspondrait à une mémoire contenant 2.13×10^{31} bits.

On est clairement très loin de ces performances, mais aussi très loin d'une technologie qui permettrait d'atteindre même une fraction de ces limites ultimes.

1.6 Comment exploiter le parallélisme

Le monde qui nous entoure contient de nombreux exemples qui permettent d'illustrer de manière simple et qualitative certains aspects du parallélisme, notamment la forme sous laquelle il apparaît. Si l'on veut accomplir plus de travail plus rapidement, on essaye d'utiliser davantage de personnes ou de main d'oeuvre. Mais cela n'est pas suffisant : faire collaborer un grand nombre de travailleurs nécessite une organisation soignée. Les problèmes d'échange d'idées et d'informations sont cruciaux et, souvent, un excès de personnes paralysent l'entreprise plutôt que d'améliorer l'efficacité du travail global.

Pour faire travailler plusieurs processeurs ensemble il faut établir une stratégie stricte définissant qui va faire quoi. Les cas de figure les plus importants, peuvent être illustrés par des situations tirées de la vie de tous les jours.

3. Voir aussi "computing in science and engineering", juin 2002

Considérons le cas de la famille Dupont qui a invité des convives à partager un repas composé de différents sandwiches ou tartines. Avant l'arrivée des convives, la famille Dupont va entreprendre la confection de ces tartines. Comment s'y prendre ?

1.6.1 Première stratégie (séquentielle ou "von Neumann") :

La première solution consiste à laisser Mme Dupont faire tout le travail pendant que M. Dupont et les enfants vaquent à d'autres occupations. Mme Dupont fera un sandwich après l'autre, de manière purement séquentielle. C'est le mode de travail des ordinateurs classiques : un programme (Mme Dupont) traitant les données (pain, beurre, confiture...), en faisant une action à la fois.

1.6.2 Deuxième stratégie (pipeline) :

Une autre approche consiste à mettre à contribution les autres membres de la famille et qu'une sorte de travail à la chaîne commence : le premier coupe le pain, le deuxième beurre la tranche qui vient d'être coupée, le troisième met la confiture et le quatrième, par exemple range les tartines confectionnées sur un plat. Ce mode de fonctionnement s'apparente à celui des machines ayant un "pipelining" des opérations (par exemple les machines vectorielles). Cette approche permet d'accélérer la fabrication d'un facteur presque 4 (il faut initialiser le processus ou remplir le pipeline). C'est donc rentable si l'on a beaucoup de tartines à faire. Mais il faut aussi que chaque opération prenne à peu près le même temps, sinon, la chaîne est rompue. De plus, on ne peut pas faire travailler une cinquième personne car la chaîne n'a que quatre places. Il faudrait être huit pour créer deux chaînes fonctionnant en parallèle. Ce mode de travail en pipeline est aussi parfois nommé MISD (Multiple Instruction Single Data) pour rappeler que la même donnée est traitée par plusieurs unités distinctes.

1.6.3 Troisième stratégie (SIMD) :

On peut encore travailler selon un troisième schéma : Mme Dupont prend la direction des opérations alors que les autres membres de la famille se placent autour de la table, avec chacun du pain, du beurre et de la confiture. Sur ordre de Mme Dupont, chacun coupe sa tranche de pain, puis en même temps, la beurre et ensuite met la confiture. Finalement chacun donne sa tartine à Mme Dupont qui la place dans un plat et l'opération peut recommencer. Ce mode d'exécution est peut-être un peu rigide mais certainement efficace. Un minimum d'échange entre les différentes personnes est nécessaire, chacun ayant ses propres ingrédients sous la main. On conçoit aussi que l'on peut sans autre avoir M. Dupont qui confectionne des tartines à la confiture d'abricot alors qu'un des enfants peut utiliser une confiture différente. On peut de même travailler sur plusieurs sortes

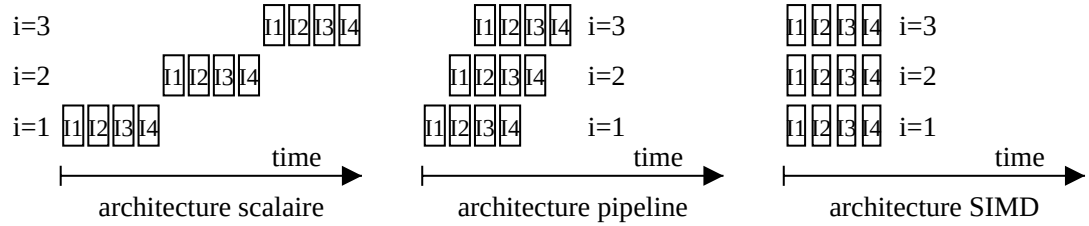


FIGURE 1.6 – *Différence de comportement pour traiter $N = 3$ données, sachant que chaque donnée (indicée par i) nécessite 4 opérations de base. Les architectures scalaires (ou séquentielles), pipelines et SIMD sont illustrées.*

de pain simultanément. Cette approche correspond au parallélisme SIMD, ce qui signifie : Single Instruction flow (Mme Dupont), Multiple Data flow (chacun à son propre pain, sa propre plaque de beurre, etc). Avec cette stratégie, on peut faire travailler autant de personnes qu'il y a de tartines à faire. De plus, si certaines opérations sont plus longues que les autres, cela n'a pas d'incidence sur le travail en commun car tout le monde, au même moment, est chargé de la même opération.

Il est intéressant d'analyser les trois organisations de travail ci-dessus en fonction des schémas de la figure 1.6 qui indiquent comment les différentes étapes du travail se déroulent dans le temps. Pour cela, on dénote par `I1`, `I2`, `I3` et `I4` les quatre opérations de base nécessaires à fabriquer une tartine, à savoir

— `I1`: Couper le pain
— `I2`: Beurrer la tranche
— `I3`: Mettre la confiture
— `I4`: Ranger la tartine sur un plat

Sur l'axe horizontal des graphiques on reporte le temps auquel chaque étape du calcul est effectué. L'axe vertical numérote les données traitées, de la première à la dernière. Dans notre exemple, on suppose que chaque tartine porte un numéro de 1 à N.

Dans la figure 1.6 on voit clairement les incidences du choix d'organisation sur le temps d'exécution du travail complet. Si τ est le temps nécessaire pour réaliser chacune des instructions de base, on voit que N tartines seront réalisées par p travailleurs dans les temps suivants (dans le cas du pipeline, on doit prendre $p = 4$) :

$$T_{seq} = 4N\tau \qquad T_{pipeline} = N\tau \qquad T_{simd} = \frac{4N\tau}{p}$$

1.6.4 Quatrième stratégie (MIMD) :

Finalement, le mode de travail qui correspond sans doute le mieux à la réalité de la situation est que chacun (y compris Mme Dupont) se charge indépendemment de la confection d'un certain nombre de tartines, en en variant la com-

position selon ses désirs, et sans se soucier de se synchroniser avec ses voisins. Cependant, en l'absence de contrôle central, il n'est pas immédiat de savoir quand arrêter le travail. Si l'objectif est de faire N tartines au total, aucun travailleur ne sait a priori combien il doit en faire. On peut bien sûr compter les tartines déjà faites mais cela suppose que chacun a un moyen de voir ce que les autres ont fait. Et même si c'est le cas, si chacun décide de faire une dernière tartine quand $N-1$ sont déjà prêtes, on en aura $N+3$ à la fin. Il faut donc mettre en place des possibilités de communication pour pouvoir se coordonner. On peut aussi se fixer à l'avance un nombre de tartines que chacun confectionnera (p. ex $N/4$ avec 4 personnes). Mais si un travailleur est plus rapide qu'un autre, il restera inoccupé à la fin du processus.

Certaines autres difficultés vont aussi apparaître si, comme cela est probable, il n'y a qu'une seule plaque de beurre pour tout le monde. Il faudra attendre qu'elle soit disponible pour s'en servir : il y a des problèmes d'accès commun à des ressources globales.

Cette manière de travailler s'apparente au parallélisme MIMD (Multiple Instruction flow, Multiple Data flow), c'est à dire que chacun suit son propre programme et interagit si nécessaire avec le reste de la famille, en communiquant par échange de messages ou de ressources communes. Sous la forme ci-dessous, c'est un cas un peu particulier de machine MIMD puisque chaque personne exécute en fait le même programme (faire des tartines). Ce mode s'appelle SPMD (Single Program, Multiple Data flow) et s'avère particulièrement efficace si chacun dispose du matériel nécessaire à la confection de ses propres tartines.

Le mode MIMD offre donc plus de souplesse mais il est aussi plus complexe et peut conduire à une certaine anarchie si l'on n'est pas très prudent : supposons que dans notre exemple, il n'y ait qu'une plaque de beurre mais aussi qu'un seul couteau (des ressources partagées). La personne qui détient le beurre ne pourra continuer sa mission que lorsqu'il aura le couteau : il va donc attendre ce dernier, ustensile indispensable pour lui. Mais la personne détenant le couteau aura peut-être une réaction analogue : il lui faut le beurre pour continuer son travail et par conséquent il attend qu'il devienne libre avant de considérer toute autre option. On débouche donc sur une situation de blocage (deadlock) que rien ne pourra résoudre si ce n'est une intervention extérieure.

L'architecture MIMD offre un nombre illimité de façons d'organiser le travail entre plusieurs processeurs. En plus de celle indiquée ci-dessus, qui est `symétrique` puisque chaque travailleur a une fonction identique à celle des autres, l'approche dite **maître-esclaves** (master-slave en anglais, ou encore host-nodes) est couramment utilisée lorsqu'un processeur spécifique effectue des tâches de gestion pour le reste du groupe. Souvent le processeur maître distribue le travail aux autres processeurs et fait la synthèse des résultas. On se rapproche ainsi du modèle SIMD mais sans la contrainte de synchronicité forte.

Un autre exemple de répartition du travail est l'approche dite de **producteur-consommateur**. Certains processeurs préparent des données qui seront ensuite

retraitées par d'autres. On peut voir cette approche comme une généralisation du travail à la chaîne. Par exemple, dans notre exemple de fabrication de tartine, l'un coupe le pain, l'un beurre les tranches et les deux derniers mettent la confiture qui est peut-être une opération à peu près deux fois plus lente que les deux autres. Cela permet d'assurer une charge équitable de travail.

1.7 Mémoire partagée et distribuée

Ces exemples mettent aussi en lumière le problème du stockage en mémoire : doit-on n'avoir qu'une seule **mémoire partagée** par tous les processeurs, auquel cas on gagne en simplicité mais on doit gérer des conflits d'accès, ou plutôt donner à chaque unité de traitement ses propres données ainsi qu'on l'a vu dans l'approche SIMD. Une **mémoire distribuée** de la sorte est avantageuse du point de vue de la fabrication de la machine mais elle oblige souvent le programmeur à adopter un mode de pensée auquel il n'est pas habitué.

Les mémoires distribuées offrent aussi la possibilité d'ajouter à un système existant de nouveaux modules processeur-mémoire sans en compromettre les performances et permettant de considérer des problèmes plus grands. Cela n'est guère envisageable avec une mémoire partagée : on voit mal 50 personnes confectionnant des tartines avec une seul plaque de beurre mise en commun. C'est le problème de l'*extensibilité* (scalabilité) d'un système auquel on reviendra plus tard et qui est un critère important des machines parallèles.

1.8 Parallélisme de données et de contrôle

En conclusion, ces exemples montrent qu'il y a deux types de parallélisme, mais plusieurs manières de le mettre en œuvre. Le parallélisme de données où beaucoup de valeurs différentes sont traitées en même temps par des processeurs différents et le parallélisme de contrôle où des tâches différentes sont exécutées simultanément. En pratique, on s'aperçoit que dans les codes scientifiques ou ceux écrits par des ingénieurs, le parallélisme de données peut être très élevé (plusieurs centaines ou milliers, voire bien davantage). Par contre, le parallélisme de contrôle est bien moins important en général et dans la plupart des codes il ne dépasse pas un facteur de 5 à 10.

Ces formes de parallélisme reflètent la nature du problème considéré alors que la classification de Flynn décrit la nature de l'architecture utilisée pour traiter le problème. Le parallélisme de données est bien adapté aux machines SIMD et MIMD alors que le parallélisme de contrôle n'est possible que dans le cas MIMD. La figure 2.1 au chapitre 2 définit les types d'architectures.

1.9 Granularité

La *granularité* est une mesure la quantité de travail fait par chaque processeur entre deux échanges avec les autres processeurs. Le parallélisme intervient à des niveaux de granularité différents selon les problèmes considérés. Il se situe typiquement aux niveaux

— des jobs ou programmes
— des tâches (à l'intérieur d'un même programme)
— des variables (à l'intérieur d'une même tâche)

Comme le montre la liste ci-dessus, la granularité peut être grossière (parallélisme entre jobs différents) ou très fine (parallélisme au niveau des variables). On reviendra plus loin, au paragraphe 6.4, sur les implications du choix de la granularité.

Chapitre 2

Architectures à haute performance

Ce chapitre décrit les architectures des ordinateurs conçus dans le but d'obtenir de grandes performances. Il s'agit bien sûr des architectures parallèles comme les machines SIMD et MIMD, mais aussi des ordinateurs vectoriels qui ont joué un rôle important dans le développement du calcul scientifique à haute performance. On mentionnera aussi les solutions architecturales adoptées dans les microprocesseurs modernes, notamment le parallélisme au niveau des instructions (ILP) qui est une caractéristique des processeurs superscalaires.

D'autres types d'architectures seront aussi brièvement discutées ici. C'est le cas notamment des machines *Dataflow* qui fonctionnent selon un principe différent des architectures von Neumann.

Nous commencerons ce chapitre en parlant de la classification de Flynn des architectures parallèles et nous passeront en revue les differentes architectures à haute performance. Un élément architectural important sera discuté en détail au chapitre 3. Il s'agit des réseaux d'interconnexion qui permettent de relier les processeurs entre-eux. Dans ce chapitre nous nous contentrons d'une vision simplifiée dans laquelle les processeurs sont interconnectés entre-eux soit à travers des câbles et des routeurs, soit à travers un switch.

2.1 Classification de Flynn

Depuis les années 70, les architectures ont été classées selon le diagramme de Flynn, en fonction de la multiplicité du flot de données et flot d'instructions. Si plusieurs données son traitées en parallèle par le système, on dira que le flot d'instruction est multiple. Sinon, il est unique (single, en anglais). De même, si plusieurs flot d'instructions sont générés simultanément par les différents processeurs, on dira que ce flot est multiple.

Le modèle séquentiel (ou de von Neumann) correspond à un flot d'instructions et de données unique. Dans la taxonomie de Flynn, l'architecture séquentielle s'appelle donc SISD (Singl Instruction flow, Single Data flow).

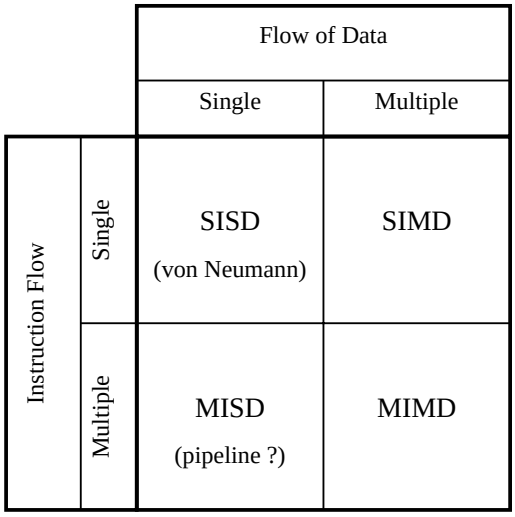


FIGURE 2.1 – *Diagramme représentant la classification de Flynn*

On a ensuite l'architecture SIMD (Single Instruction flow, Multiple Data flow) qui est l'architecture parallèle la plus simple. Elle est décrite en détail dans le paragraphe 2.2.

L'architecture MIMD (Multiple Instruction flow, Multiple Data flow) est la catégorie qui contient la grande partie de machines parallèles actuelles. Chaque processeur exécute en effet un flot d'instruction potentiellement different des autres.

Finalement, la catégorie MISD n'a pas de représentant parmi les machines qui ont été construites : elle consisterait à avoir plusieurs instructions simultanées qui traitent la même donnée. Cependant, certaines personnes ont voulu voir sous l'acronyme MIDS, le modèle de l'architecture "pipeline". Cette intreprétation est controversée.

2.2 Architecture SIMD

Actuellement, les architectures SIMD ne sont plus envisagées comme une architecture parallèle d'usage général. Les progrès technologiques permettent aujourd'hui de construire des machines MIMD offrant plus de souplesse et adaptées à un plus grand nombre d'applications. Néanmoins, les architectures SIMD restent très efficaces dans certains problèmes, et cela peut justifier leur fabrication pour un domaine d'application particulier ayant des besoins de calculs spécifiques, comme par exemple le traitement d'images ou du signal. Ainsi, certaines parties des processeurs modernes et des cartes graphiques fonctionnent sur le principe SIMD afin d'optimiser certains calculs répétitifs.

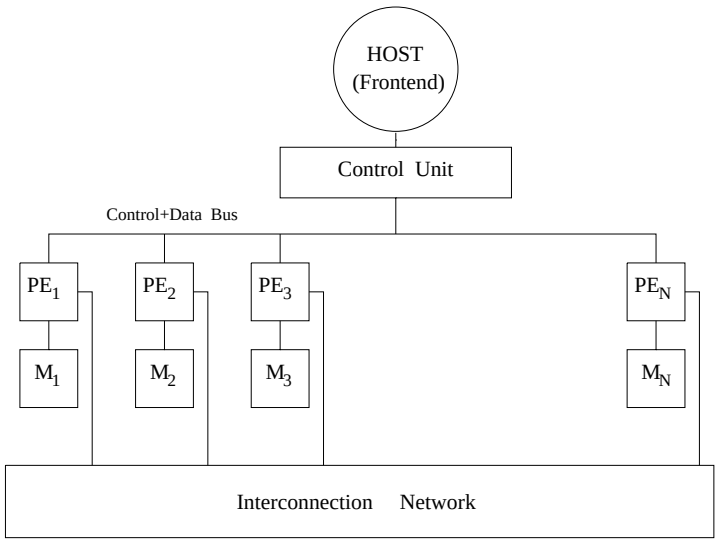


FIGURE 2.2 – *Schéma élémentaire d'une architecture SIMD.*

2.2.1 Architecture de base

Dans la classification de Flynn, l'architecture SIMD (Single Instruction, Multiple Data) se réfère à un système composé de plusieurs processeurs interconnectés, exécutant tous la *même* instruction au même moment, mais chacun sur des données qui peuvent être différentes. Ces instructions proviennent d'un programme **unique**, et l'exécution se fait de manière **synchronisée** par l'ensemble des PE (abréviation de *Processing Elements* dénotant les processeurs de calcul d'une machine parallèle).

A première vue, le modèle de calcul SIMD peut paraître quelque peu restrictif. A-t-on vraiment besoin d'avoir plusieurs processeurs qui effectuent la même tâche au même instant ? Il s'avère en fait que beaucoup de problèmes pratiques se décomposent de manière naturelle en un grand nombre d'opérations identiques qui peuvent s'effectuer simultanément sur des données différentes : c'est le parallélisme de données qui offre en général un degré de parallélisme important et de bonnes perspectives de gain en performance.

Du point de vue technologique, l'architecture SIMD a le grand avantage qu'elle reste simple dans son principe, grâce à la synchronisation des différents processeurs. Ces derniers sont connectées entre eux par un réseau d'interconnexion le plus souvent statique, comme par exemple une grille ou un hypercube. Un schéma type de machine SIMD est montré sur la figure 2.2.

Chaque processeur dispose en général d'une mémoire locale qui lui est propre et dans laquelle les données qu'il faut traiter sont conservées. Il s'agit donc d'une architecture à mémoire distribuée. Les processeurs d'une machine SIMD sont contrôlés par l'unité de contrôle (appelée CU sur la figure 2.2). Cette unité est

chargée de transmettre, à travers un bus de contrôle, les instructions qui seront exécutées simultanément par les PE. Le CU envoie (broadcast) aussi l'adresse dans la mémoire locale des PE, où les données relatives à l'instruction sont placées. De plus, les données scalaires sont aussi transmises aux PE par le CU. La liaison CU-PE doit donc être très rapide.

L'unité de contrôle est à son tour reliée à une machine hôte, en général un ordinateur à architecture classique (le frontal ou "front end"). Typiquement, le programme utilisateur réside dans le frontal et c'est là qu'il est exécuté. Alors que les instructions parallèles sont transmises aux PE par l'intermédiaire du CU, les parties scalaires du code sont entièrement exécutées par le frontal. Le programme utilisateur n'est donc pas décodé par les PE, ni stocké dans les mémoires locales.

Le type de processeur qui va constituer une architecture SIMD peut varier d'une machine à l'autre et va dépendre de l'utilisation à laquelle l'ordinateur est destiné. Ce sont typiquement des processeurs simples, comparables à des unités arithmétiques et logiques. Ils disposent en général d'un mécanisme leur permettant d'être actifs ou inactifs, c'est à dire de participer ou non à une opération dictée par le CU. Ce choix est déterminé dynamiquement, selon l'état courant des données placées en chaque processeur. De plus, dans les machines perfectionnées, les PE ont la possibilité de faire de l'adressage indirect dans leur mémoire locale, c'est à dire de travailler sur une donnée stockée à une position mémoire contenue à l'adresse unique spécifiée par le CU.

Clairement, le parallélisme SIMD, tel qu'il à été mis en oeuvre par les constructeurs, est un parallélisme à grain fin, avec un grand nombre de processeurs simple. Les machines SIMD les plus célèbres sont : l'ILLIAC IV (64 PE, années 70), le DAP (4096 PE, années 80-90), la Connection Machine CM-2/200 (jusqu'à 65536 PE, fin 80 à début 90) et la MasPar (jusqu'à 16384 PE, début années 90).

2.2.2 Communications

Les communications se font grâce aux routeurs plus ou moins complexes qui sont associés à chaque PE. En général, les routeurs sont pilotés directement par les PE mais on peut aussi avoir un contrôle du réseau d'interconnexion par l'unité de contrôle. On peut aussi avoir un bus de données qui permet un accès direct du CU aux mémoires individuelles. On distingue en général trois types de communications dans un système SIMD : locales, générales et globales. Ces communications sont faites de manière synchrone, en suivant pas à pas l'algorithme de routage approprié. Aucun calcul ne peut être réalisé par un processeur tant que la dernière donnée n'est pas arrivée à destination.

Les **communications locales** concernent le transfert d'information entre processeurs plus proches voisins. Elles sont très fréquentes pendant l'exécution de la plupart des algorithmes parallèles et utilisent des liens directs (fils de connexions entre processeurs) pour être réalisées.

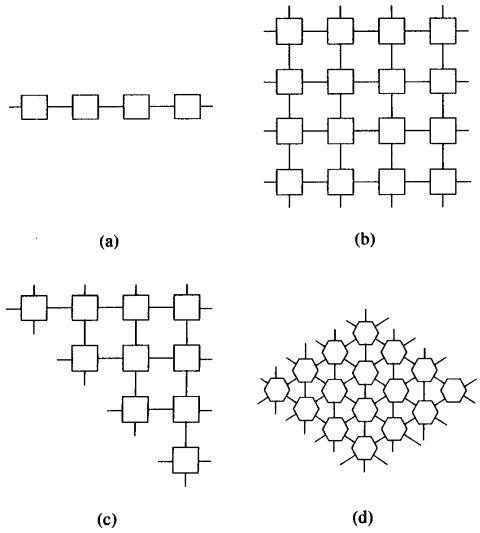


FIGURE 2.3 – *Exemples de réseaux systoliques.*

Les **communications globales** permettent d'obtenir une information globale sur le contenu de l'ensemble des processeurs d'une machine parallèle. Par exemple, il est courant de vouloir obtenir la somme des valeurs contenues dans chaque processeur. Ce sont des opérations collectives qui nécessitent un mécanisme global (mais régulier) de communication par lequel les informations contenues dans chaque processeur sont combinées et, si nécessaire, transmises à l'ordinateur frontal.

Finalement, les **communications générales** permettent des échanges quelconques entre processeurs, sans respecter un schéma de communication régulier ni prévisible. Les communications générales font appel aux mécanismes de routage des messages à travers les liens physiques du réseau. Ce type de communication est important pour assurer un maximum de souplesse aux architectures SIMD. En revanche, il est moins performant car plus général.

2.2.3 Réseaux systoliques

Un réseau systolique est une architecture SIMD de type particulier. Il consiste en un ensemble régulier de cellules simples qui sont capables d'exécuter un nombre restreint d'opérations telles des additions et multiplications. De plus, chaque cellule dispose de quelques registres internes qui font office de mémoire temporaire. Les cellules adjacentes sont typiquement interconnectées, comme le montre la figure 2.3 (cf Kumar, fig 12.1, p. 492).

Ces cellules travaillent en parallèle, de manière synchrone, comme pour une machine SIMD classique. Leur utilisation est toutefois beaucoup plus restrictive et dédiée à un algorithme spécifique. En particulier, les communications se font

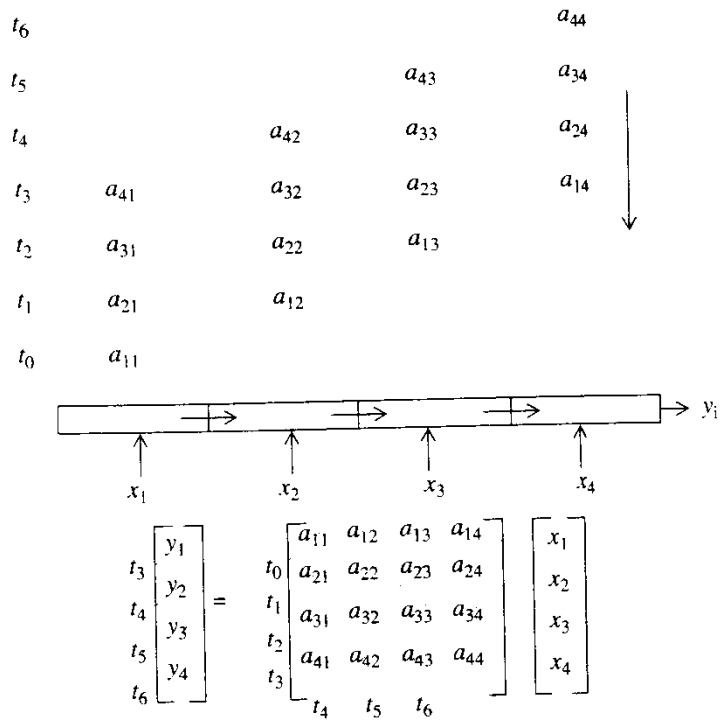


FIGURE 2.4 – *Multiplication systolique matrice-vecteur sur un réseau linéaire à quatre cellules.*

de manière très régulière et selon une structure bien précise qui correspond à l'algorithme mis en oeuvre. Le mot systolique a d'ailleurs été choisi pour décrire le flot rythmique des données à travers les cellules.

Les algorithmes systoliques sont nombreux, notamment dans le traitement de signaux et les multiplications matricielles. La particularité de ces algorithmes est que les données sont injectées par les bords du réseau systolique, puis se combinent entre elles à mesure qu'elles traversent chaque cellule. Il est donc essentiel de synchroniser le passage des données pour que les valeurs adéquates se rencontrent au bon endroit et au bon moment. La figure 2.4 (Moldovan fig. 4.28, p. 221) illustre le principe du calcul systolique pour une multiplication matrice-vecteur dans le cas d'un réseau systolique linéaire à quatre cellules.

Les coefficients a_{ij} de la matrice sont injectés dans le réseau à chaque cycle, de la manière indiquée dans la figure 2.4. Au premier cycle, a_{11} rencontre x_1 dans la première cellule, où ils sont multipliés. Au deuxième cycle, d'une part a_{21} est multiplié avec x_1 dans la première cellule et, d'autre part, le résultat précédemment obtenu arrive dans la deuxième cellule, juste au moment où a_{12} et x_2 sont combinés. Le résultat $a_{11}x_1 + a_{12}x_2$ est alors transmis à la troisième cellule où le processus continue. Au cinquième cycle, la valeur de y_1 est obtenue à la sortie du réseau. On remarque que ce processus de calcul est efficace lorsque

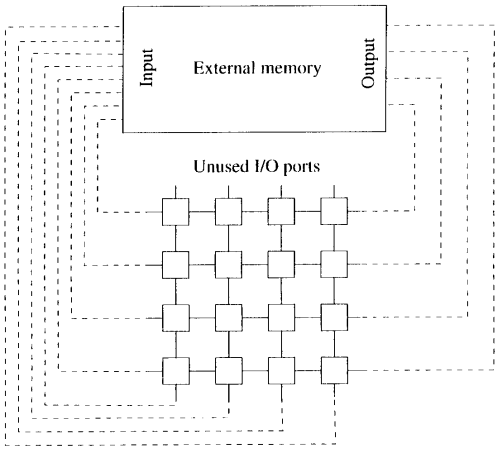


FIGURE 2.5 – *Connexions d'un réseau systolique avec l'extérieur.*

toute les cellules sont actives avec des données.

Le calcul systolique est donc essentiellement du calcul pipeline généralisé à des réseaux de processeurs. Seuls les bords du réseau sont reliés au monde extérieur, comme l'indique la figure 2.5 (cf Kumar, fig 12.2, p 493). Il s'agit toutefois d'un pipeline de données et non d'instructions, comme dans le cas des processeurs vectoriels, puisque c'est la même instruction qui est exécutée par chaque cellule.

Les réseaux systoliques sont apparus dans les années 80 lorsque les progrès dans la technologie des semi-conducteurs ont permis, grâce à l'intégration VLSI, de concevoir des circuits mettant en oeuvre des algorithmes directement au niveau du hardware.

Le calcul systolique est recommandé pour les problèmes impliquant des calculs intensifs. Les temps de communication, en raison de la spécificité des échanges de données et de l'implantation directe dans le silicium, sont souvent faibles ou négligeables.

Les algorithmes systoliques s'adaptent naturellement sur les machines SIMD. Cependant, dans ce cas, les communications sont des facteurs dont il faut tenir compte, de même que le mapping des données sur les processeurs.

2.3 Architecture MIMD

Un ordinateur MIMD (Multiple Instruction Multiple Data) se caractérise par un ensemble de processeurs pouvant communiquer de manière efficace et disposant chacun de son propre flot d'instructions (programme) qui agit sur des données qui peuvent différer d'un processeur à l'autre. Le modèle MIMD offre donc une programmation beaucoup plus riche que les machines SIMD qui n'exécutent qu'une seule et même instruction à la fois. Une autre différence importante entre ces deux architectures est que les machines MIMD sont en général asynchrones,

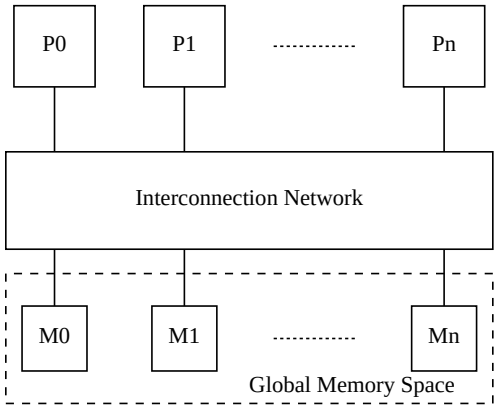


FIGURE 2.6 – *Schéma type de l'architecture à mémoire partagée.*

ce qui implique que l'état d'avancement du travail dans un processeur n'est pas prévisible par la connaissance de ce qui se passe dans un autre processeur. Deux exécutions successives d'un même programme peuvent ainsi se comporter différemment et on parle souvent de comportements **non déterministes**.

Il y a deux principales classes d'architecture MIMD : les machines à **mémoire partagée** et celles à **mémoire distribuée**. À cela vient s'ajouter les architectures à mémoire partagée **virtuelle** (NUMA et COMA) qui cherchent à concilier les avantages des deux approches : scalabilité des architectures distribuées et simplicité de programmation des systèmes à mémoire partagée.

2.3.1 Systèmes à Mémoire Partagée

Les architectures à mémoires **partagées** offrent un espace d'adresse uniforme et un accès direct à l'ensemble de la mémoire à partir de n'importe quel processeur présent. Cette architecture est représentée sur la figure 2.6. Un ordinateur à mémoire partagée est souvent dénommé **multiprocesseur**.

La mémoire se compose typiquement de plusieurs bancs, symétriquement accessibles par tous les processeurs, comme l'indique la figure 2.6. Ces différents modules de mémoire sont connectés aux processeurs à travers un réseau d'interconnexion dynamique ou commutateur (switch) dont la nature est typiquement celle d'un crossbar ou d'un réseau multiétages (voir chapitre 3).

Mais la mémoire partagée la plus simple et la moins onéreuse se base sur une architecture à **bus partagé**, à travers lequel les différents processeurs accèdent à une mémoire commune. Cette solution est typiquement adoptée dans les systèmes comportant peu de processeurs.

Les SMP (Symmetric MultiProcessors) représentent l'architecture parallèle la plus courante et la plus répandue. Les SMP ont connu un succès commercial important mais pas tellement pour leur capacité à travailler en parallèle (c'est à dire à accélérer l'exécution d'un programme unique) que par le fait qu'ils permettent

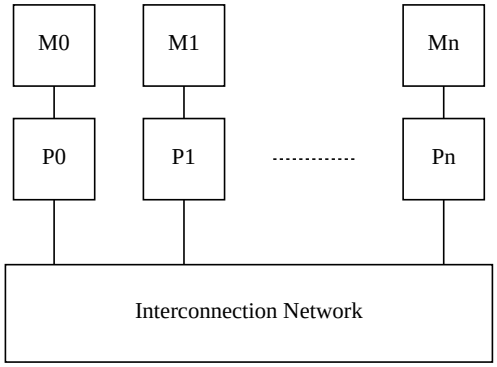


FIGURE 2.7 – *Schéma type de l'architecture à mémoire distribuée. Les processeurs ont chacun leur propre mémoire et sont connectés par un réseau d'interconnexion rapide. Chaque processeur dispose d'une unité de contrôle et peut exécuter un programme différent de son voisin.*

de mettre en oeuvre de la vraie multi-programmation. Le multi-tâches permet d'augmenter la productivité (le "throughput") en offrant à plusieurs utilisateurs la possibilité d'exécuter simultanément leurs programmes, dans un environnement où les ressources sont naturellement partagées.

2.3.2 Architecture à Mémoire Distribuée

Dans les systèmes à mémoires **distribuées** la mémoire est répartie sur chaque processeur sous forme de blocs mémoire locaux et privés, comme indiqué sur la figure 2.7. L'accès mémoire n'est possible que pour le processeur détenteur. Les communications se font exclusivement par des protocoles d'échange de messages (*message passing*) au cours desquels un processeur envoie, à travers le réseau, une de ses données propres à un autre processeur. Un ordinateur à mémoire distribuée est aussi qualifié de **multi-ordinateurs** ou **multicomputer** car il ressemble à une réunion de plusieurs ordinateur indépendants, couplés par un réseau.

Les MPP et Beowulfs

Une architecture à mémoire distribuée qui a été conçue dès le début pour un parallélisme à couplage fort et qui utilise de nombreux composants dédiés s'appelle aussi MPP (Massively Parallel Processor). La vitesse des échanges entre les processeurs est particulièrement importante (latence faible, bande passante élevée) ainsi que le degré d'élaboration des primitives de communications.

Les MPP s'opposent aux systèmes appelés **Beowulf**. Ces derniers représentent une classe d'architecture à mémoire distribuée caractérisée par un excellent rapport coût/performance. Ce sont typiquement des machines faite à partir de composants du marché, tels des PC interconnectés par un réseau fast-ethernet. Le

logiciel est lui aussi largement répandu et en général du domaine publique : Linux avec des outils GNU et MPICH ou PVM pour les échanges de message.

Beowulf est le nom d'un projet du milieu des année 90, visant à produire une machine à haute performance avec un budget modeste. L'origine du nom Beowulf provient d'un héro de la littérature anglaise du 8eme siecle qui libéra les populations d'un monstre qui les terrorisait. Dans le cadre du projet, l'analogie vient du fait que la machine plannifiée allait libérer les scientifiques de la tâche oppressante de continuellement porter leurs applications sur de nouvelles architectures à haute performance.

Il n'y a pas de définition précise quant à l'architecture d'un Beowulf si ce n'est que c'est une architecture à mémoire ditribuée. La topologie d'interconnexion est choisie selon les composants du marché disponibles et le budget à disposition.

La réalisation la plus courante comprend des PC reliés par un giga-switch ethernet. Ces switches sont actuellement courants et pour les systèmes avec beaucoup de processeurs, plusieurs switches peuvent être mis en cascade. Mais d'autres solutions sont aussi envisagées, comme des liaisons par des protocoles plus performants que TCP/IP. Myrinet est un exemple de technique d'interconnexion qui utilise le wormhole pour transmettre les messages entre les noeuds. Infiniband est une autre technologie d'interconnexion qui est bien utilisée.

Actuellement, un système Beowulf se rapproche de plus en plus d'un MPP par ses performances CPU et la richesse du logiciel. Cependant, le Beowulf reste souvent limité par ses capacités de communication s'il n'utilise pas des techniques de communication rapides.

2.3.3 Mémoire Partagée Virtuelle

Les systèmes à mémoire partagée **virtuelle**, cherchent à concilier les avantages des deux solutions "mémoire partagée" et "mémoire distribuée", notament la scalabilité des derniers et l'espace mémoire unique des premiers.

La mémoire est physiquement distribuée mais des mécanismes hardware permettent un accès à la totalité de l'espace des adresses mémoires, de manière plus au moins transparente pour l'utilisateur. L'architecture dite COMA (Cache Only Memory Access) offre une mise en oeuvre d'une mémoire partagée virtuelle en utilisant une architecture uniquement à base de mémoires caches. Elle est décrite sur la figure 2.8.

Dans un système COMA, chaque processeur dispose d'une mémoire cache locale, de taille comparable à une mémoire centrale. Ces mémoires sont appelées mémoires cache car elles sont associatives et donc adressables par le contenu. Le principe de base est que les données n'ont pas d'emplacement fixe. Elles n'appartiennent à aucun processeur en particulier. Au contraire, elles "flottent" d'une mémoire cache à l'autre, selon les processeurs qui les demandent. Alors que dans un cache usuel, les données qui s'y trouvent proviennent de la mémoire centrale,

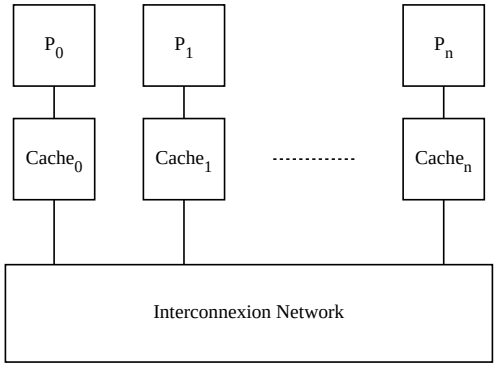


FIGURE 2.8 – *Schéma type de l'architecture COMA (Cache Only Memory Access).*

ici, elles proviennent d'un autre cache. Il y a donc des possibilités de **migration** des données.

L'adresse physique est la clé d'accès des données et la gestion des modules caches est réalisée au niveau du hardware par un dispositif particulier. Notamment, une question primordiale est la **cohérence des données** dans les caches car une même information peut être répliquée plusieurs fois. Un protocole (p.ex. le protocole MESI) implanté au niveau matériel, garantit que les modifications des données se font de façon consistante par les différents processeurs du système.

Il faut remarquer que ce problème de cohérence des données est aussi présent dans les architectures à mémoire partagée dans lesquelles chaque processeur dispose d'une mémoire cache traditionnelle. En effet, dans ce cas aussi, une même donnée peut exister dans plusieurs processeurs qui pourraient potentiellement la modifier de façon inconsistente.

On trouve aussi des architectures dites NUMA (Non Uniform Memory Access). L'accès aux mémoires est **non-uniforme** en ce sens que le temps nécessaire pour lire ou écrire en mémoire varie s'il s'agit d'une mémoire locale ou non. L'architecture NUMA s'apparente beaucoup aux systèmes à mémoire distribuée, si ce n'est qu'elles offrent des mécanismes hardware de lecture et écriture dans les mémoires locales des autres processeurs (Remote-fetch). Dans le cas d'une machine purement à mémoire distribuée, ces mêmes transferts se feraient explicitement par échange de messages. L'architecture NUMA est schématisée sur la figure 2.9.

2.3.4 Architectures hybrides

Une autre classe d'architectures parallèles est le modèle hybride combinant hiérarchiquement, au niveau matériel, les aspects mémoires partagées et mémoires distribuées. Il s'agit de ce qu'on appelle communément des clusters de SMP ou

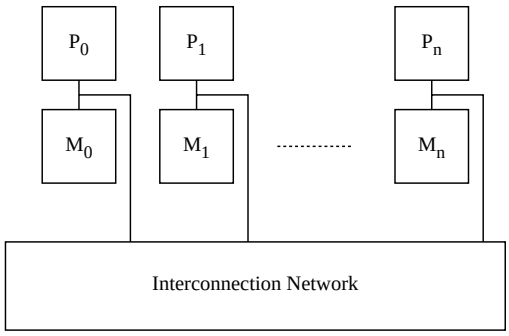


FIGURE 2.9 – *Schéma type de l'architecture NUMA (Non Uniform Memory Acces).*

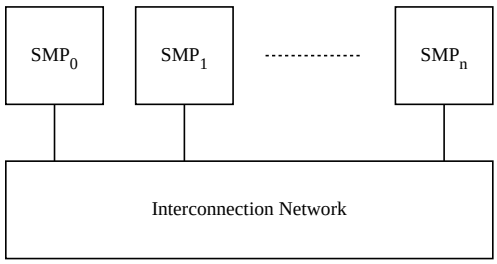


FIGURE 2.10 – *Schéma type de l'architecture hybride faite d'un ensemble de SMP ou de multi-coeur mis en réseau.*

des clusters de multicores, comme le montre la figure 2.10.

Au niveau global, c'est une architecture distribuée mais dont chaque noeud est lui même, au niveau local une machine parallèle à mémoire partagée. On combine ainsi l'avantage de la scalabilité des systèmes distribués avec la possibilité qu'offre les SMP d'avoir des noeuds individuels très puissants.

En principe, on peut imaginer que ces machines aient aussi un modèle de programmation mixte : une gestion multithread au niveau des noeuds et un échange de message entre les noeuds. En pratique, toutefois, c'est souvent MPI qui est utilisé déjà au niveau des processeurs et la structure hybride n'est plus visible pour le programmeur.

2.3.5 GPU

Les GPU (Graphical Processing Units) sont devenues une architecture très populaire pour le calcul HPC. Initiallement prévues pour le rendu d'images dans des applications graphiques exigentes (industrie du jeu), les cartes GPU ont un

FIGURE 2.11 – Architecture globale d'un GPU de Nvidia (source : documentation Nvidia).

degré de parallélisme élevé. Pour cette raison, dès le début des années 2000, des chercheurs les ont utilisées pour du calcul scientifique général (GPGPU, pour General Purpose GPU). Nvidia a par la suite encouragé cette direction en rendant la programmation plus aisée (CUDA) et en fournissant une architecture intégrant des opérations en double précision. Par ailleurs, l'environnement OpenCL offre un langage de programmation pour GPU qui n'est pas associé à l'architecture Nvidia, rendant ainsi l'utilisation des GPU plus portable.

D'un point de vue architectural, une carte GPU est un modèle hybride entre le SIMD et le MIMD, avec des mémoires partagées. Elle offre cependant une combinaison souvent peu claire de ces différents modèles, reflètant une origine historique où le rendu d'image est optimisé au dépend d'un modèle de programmation parallèle sûr et bien défini.

En gros, un GPU est une machine MIMD à mémoire partagée, dont les noeuds de calcul (baptisés SMX chez Nvidia) sont des machines SIMD à mémoire paratagée (voir figure 2.11). Il y a donc plusieurs hiérarchies de mémoire.

Entre-eux, les SMX sont asynchrones et partagent une mémoire globale commune. Cependant, les opérations atomiques contrôlant l'accès à cette mémoire ont été ajoutées seulement récemment. Une synchronisation entre les SMX a lieu lorsque les kernels (programmes tournant sur le GPU) se terminent, ce qui reste souvent la façon la plus courante de coordonner le parallélisme entre eux.

Les SMX (voir figure 2.12) sont des machines SIMD comprennant typiquement 192 coeurs synchronisés. Quatre groupes de 32 threads (appelés *warps* dans ce contexte) peuvent être exécutés en même temps, et deux instructions par thread peuvent être exécutée simultanément sur 2 coeurs, si les dépendances le

FIGURE 2.12 – Vue détaillée d'un SMX et de ses coeurs de calculs.

permettent. On notera aussi que des variables partagées ou privées peuvent être utilisées.

La programmation d'un GPU est définie par un code qui s'applique à un élément de donnée générique (par exemple un point d'une grille 2D). Dans ce code (appelé *kernel* comme indiqué plus haut), on a accès à toutes les autres variables, comme dans un modèle à mémoire partagée, mais sans pour autant avoir des primitives de coordinations garantissant par exemple une exclusion mutuelle dans une section critique.

2.4 Les ordinateurs vectoriels

Actuellement, on parle souvent d'exécution vectorielle dans les processeurs modernes. Il s'agit typiquement de la possibilité qu'offrent ces processeurs d'exécuter la même instruction simultanément sur plusieurs données, typiquement 4 éléments d'un vecteur à la fois. Dans la terminologie adoptée ici, cela s'apparente plutôt à une exécution SIMD.

Dans ce qui suit on va présenter l'architecture vectorielle à travers son évolution historique, ce qui la distingue clairement de la vision SIMD.

Les architectures vectorielles ont connu un grand succès depuis leur apparition en 1976 (Cray-1). Ce type de machine a été unanimement considéré comme le

standard de l'ordinateur à haute performance jusqu'à la fin des années 80. Les machines vectorielles ont été longtemps sans concurrence du point de vue de leur performance et il a fallu attendre l'avènement des grosses machines parallèles pour inverser cette tendance.

Il n'en reste pas moins que l'architecture vectorielle possède des caractéristiques importantes, particulièrement utiles et bénéfiques dans le domaine du calcul scientifique.

Les excellentes performances obtenues avec les machines vectorielles sont en partie dues à la technologie de pointe utilisée. Cela se reflète dans le prix élevé de ces machines qui ne sont accessibles qu'à des centres de recherche importants avec de gros besoins numériques. Un exemple célèbre d'une telle architecture est le Cray Y-MP qui combine le côté vectoriel avec la présence de plusieurs processeurs (habituellement utilisés pour augmenter le throughput du système plutôt que pour faire du parallélisme au niveau d'un même programme) Dans le même esprit, le Cray C-90 et le NEC SX-4 sont d'autres exemples d'architectures vectorielles multiprocesseurs qui ont eu un grand succès à la fin du 20ème siècle.

Cependant, la raison majeure des performances remarquables fournies par les processeurs vectoriels provient des particularités architecturales de ces machines. On peut distinguer trois éléments essentiels qui définissent une architecture vectorielle :

— Les données sont organisées en vecteurs comprenant typiquement 128 ou 256 valeurs ou mots machines (de 64 bits, en général). Ces vecteurs sont traités comme des entités élémentaires par le flot de contrôle. Le travail relatif à chaque instruction est donc important puisqu'il est répété pour chaque élément du vecteur. Par conséquent, le débit d'instructions nécessaire est réduit et il n'y a pas de risque de goulet d'étranglement dans le flot d'instructions. Pour cette raison, une architecture de type RISC ne s'est pas développée pour les processeurs vectoriels.
— Chaque opération qui est effectuée dans une machine vectorielle tire parti au maximum de la mise en pipeline.
— Un parallélisme au niveau des unités fonctionnelles est présent, comme dans les architectures superscalaires. En particulier, il y a des unités FP (floating point) et FX (integer) séparées. Il peut aussi y avoir plusieurs paires de ces unités.

Toutes ces spécificités architecturales additionnées permettent un speedup considérable, souvent supérieur à 100. Cependant, ce speedup est obtenu au prix d'une technologie de compilation avancée et coûteuse.

Globalement, une architecture vectorielle s'avère très complexe. Ici, nous nous contenterons de décrire en quelques détails les principes de base énoncés ci-dessus et d'illustrer les problèmes fondamentaux qui font obstacle à la vectorisation.

2.4.1 Vecteurs

Dans une architecture scalaire classique, le temps nécessaire pour calculer l'adresse en mémoire des données, les faire venir au CPU et, finalement les replacer en mémoire est de loin supérieur au temps de calcul proprement dit.

Si la même opération est répétée sur plusieurs données, on peut gagner un temps considérable en groupant les données en vecteur : on ne fait qu'un accès mémoire de toutes les données à la fois dans un registre vectoriel, comme présenté sur la figure 2.13. S'il y a plus de données dans le problème que ne peut en contenir le registre, l'opération est alors répétée sur les blocs de données correspondant à la taille du registre (par exemple 256 mots). Il faut remarquer que des registres scalaires supplémentaires existent aussi pour réaliser toutes les opérations non vectorielles.

La figure 2.14 illustre le gain de temps obtenu par l'emploi de registres vectoriels pour accéder à la mémoire et masquer le temps de latence.

L'accès «simultané» à toutes les données d'un vecteur n'est possible que si les données sont placées dans des **bancs** mémoire différents (c'est à dire des portions distinctes de la mémoire centrale qui peuvent être accédées en même temps). Si ce n'est pas le cas, on parle de conflit de banc et la vectorisation est entravée. En général, le compilateur se charge de bien répartir les données à traiter dans la mémoire, mais cela n'est pas toujours possible si les mêmes données sont utilisées de façon différente dans plusieurs parties du programme (par exemple, on n'accède pas forcément les données consécutivement, avec un pas de 1).

Si T_{fetch} et T_{store} sont les temps pour prendre et stocker une donnée en mémoire, et T_{cpu} le temps nécessaire pour faire un calcul sur cette donnée, la répétition de l'opération sur N valeurs par un processeur scalaire prend un temps

$$T_{scalar} = NT_{fetch} + NT_{cpu} + NT_{store}$$

Par contre, dans une architecture vectorielle, seul le temps de calcul est répété N fois. On obtient donc

$$T_{vector} = T_{fetch} + NT_{cpu} + T_{store}$$

ce qui est d'autant plus favorable que T_{fetch} et T_{store} sont grands par rapport à T_{cpu}. Par exemple, le speedup obtenu, T_{scalar}/T_{vector} peut être estimé ainsi

$$\frac{T_{scalar}}{T_{vector}} = \frac{NT_{fetch} + NT_{cpu} + NT_{store}}{T_{fetch} + NT_{cpu} + T_{store}} \approx \frac{T_{fetch} + T_{store}}{T_{cpu}}$$

en supposant que $T_{fetch} \approx T_{store} >> T_{cpu}$ et que, avec N assez grand, $NT_{cpu} >> T_{fetch}$.

2.4.2 Pipeline

Dans une architecture vectorielle, la technique du pipeline est utilisée partout où elle est possible. Chaque instruction complexe est subdivisée en instructions

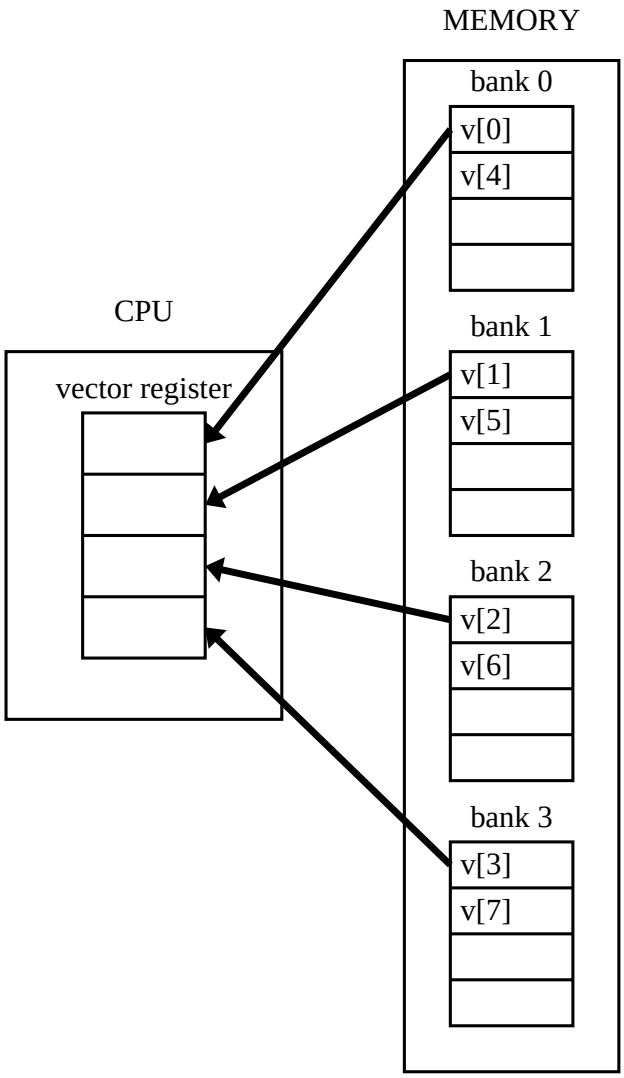


FIGURE 2.13 – Illustration d'un registre vectoriel qui peut recevoir en une fois plusieurs composantes d'un vecteur de donnée, grâce à une accès parallèle aux données réparties en mémoire dans différents bancs mémoire. Ici on considère un registre vectoriel à 4 positions, une mémoire structurée en 4 bancs et un vecteur de 8 éléments. Dans les architectures vectorielle haut de gamme, ces tailles sont bien plus grandes, avec des possibilités de traiter des vecteurs de taill 256 ou 512. Cette figure montre une opération de lecture en mémoire, mais en reversant les flèches, on obtient une écriture simulatanée en mémoire.

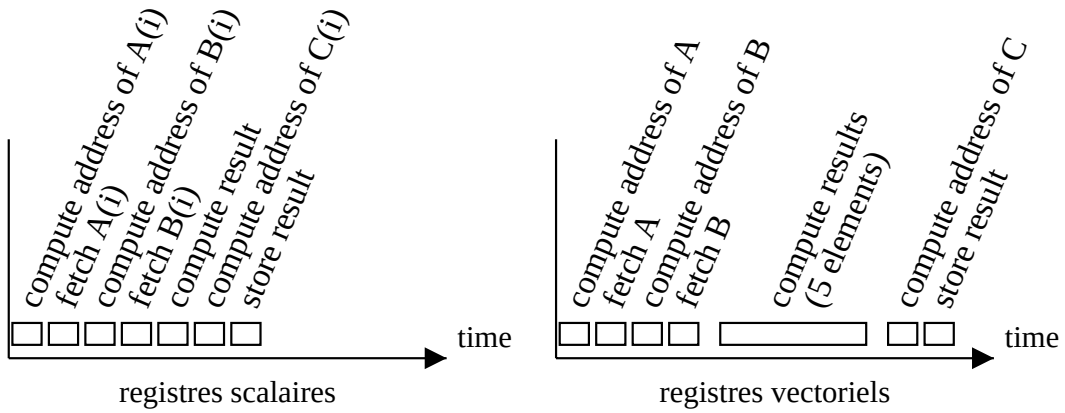


FIGURE 2.14 – *Dans une architecture vectorielle, l'accès à la mémoire se fait par blocs de données, grâce à des registres vectoriels. Cette technique, applicable si la même opération est effectuée sur de nombreuses données, permet de ne payer la pénalité d'un accès mémoire qu'une seule fois pour tout le vecteur. Ici, on compare les performances réaslisées pour l'opération* `C=A+B` *pour un accès par registre scalaire et vectoriel. A gauche, on calcule une seule valeur de* `C` *alors qu'à droite on calcule tout le vecteur (qui, sur la figure comporte 5 éléments).*

élémentaires prenant si possible un cycle d'horloge chacune. Chaque opération est ainsi un enchaînement de q instructions de base qui se suivent et qui forment le pipeline.

Si plusieurs données doivent être traitées à la suite, elles peuvent s'enchaîner (recouvrement d'instructions) dans le pipeline sans attendre que ce dernier soit entièrement vidé de la précédente donnée avant d'entrer.

Donc, si le pipeline a q niveaux, on aura jusqu'à q valeurs simultanément présentes dans le pipeline et, après les q premiers cycles nécessaires à remplir le pipeline, on aura un nouveau résultat par cycle.

La difficulté pour transformer une opération donnée sous forme d'un pipeline est de s'assurer que la subdivision peut se faire en sous-opérations de longueurs identiques. Si ce n'est pas le cas, il faut subdiviser plus finement la sous-opération la plus lente ou, si cela n'est pas possible, utiliser des solutions technologiques ou architecturales pour accélérer les étages trop lents.

L'avantage en performance d'un pipeline est un speedup d'un facteur q. En effet, une instruction prenant q cycles sur une machine scalaire d'horloge τ demandera un temps

$$T_{scalar} = Nq\tau$$

si elle est répétée N fois. Par contre, si ces q cycles sont subdivisés en q niveaux d'un pipeline, un processeur vectoriel de même cycle horloge ne prendra que

$$T_{vector} = q\tau + (N-1)\tau$$

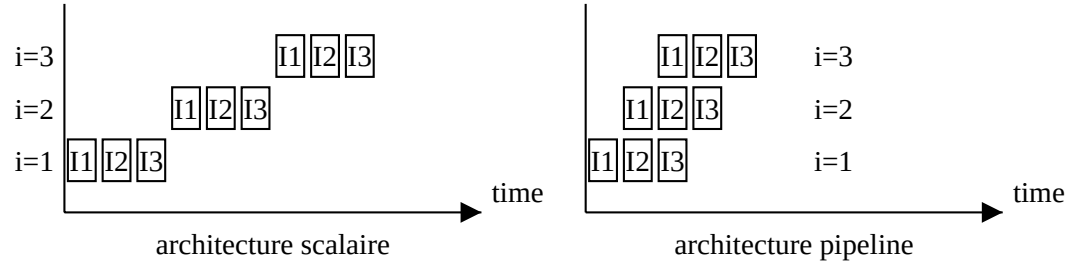


FIGURE 2.15 – *La technique du pipeline permet d'enchaîner les opérations sur plusieurs données. Ici on considère le cas d'une opération qui est décomposée en 3 instructions de base notées* `I1`*,* `I2` *et* `I3`*. On suppose que la même opération est répétée pour trois données, indicées par* `i=1`*,* `i=2` *et* `i=3`*. Dans une machine scalaire, on attend que ces 3 sous-instructions soient terminées avant de considérer la valeur suivante. Dans l'architecture vectorielle, les données peuvent se superposer dans le CPU.*

On attend q cycles pour la première valeur et, ensuite, à chaque cycle, on récolte un autre des $N-1$ résultats attendus. Pour N grand, $T_{vector} \approx (N-1)\tau$ et on obtient bien le speedup annoncé.

La figure 2.15 illustre l'avantage du pipeline dans le cas d'une opération se décomposant en $q = 3$ niveaux. On remarque sur cette figure que si le résultat du calcul pour $i = 1$ est nécessaire pour commencer celui pour $i = 2$ (c'est à dire qu'il y a dépendance entre les opérations), la structutre en pipeline n'est plus possible : l'enchaînement est rompu et le pipeline doit se vider et se reremplir.

2.4.3 Unités fonctionnelles

Une architecture vectorielle se caractérise par la présence de plusieurs unités fonctionnelles qui peuvent être actives en même temps. Il y aura ainsi une ou plusieurs unités FPU pour les opérations en virgule flottante et une ou plusieurs unités FX pour les calculs sur les entiers (virgule fixe). Ce parallélisme au niveau du processeur permet de traiter simultanément des segments différents du programme et aussi de partager le calcul sur un grand vecteur sur deux unités.

Il faut aussi remarquer que les additions et multiplications se font sur des unités FP distinctes, ce qui permet d'enchaîner sans rompre le pipeline une opération du type `D(i)=C(i) + B(i)*A(i)`. Cette particularité est très utile lors de calculs intensifs sur les matrices.

Notons encore que les opérations scalaires sont réalisées par des registres et unités d'exécution spécifiques, ce qui augmente encore le degré de parallélisme dans le processeur.

La figure 2.16 montre un diagramme schématique d'une architecture vectorielle.

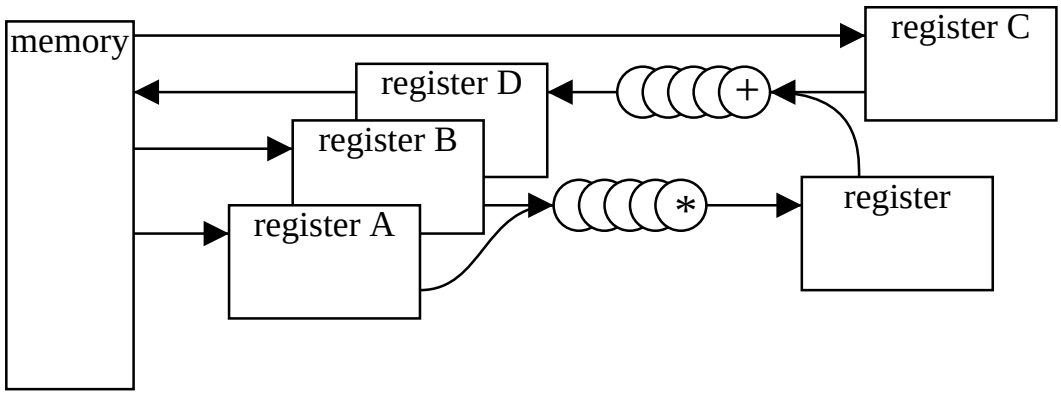


FIGURE 2.16 – *Schéma d'une architecture vectorielle avec plusieurs registres vectoriels et deux unités FP (en pipeline) actives en même temps, l'une pour la multiplication et l'autre pour l'addition. Cette figure illustre le calcul de* `D=A*B+C`*, où* `A`*,* `B`*,* `C` *et* `D` *sont des vecteurs.*

2.4.4 Problèmes liés à la vectorisation

Pour pouvoir tirer parti des ressources d'un processeur vectoriel, il faut que le problème y soit adapté. C'est le cas de nombreux problèmes de calcul scientifique qui nécessitent le traitement de grandes structures de données régulières.

Pourtant, même dans ce cas, les obstacles à la vectorisation sont nombreux. Heureusement il existe souvent des astuces de programmation qui permettent de contourner les difficultés. Ce paragraphe se propose de décrire certaines situations typiques et la façon de les réexprimer pour favoriser la vectorisation. Précisons à cet effet que les compilateurs modernes qu'on trouve sur les machines vectorielles haut de gamme sont souvent très élaborés et corrigent automatiquement la formulation du programmeur pour permettre la vectorisation du code.

Branchement : Une situation courante qui empêche la vectorisation est l'interruption du flot de données due à la présence d'un branchement `if`, par exemple. L'exemple ci-dessous en est une illustration :

```
do i=1,n
   if A(i)>0 then
       B(i)=sqrt(A(i))
   else
       B(i)=0
   endif
enddo
```

L'instruction `if` empêche le programme de traiter la totalité des éléments de `A` de façon identique (il faut un traitement différent pour les valeurs négatives et

positives), et cela est contraire au bon fonctionnement des unités vectorielles.

La solution est de réécrire ce fragement de code de la façon suivante

```
do i=1,n
   cond(i)= A(i)>0
enddo

do i=1,n
   B(i)=sqrt(cond(i)*A(i))
enddo
```

De cette façon, on a décomposé la boucle en deux parties qui chacune assure un traitement identique des données et, donc, une mise en vecteur possible.

Dépendances de données : Un autre problème qui empêche la vectorisation d'une boucle est l'existence de **dépendances** entre les données d'un vecteur. Par exemple, le code suivant (qui correspond à une multiplication matrice-vecteur)

```
do i=1,n
   do j=1,m
      y(i)=y(i) + A(i,j)*x(j)
   enddo
enddo
```

contient une dépendance car la valeur de `y(i)` à l'itération `j+1` dépend du résultat de l'itération `j`. Il faut donc totalement terminer le calcul de l'itération `j` avant de pouvoir commencer celui de l'itération `j+1`. Cela rend évidemment impossible un calcul en pipeline sur des vecteurs.

La façon de contourner le problème dans ce cas est d'inverser l'ordre des boucles sur `i` et `j`

```
do j=1,m
   do i=1,n
      y(i)=y(i) + A(i,j)*x(j)
   enddo
enddo
```

Sous cette forme, il n'y a pas de dépendance dans la boucle sur `i` et la vectorisation peut se faire. Dans cette manière de faire, on calcule les contributions du produit matrice-vecteur, progressivement pour chaque élément du vecteur colonnne `y`. Par contre dans la première disposition des boucles, chaque élément est totalement calculé avant de passer au suivant.

D'une façon générale, il est important de remarquer que l'ordre des indices de deux boucles imbriquées influence aussi les performances par rapport à l'accès mémoire. En Fortran par exemple, les éléments d'un tableau bidimensionnel

sont stockés en mémoire par colonnes (`A(i,j)` et `A(i+1,j)` sont consécutifs en mémoire) et la forme en "`ji`" est donc largement préférable et évitera des conflits de bancs au moment des chargements des registres vectoriels.

Déroulement de boucles : L'opération de réduction

```
s=0
do i=1,n
   s=s+y(i)
enddo
```

est fréquente dans de nombreuses applications. Elle ne permet pas la vectorisation, de nouveau en raison de la dépendance de la variable `s` au cours de la boucle.

La technique dite de déroulement de boucle (*loop unrolling*) est utilisée pour permettre néanmoins une certaine part de vectorisation. On réécrit la boucle de la façon suivante

```
s=0

do i=0,n,q
   do k=1,q
     t(k)=t(k)+y(i+k)
   enddo
enddo

do k=1,q
   s=s+t(k)
enddo
```

La boucle extérieur progresse par pas de `q` et la variable `t` accumule, sur `q` positions, les morceaux de somme, comme indiqué sur le schéma de la figure 2.17.

La boucle intérieure sur `k` se vectorise maintenant car il n'y a plus de dépendance. Par contre, la dernière boucle, celle qui calcule `s` ne se vectorise toujours pas. Mais elle est beaucoup plus courte qu'avant si `q` est beaucoup plus petit que `n`. On a donc intérêt à choisir `q` aussi petit que possible. Cependant `q` ne doit pas être choisi inférieur au nombre d'étages q_+ du pipeline de l'unité qui réalise l'addition, sans quoi, on ne tirerait pas pleinement parti de la vectorisation de la boucle interne. En général, `n` est ainsi largement supérieur à `q` et l'opération de réduction se fait plus efficacement avec le déroulement de la boucle.

Une façon encore plus astucieuse de faire est de choisir `q` grand (afin de mieux vectoriser la boucle) et ensuite de répéter la même procédure plusieurs fois sur la boucle non-vectorisable :

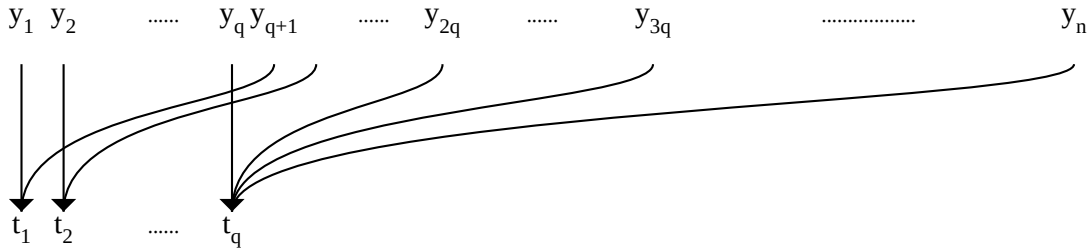


FIGURE 2.17 – *Manière de formuler une opération de réduction avec la technique de déroulement de boucle. La variable* `t` *stocke les valeurs de* `y` *selon la structure indiquée par les flèches.*

```
do k=1,q
   s=s+t(k)
enddo
```

En effet, en choisissant par exemple `q=n/2` on se ramène au même problème de réduction, mais deux fois plus petit. On peut donc recommencer la démarche de déroulement avec `q=n/4` et ainsi de suite, jusqu'à ce que `q` atteigne la valeur limite de vectorisation $\texttt{q=q}_+$.

2.5 L'architecture Data-flow

Les ordinateurs les plus courants sont basés sur une architecture dite **control-driven**, c'est à dire que l'exécution suit l'ordre des instructions spécifiée par le programme. C'est l'idée de base du modèle de von Neumann qui nous est très familière et qui correspond aussi au modèle de base des architectures parallèles SIMD et MIMD.

Mais ce n'est pas la seule solution et d'autres modèles calculatoires ont été réalisés en hardware, comme les architectures *data-driven* (data-flow) et *demand-driven* (machine à réduction) qui toutes deux font jouer un rôle prépondérant aux données.

Dans l'architecture data-driven, c'est la disponibilité des données qui déclenche les instructions correspondantes. Ainsi, l'instruction

$$\texttt{a} = \texttt{b} + \texttt{c}$$

sera exécutée dès que les valeurs de `b` et `c` seront calculées. Ensuite, ce résultat est dupliqué autant que nécessaire pour toutes les autres intructions qui en ont besoin. Il n'y a plus de compteur d'instruction qui, à un moment déterminé, décide de l'exécution et l'ordre des instructions du programme n'est pas respecté.

Une structure en arbre, avec des opérations arithmétiques en chaque noeud et des données placées sur les feuilles est une bonne façon de représenter un calcul

dirigé par les données : les données "s'écoulent" des feuilles vers les noeuds et sont combinées lorsqu'elles s'y croisent.

Cette approche est intéressante pour exploiter naturellement le parallélisme à grain fin entre les intructions. Il n'y a pas d'analyse de dependance explicite à faire puisque que la présence des données implique que l'opération est valide. De la sorte, toutes les instructions réalisables en même temps peuvent être activables.

Mais, d'un point de vue hardware, il faut réaliser un mécanisme qui détecte la disponiblité des données et leur faire correspondre les instructions où elles sont impliquées. Une solution est de repérer chaque instruction par un tag relatif aux données qu'elle met en jeu. Quelques réalistions pratiques de machine data-flow existent, tel le *tagged-token computer* du MIT développé au début des années 80.

Les machine demand-driven (aussi appelées machines à réduction) sont un peu semblables si ce n'est qu'elles déclenchent les opérations lorsqu'un résultat est requis. Par exemple, l'instruction

$$\mathtt{a} = \mathtt{b} + \mathtt{c}$$

ne sera considérée que quand la valeur de `a` est demandée par une autre instruction.

De nouveau, une structure en arbre, avec des opérations arithmétiques en chaque noeud et des données placées sur les feuilles permet de représenter un calcul régit par la demande : le résultat est d'abord demandé pour la racine, ce qui implique que les deux valeurs filles soient connues. Récursivement, cela déclenche les opérations depuis les feuilles.

La figure 2.18 illustre le calcul de l'espression

$$\mathtt{a} = (\mathtt{b} + 1) * (\mathtt{b} - \mathtt{c})$$

en modes control-driven, data-driven et demand-driven. On voit que le parallélisme disponible entre les instructions

$$\mathtt{i1} : \mathtt{b} + 1 \qquad \text{et} \qquad \mathtt{i2} : \mathtt{b} - \mathtt{c}$$

est naturellement pris en compte dans les systèmes data-driven et demand-driven.

2.6 Parallélisme interne pour les microprocesseurs

De nombreuses techniques **architecturales** ont été mises en oeuvre dans les processeurs afin d'en augmenter la performance. Par exemple, la mémoire cache est un composant qui permet de réduire le goulet d'étranglement de von Newmann en diminuant les besoins d'accès à la mémoire centrale.

Une autre direction est d'amener du parallélisme directement dans le CPU afin d'exécuter plusieurs instructions simultanément. Les architectures qui permettent

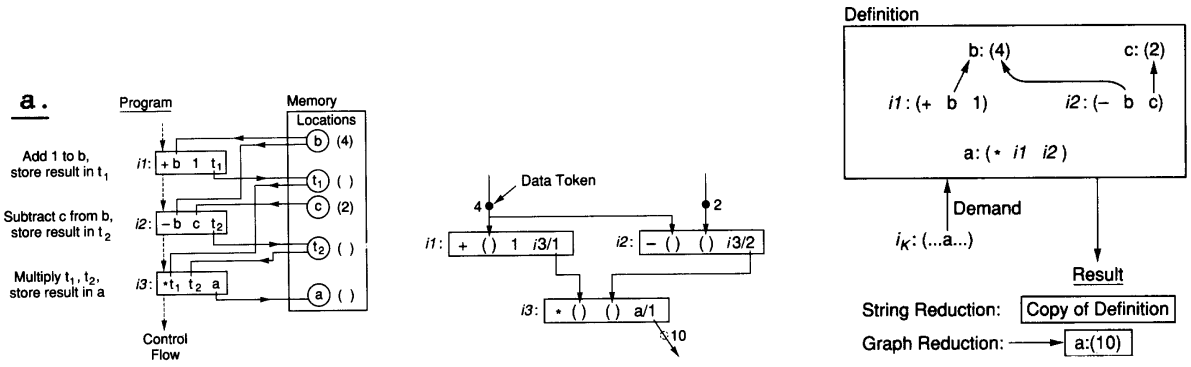


FIGURE 2.18 – *Illustration d'un même calcul avec trois architectures différentes (cf Almasi/Gottlieb 1989) : (gauche) control-driven, (milieu) data-driven et (droite) demand-driven. Les indications* `i3/1` *et* `i3/2` *indiquent que le résultat de l'opération est utilisé par l'instruction* `i3` *comme premier ou deuxième argument.*

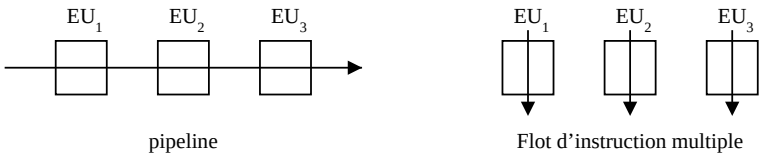


FIGURE 2.19 – *Le pipelining et les unités fonctionnelles multiples sont les solutions de base des processeurs ILP. Les entités notées EU sont les unités fonctionnelles ou les "Execution Units." Les flèches indiquent le flot d'instruction.*

du parallélisme au niveau des instructions sont appelés **"ILP-processors"** où ILP signifie Instruction Level Parallelism.

Les deux façons génériques d'introduire du parallélisme au niveau des instructions est le **pipelining** et l'utilisation d'**unités fonctionnelles multiples**, comme l'indique la figure 2.19. Il s'agit donc d'un parallélisme à grain très fin, basé sur les stratégies générales discutées au chapitre 1.

Dans le pipelining, plusieurs unités fonctionnelles EU_i (Execution Units) sont chaînées pour pouvoir traiter plusieurs instructions à la suite, sans attendre que le traitement de la première soit terminé. Les différentes étapes du pipeline sont par exemple : *instruction fetch*, *instruction decode*, *execute* et *write-back*. Cette solution donne ce qu'on s'appelle un ILP scalaire car le flot d'instruction est séquentiel.

Dans le cas d'unités fonctionnelles multiples, on a un flot d'exécution multiple et les deux architectures qui mettent en oeuvre cette approche sont les machines **VLIW** (Very Long Instruction Word) et les processeurs superscalaires. Dans les deux cas, les EU peuvent être «pipelinées». Dans le cas des VLIW, c'est le

compilateur qui construit avec les instructions séquentielles du programme, des mots multi-instructions alors que dans le cas superscalaire, le parallélisme est mis en œuvre à un niveau hardware par un principe data-flow : les EU dont les données d'entrées sont disponibles sont activées.

En plus de l'introduction de l'ILP, un autre concept est considéré pour améliorer l'architecture des processeurs : on cherche à adapter la conception du processeur au comportement du programme. Cela se traduit par une simplification du jeu d'instruction des processeurs ainsi que des stratégies de prédictions lors d'instructions de branchements conditionnels.

Les processeurs RISC (Reduced Instruction Set Computer) se caractérisent par une simplification de leur jeu d'instructions conduisant à un gain en performance. Ils se distinguent ainsi des processeurs CISC (Complex Instuction Set Computer) qui offrent un jeu d'intructions beaucoup plus riche mais où chaque instructions prend plusieurs cycles machine pour s'exécuter.

Dans ce contexte, une mesure fondamentale des performances des processeurs est le CPI (Cycle Per Instruction), c'est à dire le nombre de cycles nécessaire à l'exécution d'une instruction. On veut évidemment diminuer le CPI pour augmenter les performances : moins de cycles par instruction signifie plus d'instructions par cycle. Le but du RISC est d'atteindre CPI=1 pour la plupart des instructions.

On peut représenter les différentes familles de processeurs et les tendences de l'évolution dans le diagramme de la figure 2.20. Dans cette figure, on peut dire que l'axe vertical est l'axe des **progrès architecturaux** alors que l'axe horizontal est l'axe des **progrès technologiques**. En effet l'augmentation de la vitesse d'horloge est principalement due aux progrès technologiques et aussi un peu à la simplification matérielle des processeurs RISC par rapport aux CISC. Par contre, la diminution du CPI est l'oeuvre des architectures ILP. Ainsi, avec plusieurs instructions émises en parallèle, on peut atteindre un CPI<1.

Les processeurs CISC scalaires sont caractérisés par un CPI grand et datent d'une époque où la fréquence d'horloge était de l'ordre de 100 Mhz. A l'autre extrêmité de l'axe, correspondant aux CPU les plus performants, on trouve les processeurs vectoriels qui cumulent une technologie de pointe, du pipelining à grande échelle et des unités fonctionnelles multiples.

La mise en œuvre du parallélisme au niveau des instructions permet de faire baisser le CPI en dessous de 1. Mais les dépendences possibles entre les instructions successives d'un programme réduisent souvent le potentiel du parallélisme interne. Plusieurs téchniques ont été développées pour détecter ces dépendances et les gérer au mieux, directement au niveau du hardware (p. ex. le shelving et l'exécution dans le désordre).

Une autre difficulté est liée à la presence de branchements (IF et autre structures de contôle). Les branchements sont des instructions simples mais elles représentent un grand obstacle au parallélisme interne des processeurs superscalaires car ils provoquent une dépendance de contrôle : on ne sait plus quelle sera la prochaine instruction à exécuter. Il faut donc attendre que la condition du bran-

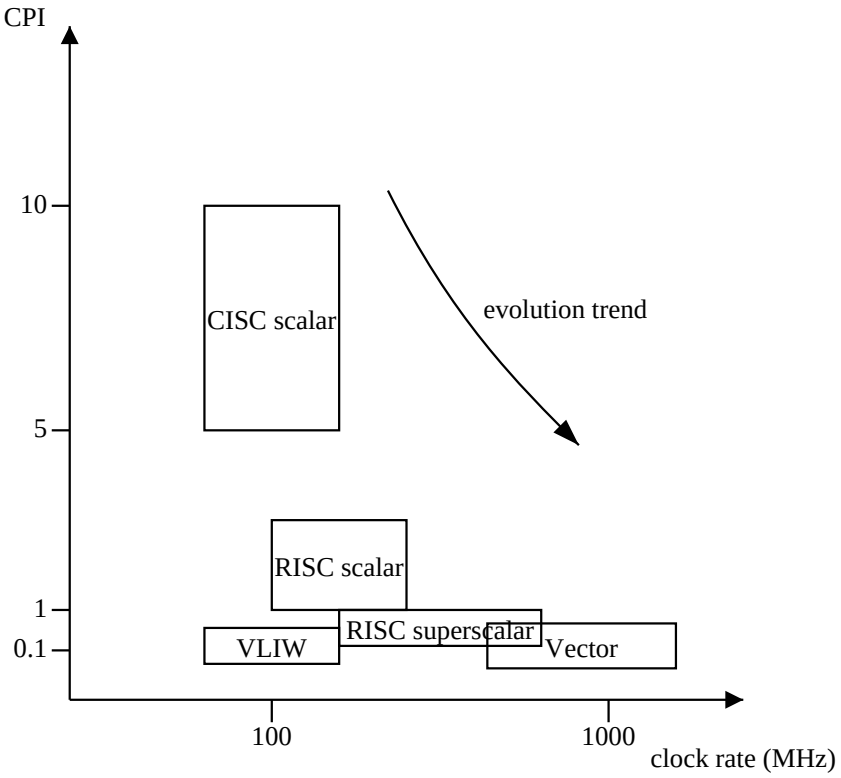


FIGURE 2.20 – *Tendance de l'évolution des familles de processeurs au début des années 2000. Avec les nouvelles générations, les machines ont une fréquence d'horloge qui augmente et un CPI qui diminue.*

chement soit calculée et, si le branchement est pris par l'exécution du programme, il faut aussi calculer l'adresse de la nouvelle instruction. Pendant ce temps, les unités fonctionnelles restent inactives.

Quand on sait qu'environ 25% des instructions sont des branchements, on mesure l'importance de concevoir des techniques qui ne bloquent pas le parallélisme. Parmi les solutions proposées, la plus courante est le **branchement spéculatif** qui fait une prédiction sur le chemin que va suivre le programme. Cette prédiction est typiquement basée sur l'historique du comportement du programme, ou sur des considérations statistiques. En cas d'erreur de prédiction (qu'on espère être aussi rare que possible), le processeur oublie les résultats intermédiaires et repart sur l'autre branche.

Chapitre 3

Réseaux d'interconnexion

3.1 Introduction et définitions

Un ordinateur parallèle se compose de processeurs, de mémoires et d'un réseau d'interconnexion reliant ces éléments entre eux. Ce réseau permet la communication (l'échange de données intermédiaires ou d'informations) entre les différentes unités de traitement et joue, pour cette raison, un rôle primordial dans l'architecture des machines parallèles. Un bon réseau doit, à un prix raisonnable et avec fiabilité, être capable de transmettre assez rapidement les données là où elles sont requises.

Dans de nombreux problèmes, la phase de communication peut être aussi importante que la phase de calcul proprement dite. Un mauvais réseau de communication peut être responsable d'une dégradation importante des performances, pouvant conduire l'utilisateur à se tourner vers une autre machine. Dans une machine parallèle, les processeurs sont fortement couplés et on s'attend à des connexions inter-processeurs à fort débit (de l'ordre du GigaBytes/s entre pairs de processeurs) et faible latence (de l'ordre de la micro-seconde).

3.1.1 Topologie et propriétés des réseaux d'interconnexion

L'idéal serait évidemment d'avoir un système de connexions qui relie chaque processeur avec tous les autres mais cela n'est possible que pour des machines avec peu de processeurs. Le prix et les problèmes technologiques qui en résulteraient rendent cette approche irréalisable pour une architecture qu'on souhaite massivement parallèle.

En pratique, les processeurs sont reliés entre eux selon une topologie déterminée. On verra de nombreux exemples concrets tout au long de ce chapitre. La **topologie** du réseau définit la structure des liens entre les processeurs. Certains processeurs sont adjacents s'il existe une connexion directe qui les relie. D'autres sont éloignés s'il faut passer par plusieurs processeurs intermédiaires pour trouver un chemin qui les relie. La figure 3.1 donne un exemple d'une topologie de

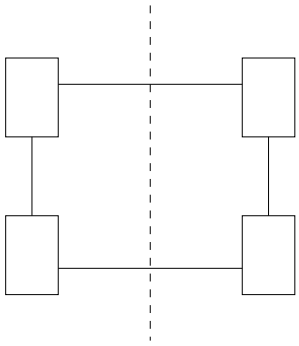


FIGURE 3.1 – *Exemple de réseau d'interconnexion entre 4 processeurs. La ligne pointillée divise le réseau en deux sous systèmes identiques, ce qui donne une largeur bissectionnelle de 2.*

connexion avec 4 processeurs.

On peut comparer un réseau d'interconnexion à un graphe : les **noeuds** du graphe sont les processeurs ou des modules mémoires ; les arcs sont les câbles de connexion qui peuvent être bidirectionnels ou ne permettre des échanges que dans un sens à la fois (semi-duplex). La théorie des graphes permet d'ailleurs de formaliser plusieurs problèmes importants liés à la topologie des réseaux : comment trouver les chemins de longueur minimale entre deux noeuds, chercher un chemin qui relie une et une seule fois chaque noeud, etc.

Ces questions sont liées à la nécessité de faire transiter le plus efficacement possible les messages entre divers noeuds. Cette opération importante s'appelle le **routage**. Elle dépend évidemment de la topologie du réseau considéré. On distingue deux types de topologies dans les machines parallèles :

— Les topologies statiques dans lesquelles les connexions physiques entre les noeuds du réseau sont fixes et ne changent plus après la construction de la machine. Dans ce cas, un noeud est typiquement constitué d'un processeur et d'un routeur.

— Les topologies dynamiques qui permettent une reconfiguration des connexions entre les noeuds du réseaux. On peut ainsi établir des chemins directs entre certaines paires de noeuds au détriment d'autres. Cette reconfiguration peut se faire totalement dynamiquement, en cours d'exécution d'un programme ou, plus simplement, être spécifiée avant exécution par un fichier de configuration. Dans ce cas, les noeuds internes de la topologie sont des switches et les processeurs sont les noeuds externes.

Les performances de chaque réseau sont déterminées par un ensemble de critères et de propriétés. Nous décrivons ci-dessous les principales. Il faut remarquer que certaines de ces propriétés sont plus ou moins significatives selon qu'il s'agit de topologies statiques ou dynamiques.

Le diamètre D d'un réseau est la longueur du chemin minimal (en nombre de noeuds à traverser) qui relie les deux noeuds les plus éloignés.

La connectivité qui est le nombre de liens de chaque noeud, ou encore le nombre de voisins qui lui sont directement accessibles. Ce nombre s'appelle aussi le **degré** de chaque noeud. En anglais on parle parfois du «radix» d'un noeud pour indiquer le nombre de liens dont il dispose.

La latence qui est le temps (exprimé en micro-seconde, typiquement) de transit maximum d'un signal à travers le réseau. C'est aussi une mesure de la distance qui relie les deux points les plus éloignés du réseau. Cependant, à la différence du diamètre, c'est une grandeur qui ne dépend pas uniquement de la topologie mais aussi de la technologie utilisée : longueur et nature des connexions, rapidité des circuits, etc. De plus, la latence inclus souvent aussi le temps t_s dit de *startup* qui est le temps de préparation d'un message : création de l'entête, de la fin de message, des codes correcteurs d'erreurs (ECC), ainsi que le temps d'activation du dispositif d'envoi. Le temps t_s n'intervient qu'une fois pour chaque message. L'ordre de grandeur du temps de latence dans une architecture parallèle est de plusieurs nanosecondes pour la partie hardware du réseau et quelques μsec en incluant l'overhead du logiciel. La figure 3.2 compare ce temps de latence (noté "remote memory" pour indiquer qu'on accède à des données hors du processeur) aux autres temps de latence caractéristiques d'un ordinateur.

La bandwidth qui mesure le débit d'information (généralement en MByte/s) qu'un noeud peut transmettre dans le réseau. La bandwidth dépend de la nature physique des liens existants, de leur nombre (largeur des canaux) et de la vitesse d'horloge du réseau. Une valeur de 500 Mbytes/s par connexion, est considéré comme une performance moyenne.

La largeur bissectionnelle qui est une mesure topologique globale du débit d'information qui peut circuler dans un réseau d'interconnexion. Pour la déterminer, on divise le réseau en deux moitiés identiques et on compte le nombre de canaux ou de connexions qui vont d'une moitié vers l'autre. Cela donne une information réaliste sur le volume d'informations qui peut être échangé dans le réseau. La figure 3.1 illustre cette construction. Il y a parfois plusieurs manières de diviser un réseau en deux moitiés égales. Dans ce cas la largeur bissectionnelle est définie pour la subdivision la plus défavorable, c'est à dire celle pour laquelle le nombre de canaux reliant les deux moitiés est le plus petit. On définit aussi la bandwidth bissectionnelle comme la largeur bissectionnelle multipliée par la bandwidth d'une connexion. Par exemple, le Cray T3D, avec 256 processeurs offrait une bandwidth bissectionnelle d'environ 20Gbytes/s.

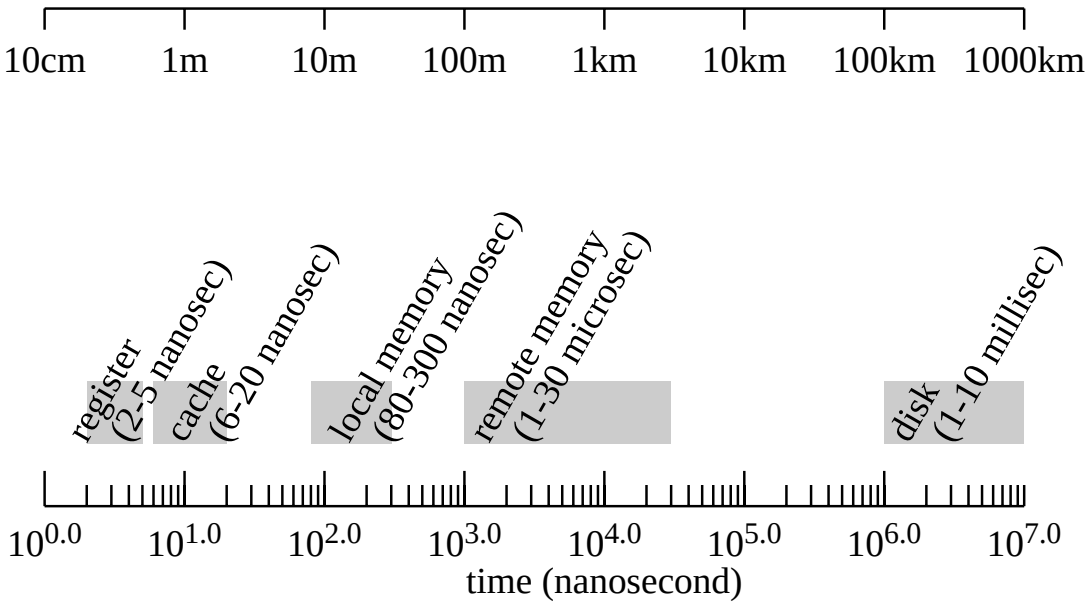


FIGURE 3.2 – *Ordre de grandeur des temps d'accès caractéristique (latence) aux différentes ressources d'un ordinateur (en l'an 2000). Pour mieux appréhender les écarts qui existent entre ces latences, l'échelle en haut de l'image donne une représentation en terme d'une distance à parcourir. On a choisi arbitrairement d'associer 10 cm au temps d'une nano-seconde (cela correspond en fait au parcours effectué en se déplaçant à 1/3 de la vitesse de la lumière). Cette échelle montre que si les données dans la mémoire cache sont à portée de main, celles qui sont sur le disque ont un voyage long comme la traversée de la Suisse.*

La scalabilité qui est la possibilité d'augmenter la taille du réseau en ajoutant des processeurs. Pour des raisons pratiques, cela implique que le degré de chaque noeud est indépendant de la taille du système (car le nombre de canaux dont dispose un processeur est fixe). La scalabilité décrit ici la nature modulaire du réseau.

Le prix (ou la complexité) de fabrication du réseau. C'est essentiellement le nombre de composants dans le réseau en fonction du nombre de noeuds présents.

La fonctionnalité du réseau qui est sa capacité de combiner des messages allant vers une même destination et à gérer des conflits de routage.

La capacité qui mesure le nombre maximum de messages pouvant être contenus en même temps dans le réseau.

La symétrie. Un réseau est dit symétrique s'il est le même vu de chaque noeud.

3.1.2 Le routage

Le routage est un algorithme adapté à un réseau donné et définissant une recette permettant aux noeuds de diriger un message *rapidement* vers sa destination finale. Cet algorithme utilise les propriétés mathématiques et géométriques du réseau pour offrir un routage dit **minimal**, c'est à dire un choix de chemins les plus courts entre la source et la destination.

Le circuit chargé du routage est appelé le **routeur**. Un message est composé tout d'abord d'une adresse de destination, puis d'un ensemble de données à transmettre (corps du message). Le routeur, en fonction de cette adresse et de l'algorithme de routage qu'il connaît, redirige le message sur un noeud voisin, afin de s'approcher de la destination finale. Le routeur doit être capable de gérer des conflits de chemin (deux messages désirant emprunter simultanément le même canal) c'est à dire, en pratique, de pouvoir mémoriser temporairement un message pour l'envoyer plus loin une fois la voie libre. L'algorithme de routage, en raison des conflits potentiels peut devenir assez complexe. Il faut être sûr qu'aucun message ne puisse être bloqué éternellement dans le réseau : risque de **deadlock** causé par deux ou plusieurs messages se bloquant mutuellement ou de **livelock** indiquant un message tournant sans fin sans jamais arriver à destination.

Plus loin dans ce chapitre, on décrira les algorithmes élémentaires de routage pour plusieurs réseaux différents, en supposant toutefois qu'un seul message circule à la fois. Les algorithmes de routage utilisés en pratique doivent en plus faire face à tous les cas de conflits qui peuvent se produire et c'est un problème difficile que de s'assurer théoriquement qu'un algorithme de routage donné fonctionne effectivement de façon correcte.

Les liens qui relient les noeuds d'un réseau d'interconnexion peuvent être soit bidirectionnels (deux messages peuvent circuler en même temps dans chaque sens) ou unidirectionnels (un seul sens à la fois). Dans les systèmes sophistiqués, le routeur peut envoyer et recevoir plusieurs messages en même temps par des liens différents. Mais souvent, l'architecture considérée ne permet qu'un message par direction. La deuxième option est plus simple techniquement et, bien sûr, moins chère. Par contre elle est plus lente car elle nécessite de parcourir séquentiellement les directions du réseau pour envoyer et recevoir les messages. Les communications n'utilisant qu'un seul canal à la fois sont dites **single-port** ou encore *processor bound*. Par contre, celles qui utilisent plusieurs canaux simultanément sont dites **multi-port** ou encore *link bound*. Il faut remarquer que les transferts multi-port demandent des accès mémoire plus rapides pour alimenter la communication et aussi des algorithmes plus complexes pour réaliser certains échanges de données spécifiques.

L'action du routeur est **locale**, par opposition à une situation **centralisée** dans laquelle les chemins seraient déterminés sur la base de l'ensemble des messages à acheminer. Une stratégie globale peut s'avérer très complexe avec des grands systèmes et des communications irrégulières mais elle permet en principe de contribuer à empêcher les conflits entre divers messages qui veulent emprunter la même connexion.

Dans une stratégie de routage **locale** (ou répartie), chaque noeud achemine les messages qui lui arrivent uniquement sur la base d'une information purement locale. Si cette information est l'adresse de destination portée par le message, on parle de routage local **déterministe**. Il existe aussi des stratégies de routage dite **adaptatives** qui utilisent le fait que, souvent, plusieurs chemins de même longueur sont possibles. Ces stratégies prennent en compte une information sur la contention locale du réseau (ou éventuellement des pannes) pour choisir vers quel noeud suivant le message doit être envoyé. Lorsque les chemins alternatifs sont tels que le message s'approche toujours de la destination, on parle de routage **profitable**. Dans le cas contraire, on accepte que le message fasse un détour localement. Un routage profitable évite les livelocks mais pas les deadlocks.

De plus, le routage est soit *synchrone*, soit *asynchrone*. Dans le premier cas, tous les messages sont envoyés simultanément et sont soumis à des cycles de routage complets. Le temps de transfert est alors celui du message le plus lent. C'est la stratégie utilisée dans les machines SIMD.

Le mode d'opération asynchrone est, quant à lui, plus souple et mieux adapté à une architecture MIMD : chaque noeud envoie ou reçoit les messages selon les besoins du programme, à des moments différents.

On définit le **trafic** dans le réseau comme le nombre de canaux d'interconnexion utilisé lors d'une communication. Souvent le trafic n'est pas homogène sur tout le réseau et il arrive qu'un noeud unique reçoive un volume disproportionné de données. Un tel noeud est alors un *hot-spot* et il est important d'avoir des capacités de routage ou de combinaison de messages efficaces afin de ne pas

dégrader les performances du réseau.

Il peut paraître curieux de remarquer que les réseaux à basses dimensions, avec une faible connectivité, telles les grilles bidimensionnelles, peuvent être plus efficaces que des réseaux à grande connectivité (hypercubes). En effet, avec moins de chemins disponibles, mais une bandwidth importante, il est plus facile de bien utiliser les ressources disponibles et d'équilibrer le trafic. S'il y a trop de chemins possibles, on aura souvent plusieurs chemins inoccupés, ce qui résulte en une faible utilisation du réseau.

3.1.3 Primitives de communication

Dans une machine parallèle, il y a certains types de communications qui reviennent fréquemment et que le routeur doit être capable de réaliser efficacement. Souvent, ces communications donnent lieu à des instructions particulières dans les bibliothèques d'échange de messages. Parmi les primitives de communication les plus courantes, on peut mentionner

- Les communications **point-à-point** ou **one-to-one** qui consistent en des échanges de données entre paires de processeurs. C'est le cas le plus simple et le plus générique de communication.
- Les **permutations** : le processeur i communique avec le processeur $f(i)$ où f est une bijection quelconque dans l'ensemble des noeuds du réseau. C'est un cas particulier où tous les processeurs sont simultanément impliqués dans une communication point à point.
- Le **broadcast**, **one-to-all** ou la **diffusion** qui est une opération très courante : un processeur donné envoie un message à tous les autres. Dans le même ordre d'idée, on a aussi le **multicast** ou le **one-to-many**, où un processeur communique une information à un groupe de processeurs.
- L'opération de **réduction**, ou le **many-to-one**, par laquelle un ensemble de processeurs envoie un message différent sur un processeur unique où toutes les informations sont combinées (par exemple au moyen une opération arithmétique ou logique).
- L'**échange total** ou le **all-to-all** qui implique que tous les processeurs envoient une information à tous les autres. C'est en quelque sorte un broadcast multiple simultané.
- Le **one-to-all personnalisé**, la **distribution** ou encore l'opération de **scatter**. Un processeur source envoie un message **différent** à tous les autres.
- L'opération de **gather** qui est en quelque sorte l'inverse du scatter : un processeur destination reçoit un message différent de tous les autres et les stocke dans un buffer de sorte que le message venant du $i$ème processeur soit en position i.
- L'**échange total personnalisé** qui désigne une communication dans laquelle chaque processeur envoie un message différent à tous les autres.

— Les opérations de **préfixes parallèles** ou **scans** qui combinent communications et calcul de sorte que la donnée finale du $i$ème processeur soit la somme (ou le résultat d'une autre opération arithmétique ou logique) des données originales contenues dans les processeurs 1 à $i-1$. Ces opérations permettent de faire des sommes partielles à travers les processeurs, ou encore, si l'opération utilisée est par exemple un `max`, d'obtenir la valeur maximum d'une variable dans tous les processeurs de rang inférieurs.

Plusieurs de ces primitives (par exemple l'échange total, les scans ou le broadcast) impliquent la participation de tous les processeurs et donc éventuellement leur coordination ou synchronisation. On désigne souvent ces primitives par le terme de **communications collectives**.

3.1.4 Techniques de commutation

Il y a plusieurs techniques associées à la manière dont un message (paquet) est transmis de proche en proche, du noeud initial jusqu'à sa destination. Le mode de commutation le plus simple est le **Store-and-Forward** qui correspond à une commutation de message. Le principe est d'envoyer un message du noeud i au noeud $i+1$ où il est stocké à mesure qu'il arrive. Après réception complète du message, le noeud $i+1$ l'envoie de la même manière au noeud $i+2$, selon le chemin spécifié par le routeur. Le temps de transfert du message est ainsi proportionnel au nombre de noeuds à traverser et au temps de transit du message entre deux noeuds consécutifs (c'est à dire, la longueur du message). Cette technique est illustrée sur la figure 3.3. Si W est la bandwidth du réseau (c'est à dire le nombre de Bytes qui peuvent traverser chaque ligne en une seconde), alors le temps de transfert T_{sf} du message est

$$T_{sf} = t_s + (n-1)\frac{L}{W} = t_s + L(n-1)t_w$$

où L est la longueur du paquet en bytes, n le nombre de noeuds à traverser, t_s le temps de préparation du message et $t_w = 1/W$ le temps de transfert d'un byte.

La technique du Store-and-Forward n'est pas rapide et n'utilise pas efficacement les possibilités de communication du réseau. Pour cette raison elle n'est plus guère utilisée dans les réseaux modernes. La commutation de circuit dynamique ou **cut-through** permet des performances beaucoup plus intéressantes. C'est une technique très utilisée depuis le début des années 90. Le principe est de permettre au message de poursuivre sa route tant qu'il n'est pas complètement reçu. La technique du **Wormhole** (ou commutation par **trains de bits**) est un exemple. Les messages sont découpés en petits morceaux de taille fixe appelés *flits* (flow-control digits). Le premier flit contient l'adresse de destination et les suivant les données. Typiquement, un flit correspond à 8 bits, mais ceci peut dépendre de la taille du réseau. Avec 256 noeuds, 8 bits sont suffisants pour coder toutes les destinations. Dans la technique du wormhole, la taille des buffers sur

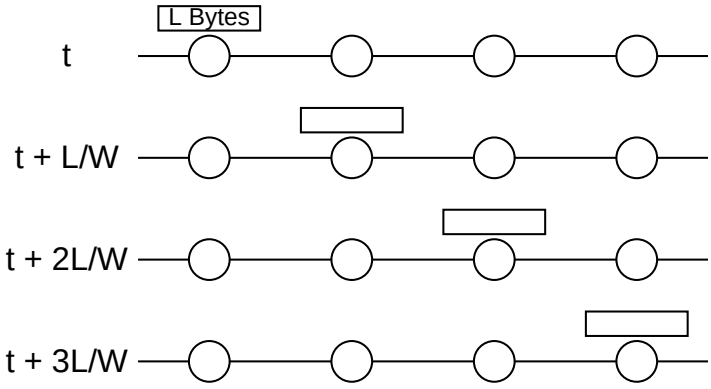


FIGURE 3.3 – *Illustration de la technique d'acheminement Store-and-Forward. Un message de L Bytes (indiqué par le rectangle) traverse 4 noeuds du réseau d'interconnexion (cercles). Le message est totalement reçu par chaque noeud avant d'être réexpédié plus loin.*

les noeuds intermédiaires est réduite à un seul flit, alors que dans le cut-through plus général, ces buffers peuvent contenir plusieurs flits.

Les flits traversent le réseau de façon "pipeline," à la manière d'un ensemble de wagons qui suivent la voiture de tête, seule à connaître l'adresse de destination du message et à "creuser" le chemin. Cette technique est illustrée sur la figure 3.4.

Le temps de transfert du message est dans ce cas le temps mis par l'entête (qui est proportionnel à la longueur du chemin) plus un temps proportionnel au nombre de flits restants du message (c'est à dire à la longueur du message). Si la bandwidth du réseau est W, chaque flit (1 byte) met un temps $t_w = 1/W$ pour passer entre deux noeuds adjacents. Donc, si n est le nombre de noeuds à traverser et L la longueur du message, le Wormhole nécessite un temps de transfert

$$T_{wh} = t_s + \frac{(n-1+L-1)}{W} = t_s + (n-1+L-1)t_w$$

On constate ainsi que la commutation Wormhole est beaucoup plus efficace que la commutation Store-and-Forward car elle est proportionelle à la somme $L+n$ de la longueur du chemin et de la longueur du message plutôt qu'à leur produit. Ainsi, si le diamètre du réseau est suffisamment petit, on peut négliger n par rapport à L et dire que le temps de transfert par Wormhole est approximativement proportionnel à la longueur du message, si celui-ci est assez long.

Un autre avantage de la commutation Wormhole est qu'elle utilise moins de mémoire que le Store-and-Forward.

La technique du cut-through est une amélioration de la commutation de circuit usuelle dans laquelle l'ensemble du chemin est réservé pour le passage du message. L'entête du message est là aussi responsable d'établir le chemin mais c'est seulement lorsqu'elle est a destination que le reste du message est envoyé

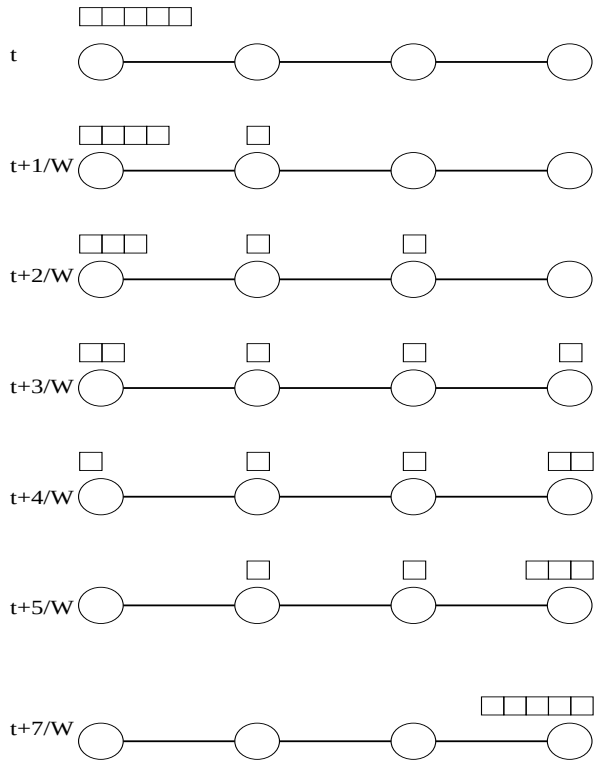


FIGURE 3.4 – *Illustration de la technique d'acheminement Wormhole. Le message initial (rectangle) est divisé en flits (carrés) et traverse le réseau à la manière d'une locomotive qui tire ses wagons.*

d'une traite. De plus, dans la technique du Wormhole, le chemin est libéré immédiatement après le passage du dernier flit et non à la fin du transfert. Le Wormhole est très efficace pour les messages relativement courts, tels que ceux que l'on rencontre dans le calcul parallèle. Elle permet aussi de facilement dupliquer un message sur un noeud donné, ce qui est utile pour les opérations comme le broadcast. La commutation de circuit classique ne permet pas cette duplication des messages mais reste une solution intéressante pour les très longs messages.

Si un chemin est bloqué par un autre message en cours, la tête du message arrivant est arrêtée, ainsi que tout le reste, jusqu'à libération du chemin. Il faut évidemment éviter tout risque de deadlock lorsque plusieurs messages se bloquent mutuellement. Une solution pour résoudre une telle situation est d'avoir des canaux supplémentaires entre les routeurs, par exemple en créant des **canaux virtuels** par un multiplexage en temps du canal physique. Ainsi, chaque direction dispose d'une tranche de temps pour passer et chaque routeur dispose de plusieurs buffers pour accueillir les flits provenant des canaux virtuels différents.

Le concept de canal virtuel est aussi très important pour augmenter le débit global du réseau. Par exemple, si le corps d'un message M_1 bloque un noeud intermédiaire P_j car sa tête ne peut momentanément pas progresser, un autre

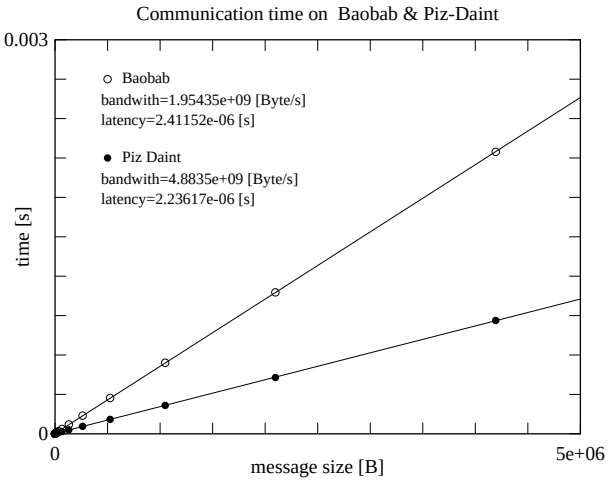


FIGURE 3.5 – Temps de communication entre deux processeurs, pour les machines Baobab de l'UNIGE et Piz Daint du CSCS. Mesure par C. Kotsalos, 2020

message M_2 ne pourra pas, sans canal virtuel, traverser P_j, même si c'est pour se rendre dans une direction qui est libre. Le partage en temps du routeur permet, par le mécanisme des canaux virtuels de libérer P_j et de permettre l'acheminement du deuxième message. On augmente ainsi le throughput par une meilleure utilisation des ressources.

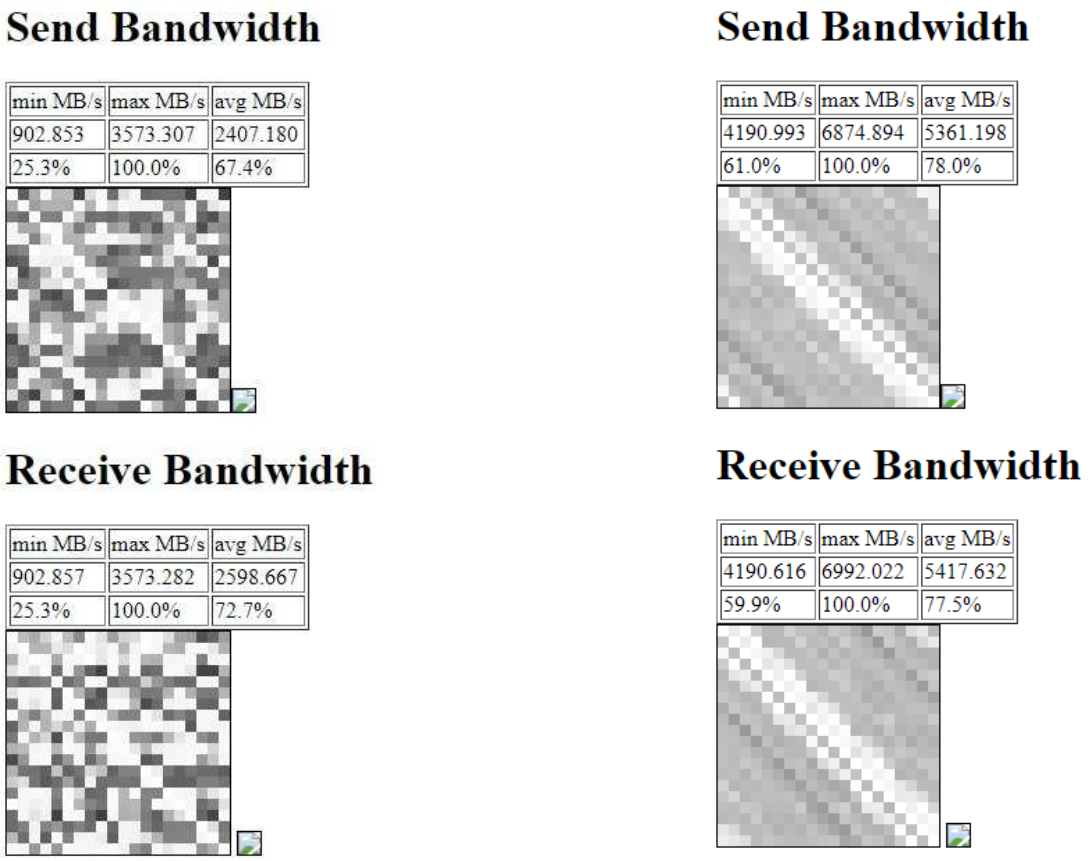
3.1.5 Exemple de performances

La figure 3.5 illustre le résultat de la section précédente, à savoir que le temps de communication entre deux processeurs peut s'approximer par une dépendance linéaire dans la taille M du message

$$T_{comm} = t_s + \frac{M}{b}$$

où t_s est un temps de latence et b est la bande passante. On obtient une latence de l'ordre de 2 μs et une bande passante variant entre 2 et 5 GB/s.

La figure 3.6 montre que si le réseau d'interconnexion est puissant et homogène (cas de Piz Daint) les mêmes performances se retrouvent entre n'importe quelle paire de noeuds. Dans le cas de Baobab, le réseau d'interconnexion est heterogène car beaucoup d'utilisateurs n'ont pas de grands besoins de communication. Dès lors le réseau infiniband n'est pas déployé entre tous les noeuds, ce qui explique la diversité des bandes passantes mesurées dans la figure 3.5.



Baobab Piz Daint

FIGURE 3.6 – Bande passante entre différentes paires de noeuds dans les machines Baobab (UNIGE) et Piz Daint (CSCS). Measures : C. Kotsalos, 2020

3.2 Les réseaux statiques

Les réseaux statiques sont ceux qui disposent de connexions physiques fixes entre les noeuds et dont la topologie ne peut être modifiée. En pratique, chaque noeud d'un réseau d'interconnexion statique contient un processeur muni d'un routeur, alors que dans un réseau dynamique, il peut contenir un switch (commutateur). Dans ce paragraphe, nous allons étudier les réseaux statiques les plus courants et discuter leurs propriétés.

3.2.1 Anneaux et chaînes

Le réseau le plus simple que l'on puisse imaginer est sans doute une chaîne où chaque processeur n'est relié qu'à deux autres processeurs. Les extrémités de la chaîne peuvent soit être connectées entre elles (chaîne cyclique), ou bien être indépendantes (chaîne linéaire). La chaîne cyclique ou anneau est un réseau symétrique alors que la chaîne linéaire ne l'est pas (voir figure 3.7)

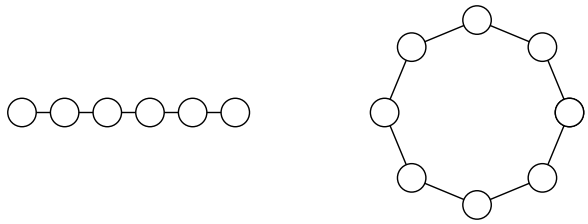


FIGURE 3.7 – *Topologie linéaire et en anneau*

La connectivité d'un anneau de processeurs est faible (égale à 2) et le temps de transit d'un message est directement proportionnel à la distance qui sépare les processeurs qui communiquent entre eux, puisque le message doit être transmis de proche en proche jusqu'à sa destination. Pour un anneau contenant N noeuds, on a donc un diamètre $D = N/2$. La largeur bissectionnelle vaut 2 et la scalabilité ne pose pas de problème (on peut agrandir l'anneau en mettant bout à bout des noeuds identiques). Le prix d'un tel réseau est faible ($\sim N$), car le nombre de connexions augmente avec le nombre d'éléments de la chaîne. Une telle topologie est adaptée au traitement de problèmes unidimensionnels. Cependant, les réseaux à une dimension sont un peu restrictifs pour justifier leur fabrication dans le cadre d'une machine parallèle. Néanmoins, les réseaux linéaires sont couramment utilisés pour connecter ensemble des ordinateurs personnels tels des stations de travail. Des environnements comme PVM (Parallel Virtual Machine) offrent une bibliothèque de primitives de communications qui permettent de transformer un réseau local en une machine parallèle. Cependant, les temps de communications importants peuvent limiter ce type d'approche à des applications plus distribuées que parallèles.

3.2.2 Grilles de processeurs (mesh)

La généralisation de la topologie linéaire est la topologie de grille (ou *mesh*), dans laquelle les processeurs sont connectés selon un réseau cartésien de dimension d (voir figure 3.8).

Les grilles de dimension 2 sont les plus répandues car elles s'adaptent naturellement à un grand nombre d'applications tout en restant architecturalement simples donc efficaces. Ce type de réseau était notamment utilisé dans l'ILLIAC IV, le DAP et la Paragon d'Intel. Chacune de ces machines est toutefois différente en ce qui concerne la connectivité des processeurs aux extrémités du réseau (bords libres, conditions périodiques ou spirales). Notons encore que le T3E de Cray Research est basé sur une topologie tridimensionnelle, de même que l'étaient les machines de Parsytec.

La figure 3.8 illustre une topologie de grille à deux et trois dimensions.

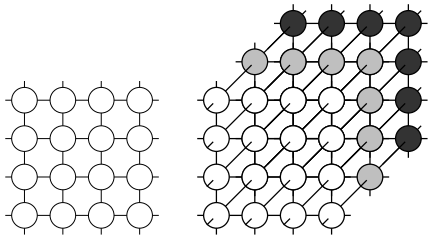


FIGURE 3.8 – *Grille de processeur à 2 et 3 dimensions.*

Comme l'illustre la figure 3.8, la connectivité d'une grille de dimension d est $2d$. Si N est le nombre total de processeurs, le diamètre vaut $dN^{1/d}$, qui est la distance entre les noeud diamétralement opposés. Si les bords sont connectés de façon périodique, selon un tore, le diamètre est divisé par 2. Le prix d'un tel réseau croît proportionnellement à N. Les grilles sont scalables. La largeur bissectionnelle s'obtient en calculant la surface résultant de la coupe du réseau en deux parties. Comme $N^{1/d}$ est le nombre de noeud le long de chaque arête de la grille, la surface de coupe contiendra $N^{(d-1)/d}$ processeurs desquels partent une seule connexion vers l'autre moitié du système. La largeur bissectionnelle vaut donc $N^{(d-1)/d}$.

Le routage s'effectue de façon naturelle, le long des axes de la grille. Par exemple, pour aller du noeud (i_0, j_0) au noeud (i_1, j_1), dans une grille bidimensionnelle, on parcourt tout d'abord l'axe X de la grille, soit les noeuds (i_k, j_0) pour i_k variant de de i_0 à i_1. Puis on change de direction pour aller du noeud (i_1, j_0) au noeud (i_1, j_1), en suivant la direction Y. Cette technique se généralise facilement à des dimensions de grilles plus grandes.

Ce routage s'appelle le routage $X - Y$. Dans sa version la plus simple, on chemine d'abord selon l'axe des X puis selon l'axe Y. Cependant, dans le cas d'un ensemble de communications point à point simultanées, le risque de congestion de

réseau est fréquent, comme l'indique la figure 3.9. Si on suppose que l'on considère un routage *synchrone* qui respecte strictement le cheminement selon X d'abord, puis selon Y, on voit que, dans une première phase, le message du noeud A vers le noeud C transite vers le noeud B. A la deuxième phase (routage vertical), le noeud B devrait envoyer simultanément deux messages dans le lien $B - C$, ce qui produit une congestion. Par conséquent, un des messages doit être retardé et mémorisé par le router B, puis envoyé lors d'un cycle ultérieur de routage, ce qui évidemment implique une pénalité de temps.

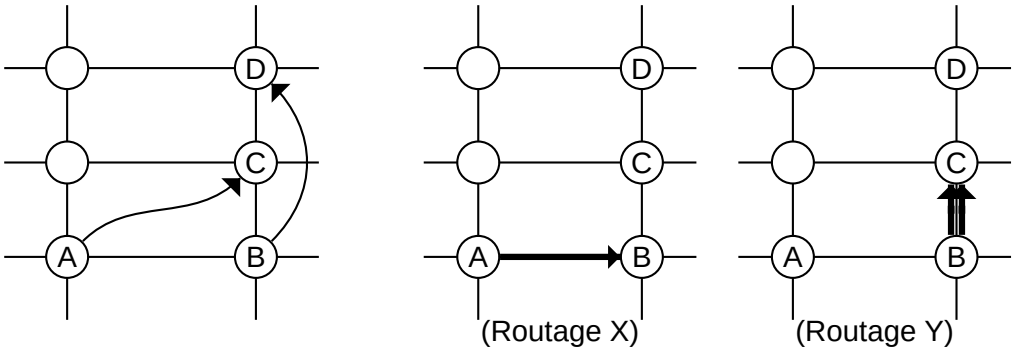


FIGURE 3.9 – *Situation de congestion sur une grille lors d'un routage synchrone local non adaptatif. L'image de gauche indique les expéditeurs et destinataires des deux messages. Les deux images suivantes montrent les phases $X - Y$ du routage, jusqu'à apparition de la congestion.*

Cependant, comme l'ordre de déplacement des messages le long des axes de coordonnées est arbitraire, une stratégie de routage *adaptative* permet de modifier localement cet ordre si un excès de trafic le rend nécessaire. Par exemple, dans le cas présenté dans la figure 3.9, le noeud A pourrait d'abord faire une étape de routage Y avant de faire celle en X. Cela éviterait la surcharge du lien $B - C$ et assurerait un routage en un temps minimum.

Exemple de broadcast sur une grille

Une autre opération de communication importante est le broadcast où un processeur envoie un message à tous les autres. Sur une grille, on peut réaliser un tel broadcast en distribuant d'abord le message sur la ligne du processeur source, puis en parallèle, sur toutes les colonnes. Cette approche est illustrée sur la figure 3.10.Une autre technique consiste à distribuer le message dans plusieurs directions simultanément (communication multi-port). Il est facile de voir que la complexité de ces deux approches (c'est à dire le nombre d'étapes nécessaires) est en $O(\sqrt{N})$, où N est le nombre total de noeuds.

Finalement, la figure 3.11 présente une variante astucieuse d'un réseau en grille 2D. C'est le réseau X-net qui a été conçu pour la MaPar, une machine SIMD du début des années 90 et qui permet des communications directes entre les 8

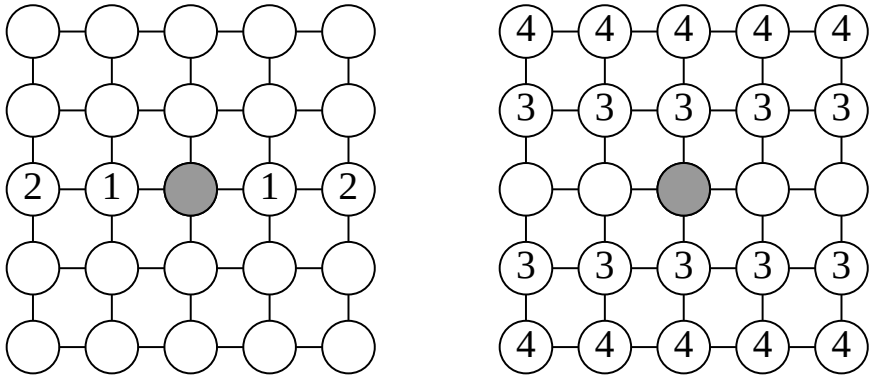


FIGURE 3.10 – *Broadcast sur une grille de processeurs bidimensionnelle, réalisé en distribuant d'abord le message le long de la ligne du processeur source (image de gauche), puis le long des colonnes, simultanément par tous les processeurs de la ligne (image de droite). Les nombres indiquent l'étape de temps à laquelle le message est reçu.*

processeurs plus proches voisins, grâce à des connexions selon les diagonales. A chaque intersection de la grille en X se trouve un noeud tridirectionnel permettant de rediriger un message vers l'un des quelconques trois voisins possibles.

3.2.3 L'hypercube

Le réseau en hypercube représente un moyen très courant pour relier entre eux un ensemble de processeurs. Cette topologie est utilisée dans la Connection Machine en particulier, mais est aussi une option importante pour les architectures MIMD. C'était sans doute le réseau le plus populaire dans les années 80.

Au sens mathématique, un hypercube désigne la généralisation d'un cube dans des espaces de dimension supérieure à 3. Dans notre contexte, ce terme suppose que les côtés sont de longueur 2. Ainsi, dans un hypercube de dimension k, on a exactement $N = 2^k$ processeurs, chacun de degré k (c'est-à-dire liés à k autres noeuds du réseau). Des exemples de réseaux hypercubiques de dimension $k = 0, 1, 2, 3$ sont donnés sur la figure 3.12. La figure 3.13 montre un hypercube de dimension $k = 4$.

On constate que le réseau hypercube se construit récursivement de la manière suivante. Un hypercube de dimension k se construit à partir de deux hypercubes de dimension $k - 1$. On relie ensuite les sommets correspondants de ces deux hypercubes.

Un hypercube de dimension k est donc une grille de dimension k et d'arêtes égales à 2. Cependant, la différence dans la discussion que l'on fait des propriétés de ces deux types de réseaux provient du fait que, pour l'hypercube, on considère une arête de taille fixe (=2) avec une dimension k qui varie, alors que pour la

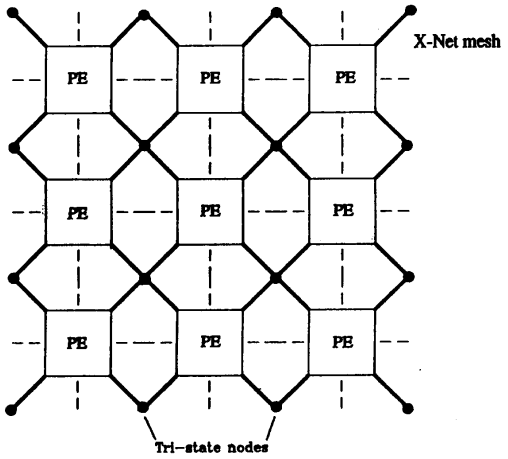


FIGURE 3.11 – *Réseau X-net de la MasPar (Past, Present and Parallel, fig. 2.3, p. 30).*

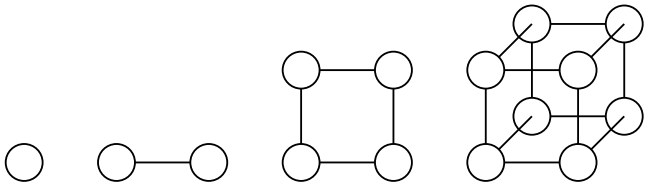


FIGURE 3.12 – *Hypercubes de dimension 0, 1,2 et 3.*

grille, on a une dimension d donnée et on varie la taille des arêtes.

Le diamètre D d'un hypercube est égal à sa dimension k car c'est le nombre d'arêtes qu'il faut suivre pour relier les deux noeuds les plus éloignés. La largeur bissectionnelle vaut $2^{k-1} = N/2$, soit la taille d'un des deux sous-cubes de dimension $k-1$ utilisés pour la construction de l'hypercube de dimension k.

Le prix d'un réseau en hypercube est proportionnel à $kN = N \log_2 N$ car $k = \log_2 N$ est le nombre de connexions par noeuds et N le nombre de noeuds.

La topologie en hypercube n'est pas véritablement scalable, malgré le fait que l'on peut envisager des hypercubes de taille arbitraire. Le problème est qu'en agrandissant un hypercube, on augmente la connectivité de chaque noeud, ce qui n'est évidement pas possible pour des processeurs donnés qui ont un nombre fixe de liens vers l'extérieur. On peut éviter ce problème en plaçant en chaque noeud de l'hypercube une chaîne cyclique de k processeurs (*cube connected cycles*), plutôt qu'un processeur unique. Avec des processeurs ayant trois liens extérieurs, on peut former un anneau de k processeurs ayant exactement les k liens extérieurs nécessaires aux noeuds de l'hypercube. De cette manière, on garde la plupart des avantages de l'hypercube tout en assurant sa scalabilité. Dans ce qui suit, nous nous contenterons de discuter l'hypercube simple.

La structure en hypercube a plusieurs avantages. Premièrement, le nombre

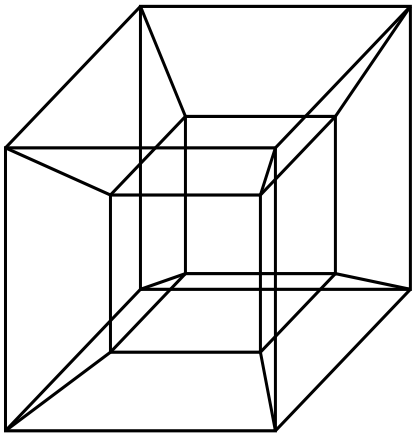


FIGURE 3.13 – *Hypercube de dimension 4.*

de noeuds du réseau croît comme une puissance de 2 du nombre de connexions. Avec une augmentation faible du nombre de connexions, on augmente de façon significative la taille du système. Deuxièmement, le nombre de chemins disponibles entre 2 noeuds augmente à mesure que l'hypercube s'agrandit, ce qui aide à éviter une congestion du trafic des messages.

Routage Il existe une manière efficace de diriger un message à travers un réseau hypercubique. Pour cela, on utilise tout d'abord une numérotation adéquate des noeuds. Cette numérotation se base sur la représentation binaire des nombres de 0 à $2^k - 1$. Chaque noeud est représenté par un nombre de k bits. On choisit arbitrairement le noeud initial qui reçoit l'adresse 0. Ensuite, on applique l'algorithme suivant : deux noeuds qui sont reliés le long de la dimension ℓ de l'hypercube ont une adresse qui ne diffère que par le ℓ^{eme} bit. Cette numérotation est illustrée dans la figure 3.14. Il faut remarquer que cette numérotation correspond aussi au coordonnées cartésiennes dans un espace de dimension k pour lequel les deux seules valeurs possibles dans chaque direction sont 0 et 1.

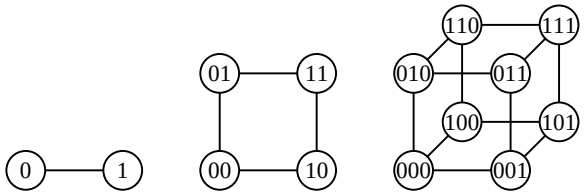


FIGURE 3.14 – *Numérotation des noeuds dans un hypercube : les noeuds voisins dans la direction i ont une adresse qui ne diffère que par le bit i.*

Le chemin que doit parcourir un message devant transiter entre deux noeuds est obtenu par un XOR (ou exclusif) des adresses respectives. Cette opération donne en effet un 1 dans chaque dimension à suivre pour atteindre la destination

désirée. Cela se comprend facilement dans le cas à trois dimensions de la figure 3.14. Pour aller du noeud 001 au noeud 111, il faut voyager dans la direction 1 puis la direction 2 (001 XOR 111 =110), ou inversement. Plus généralement, l'algorithme de transfert de messages ("routing algorithm") dans un hypercube de dimension k contient k étapes. Durant l'étape i, le message est transmis dans la direction i si le XOR de l'adresse contient un 1 en position i. Sinon, le message reste là où il est et attend l'étape suivante. Cela montre que le temps de transit d'un message par cette méthode est proportionnel à k, ou encore au logarithme du nombre N de processeurs ($k = \log_2(2^k)$).

Exemple de broadcast sur un hypercube

La figure 3.15 présente une opération de broadcast sur un hypercube de dimension $k = 4$, à partir du noeud racine (000). On constate que $k = \log_2 N$ étapes sont suffisantes pour que le message initial atteigne tous les noeuds. C'est donc beaucoup plus efficace qu'un broadcast sur un anneau qui requiert $N/2$ étapes.

L'algorithme de broadcast présenté ici est le suivant : un message arrivant de la dimension i de l'hypercube est redistribué simultanément dans les dimensions $i-1$, $i-2$,...,1. C'est un algorithme qui implique des communications multi-liens (ou multi-port) puisqu'un même processeur envoie simultanément un message dans plusieurs directions.

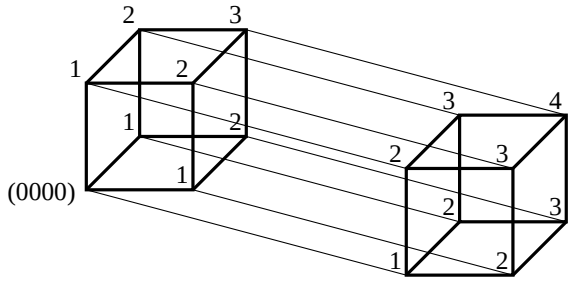


FIGURE 3.15 – *Broadcast sur un hypercube. Le noeud source (0000) envoie le même message dans les quatre dimensions de l'hypercube. Ensuite, chaque noeud redistribue le message selon l'algorithme décrit dans le texte. Les nombres indiquent l'étape de temps à laquelle le message atteint chaque noeud.*

On peut aussi imaginer un algorithme qui n'utilise qu'un lien à la fois (single port). Il est illustré sur la figure 3.16 et nécessite aussi autant d'étapes qu'il y a de dimension dans l'hypercube (il est donc logarithmique dans le nombre de processeurs). Le principe de cet algorithme de broadcast est le suivant : à la première étape, le processeur racine envoie le message dans le direction 1. Durant la deuxième étape, le processeur racine et celui qui a reçu le message à l'étape précédente envoient tous deux le message dans le direction 2 de l'hypercube. A ce stade, 4 processeurs sont "touchés." A l'étape suivante, ces 4 processeurs envoient

le message dans la direction 3 et ainsi de suite jusqu'à ce que tous les processeurs aient reçu le message d'origine.

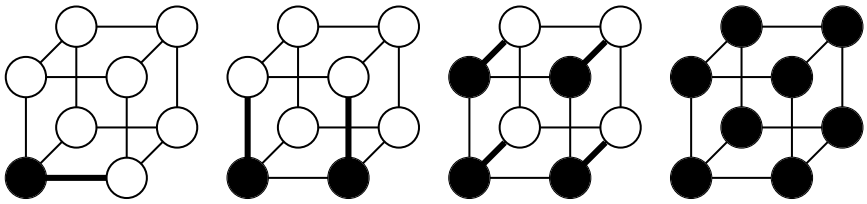


FIGURE 3.16 – *Broadcast sur un hypercube en utilisant des communications "single port." A l'étape i, tous les processeurs ayant reçu copie du message initial (il y en a 2^{i-1}) le renvoient dans la direction i de l'hypercube. Après $k = \log_2 N$ étapes, le broadcast est terminé.*

Exemple d'échange total sur un hypercube

Dans une opération d'échange total, chaque noeud envoie une donnée différente à tous les autres noeuds. A la fin de l'échange, chaque noeud contiendra autant de données qu'il y a de processeurs. Une méthode simple pour réaliser un échange total sur un hypercube est de procéder à des échanges de paires de données, successivement le long de chaque dimension de l'hypercube. A la première étape, chaque noeud envoie une seule donnée au noeud voisin dans la dimension 1. A la ℓ^{eme} étape, chaque noeud envoie son contenu initial, plus toute les données reçues aux étapes précédentes. Sur un hypercube de dimension k, il faut k étapes pour achever un tel échange total. Comme le nombre de données à communiquer au cours de l'étape i vaut 2^{i-1}, le temps total T de la communication est

$$T = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 = N - 1$$

ce qui est proportionnel au nombre de processeurs. Il existe des techniques plus complexes et plus rapides permettant de réaliser un échange total en distribuant les données simultanément dans toutes les dimensions de l'hypercube.

Mapping d'une grille sur un hypercube

D'une manière générale, le problème du mapping est de plonger une structure de données sur un réseau de communication physique ayant une topologie différente. On aimerait par exemple répartir un arbre sur une topologie en hypercube, tout en préservant la connectivité naturelle de l'arbre.

Un autre exemple courant est le mapping d'une grille sur un hypercube, de sorte que les plus proches voisins sur la grille soient répartis sur des noeuds

adjacents de l'hypercube. Cette contrainte est importante car nombreux sont les algorithmes numériques utilisant des données disposées selon une grille de points et nécessitant des échanges entre points voisins.

De façon générale, ce mapping est réalisé à l'aide d'une fonction G appelée code de Gray (binary reflected Gray code).

Considérons tout d'abord le cas d'une grille unidimensionnelle (vecteur) composée de 2^d points, numérotés $0, 1, 2, ..., 2^d - 1$. On peut associer chaque point i du vecteur à un processeur n dans un hypercube de dimension d par la relation

$$n = G(i, d)$$

où la fonction G est définie récursivement par

$$G(0, 1) = 0 \qquad G(1, 1) = 1$$

pour $d = 1$ et

$$G(i, d+1) = \begin{cases} G(i, d) & \text{pour } i < 2^d \\ 2^d + G(2^{d+1} - 1 - i, d) & \text{pour } i \geq 2^d \end{cases}$$

pour les dimensions supérieures. La numérotation des processeurs dans l'hypercube se fait selon le codage standard : deux processeurs voisins le long de la dimension $k \leq d$ ont une adresse qui ne diffère que par le $k$ème bit.

La définition de G indique que lorsqu'on considère un hypercube de dimension $d + 1$, les 2^d premiers points du vecteur sont envoyés sur le sous hypercube de dimension inférieure d. Les suivants sont envoyés sur les processeurs dont l'adresse s'obtient par "réflexion", selon la construction illustrée sur la figure 3.17. On constate que ce mapping conduit naturellement à des conditions aux bords périodiques puisque le dernier point du vecteur est connecté au premier et que, par construction, les points voisins sur le vecteur sont placés sur des processeurs dont l'adresse ne diffère que par un seul bit (voir figure 3.18).

Le mapping d'une grille de dimension supérieure se fait par une extension de la méthode du code de Gray que l'on vient de voir. Illustrons ceci pour le cas d'une grille rectangulaire bidimensionnelle de taille $2^r \times 2^s$ plongée dans un hypercube de dimension 2^{r+s}. Le point de coordonnée (i, j) de la grille est assigné au processeur d'adresse $G(i, r)\|G(j, s)$ où l'opération $\|$ dénote la concaténation des deux codes de Gray. Comme chacun des codes de Gray préserve la connectivité à une dimension, on a aussi que les points (i, j) et $(i \pm 1, j)$ ou $(i, j \pm 1)$ sont adjacents sur l'hypercube.

Par exemple, pour un tableau de taille 4×8, l'élément $(2, 4)$ est placé sur le processeur d'adresse 11110 d'un hypercube de dimension 5 ($11110 = 11\|110$ qui sont respectivement les code de Gray $G(2, 2)$ et $G(4, 3)$).

Le mapping d'un tableau 4×4 est illustré ci-dessous. Dans chaque élément du tableau figure un nombre indiquant l'adresse en binaire du processeur où il

1-bit Gray code	2-bit Gray code	3-bit Gray code	3-D hypercube	8-processor ring
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

FIGURE 3.17 – *Construction récursive du code de Gray réfléchi. Pour passer de la dimension d à la dimension $d+1$ on répète le mapping obtenu (régions ombrées) et on le réplique par symétrie par rapport aux lignes pointillées. On complète le mapping en faisant précéder le partie supérieure par un 0 et la partie inférieure par un 1.*

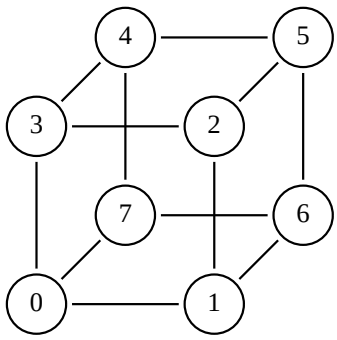


FIGURE 3.18 – *Mapping, par le code de Gray, d'un anneau à 8 noeuds sur un hypercube de dimension 3.*

sera placé.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0000 & 0001 & 0011 & 0010 \\ 0100 & 0101 & 0111 & 0110 \\ 1100 & 1101 & 1111 & 1110 \\ 1000 & 1001 & 1011 & 1010 \end{pmatrix} \end{array}$$

On remarque que ce mapping a de plus la propriété que les processeurs associés à une même ligne du tableau ont les deux bits d'adresse les plus significatifs identiques, alors que ceux qui correspondent à une même colonne ont les même deux derniers bits.

Les grilles dont la dimension n'est pas une puissance de 2 peuvent être plongées dans un hypercube de la même façon à condition d'être élargie à la puissance de 2 supérieure. Cependant, dans ce cas, on perd les conditions aux bords périodiques.

3.2.4 Les arbres binaires

Les réseaux en arbre binaire statiques sont peu utilisés dans les architectures parallèles. Cependant, une version améliorée est à la base du réseau d'interconnexion dit *fat tree* qui est un arbre dynamique que nous étudierons au paragraphe 3.3.3.

Pour l'instant, nous discutons les propriétés élémentaires d'un arbre binaire statique. Plusieurs d'entre elles se généraliseront aisément au cas du fat tree.

Le schéma d'un tel réseau est donné sur la figure 3.19. A l'exception des extrémités, chaque noeud est relié à 3 autres noeuds, un ancêtre et deux descendants. Chaque ligne horizontale de processeurs est appelée niveau ou étage de l'arbre. Le niveau ℓ ($\ell = 0, k$) contient 2^ℓ noeuds et le nombre total de noeuds est $N = 2^0 + 2^1 + 2^2 + ... + 2^k = 2^{k+1} - 1$.

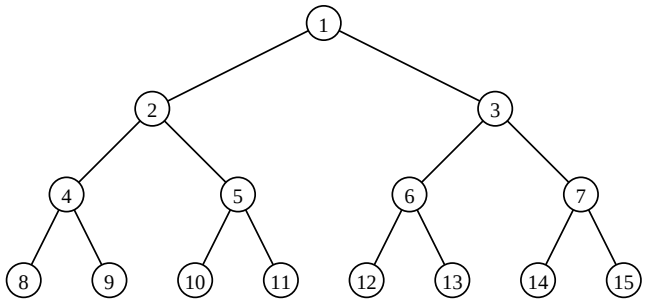


FIGURE 3.19 – *Topologie d'arbre binaire statique.*

Le diamètre d'un arbre binaire est $D = 2k$, car au pire, il faut remonter jusqu'à la racine et redescendre jusqu'en bas. La largeur bissectionnelle vaut 1, ce qui est le défaut majeur de cette architecture : la racine constitue un "bottleneck"

pour les communications. Par contre, un tel réseau est scalable et son prix est proportionnel à N, le nombre de noeuds.

L'arbre binaire est un réseau qui se prête bien à sommer un ensemble de nombres. Dans ce sens, ce réseau est important du point de vue théorique car il permet de formaliser simplement de nombreuses opérations parallèles impliquant l'ensemble des processeurs. Par exemple, on obtient la somme de 2^k nombres en les répartissant d'abord sur les processeurs du niveau k. Chaque processeur du niveau $k-1$ calcule ensuite la somme des 2 processeurs qui lui sont liés. On poursuit cette méthode pour les niveaux suivants, jusqu'à atteindre la racine de l'arbre qui va contenir la somme de toutes les valeurs. Cette opération nécessite k étapes, c'est-à-dire le logarithme du nombre de valeurs à additionner.

Routage Le transfert d'un message d'un noeud X à un noeud Y à travers un tel réseau arborescent requiert de "remonter" jusqu'à un ancêtre commun de Y, puis de "redescendre" sur Y. En numérotant les noeuds comme sur la figure 3.19, on constate que les descendants d'un noeud x portent les adresses $2x$ et $2x+1$. En représentation binaire, chaque noeud du niveau k est donné par $k+1$ bits. Les adresses des descendants à gauche et à droite sont donc obtenues en ajoutant respectivement 0 et 1 à l'adresse du noeud parent. Le plus proche ancêtre commun de X et Y aura une adresse correspondant au plus long préfixe de bits commun dans l'adresse de X et Y (en lisant de gauche à droite). Par exemple, le noeud 14 (1110) et le noeud 6 (110) ont comme ancêtre commun le noeud 3 (11). Cela signifie que du noeud 14, il faut d'abord remonter de deux niveaux (14 a 4 bits alors que 3 n'en a que 2), puis descendre d'un niveau à droite (car 6 s'obtient de 3 en ajoutant un zéro).

Les architectures en arbre peuvent être modifiées en ajoutant des liens entre noeuds d'un même niveau. De plus, chaque noeud peut lui-même être composé de plusieurs sous-noeuds interconnectés d'une manière quelconque. L'architecture en arbre est la base d'un système hiérarchique de connexions.

3.2.5 Les réseaux de permutation et "shuffle exchange"

Les topologies que nous avons étudiées jusqu'à présent ont été obtenues par une construction géométrique. Dans ce paragraphe, nous considérons une autre manière de générer des réseaux d'interconnexions, en utilisant des fonctions de permutations. L'idée est de relier entre eux les noeuds i et j si $j = f(i)$ où f est une bijection définie sur les N noeuds du réseau. De tels réseaux sont aussi appelés réseau de De Bruijn.

Un exemple bien connu est la topologie dite "shuffle exchange" qui, intuitivement, consiste à "mélanger" les noeuds d'un réseau afin, qu'en moyenne, chacun soit proche de chacun. Ce type de réseau a de bonnes propriétés de routage, malgré une connectivité faible. Ils ont été utilisés pour calculer des transformées de Fourier rapides et ont des caractéristiques qui les rendent intéressants pour

plusieurs algorithmes numériques. Notons encore que les réseaux de permutations sont à la base d'une famille de réseaux dynamiques multiétages que nous verrons au paragraphe 3.3.3.

Permutation Perfect shuffle

La première bijection que nous allons discuter ici est la permutation, dite de *perfect shuffle*. Elle est analogue à celle obtenue en mélangeant un paquet de cartes à jouer (lorsque les cartes de la moitié supérieure sont entrecroisées avec celles de la moitié inférieure), comme le montre la figure 3.20.

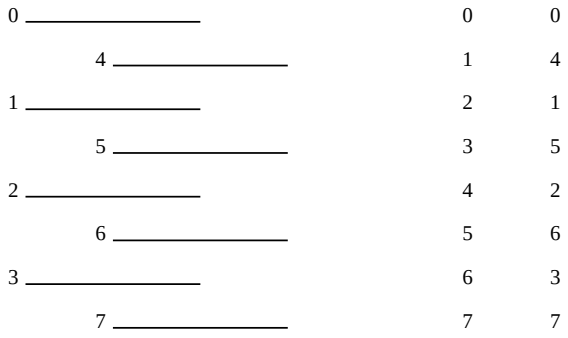


FIGURE 3.20 – *Opération de mélange qui donne la permutation de perfect-shuffle. La partie gauche illustre l'analogie avec le brassage d'un jeu de carte et la partie droite donne la bijection résultante.*

Après mélange, la première carte du paquet reste en première position ($0 \to 0$, alors que la carte qui occupe la deuxième position était initialement au milieu ($1 \to N/2$), et ainsi de suite. La figure 3.21 montre la permutation *perfect shuffle* dans son ensemble, ainsi que la permutation réciproque, appelée *inverse shuffle*.

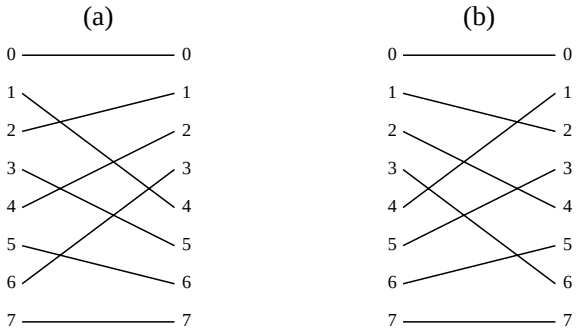


FIGURE 3.21 – *Permutation "perfect shuffle" et "inverse shuffle."*

En représentation binaire, la permutation de perfect shuffle se réalise par une rotation circulaire vers la droite des bits de l'adresse de chaque noeud

$$b_m b_{m-1} ... b_2 b_1 \rightarrow b_1 b_m b_{m-1} ... b_2$$

et le shuffle inverse est donné par une rotation circulaire vers la gauche

$$b_m b_{m-1} ... b_2 b_1 \rightarrow b_{m-1} ... b_2 b_1 b_m$$

Dans le même ordre d'idée, on introduit d'autres fonctions de permutations.

Permutation Butterfly La permutation butterfly s'obtient par un échange du premier et dernier bit de l'adresse

$$b_m b_{m-1} ... b_2 b_1 \rightarrow b_1 b_{m-1} ... b_2 b_m$$

Permutation Bit Reversal La permutation bit reversal est construite en interchangeant l'ordre des bits

$$b_m b_{m-1} ... b_2 b_1 \rightarrow b_1 b_2 ... b_{m-1} b_m$$

Pour un système de 8 noeuds (nécessitant des adresses de 3 bits), l'opération de *butterfly* est équivalente à celle de *bit reversal*.

Permutation d'échange La permutation d'échange est obtenue en échangeant le premier bit par son complément à 1

$$b_m b_{m-1} ... b_2 b_1 \rightarrow b_m b_{m-1} ... b_2 \bar{b}_1$$

où $\bar{b} = 1 - b$

Le Réseau Shuffle-Exchange

Revenons maintenant à la construction du réseau shuffle-exchange. Les connexions physiques entre processeurs sont données par les permutations de perfect shuffle et d'échange, comme l'indique la figure 3.22. Il est important de combiner deux permutations différentes si l'on veut transmettre un message entre deux noeuds quelconques. La permutation de perfect shuffle n'offre en effet aucun lien entre des processeurs dont les adresses ne sont pas une permutation cyclique l'une de l'autre.

La connectivité du réseau shuffle-exchange vaut 3. De chaque noeud partent deux connexions (l'une du perfect shuffle et l'autre de l'échange). De même, sur chaque noeud il arrive aussi deux connexions. Cependant, la connexion d'échange qui quitte le noeud est la même que celle qui rentre. Cela donne donc bien une connectivité de 3 (sur la figure 3.22, les liens 0-0 et 7-7 sont doubles).

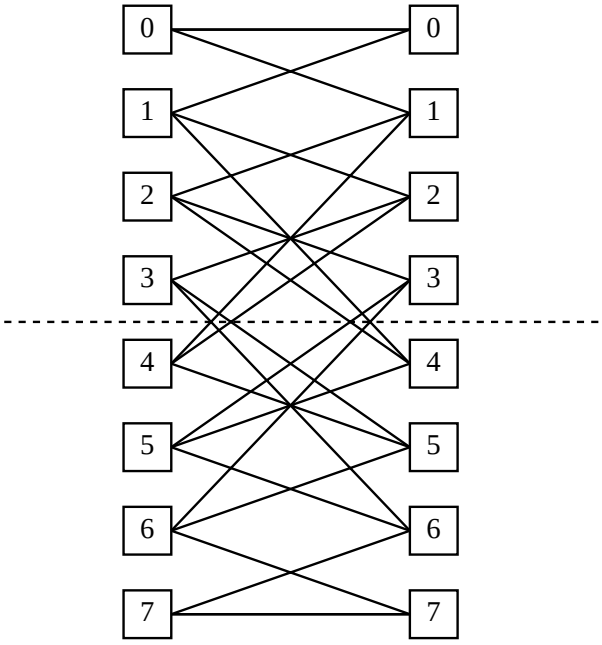


FIGURE 3.22 – *Réseau shuffle-exchange. Pour des raisons de clarté, les connexions sont représentées sur un schéma tel que chaque processeur est répété deux fois (colonne de gauche et colonne de droite). Pour cette raison, chaque connexion est aussi répétée deux fois (le lien 1-4 et le lien 4-1 peuvent être supposés identiques).*

Le prix du réseau shuffle-exchange est ainsi proportionnel à $3N$, où N est le nombre total de noeuds.

La scalabilité pose des problèmes car les liens changent lorsque N augmente. Avec quatre processeurs, le noeud 1 serait relié au noeud 2 par l'opération de perfect shuffle, alors qu'avec huit processeurs, il est relié au noeud 4.

La largeur bissectionnelle s'obtient en coupant le réseau en deux parties égales, comme le montre la ligne pointillée de la figure 3.22. La moitié supérieure (appelons-la A) est caractérisée par les noeuds

$$A : \qquad 0b_{m-1}b_{m-2}...b_2b_1$$

et la moitié inférieure (B) possède les noeuds

$$B : \qquad 1b'_{m-1}b'_{m-2}...b'_2b_1$$

Le noeud $x = b_m b_{m-1}...b_1$ de A est équivalent au noeud $y = \bar{x} = \bar{b}_m\bar{b}_{m-1}...\bar{b}_1$ de B. En effet, par l'opération de perfect shuffle f, on a que $x \to f(x)$ et $y \to f(y) = \overline{f(x)}$. De même, par l'opération d'échange g, on a que $x \to g(x)$ et $y \to g(y) = \overline{g(x)}$

Comme on le voit facilement, seule l'opération de perfect shuffle mène de la moitié A vers la moitié B. Ce sont en effet les noeuds $x = 0b_{m-1}...b_21 \in A$ et

$y = 1b_{m-1}...b_20 \in B$ qui seront envoyés d'une moitié à l'autre. Leur nombre donne la largeur bissectionnelle : elle vaut $N/2$ qui est le nombre de noeuds de la forme $0b_{m-1}...b_21$ dans A plus le nombre de noeuds de la forme $1b_{m-1}...b_20$ dans B.

Le diamètre du réseau shuffle-exchange vaut $2\log_2(N)-1$, ce qui est le nombre maximum d'étapes de routage, selon l'algorithme standard décrit ci-dessous.

Pour comprendre l'algorithme de routage, remarquons tout d'abord qu'un noeud voisin du noeud X se caractérise par une adresse qui diffère de celle de X soit par une rotation circulaire de ses bits (opération de perfect shuffle), soit par le complément du dernier bit (opération d'échange). Pour trouver un chemin reliant un noeud X à un noeud Y, il faut atteindre des noeuds intermédiaires à l'aide des deux opérations de perfect shuffle ou d'échange.

En appliquant un nombre de rotations égal au nombre total de bits (c'est-à-dire $\log N$ shifts circulaires), l'adresse du noeud X est renvoyée sur elle-même. Pour construire l'adresse de destination à partir de l'adresse source, on corrige les bits faux à l'aide de l'opération d'échange, à mesure qu'ils passent en première position.

Ainsi, afin d'obtenir le noeud Y en partant de X, on fait précéder le i^{eme} shift par une opération d'échange (en complémentant le dernier bit), si les i^{emes} bits de X et Y sont différents. Ainsi, dans l'exemple de la figure 3.22, on va du noeud 7 (111) au noeud 2 (010) par la séquence d'opération suivante :

$$111 \underbrace{\xrightarrow{E} 110 \xrightarrow{S} 011}_{\text{Etape 1}} \underbrace{\xrightarrow{NE} 011 \xrightarrow{S} 101}_{\text{Etape 2}} \underbrace{\xrightarrow{E} 100 \xrightarrow{S} 010}_{\text{Etape 3}}$$

où $\xrightarrow{E}$ correspond à une opération d'échange et $\xrightarrow{S}$ une opération de shuffle. Le symbole $\xrightarrow{NE}$ indique que que l'opération d'échange n'est pas faite à la deuxième étape puisque les deuxièmes bits de l'adresse d'expédition et de destination sont identiques. Ce parcours demande donc $\log(N)$ étapes pour l'échange et $\log(N)$ étapes de shift.

On remarque que, souvent, un chemin plus court peut être trouvé. Cependant, toutes les étapes mentionnées ci-dessus sont nécessaires pour passer du noeud (000) au noeud (111).

3.2.6 Les réseaux totalement connectés et les réseaux aléatoires

Comme on l'a souligné plus haut, un réseau totalement connecté n'est guère envisageable dans le cas de systèmes massivement parallèles. Il n'est évidemment pas scalable et atteindrait vite un prix prohibitif.

Cependant, cette situation théorique est intéressante pour en comparer les propriétés. Chacun des N noeuds d'un tel système serait de degré $N-1$, ce qui implique $N(N-1)/2 \sim N^2$ connexions. Le diamètre d'un tel réseau est $D = 1$, indépendant de N. Quant à la largeur bissectionnelle, elle est de $N^2/4$, puisque

de chacun des $N/2$ noeuds d'une moitié de réseau partent $N/2$ liens vers l'autre moitié.

Mentionnons encore que certaines études ont porté sur des réseaux dit aléatoires, qui sont constitués d'une topologie de base (par exemple un anneau) sur laquelle sont ajoutés aléatoirement des liens entre paires quelconques de processeurs. Les réseaux aléatoires offrent en général de bonnes propriétés de routage et sont tolérants aux pannes.

3.3 Les réseaux dynamiques

Les topologies statiques telles que celles que nous venons de voir sont d'autant plus efficaces que le problème considéré ait une structure adaptée au réseau disponible : une topologie en forme de grille est adaptée à la discrétisation d'une équation différentielle sur un espace à deux dimensions, mais pas forcément au problème du tri d'une liste de nombres. Les topologies dynamiques, en offrant la possibilité de changer les liens du réseau, garantissent une souplesse qui s'approche mieux de ce qu'on attend d'une architecture parallèle à usage général. Cependant, il y a toujours une pénalité de prix et de complexité dont il faut tenir compte. Du moins complexe et moins cher au plus performant et plus onéreux, on peut citer :

1. Les connexions par un bus.
2. Les réseaux de commutateurs multiétages et arbres dynamiques.
3. Les réseaux crossbar (commutateurs matriciels)

A part pour le cas d'un bus de connexion, les topologies dynamiques se caractérisent par un réseau de commutateurs ou *switches*. Les processeurs et éventuellement les modules mémoire sont reliés aux points d'entrées et de sorties de ces réseaux de commutateurs. Ainsi, on obtient une architecture qui dissocie bien la partie réseau de la partie processeurs, ce qui est un avantage pour la conception des systèmes et leur possibilités d'évolution.

3.3.1 Les connexions par bus

Le bus est la manière de connecter entre eux différents processeurs qui s'apparente le plus à l'architecture des ordinateurs séquentiels classiques. Un bus évite de relier chaque unité à toutes les autres : une même ligne est partagée par plusieurs processeurs, une mémoire commune et éventuellement des unités périphériques, comme le montre la figure 3.23.

Une connexion par bus utilise une technologie proche de celle utilisée pour les machines séquentielles, pour lesquelles de nombreux standards industriels ont été développés. Ce mode de connexion est utilisé dans plusieurs multiprocesseurs symétriques (SMP).

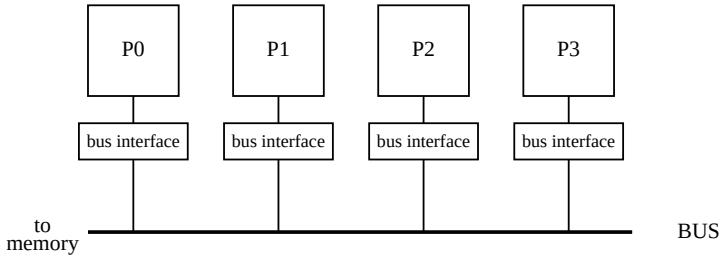


FIGURE 3.23 – *Schéma d'une interconnexion par bus partagé.*

Si ce mode de connexion est relativement simple et bon marché (coût proportionnel au nombre d'unités connectées), il reste limité à des systèmes comprenant peu de processeurs. En pratique, un bus est constitué de plusieurs lignes physiques : typiquement, il y a un bus de données, un bus d'adresses, un bus de synchronisation et d'interruption et, finalement, un bus d'arbitrage qui est nécessaire pour gérer la compétition entre les processeurs accédant au bus.

Pour une connexion par bus, le concept de diamètre du réseau ne fait pas grand sens. La latence est la grandeur correspondante à considérer. Le temps de latence d'un bus est pratiquement indépendant du nombre N de noeuds présents. Par contre la bandwidth d'un bus est inversement proportionnelle à N car tous les processeurs doivent se partager la même voie de communication, l'un après l'autre.

La bandwidth totale maximum d'un bus est limitée par des contraintes technologiques : elle est donnée par la largeur du chemin disponible et la fréquence d'horloge du bus, limitée elle par la nature physique de la connexion. Ces contraintes limitent le nombre maximum de processeurs que l'on peut attacher à un même bus puisque seul un accès à la fois est possible. Heureusement, l'existence de mémoires cache permet de diminuer le trafic de données sur le bus.

Ces mémoires, en stockant localement des données, évitent en effet les allerretours vers la mémoire centrale. Le problème est alors d'assurer la cohérence des données dans l'ensemble des mémoires caches (une même donnée peut être recopiée plusieurs fois dans des caches différents). En utilisant plusieurs bus et des mémoires caches, on peut avoir jusqu'à 30 processeurs interconnectés selon cette méthode.

Remarquons encore que pour assurer rapidité et fiabilité du transfert d'informations dans un bus, il est important de le faire aussi court que possible : un conducteur s'oppose à un changement de voltage en raison de sa capacité et de son inductance propre. Ces quantités sont proportionnelles à la longueur du conducteur. Cependant, plus un bus est court, plus il est difficile d'y brancher un grand nombre d'éléments.

Arbitrage d'un bus La gestion du partage d'un bus se fait par le contrôleur du bus qui possède une logique d'arbitrage permettant d'allouer et désallouer le bus. Cet arbitrage se fait directement au niveau hardware, selon plusieurs stratégies courantes :

1. Priorité fixe ou *daisy chaining* : les différentes unités rattachées au bus reçoivent une priorité fixe qui règle leur accès au bus. Si deux unités demandent simultanément le bus, celle de priorité maximum l'emporte. Les unités sont d'ailleurs réparties physiquement le long du bus selon l'ordre de ces priorités (daisy chaining). Cette technique est simple à mettre en oeuvre mais pas symétrique. Une unité peut en principe attendre un temps infini avant d'accéder au bus.
2. Découpage en tranches : chaque unité reçoit un intervalle de temps d'accès au bus, selon un ordre fixe. Si une unité n'utilise pas le bus pendant cette période d'allocation, le bus reste inactif. Cela conduit à une faible utilisation des ressources mais garantit un temps d'attente fini et une symétrie entre les unités. Cette stratégie est plus adaptée à des systèmes synchrones qui permettent un meilleur usage des tranches de temps.
3. Priorités dynamiques : dans ce cas, on attribue des priorités comme dans le premier cas, mais celles-ci peuvent changer au cours du temps. Par exemple, le premier arrivé est le premier servi pour acquérir une priorité plus grande. Cette stratégie est techniquement plus difficile à réaliser mais utilise efficacement les ressources de façon symétrique et minimise les temps d'attente.

3.3.2 Le crossbar switch

Le réseau crossbar, ou matrice de points de croisement est l'opposé du bus : il a une haute complexité et permet un trafic important de message. Un crossbar switch est une grille dont chaque point d'intersection est un commutateur qui permet de relier entre eux deux éléments quelconques du réseau (par exemple un processeur à une mémoire, ou deux processeurs ensemble). C'est une solution couramment utilisée dans les ordinateurs haut de gamme ayant plusieurs processeurs et une mémoire partagée composée de plusieurs bancs. Un crossbar permet aussi de réaliser une topologie en «étoile» dans une architecture à mémoire distribuée, ainsi que l'illustre la figure 3.24.

Un crossbar peut être considéré comme un commutateur à N entrées et N sorties et permettant de réaliser n'importe quelle permutation $i \rightarrow f(i)$ de l'ensemble des entrées dans l'ensemble des sorties. Il y a $N!$ états différents du crossbar produisant des communications *one-to-one* différentes. La figure 3.25 montre un crossbar switch avec N processeurs en entrée et N modules mémoire en sortie.

La connectivité d'un crossbar est donc totale. Mais, si tous les chemins sont possibles, seul l'accès à des noeuds non occupés est possible. Si 2 processeurs

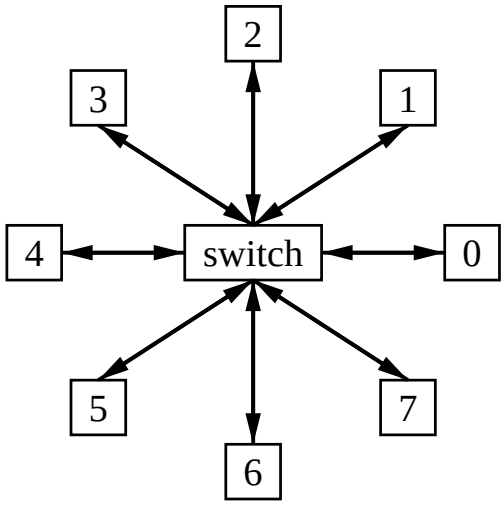


FIGURE 3.24 – Plusieurs processeurs sont courament interconnecté par un switch central qui a la fonctionnalité d'un crossbar.

veulent accéder à un même noeud, l'un d'eux devra attendre. Par contre, s'il n'y a pas de tels conflits, N chemins indépendants et sans intersections pourront être simultanément établis pour un élément de commutation matriciel de taille $N \times N$. Le temps de latence est dépendant du temps de réaction des commutateurs situés aux intersections de la grille et est indépendant de N.

Le coût d'un tel réseau croît comme N^2 en ce qui concerne les commutateurs et les fils de connexions, ce qui limite l'usage de ce type d'interconnexion à des systèmes comprenant relativement peu d'entrées.

La figure 3.26 donne une réalisation simple d'un crossbar à l'aide de commutateurs élémentaires 2×2 placés en chaque point (i, j) de croisement. Ces commutateurs sont caractérisés par deux états internes c_{ij} indiquant une commutation directe ($c_{ij} = 0$) ou croisée ($c_{ij} = 1$), comme l'indique la figure 3.26. Pour relier l'entrée E_i à la sortie S_j il suffit que

$$c_{ij} = 1, \qquad c_{kj} = 0, k > i \qquad \text{et } c_{i\ell} = 0, \ell < j$$

Dans l'exemple de la figure 3.26, on établit les connexions $E_1 \to S_2$, $E_2 \to S_1$ et $E_3 \to S_3$. Cependant, ce réseau ne peut réaliser des broadcasts *one-to-many* où une même entrée est connectée simultanément à plusieurs sorties. De même, si plusieurs entrées veulent envoyer simultanément un message sur la même sortie, le réseau de la figure 3.26 va systématiquement privilégier l'entrée dont l'indice de ligne est le plus grand.

Pour gérer correctement ces conflits il faut introduire de la logique supplémentaire. Ainsi, dans un crossbar, chaque point de croisement a la complexité d'un bus, ce qui explique le coût important d'un tel réseau.

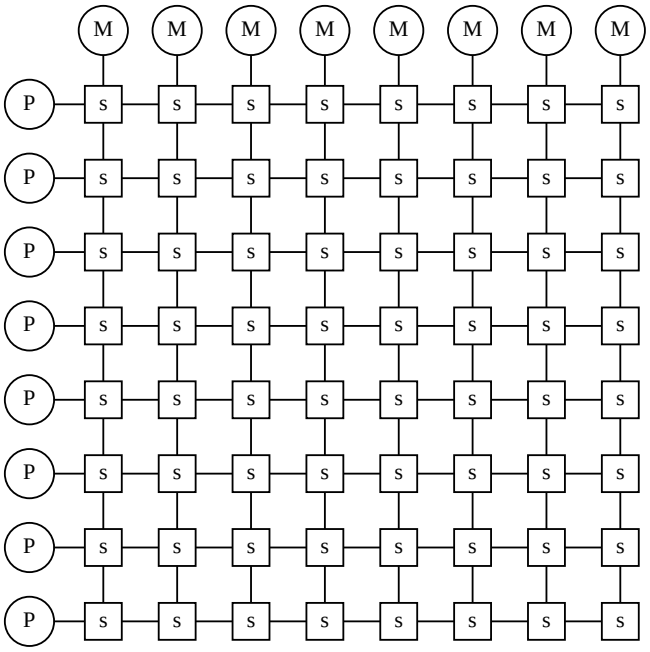


FIGURE 3.25 – *Schéma d'un crossbar* 8×8. *Les éléments notés s apparaissant aux points de croisement sont les switches. Cet exemple montre une situation où 8 processeurs sont reliés à 8 mémoires.*

3.3.3 Les réseaux multiétages

Ce type de réseau se situe entre le bus et l'interconnexion matricielle. Il est plus complexe et plus performant que le premier mais moins onéreux que le deuxième. L'idée de l'interconnexion multiétages (*multistage crossbar switch* en anglais) est de remplacer un commutateur matriciel $N \times N$ par un ensemble de couches comprenant des commutateurs 2×2 (ou éventuellement un peu plus grand). Par exemple, pour un système de 4 entrées et 4 sorties, on peut imaginer le dispositif illustré sur la figure 3.27.

Dans cette figure, chacune des boîtes est un commutateur 2×2 dont les fonctions internes de commutation sont schématisées sur la figure 3.28. Elles permettent une commutation directe ou croisée.

Ce dispositif permet de relier une quelconque des quatre entrées avec une quelconque des quatre sorties. Par exemple, pour mettre en contact l'entrée 1 avec la sortie 4 il faut avoir une commutation croisée dans les commutateurs en haut à gauche et en bas à droite. Cependant, ce choix contraint les autres connexions : l'entrée 2 ne peut alors être connectée qu'à la sortie 1 ou 2, selon l'état du commutateur en haut à droite. Ce système ne peut donc pas réaliser toutes les permutations des quatre entrées sur les quatre sorties, à la différence d'un crossbar 4×4. Il permet néanmoins d'assurer quatre communications simultanées.

Il faut remarquer qu'il est courant dans les machines vectorielles ou parallèles

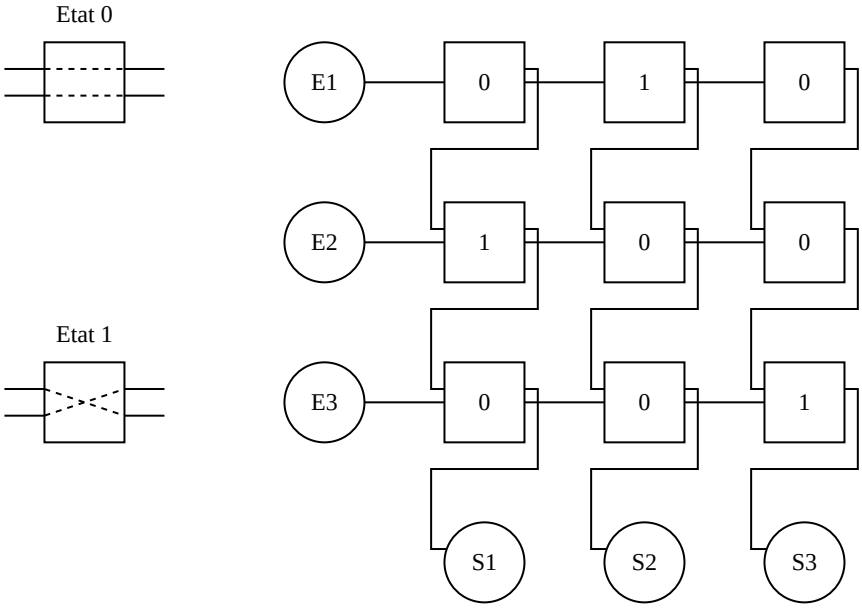


FIGURE 3.26 – *Réalisation d'un crossbar simplifié à l'aide de commutateurs binaires c_{ij} dont les deux états sont décrits sur la gauche de la figure. L'indice i dénote les lignes et j les colonnes. Les points d'entrées sont notés E_1, E_2, E_3 et les sorties S_1, S_2 et S_3.*

d'avoir la mémoire centrale divisée en plusieurs parties, chacune ayant son propre canal d'entrée-sortie. Chacune de ces parties s'appelle un **banc** de mémoire. Une structure avec des bancs permet un accès plus rapide à la mémoire car plusieurs données peuvent être accédées simultanément. Cela ne résout évidemment pas le problème de *conflit de bancs* qui est une situation où des données appartenant au même banc doivent être lues simultanément.

Les réseaux Oméga

Les réseaux Oméga sont un exemple type d'interconnexion multiétages. Ils sont basés sur une topologie de connexions provenant des réseaux "perfect-shuffle," comme le montre la figure 3.29 et servent de base à toute une famille de réseaux multiétages. Un réseau Oméga consiste en $\log_2 N$ colonnes contenant chacune $N/2$ commutateurs 2×2. Son coût est donc proportionnel à $N \log_2 N$.

Chaque élément commutateur a les mêmes fonctions que celles décrites par la figure 3.28, plus encore deux fonctions dites de "broadcast" qui permettent de dupliquer un même message sur deux destinataires à la fois (voir figure 3.30). Avec cette fonctionnalité, un broadcast "one-to-all" est obtenu en une traversée des $\log_2 N$ couches.

Dans un réseau Oméga, chaque paire d'entrée-sortie ne peut être connectée

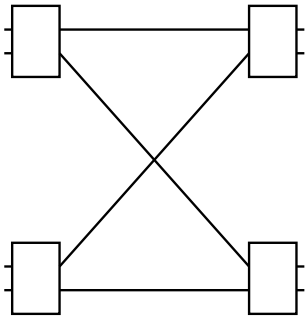


FIGURE 3.27 – *Exemple simple d'un réseau multiétages.*

FIGURE 3.28 – *Switches ou éléments de commutation.*

que d'une seule manière, c'est-à-dire qu'il n'existe qu'un seul chemin possible pour les relier. Ceci constitue une faiblesse de ce réseau, comme nous allons le voir. En contrepartie, le routage se fait de manière simple. Les communications se font typiquement par l'échange de messages qui contiennent eux-mêmes l'information pour se déplacer dans le réseau (le contrôle de la commutation se fait de manière locale, sans nécessiter une stratégie commune).

Routage : L'algorithme pour l'acheminement d'un message est le suivant : l'adresse de destination est un nombre entre 0 et $N-1$. Par conséquent, en binaire, elle contient $k = \log_2 N$ bits, soit autant de bits qu'il y a d'étages de commutateurs. A chaque étage, le bit dominant de l'adresse est examiné et supprimé de l'adresse (ainsi, à chaque étape le nombre de bits de l'adresse diminue de un). Si ce bit dominant est 0, le message est envoyé par la sortie supérieure du commutateur alors que s'il vaut 1, le message quitte le commutateur par la sortie inférieure.

Cet algorithme s'inspire de celui discuté précédemment pour le réseau shuffle-exchange (paragraphe 3.2.5) : on remarque que les liens entre les différents étages sont construits avec la permutation "shuffle inverse". Par ailleurs, les commutateurs 2×2 réalisent la permutation d'échange. Pour mieux comprendre l'algorithme de routage, supposons que, en représentation binaire, l'adresse de l'expéditeur est $e_1 e_2 \dots e_k$ et celle du destinataire $d_1 d_2 \dots d_k$. A l'entrée du premier étage de commutateurs, on se trouve, par la permutation shuffle inverse, en position $e_2 e_3 \dots e_k e_1$. Si alors on choisit la sortie du haut du commutateur, on se retrouve en $e_2 e_3 \dots e_k 0$. Par contre, la sortie du bas mène en $e_2 e_3 \dots e_k 1$. Comme ce choix est dicté par la valeur du bit le plus significatif d_1 de l'adresse du destinataire,

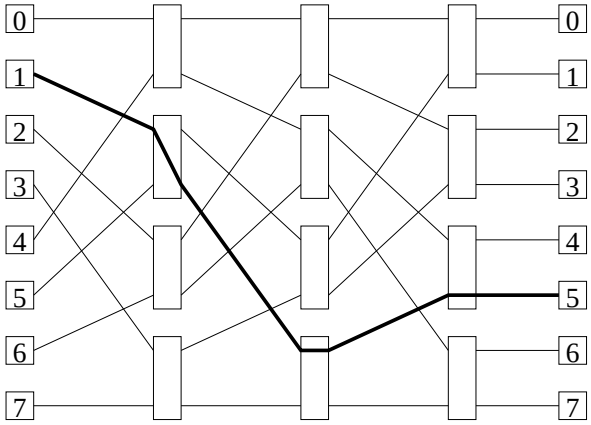


FIGURE 3.29 – *Réseau Oméga à huit entrées.*

FIGURE 3.30 – *Switches du réseau oméga pour le broadcast.*

on choisit donc la sortie $e_2 e_3 \dots e_k d_1$. De même, au deuxième niveau, on entrera en position $e_3 \dots e_k d_1 e_2$ et on en resortira en $e_3 \dots e_k d_1 d_2$. En poursuivant ainsi, on reconstruit, étage après étage, l'adresse de destination. Cela démontre l'algorithme de routage et prouve qu'il existe un chemin unique reliant chaque paire d'entrée-sortie.

Cet algorithme de routage donne un temps de latence du réseau en $\log_2 N$, correspondant au nombre d'étapes qu'un message doit franchir pour arriver à destination. On vérifie aisément sur un exemple qu'un message issu de l'entrée 1 à destination de la sortie 5 emprunte, par cet algorithme, le chemin décrit par la figure 3.29 du réseau oméga. On s'aperçoit aussi qu'un message partant de l'entrée 5 pour la sortie 7 va aussi être redirigé à travers le même fil de connexion par le commutateur du premier niveau (auquel les entrées 1 et 5 sont toutes deux reliées).

Les deux messages ne peuvent donc simultanément poursuivre leur route. L'un doit être retardé par rapport à l'autre, ce qui nécessite des capacités de mémoire au niveau des commutateurs. Cette situation de conflit de chemin est décrite en disant que le réseau Oméga est *bloquant*. La connectivité, bien que totale en principe, est limitée par ces effets de blocage.

Il faut remarquer que les réseaux à interconnexion multiétages permettent de renvoyer une deuxième série de messages avant que la première ne soit arrivée et ainsi de suite jusqu'à $\log_2 N$ groupes de messages. Cette possibilité de faire du "pipelining" augmente la "bandwidth" effective du réseau. Cela n'est par contre plus possible si on utilise une version plus simple du réseau Oméga, constitué

d'une seule couche d'éléments commutants sur lesquels les messages recirculent $\log_2 N$ fois.

Le fat-tree

Le fat-tree (arbre gras) est un réseau d'interconnexion en arbre, mais qui est dynamique en ce sens que les processeurs (et les noeuds d'entrée-sortie) occupent les extrémités de l'arbre (feuilles), alors que les noeuds intérieurs sont des switches.

Le fat-tree a la propriété que son épaisseur croît à mesure que l'on se rapproche de la racine. On augmente ainsi la bandwidth là où il y en a le plus besoin, c'est à dire là où convergent les messages qui doivent traverser tout l'arbre. Un exemple de fat-tree est représenté sur la figure 3.31. Les cercles indiquent les processeurs et les rectangles les commutateurs. On remarque que le nombre de liens entre commutateurs augmentent d'une unité à chaque niveau de l'arbre. La bandwidth au niveau de la racine est ici trois fois plus élevée qu'au niveau des processeurs. De plus, le trafic local de message n'a pas besoin de remonter au sommet de l'arbre.

On constate aussi qu'un arbre dynamique permet de définir facilement des partitions de processeurs, comme indiqué sur la figure 3.31. Ces partitions peuvent être attribuées à différents utilisateurs sans que les trafics de messages ne se perturbent. Dans les architectures parallèles actuelles, on préfère en effet le "space-sharing" au time-sharing pour une utilisation multi-utilisateurs. Le découpage en tranches temporelles est rendu plus complexe à cause des messages qui circulent de manière asynchrone dans le réseau.

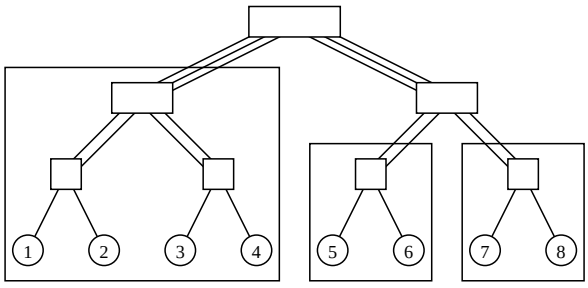


FIGURE 3.31 – *Exemple de fat-tree avec 8 processeurs et 3 partitions*

Le routage dans un fat-tree se fait selon le même principe que dans un arbre binaire statique. On peut toutefois choisir le lien physique utilisé de façon à équilibrer la charge du réseau. Le temps de latence est proportionnel au nombre de switches à traverser pour aller d'une extrémité à l'autre de l'arbre, c'est à dire $2\log_2(N) - 1$.

La largeur bissectionnelle est obtenue en considérant le nombre de liens qui arrivent au commutateur racine. Dans le cas illustré sur la figure, on obtient une

largeur bissectionnelle de $\log_2 N$. On peut aussi faire croître plus rapidement le nombre de connexions sur chaque switch, mais cela est évidement au détriment du prix et de la complexité des switches.

En général, les switches présents dans les fat-trees ont une fonctionnalité élevée qui leur permet de combiner des messages, de faire des broadcasts et évidement de mémoriser temporairement des messages. On peut ainsi réaliser très naturellement des primitives de communications collectives comme un SCAN (parallel prefix) : les données issues des feuilles remontent dans l'arbre. En chaque noeud, on calcule la somme des valeurs "enfants." On mémorise temporairement la valeur venue de droite et on envoie au processeur "père" la somme. Arrivé à la racine, la somme complète est redistribuée intégralement sur le fils droit. Par contre, le fils gauche reçoit la quantité obtenue en soustrayant de cette somme la valeur précédemment stockée dans le noeud. En poursuivant cette démarche jusqu'aux feuilles, on obtient le résultat de toutes les sommes partielles des valeurs initiales.

Le fat-tree (et ses variantes) est un réseau d'interconnexion hiérarchique devenu très apprécié. Il a pris peu à peu le relais de l'hypercube dans les machines massivement parallèles. Il a été utilisé notamment dans la Connection Machine CM-5 (qui fut la première à mettre en oeuvre cette solution) et d'autres architectures comme la KSR de Kendall Square Research et la CS-2 de Meiko, Parsys et Telmat. La raison de ce succès est que l'on peut montrer que les autres topologies peuvent efficacement être plongées dans un fat-tree. De plus, le fat-tree est un réseau dynamique bien adapté aux architectures à mémoire distribuée, pour lesquelles le principe de localité est important (les communications les plus fréquentes ont lieu entre processeurs voisins).

Finalement, le fat tree offre une tolérance aux fautes, grâce à la redondance de ses connexions. Même si un lien est interrompu, le réseau reste un fat tree, peut-être moins épais, mais un fat tree quand même. Les mêmes algorithmes de routage sont toujours valables. Ce n'est évidemment pas le cas d'un réseau en grille ou en hypercube où la disparition d'un lien change localement la topologie.

En guise d'illustration, la figure 3.32 montre un fat-tree d'ordre 4 dont chaque noeud interne est composé de plusieurs routeurs reliés à quatre noeuds "enfants" et deux ou quatre noeuds "parents." Les switches des deux premiers niveaux du réseau ont deux parents, puis les suivants en ont quatre, ce qui garantit une bandwidth constante à partir du troisième niveau. Un réseau comprenant deux niveaux contient $4^2 = 16$ feuilles et un réseau avec 3 niveaux en contient $4^3 = 64$. Cette dernière configuration est utilisée pour les machines avec 32 processeurs de calculs plus des processeurs de contrôle et des noeuds pour les entrée-sorties. Les figures 3.33 et 3.34 montrent des configurations avec 64 et 256 noeuds (feuilles).

Dans un tel arbre, les messages sont acheminés selon les algorithmes standards de routage : on remonte jusqu'au premier ancêtre commun, puis on redescend vers le processeur de destination. A la montée, cependant, plusieurs chemins sont possibles, ce qui assure un équilibre du trafic dans le réseau. Pour "redescendre," il n'y a qu'un seul chemin possible et sa description ne dépend pas du chemin

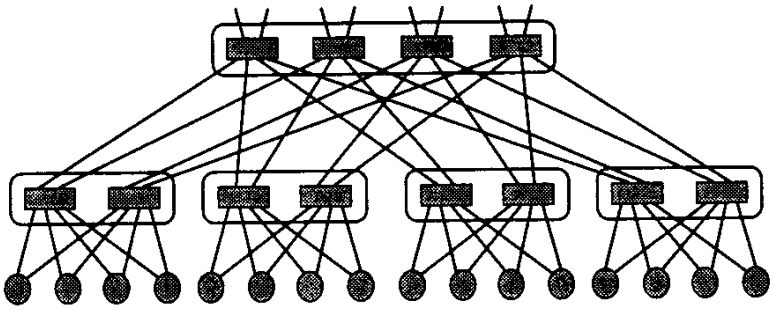


FIGURE 3.32 – *Réseau fat-tree d'ordre 4 (Connection Machine CM-5). Les feuilles de l'arbre (indiquées par les cercles) représentent les interfaces réseau sur lesquels viennent se placer les processeurs ou les entrée-sorties.*

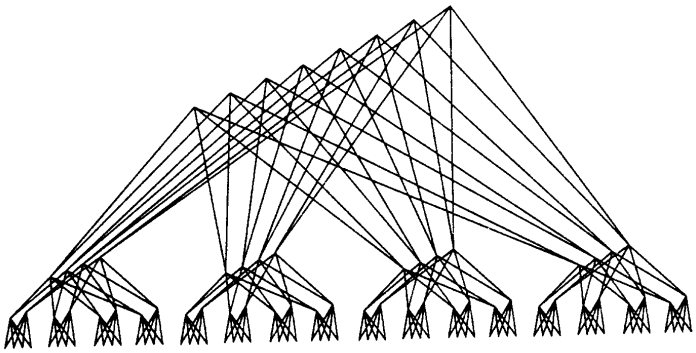


FIGURE 3.33 – *Réseaux fat-tree d'ordre 4 avec 64 noeuds (CM-5).*

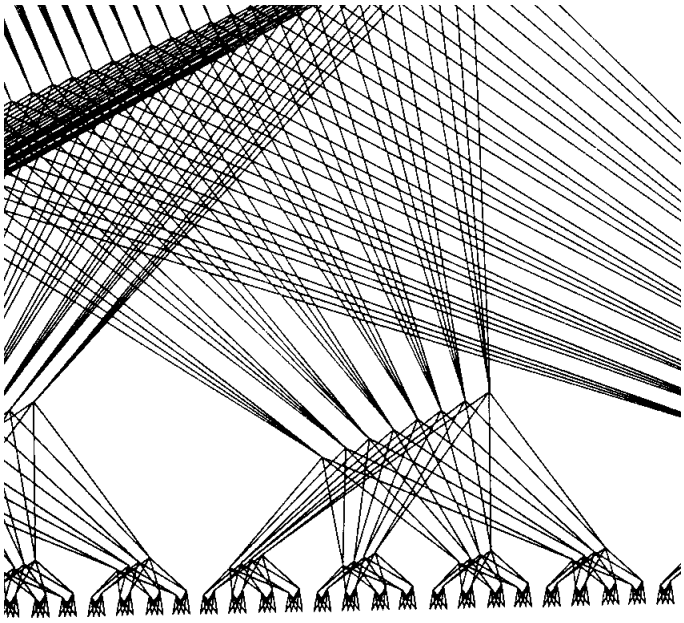


FIGURE 3.34 – *Réseaux fat-tree de la CM-5 avec 256 noeuds.*

emprunté à la montée.

Le Réseau haute-performance de l'IBM SP2

La ligne des ordinateurs SPx (Scalable Power) d'IBM a obtenu un grand succès commercial à son apparition, au début des années quatre-vingt-dix. Les noeuds de calculs (soit des processeurs ou des SMP) sont reliés entre eux par un réseau d'interconnexion appelé *High Performance Switch*. La machine peut indifféremment fonctionner comme un ensemble de mono-processeurs ou comme une machine parallèle extensible.

Le high performance switch est un exemple de réseau multiétage moderne. Il se compose de commutateurs élémentaires qui sont des crossbars 8×8 (voir figure 3.35). Sur chacun, quatre processeurs peuvent être connectés. Quatre connexions sont utilisées pour expédier les données dans le réseau et quatre autres pour recevoir des données provenant du réseau. Ainsi, chaque processeur utilise deux connexions du crossbar, soit une entrée et une sortie.

Un tel crossbar suffit à relier complètement 4 processeurs. Afin de connecter davantage d'éléments, une structure multiétages est nécessaire. Les 8 connexions restantes du crossbar sont alors reliées, par une topologie de perfect shuffle inverse, à d'autres crossbars identiques. Ces derniers constituent le deuxième niveau du réseau.

La structure de base de l'architecture du réseau est une carte nommée *switch board* qui comporte ces deux étages de crossbars. La figure 3.36 en montre le

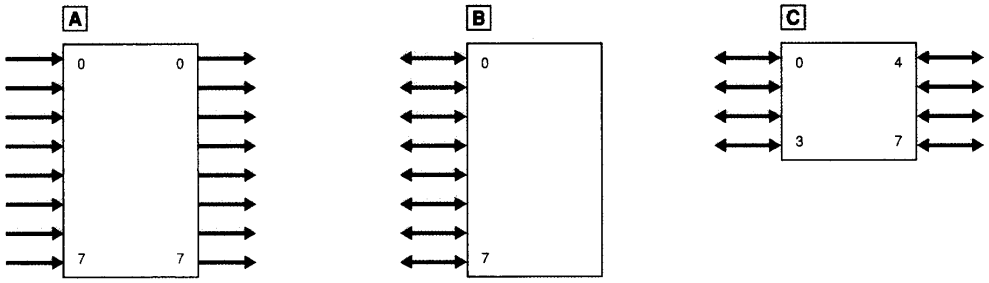


FIGURE 3.35 – *Trois façons équivalentes de représenter le crossbar* 8×8 *(switching element) de la SP.*

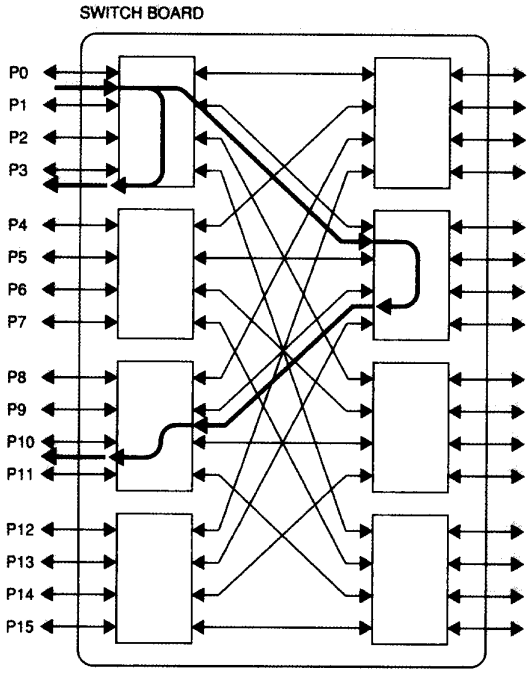


FIGURE 3.36 – *L'élément de base du réseau d'interconnexion de la SP est composé d'une carte (switch board) sur laquelle 8 crossbars* 8×8 *sont reliés par une permutation perfect shuffle inverse. Seize processeurs sont reliés à la partie gauche de la carte.*

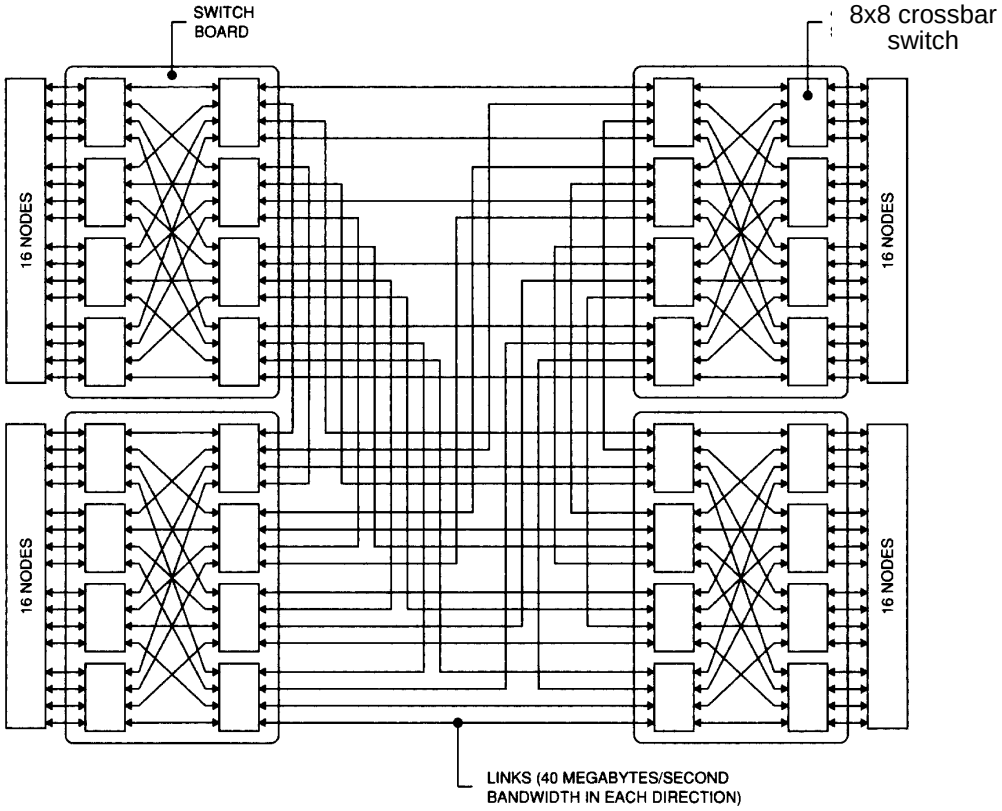


FIGURE 3.37 – *Scalabilité du switch : quatre switch boards peuvent être connectées comme l'indique la figure pour donner une configuration à 64 processeurs.*

schéma. Le switch board suffit pour assurer la connexion entre 16 processeurs, en offrant des chemins multiples et un partitionnement naturel en sous-groupes de processeurs. La figure montre le chemin d'un message partant du noeud 0 en direction des noeuds 3 et 10.

La scalabilité du réseau est assurée en connectant plusieurs switch boards entre eux. La figure 3.37 montre une configuration à 64 noeuds.

Le réseau Dragonfly

Le réseau dit «dragonfly» est un réseau d'interconnexion moderne. Il équipe la machine Piz Daint du CSCS (2020). Il est composé de groupes de noeuds fortement connectés desquels partent de nombreuses connexions vers les autres groupes. Cette hiérarchie, qui fait penser à un réseau à étages, permet de réduire la distance entre les différents noeuds de la machine. Les groupe de noeuds doivent être de haut degré (on parle en anglais de «high radix nodes»), afin d'offrir suffisament de liens vers les autres groupes. Ce haut degré est typiquement obtenu par le nombre de ports libres des processeurs de chaque groupe. Ce nombre augmente avec la taille des groupes.

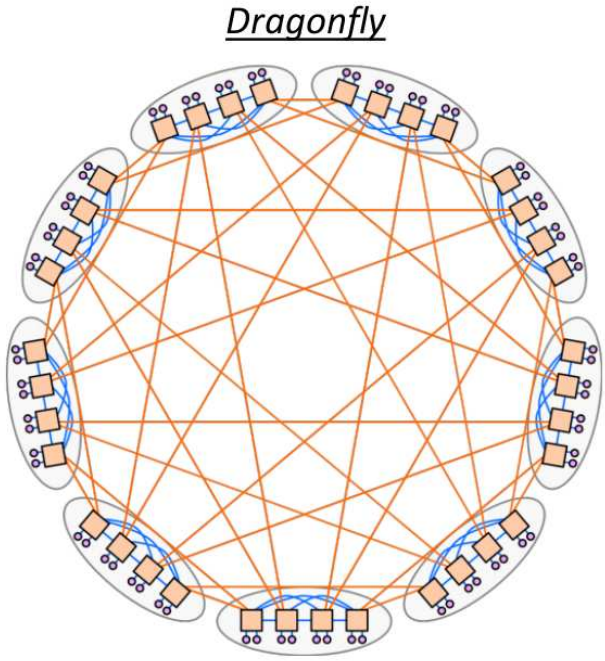


FIGURE 3.38 – Exemple de réseau dragonfly.

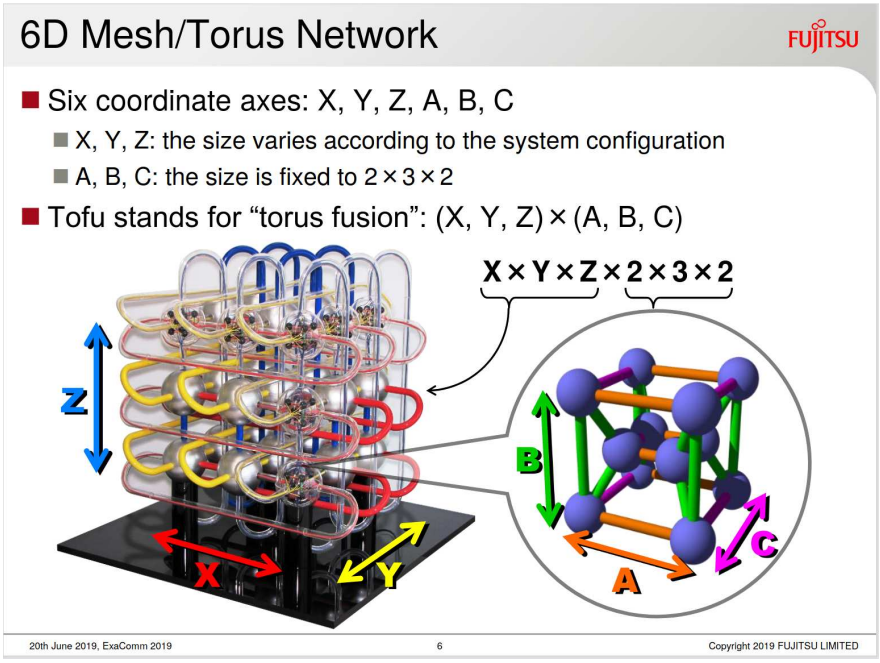


FIGURE 3.39 – Diagramme du réseau torique Tofu à 6 dimension de la machine Fugaku (Fujitsu)

La terminologie «dragonfly», ou «libellule» en français, vient du fait que ce réseau fait penser à de longues ailes (les liens entre groupes) et un corps dense (les groupes).

Le réseau Tofu à 6D

Le réseau dit «Tofu» équipe la machine japonaise Fugaku qui domine le top 500 depuis juin 2020. C'est un réseau dont la topologie de base est une grille tridimensionnelle, mais telle que chaque noeud de cette grille est lui-même un petit réseau à 3 dimensions. Il faut donc 6 coordonnées pour spécifier un processeur. La figure 3.39 illustre cette architecture. Le réseau est dit torique pour indiquer que les processeurs d'un bord du réseau sont reliés à ceux du bord opposé. Chaque lien du réseau à une bande passante de 100 Gbits/s.

Deuxième partie

Performances

Chapitre 4

Travail, Speedup et Efficacité

4.1 Gain et limitation des performances

Le parallélisme a pour but d'exécuter plus rapidement un problème donné. La question est : quel gain peut-on espérer en parallélisant une application donnée. On verra dans ce paragraphe que l'on peut donner certaines limites simples à l'augmentation de performances et l'on verra comment tirer le meilleur parti d'un ensemble de processeurs fortement connectés. En particulier, on apprendra qu'il est plus avantageux d'utiliser le parallélisme pour résoudre des problèmes plus gros que pour résoudre un problème donné plus rapidement. La remarque suivante illustre d'ailleurs ce propos de façon humoristique :

> Une femme peut donner naissance à un enfant en neuf mois. Il est par contre difficile à neuf femmes de donner naissance à un enfant en un mois seulement. Mais, neuf femmes peuvent très bien avoir neuf enfants en neuf mois.

Certaines tâches ne se parallélisent pas, le gain n'est obtenu que si on multiplie, sur plusieurs processeurs, l'exécution de telles tâches pour produire *plus* de travail.

4.2 Le degré de parallélisme et speedup

Considérons une application donnée qui est exécutée par un ensemble de processeurs. Ces derniers ne sont en général pas tous continuellement actifs. Définissons le degré de parallélisme $P(t)$ (degree of parallelism) de cette application comme le nombre de processeurs travaillant simultanément au temps t. On peut tracer un profil d'exécution comme par exemple celui de la figure 4.1.

De ce profil d'exécution, nous pouvons obtenir le gain de performance de cette application parallèle par rapport à une exécution purement séquentielle. Chaque processeur (tous supposés identiques) est caractérisé par sa puissance ou son taux d'exécution R (rate) qui correspond au nombre d'opérations qu'il peut effectuer par seconde (on utilise souvent le flops ou les MIPS pour mesurer R).

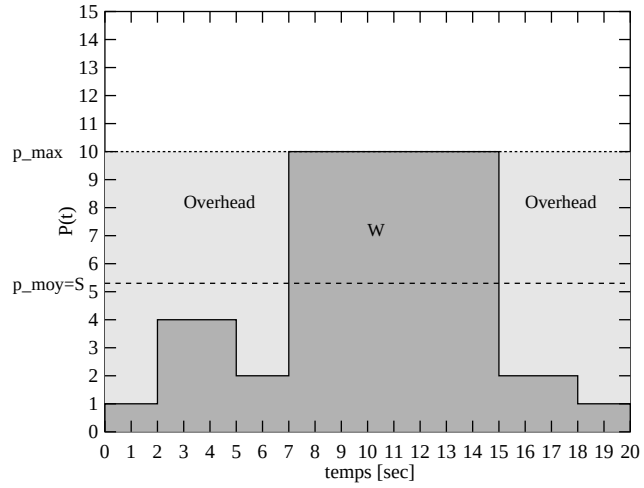


FIGURE 4.1 – *Degré de parallélisme d'une application donnée. $P(t)$ indique combien de processeurs sont simultanément actifs au temps t.*

Le travail total (le nombre total d'opérations) effectué dans notre exemple entre les temps t_{min} et t_{max} est donné par

$$W = \int_{t_{min}}^{t_{max}} P(\tau) R d\tau$$

La quantité $Rd\tau$ est le travail fait pendant l'intervalle de temps $d\tau$ par un processeur. Le produit $P(\tau)Rd\tau$ est donc le travail effectué par tous les processeurs en activité entre les temps τ et $\tau + d\tau$. Graphiquement, W est l'aire hachurée en gris foncé sur la figure 4.1.

Le gain en performance de l'exécution parallèle est mesuré par le quotient du temps T_{serial} qu'un seul processeur aurait mis pour exécuter le travail W par le temps mis par l'ensemble des processeurs. Ce rapport de temps s'appelle le **speedup** S. Le temps d'exécution en parallèle est $T_{parallel} = t_{max} - t_{min}$ et le temps séquentiel est le travail total W divisé par la puissance R d'un processeur : $T_{seq} = W/R$

On obtient donc

$$S = \frac{T_{seq}}{T_{parallel}} = \frac{(W/R)}{t_{max} - t_{min}} = \frac{1}{(t_{max} - t_{min})} \int_{t_{min}}^{t_{max}} P(\tau) d\tau \qquad (4.1)$$

Ceci exprime que le gain de performance S est égal au *parallélisme moyen* de l'application considérée (c'est à dire la hauteur d'un rectangle dont l'aire serait la même que celle découpée par $P(t)$. Cette valeur est indiquée sur la figure 4.1 avec la ligne en traitillés.

Idéalement, on souhaiterait que le speedup soit égal au nombre $p = p_{max}$ de processeurs disponibles ou, en d'autres termes, que $P(t) = p$ en tout temps. Cela

est rarement le cas dans la pratique et la relation (4.1) montre qu'un déséquilibre des charges parmi les processeurs peut fortement influencer les performances d'une application parallèle.

La fait que $P(t) \neq p_{max}$ a une autre conséquence. Comme il est rarement possible d'utiliser les processeurs inactifs pour d'autres tâches que le problème considéré, les cycles d'horloge des processeurs inactifs sont simplement perdus. Leur quantité est représentée sur la figure 4.1 par la zone hachurée en gris clair. Ces cycles perdus, ou plus généralement ceux qui ne correspondent à aucune instruction dans l'exécution séquentielle, sont appelés overhead.

Les surfaces gris clair est gris foncé representent le nombre total de cycles utilisés dans l'exécution parallèle. Ce nombre s'appelle le travail parallèle W_{par} et il est égal au travail séquentiel W additionné de l'overhead Δ. Comme la figure 4.1 l'indique, il vaut aussi le produit $T_{parallel} \times p_{max}$.

En notant $p = p_{max}$ et $T_{par} = T_{parallel}$, on a donc

$$W_{par} = pT_{par}R = W + \Delta \tag{4.2}$$

où R est la puissance d'un processeur.

Parmi les grandeurs qui spécifie le parallélisme, il y a aussi l'**efficacité** E, définie comme

$$E = \frac{S}{p} \tag{4.3}$$

Si le speedup valait p, on aurrait une efficacité de 1, ou de 100%.

Sur la figure 4.1 l'efficacité correspond au rapport p_{moy}/p_{max} et indique le rapport du travail utile au travail total :

$$E = \frac{S}{p} = \frac{T_{seq}}{pT_{par}} = \frac{W}{W_{par}} \tag{4.4}$$

Une efficacité de 100% signifie l'absence d'overhead.

Il y a deux relations importantes qui découlent des résultats précédents. La première se construit en écrivant

$$E = \frac{T_{seq}}{pT_{par}} = \frac{W}{RpT_{par}} \tag{4.5}$$

ce qui implique que

$$T_{par} = \frac{W}{pER} \tag{4.6}$$

On peut interpréter cette relation on disant que le temps parallèle est le travail séquentiel divisé par le nombre de processeurs, mais dont la puissance de ces derniers est reduite d'un facteur $E \leq 1$ et devient $R_{eff} = ER$.

La deuxième relation s'obtient à partir de $pRT_{par} = W + \Delta$, ou Δ est l'overhead. On en déduit que

$$T_{par} = \frac{W}{Rp} + \frac{\Delta}{Rp} \tag{4.7}$$

et que

$$S = \frac{T_{seq}}{T_{par}} = \frac{W/R}{\frac{W}{Rp} + \frac{\Delta}{Rp}} = \frac{p}{1 + \frac{\Delta}{W}} \tag{4.8}$$

La grandeur Δ/W s'appelle l'overhead fractionnaire et exprime la reduction de speedup par le fait que le dénominateur n'est pas égal à 1.

4.3 Lois d'Amdahl et de Gustafson

La relation 4.1 peut être considérée dans un cas particulier où le profil d'exécution est soit purement séquentiel, soit purement parallèle : $P(t) = 1$ ou bien $P(t) = p$. Cela correspond à la situation fréquente où un programme a une partie entièrement séquentielle et une autre entièrement parallélisable sur p processeurs. On va voir comment la partie purement séquentielle influence le speedup en envisageant deux situations différentes.

Loi d'Amdahl : On suppose qu'on a un travail W donné à effectuer et on regarde comment varie S en changeant la portion α ($0 \leq \alpha \leq 1$) de code qui s'exécute séquentiellement. On a évidemment

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

En parallèle, une quantité de travail αW sera quand même exécutée en séquentiel et durera un temps $(\alpha W)/R$. Le reste, soit une quantité de travail $(1 - \alpha)W$ se fera sur p processeurs, p fois plus vite. Donc

$$T_{par} = \frac{\alpha W}{R} + \frac{(1 - \alpha)W}{pR}$$

Ainsi, le speedup s'exprime comme

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \leq \frac{1}{\alpha}$$

Ainsi, quel que soit le nombre de processeurs qui vont travailler dans la partie parallèle du code, le speedup sera limité par la portion du travail qui se fait séquentiellement. Par exemple, si α=10% et que 1000 processeurs sont actifs pour résoudre les 90% restant du travail, le speedup ne sera que de 9.91. Cela est très faible au regard des 1000 processeurs qui ont été utilisés. Ce problème s'appelle le *bottleneck* séquentiel.

Loi de Gustafson : La loi d'Amdahl nous apprend qu'on n'obtient pas toujours un bon speedup en parallélisant une application donnée dont une partie est inévitablement séquentielle. Bien que cette loi s'applique dans plusieurs cas pratiques, elle montre que la bonne manière de faire du parallélisme n'est pas de garder le même problème et d'espérer un speedup conséquent en utilisant beaucoup de processeurs. La loi de Gustafson donne le speedup en n'imposant pas que la taille du problème soit fixe. Plus un problème est grand (p.ex. contient beaucoup de données), plus le parallélisme sera intéressant. Beaucoup d'applications scientifiques ont d'ailleurs besoin de considérer des systèmes de plus en plus grands.

La loi de Gustafson considère un temps total d'exécution parallèle T_{par}. On suppose qu'une portion βT_{par} de ce temps est passé dans la partie séquentielle du programme. Le reste du temps, $(1-\beta)T_{par}$, correspond à une exécution parallèle sur p processeurs.

Le travail total effectué est donc

$$W = \beta T_{par} R + p(1-\beta)T_{par} R$$

où R est toujours le taux d'exécution d'un processeur. Le temps d'exécution total sur un seul processeur est donc $T_{seq} = W/R$ et le speedup s'écrit

$$S = \frac{T_{seq}}{T_{par}} = \beta + p(1-\beta) = O(p)$$

On constate que maintenant, le speedup est linéaire avec le degré de parallélisme p de l'application. Si on prend $\beta = 0.1$ et $p = 1000$, on obtient $S \approx 900$, ce qui est plus encourageant que la valeur obtenue de la loi d'Amdahl. La raison de cette différence est que maintenant le travail total W varie avec la valeur de p et que la fraction de travail séquentiel $W_{seq} = \beta R T_{par}$ par rapport au travail parallèle $W_{par} = p(1-\beta)RT_{par}$ est inversement proportionnelle à p.

Cette situation correspond d'ailleurs mieux à une certaine classe d'applications scientifiques pour lesquelles on peut augmenter le travail parallèle tout en gardant les tâches intrinsèquement séquentielles de taille à peu près constante.

La nécessité d'augmenter la taille d'un problème à mesure qu'on utilise davantage de processeurs est un point fondamental de la parallélisation. Il se comprend aisément sur l'exemple suivant : supposons une situation où chaque processeur contient une seule donnée sur laquelle un calcul est effectué. Si maintenant on augmente le nombre de processeurs sans augmenter le nombre de données, les nouveaux processeurs seront inactifs car ils n'auront aucune donnée à traiter. En conséquence, le speedup ne va pas augmenter. Par cet exemple, on voit qu'il faut s'attendre à augmenter la taille du problème au moins proportionnellement à l'accroissement du nombre de processeurs afin de maintenir les performances de l'exécution.

4.4 Speedups super-linéaires

Nous avons montré dans le paragraphe 4.2 que le speedup d'une application était égal au degré de parallélisme moyen. En conséquence, on a que

$$S \leq p \tag{4.9}$$

où p est le nombre maximum de processeurs impliqués. Pour dériver ce résultat, on a fait l'hypothèse de comparer le temps d'exécution séquentiel avec le temps d'exécution parallèle pour exactement le *même* code ou le *même algorithme*. On a ainsi fait l'hypothèse que le travail séquentiel W_{seq} était égal au travail parallèle W_{par}.

En réalité, on devrait définir

$$S = \frac{T_{sequential}}{T_{parallel}}$$

où $T_{sequential}$ est le temps d'exécution du meilleur algorithme séquentiel et $T_{parallel}$ le temps d'exécution de l'algorithme parallèle considéré. Il se trouve en effet que pour beaucoup de cas pratiques

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

Par exemple, un algorithme séquentiel qui prend m étapes en prendra davantage en parallèle, par exemple $m \log_2(m)$. Sur m processeurs, on aura alors un temps d'exécution parallèle proportionnel à $\log_2(m)$, ce qui rend la relation 4.9 tout à fait réaliste puisque $m/\log_2(m) < m$.

Toutefois, il existe des situations pour lesquelles

$$W_{par} < W_{seq}$$

Le speedup peut ainsi excéder le nombre de processeurs utilisés. On parle alors de speedup super-linéaires. Cela arrive par exemple dans des cas de recherches non déterministes, sur des arbres. Le fait de rechercher en parallèle sur plusieurs branches de l'arbre en même temps permet d'éliminer rapidement certaines d'entre-elles qu'il aurait fallu explorer plus profondément avec un algorithme séquentiel avant de s'apercevoir qu'elles pouvaient être écartées.

Ce problème est illustré par l'exemple donné dans la figure 4.2 qui représente un arbre de recherche. Avec deux processeurs on peut rechercher en parallèle sur deux parties distinctes de l'arbre. Le temps de visite de chaque événement est indiqué sur la figure pour les deux cas d'exécution, parallèle (a) ou séquentielle (b). Supposons que la solution cherchée soit le cercle trouvé au temps $T_p = 3$ par le processeur P_2. Ce même point est visité au temps $T_s = 10$ lors de la recherche séquentielle.

Donc le speedup est $S = T_s/T_p = 10/3$, ce qui est plus grand que 2. On a bel et bien un speedup super-linéaire. Un moyen d'éviter cela est de modifier le mode

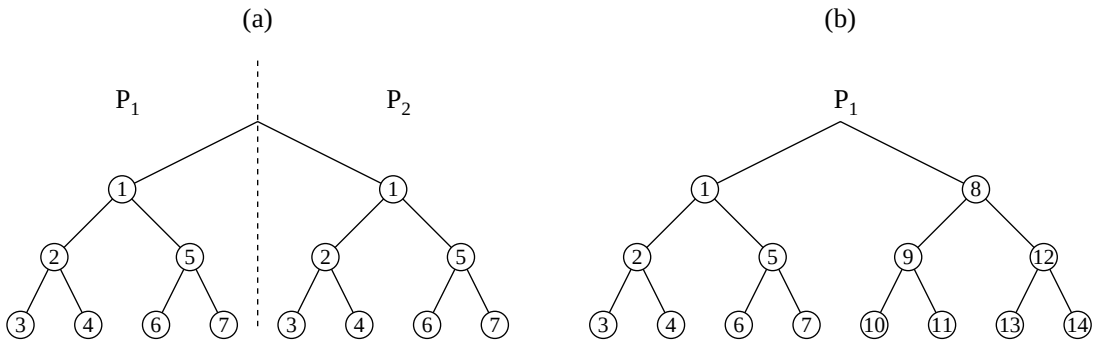


FIGURE 4.2 – *Arbre de recherche parcouru (a) en parallèle par deux processeurs et (b) en séquentiel par un seul processeur. Les nombres indiqués donnent le temps où les événements, représentés par les cercles, sont considérés par un processeur*

de recherche séquentiel, comme indiqué sur la figure 4.3. Si le point recherché est le même que précédemment, ce nouveau parcours trouvera la réponse en 6 unités de temps, c'est a dire exactement deux fois plus lentement que la recherche parallèle.

Malheureusement, cet algorithme n'est pas forcément le meilleur. Supposons en effet que la solution cherchée soit maintenant l'événement visité au temps $t = 11$, dans la figure 4.3. Avec l'ancienne méthode de recherche séquentielle, on aurait trouvé ce même événement en 6 unités de temps, seulement. Comme on ne sait pas d'avance où se trouvera la solution recherchée, on n'a pas de raison de préférer le mode de recherche indiqué dans la figure 4.3 plutôt que celui de la figure 4.2 (b).

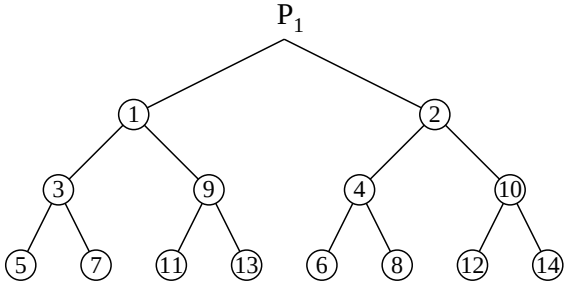


FIGURE 4.3 – *Arbre de recherche visité séquentiellement en imitant le parcours parallèle*

Il y a un autre cas encore de speedup super-linéaire qui, cette fois, n'est pas dû à un avantage algorithmique du parallélisme mais à un avantage hardware. Cela se traduit par le fait que la puissance *effective* du processeur R est supérieure en parallèle qu'elle n'est en séquentiel : $R_{par} > R_{seq}$.

Cela peut arriver dans la situation suivante : les processeurs actuels possèdent des mémoires caches très rapides mais relativement petites. Lors d'une exécution en parallèle, les données du problème sont souvent réparties entre les processeurs. Si la quantité de données distribuées n'est pas trop grande, celles-ci pourront toutes être placées dans la mémoire cache de chaque processeur et il s'en suivra une exécution très efficace du code. Par contre, lors d'une exécution séquentielle, le nombre total de données sera de toute manière trop grand pour une seule mémoire cache et il faudra inévitablement recourir à des lectures et écritures de données en mémoire centrale. Le placement avantageux des données dans le cas parallèle peut conduire à un speedup super-linéaire.

4.5 Overhead résultant de la parallélisation

Jusqu'à présent, nous avons évalué les performances d'une application parallèle en ne considérant que la façon dont le travail total était exécuté par plusieurs processeurs. Mais l'analyse que nous avons faite néglige une partie essentielle du problème, à savoir le **surplus de travail** (ou **overhead**) inhérent à la parallélisation d'une application. En effet, des tâches nouvelles apparaissent lorsque plusieurs processeurs coopèrent à la résolution d'un même problème.

Ces sources d'overhead sont principalement : (1) les communications entre processeurs, indispensables pour permettre l'échange des données ; (2) l'attente résultant de la synchronisation des processeurs après certaines étapes de calcul ; (3) les modifications algorithmiques qui résultent de la parallélisation ; et (4) la coordination, par le système d'exploitation de plusieurs processeurs qui coopèrent (allocation des processus, gestion d'accès concurrents à des ressources partagées,...).

Les communications entre processeurs sont, avec le déséquilibre de charge, les causes principales d'une dégradation des performances d'une application parallèle.

Les communications jouent un rôle important car il est difficilement possible de réaliser une connexion entre processeurs qui soit aussi performante que l'accès d'un processeur à sa propre mémoire. Donc, un échange de donnée s'accompagnera toujours d'un overhead.

Nous allons montrer, par des exemples simples, que l'augmentation du nombre de processeurs ne conduit pas toujours à l'accroissement de performance souhaité, en raison des tâches supplémentaires qui sont nécessaires au processus de parallélisation.

4.6 Systèmes hétérogènes

Comment peut-on mesurer et définir le gain du parallélisme dans le cas où les processeurs ne sont pas tous identiques ? Il peut arriver, en effet, qu'on utilise des processeurs de types différents et de performances inégales que l'on fait coopérer grâce à une bibliothèque d'échange de messages. Dans ce cas, il n'y a pas de référence immédiate pour comparer le cas parallèle au cas séquentiel.

Néanmoins, on peut procéder ainsi pour définir l'exécution parallèle idéale. Soient R_i la puissance du processeur P_i, $i = 1, \dots p$ et W_i la quantité de travail qui lui est attribué. Le temps d'exécution du processeur P_i est donc

$$T_i = \frac{W_i}{R_i}$$

Clairement, l'exécution idéale requiert que tous les T_i soient égaux. Sans quoi, on pourrait toujours imaginer enlever une partie du travail au processeur le plus lent et le redistribuer aux autres afin de diminuer le temps d'exécution parallèle T_{par}.

Avec $T_i = T$, pour tout i, on en déduit que la charge de travail idéale est

$$W_i = R_i T$$

Comme le travail total est la somme des travaux de chaque processeurs, on a

$$W = \sum_{i=1}^{p} W_i = \sum_{i=1}^{p} R_i T = T R_{tot}$$

où, par définition, $R_{tot} = \sum_i R_i$ est la puissance agrégée de tous les processeurs.

Ainsi, le temps d'exécution idéal T s'obtient comme

$$T = \frac{W}{R_{tot}}$$

et on définit **l'efficacité** d'un système hétérogène comme le rapport du temps idéal au temps parallèle observé

$$E = \frac{T}{T_{par}}$$

Cette définition est cohérente avec la définition de l'efficacité dans le cas de processeurs homogènes. Posons en effet que

$$R_{moy} = \frac{R_{tot}}{p}$$

est la puissance du processeur sequentiel utilisé pour la comparaison. Supposons de plus que le travail séquentiel est identique au travail parallèle : $W_{seq} = W_{par} = W$. Dès lors, on a $T_{seq} = W/R_{moy}$ et donc

$$E = \frac{T}{T_{par}} = \frac{W}{R_{tot} T_{par}} = \frac{W}{p R_{moy} T_{par}} = \frac{T_{seq}}{p T_{par}}$$

ce qui est bien la relation attendue.

Il est aussi intéressant de remarquer l'analogie qu'on peut tirer des relations ci-dessus avec un circuit électrique contenant des résistances branchées en parallèle. La relation $W = \sum_i W_i$ suggère que le travail est l'analogue du courant électrique. Le fait que le temps d'exécution idéal est le même pour tous les processeurs suggère que le temps d'exécution est l'équivalent de la tension électrique. De plus, la relation $W_i = R_i T$ (loi d'Ohm) indique que la puissance du processeur correspond à *l'inverse* d'une résistance, en accord avec l'intuition et le fait que $R_{tot} = \sum R_i$ qui, dans le cas de résistances en parallèle s'exprime par $1/r_{eq} = \sum(1/r_i)$.

Chapitre 5

Modèles de Performance

Dans ce chapitre on va s'intéresser à décrire comment le temps d'exécution d'un programme parallèle dépend des différents paramètres du problème et de l'ordinateur. En particulier, on verra précisément le rôle que jouent les communications inter processeurs ainsi que les augmentations de travail liées à des algorithmes parallèles différents des algorithmes séquentiels. Les comportements décrits vont dépendre du problème considéré de même que des hypothèses que l'on fait sur l'architecture de la machine parallèle. On obtiendra ainsi que les speedup et efficacité sont plus ou moins prometteurs. Dans ce chapitre, on supposera que la totalité du travail est parallélisable mais on aura que $W_{par} > W_{seq}$ en raison des overhead algorithmique et de communications.

5.1 Sommer des valeurs en parallèle

L'opération de sommer n valeurs réparties sur p processeurs est très courante dans le cadre des applications parallèles. C'est un exemple important car il illustre l'overhead algorithmique qui résulte de la façon dont une telle opération est réalisée en parallèle. En plus, une autre source d'overhead provient des communications entre les processeurs impliqués.

L'algorithme de sommation que nous allons proposer implique que les processeurs ont une mémoire distribuée (chacun stocke ses propres données) et que n'importe quelle paire de processeurs peut échanger des données en un temps qui ne dépend que de la quantité d'information transmise. En d'autre terme, le temps de communication T_{comm} nécessaire à échanger k mots (de 32-bits, par exemple) entre deux processeurs vaut :

$$T_{comm} = kC$$

où C est une constante caractéristique du matériel utilisé.

Cas $n = p$: Nous allons, dans un premier temps, supposer que p processeurs sont utilisés et que chacun contient une valeur a_i, $i = 1, ..., p$. Le but de l'algo-

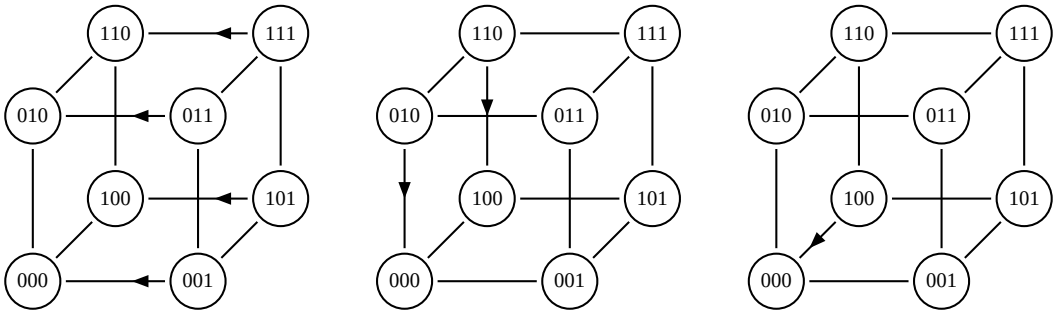


FIGURE 5.1 – Algorithme de réduction sur un hypercube pour sommer p nombres répartis sur p processeurs. Seuls des échanges entre processeurs voisins sont nécessaires.

rithme de sommation que nous présentons ici est de calculer la somme s de ces p valeurs ($s = \sum_{i=1}^{p} a_i$) de sorte que le résultat soit stocké dans le dernier processeur P_p.

Pour simplifier la discussion, nous allons encore supposer que p est une puissance de 2. L'algorithme procède de la façon illustrée ci-dessous pour le cas $p = 8$

	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0
	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$t=1$	–	$\sum_6^7 a_i$	–	$\sum_4^5 a_i$	–	$\sum_2^3 a_i$	–	$\sum_0^1 a_i$
$t=2$	–	–	–	$\sum_4^7 a_i$	–	–	–	$\sum_0^3 a_i$
$t=3$	–	–	–	–	–	–	–	$\sum_0^7 a_i$

Les échanges de données qui correspondent à cet algorithme sont illustrés dans le cas d'un hypercube de dimension 3, sur la figure 5.1. On constate qu'il s'agit chaque fois d'échange entre processeurs voisins.

Au cours de la première étape ($t = 1$), les processeurs P_i, pour i pair, reçoivent la valeur a_{i+1} de leur voisin de gauche et l'additionnent avec la valeur locale a_i. Comme tous les processeurs réalisent cette opération simultanément, cette première étape prend un temps ΔT

$$\Delta T = C + T$$

où T représente le temps nécessaire pour une addition.

Durant la deuxième étape ($t = 2$), la même démarche est répétée, mais cette fois entre les paires de processeurs (P_4, P_6) et (P_0, P_2). Les autres processeurs restent inactifs. En résumé, P_4 va recevoir $\sum_6^7 a_i$ de P_6 et l'additionner à la somme partielle obtenue à l'étape précédente, à savoir $\sum_4^5 a_i$. Ainsi, à la fin de cette deuxième étape qui dure aussi un temps $T + C$, P_4 contiendra $\sum_4^7 a_i$ et P_0 aura calculé $\sum_0^3 a_i$.

La dernière étape ($t = 3$) continue avec le même scénario entre les processeurs P_4 et P_0 : P_0 contient alors $\sum_0^7 a_i$, ce qui est le résultat escompté.

En général, cet algorithme nécessite $\log_2 p$ étapes, puisqu'on réduit chaque fois le problème d'un facteur 2. Donc le temps d'exécution parallèle vaut

$$T_{par} = (C + T) \log_2 p$$

Si p n'est pas une puissance de 2, il faut compléter la taille du problème à la prochaine puissance de 2 et le nombre d'étapes de l'algorithme correspond à $\lceil \log_2 p \rceil$.

Séquentiellement, la somme de p valeurs nécessite $p - 1$ additions

$$T_{seq} = (p - 1)T$$

Par conséquent, le speedup vaut

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

On constate donc que, même pour beaucoup de processeurs, le speedup est inférieur à p pour deux raisons :

— Il y a au dénominateur un facteur $\log_2 p$ qui provient de l'algorithme parallèle utilisé. En fait, le travail proprement dit est le même (il y a 4+2+1=7 additions, comme en séquentiel) mais il y a de moins en moins de processeurs actifs. Les cycles perdus correspondent à un travail inutile mais néanmoins présent.

— Le rapport C/T est non nul car le temps de communication n'est pas négligeable en général.

Cette exemple illustre deux sources courantes d'overhead liées à la parallélisation d'une application.

Cas $n >> p$: Il est intéressant d'analyser le même problème quand le nombre de valeurs n à additionner est suppérieur au nombre de processeurs p. Pour simplifier, on va supposer que p divise n et que chaque processeur détient n/p valeurs.

Une façon simple (mais mauvaise) de généraliser l'algorithme de somme parallèle ci-dessus dans ce cas est de répéter la même procédure pour chacune des n/p "couches" de valeurs : si a_i^k est la k^{eme} valeur du processeur P_i, on commence par sommer tous les a_i^1 dans P_8, puis les a_i^2 et ainsi de suite. Il faut donc n/p étapes qui prennent chacune $(T + C) \log_2 p$ unités de temps. Finalement, il faut encore sommer les n/p sous-totaux ainsi obtenus et le temps d'exécution parallèle de cette procédure est

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

Le temps séquentiel pour sommer n valeurs est

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

Par conséquent, le speedup vaut

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

Pour mieux analyser cette expression, il est souhaitable de la réécrire dans le cas où n est grand et largement supérieur à p. On a alors $(n-p)/(n-1) \approx 1$ et $n/(n-1) \approx 1$. On a donc

$$S = \frac{p}{1 + \left(1 + \frac{C}{T}\right)\log_2 p} \tag{5.1}$$

Comme on le verra par la suite, le grand défaut de cet algorithme est que le speedup ne dépend pas de n. Il ne s'améliore pas quand n augmente.

Bon algorithme de sommation : On peut remédier à cela en modifiant l'algorithme de sommation parallèle : chaque processeur commence par sommer les n/p valeurs qu'il détient. On retombe alors dans le premier cas d'avoir p valeurs sur p processeurs. La phase de sommation locale prend $(n/p-1)T$ unités de temps. A cela, il faut ajouter la partie non locale discutée ci-dessus et on obtient :

$$T_{par} = \left(\frac{n}{p} - 1\right)T + (C+T)\log_2 p \tag{5.2}$$

Avec toujours $T_{seq} = (n-1)T$, on trouve pour le speedup

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

ce qui, dans la limite de n grand, donne

$$S = \frac{p}{1 + \left(1 + \frac{C}{T}\right)\frac{p}{n}\log_2 p} \tag{5.3}$$

Dans cette expression, l'overhead est divisé par n. Donc, pour un p donné, le speedup peut être arbitrairement proche de p si la taille n du problème est assez grande.

La figure 5.2 illustre le temps d'exécution de cet algorithme parallèle de sommation dans le cas où $n = 64$ et $1 \leq p \leq n$. L'overhead est $(C+T)\log_2 p$ et, pour des raisons de simplicité, on approxime le temps de calcul par $(n/p)T$. On suppose aussi que ces formules restent valables quand p n'est pas une puissance

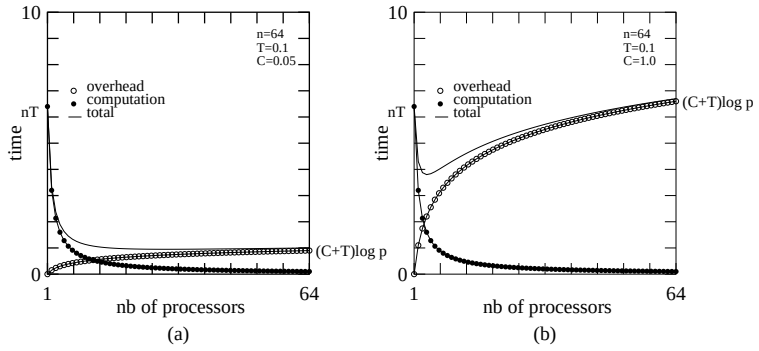


FIGURE 5.2 – *Temps de calcul et d'overhead pour une parallélisation de l'algorithme de sommation de $n = 64$ valeurs en fonction du nombre p de processeurs. (a) Situation acceptable : $C << T$. (b) situation inacceptable : $C >> T$.*

de 2 et quand p ne divise pas n. La figure montre deux situations qui se différencient par les valeurs de T et C : à gauche, le parallélisme est bénéfique car le temps d'exécution parallèle est inférieur au temps séquentiel (nT) et décroît avec p dans les limites du graphique ; à droite, par contre, la parallélisation n'est acceptable que pour des petites valeurs de p : à partir de $p = 6$, le temps parallèle est dominé par l'overhead qui augmente continuellement et qui, pour $p = n = 64$, est supérieur au temps d'exécution sur un seul processeur.

5.2 Remarque sur la répartition des tâches

Dans l'exemple précédent, nous avons supposé que n était divisible par p de sorte qu'une répartition équilibrée des données était possible. Mais si n n'est pas un multiple de p, le nombre k de données associées à chaque processeur variera. Les p' processeurs les plus chargés hériterons de $k' = \lceil N/p \rceil$ données (c'est à dire N/p arrondi vers le haut) et les autres p'' processeurs auront $k'' = \lfloor N/p \rfloor = k' - 1$ valeurs. On a donc $p' + p'' = p$ et $p'k' + p''k'' = n$ dont la solution est

$$p' = n \bmod \mathrm{p} \qquad p'' = p - p'$$

Si les différentes étapes de l'algorithme se font de manière synchrone entre les processeurs, la valeur k' sera déterminante pour le temps de calcul, car les éventuels processeurs qui auraient hérité de $k'' = k' - 1$ tâches devront attendre les autres à la fin de chaque étape.

En raison des temps de communication nécessaires aux transferts de données (equation 5.2), il est plus avantageux ici de placer k' tâches sur le plus grand nombre possible de processeurs, quitte à en laisser certains inutilisés, plutôt que d'avoir la totalité des processeurs au travail.

Par exemple, si $n = 19$, $p = 6$, on a $k' = 4$, $p' = 1$ et $p'' = 5$, ce qui donne la répartition 4,3,3,3,3,3 sur les 6 processeurs. Mais on peut aussi avoir la répartition

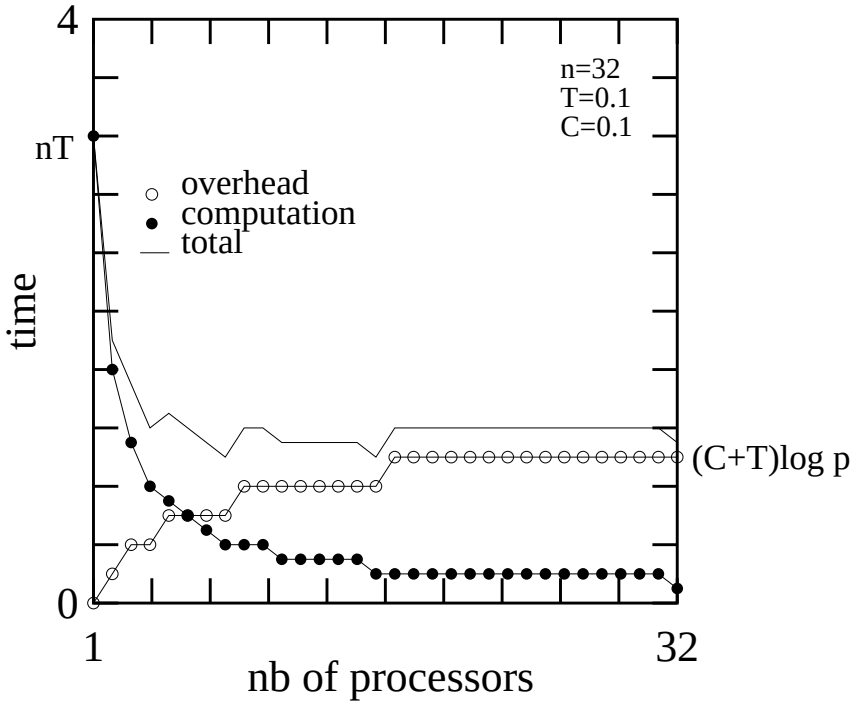


FIGURE 5.3 – *Temps de calcul et d'overhead pour une parallélisation de l'algorithme de sommation, avec une répartition des tâches par processeur donnée par* $k = \lceil n/p \rceil$*, un overhead* $k = (C + T)\lceil \log_2 p \rceil$ *et en utilisant la totalité des* p *processeurs.*

4,4,4,4,3,0 qui laisse un processeur inactif. Les deux solutions sont equivalentes du point de vue du temps de calcul mais la deuxième est plus avantageuse pour les communications (car p est inférieur).

Finalement, pour l'algorithme de sommation parallèle, remarquons encore que le fait de ne pas pouvoir diviser exactement n données sur p processeurs et de devoir arrondir $\log_2 p$ à la valeur supérieure donne des courbes de temps d'exécution avec des sauts, comme le montre la figure 5.3.

5.3 L'équation de Laplace

Ce paragraphe décrit une autre situation de parallélisation qui est générique de nombreux problèmes de calcul scientifique qui ne nécessitent que des **communications locales**. On va voir de nouveau qu'en choisissant bien la taille du problème, on peut augmenter le temps de calcul suffisamment par rapport au temps de communication pour se trouver dans une situation tout à fait favorable.

Considérons le cas de la solution de l'équation de Laplace en trois dimensions sur un domaine spatial D.

$$0 = \nabla^2\phi(x,y,z) \equiv \left(\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2}\right)$$

En se fixant la valeur de la fonction ϕ sur le bord de D, il existe une fonction unique vérifiant cette équation.

Une manière simple de résoudre numériquement ce problème est de discrétiser l'espace sous forme d'une grille et d'utiliser un schéma d'itération. On divise le domaine en N^3 points de coordonnées (i,j,k), également répartis sur D. On peut montrer que l'itération suivante tend vers la solution recherchée :

$$\phi_{i,j,k}^{(n+1)} = \frac{1}{6}(\phi_{i-1,j,k}^{(n)} + \phi_{i+1,j,k}^{(n)} + \phi_{i,j-1,k}^{(n)} + \phi_{i,j+1,k}^{(n)} + \phi_{i,j,k-1}^{(n)} + \phi_{i,j,k+1}^{(n)}) \qquad (5.4)$$

où $\phi^{(0)}$ est une condition initiale aléatoire donnée. L'itération signifie donc que la valeur ϕ en chaque point est remplacée par la moyenne arithmétique des 6 voisins les plus proches.

Supposons maintenant que l'on dispose de p processeurs se partageant le travail ci-dessus. Pour cela, on divise le domaine D en p morceaux de taille ℓ (contenant donc chacun ℓ^3 points de grille), de sorte que $N^3 = p\ell^3$. Chaque processeur se chargera du calcul des $\phi^{(n)}$ dans son propre sous-domaine. L'équation 5.4 montre que pour chaque point de grille, il faut 6 opérations en virgule flottante par itération. Le temps de calcul T_{cal} vaut donc

$$T_{\text{cal}} = \frac{6\ell^3}{R}$$

où R est la puissance d'exécution du processeur, donnée en nombre d'opérations virgule flottante par seconde.

Cependant, les points qui sont au bord de chaque sous-domaine auront certains de leur voisins dans des processeurs différents. Il faudra donc recourir à une communication inter-processeurs pour obtenir les valeurs de $\phi^{(n)}$ voisines. Comme on a divisé le domaine D en un ensemble de cubes de côtés ℓ, il y a $6\ell^2$ points à la surface de chaque sous-domaine car il y a six faces contenant chacune ℓ^2 points. Le temps nécessaire au transfert de ces données peut s'écrire

$$T_{\text{comm}} = 6\ell^2 C$$

où C est le temps de transfert d'un mot machine d'un processeur à l'autre.

Il s'en suit que

$$T_{\text{par}} = T_{\text{cal}} + T_{\text{comm}} = \frac{6\ell^3}{R} + 6\ell^2 C$$

De ce qui précède, on a que le rapport des temps de communications et de calcul se comporte comme :

$$\frac{T_{\text{comm}}}{T_{\text{cal}}} = \frac{6\ell^2 C}{6\ell^3/R} = \frac{RC}{\ell}$$

Si le problème est assez grand ($\ell \to \infty$), ce rapport peut être rendu aussi petit que l'on veut et les temps de communications deviennent négligeables par rapport aux temps de calcul. Le speedup s'écrit

$$S = \frac{(6p\ell^3)/R}{6\ell^2 C + (6\ell^3/R)} = \frac{p}{1 + \frac{RC}{\ell}} = \frac{p}{1 + \frac{T_{\text{comm}}}{T_{\text{cal}}}}$$

Dans la limite $\ell \to \infty$, on a bien $S \sim p$ et on dit que l'application est scalable (voir paragraphe 5.4).

Il est intéressant de constater que ce qui limite le speedup est le rapport $T_{\text{comm}}/T_{\text{cal}}$. Le temps de communication est proportionnel à la *surface* des sous-domaines, car chaque face doit être échangée avec le processeur voisin. Par contre, le temps de calcul est lui proportionnel au *volume* des sous-domaines. Ainsi $T_{\text{comm}}/T_{\text{cal}}$ est propotionnel au rapport surface sur volume des sous-domaines. On sait qu'un tel rapport est minimum pour un disque en 2D, ou une boule en 3D. De tels sous-domaines sont cependant inadaptés pour partitionner une grille régulière. Un carré ou un cube sont alors de bonnes alternatives et présentent un rapport surface/volume bien plus avantageux qu'un rectangle long et mince. Dans cet esprit, un paritionnement en bloc est donc préférable à un partitionnement en tranche, ou un paritionnement cyclique (ou modulo), comme décrit à la section 7.3.

5.4 Scalabilté

Le terme de scalabilité (ou scalability, en anglais) exprime la possibilité d'augmenter arbitrairement le nombre de processeurs d'une architecture parallèle afin d'en augmenter proportionnellement les performances.

Le mot scalabilité pourrait se traduire par **extensibilité** ou **passage à l'échelle**. C'est un concept très important dans les architectures parallèles car une machine dite scalable peut être dimensionnée aux exigences et moyens de chaque centre de calcul et peut être étendue d'année en année. On peut ainsi augmenter la puissance de calcul ou plutôt les capacités d'entrée-sortie, selon les besoins des applications spécifiques qui sont envisagées.

Comme l'ont montré les discussions précédentes, l'accroîssement des performances d'une machine parallèle n'est pas toujours linéaire avec le nombre de processeurs. Ce fait est plus ou moins marqué selon les architectures et les applications considérées. Certaines architectures sont scalables et d'autres ne le sont pas.

Le concept de scalabilité a plusieurs facettes. En pratique, on distingue la scalabilité d'architecture, la scalabilité de génération et la scalabilité d'application.

5.4.1 La scalabilité d'architecture

La scalabilité d'architecture décrit la propriété d'une architecture donnée à être augmentée en taille. Certaines contraintes sont à respecter, d'abord du côté hardware, mais aussi du côté du logiciel. On peut mentionner les points suivants :

- Le réseau d'interconnexion qui relie les processeurs entre eux doit permettre cet accroîssement du nombre d'éléments (possibilité de branchement supplémentaire, débit d'information adapté au trafic d'information accru).
- La capacité mémoire doit aussi augmenter avec le nombre de processeurs supplémentaires, de même que les possibilités d'accès à cette mémoire.
- Les possibilités d'entrée-sortie doivent s'accroître pour correspondre à l'augmentation du nombre de données traitées.
- Le coût de la machine doit être proportionnel au nombre de processeurs.
- Le logiciel doit pouvoir s'adapter à un nombre quelconque de processeurs. Le même système d'exploitation doit pouvoir tourner quelle que soit la configuration de la machine, avec des temps d'overhead qui ne croissent pas à mesure que l'on rajoute des processeurs.
- Les mêmes programmes doivent pouvoir s'exécuter sur un nombre arbitraire de processeurs. Le compilateur ou le système d'exploitation doivent être capables de répartir de manière efficace une application sur le nombre de processeurs disponibles.

En pratique, la scalabilité d'une architecture est envisagée dans une certaine plage. Il y a des considérations d'ordre pratique et parfois d'ordre technique qui font que le nombre de processeurs maximum d'une architecture ne peut croître à l'infini :

- Le prix devient vite prohibitif. Par exemple, dans le milieu des année 90, un MPP (Connection Machine CM-5) contenant 8192 processeurs aurait coûté de l'ordre de US$ 200 Millions.
- La consommation électrique, la chaleur dissipée et l'encombrement sont aussi des facteurs limitatifs.

5.4.2 La scalabilité de génération

La scalabilité de génération dénote la possibilité de substituer les processeurs ou les composants d'une architecture par de plus puissants, à mesure qu'ils sont disponibles sur le marché. Cela permet d'évoluer "linéairement" dans le temps, en incorporant les progrès technologiques récents. Cette possibilité est très importante pour rentabiliser les coûts de développements élevés qui résultent de

l'élaboration d'une nouvelle machine que l'on souhaite être compétitive le plus longtemps possible.

Les machines parallèles actuelles sont principalement MIMD. Il y a soit des MPP à mémoire distribuée (chaque processeur possède sa propre mémoire et communique avec les autres par échange de messages à travers un réseau d'interconnexion), ou des clusters fortement couplés de SMP (mémoire partagée) ou encore (et de plus en plus) des "fermes" de station de travail.

Ces architectures se basent de plus en plus sur le concept de scalabilité de génération : les constructeurs choisissent les processeurs RISC les plus performants du marché (en général les mêmes que ceux utilisés dans les stations de travail) et peuvent les remplacer lorsque de plus puissants sont disponibles.

Les constructeurs préconisent de plus en plus d'utiliser des technologies courantes pour le réseau d'interconnexion entre les processeurs. En effet, ces dernières bénéficient d'un développement considérable et indépendant du seul marché du parallélisme. La technologie Infiniband qui se développe actuellement est un exemple d'un moyen de connexion rapide utilisable dans de nombreux secteurs de l'informatique.

La scalabilité de génération est ainsi à l'opposé d'une architecture dédiée (telle que le fut la Connection Machine CM-2/200) qui n'utilise pratiquement que des composants spécifiques, imposant des frais de développement excessifs.

5.4.3 La scalabilité d'application

La scalabilité d'application décrit la relation entre les performances d'une machine parallèle, la taille du problème considéré et le nombre de processeurs utilisés.

L'analyse de scalabilité doit se faire pour une architecture donnée et pour une application donnée. En principe, ces deux éléments sont indissociables. La question à laquelle on veut répondre est la suivante : quelles sont les lois d'échelle qui régissent les performances d'une architecture parallèle pour une application donnée. Les quantités qui interviennent dans cette analyse de scalabilité sont nombreuses. On citera les principales :

- p : la taille de la machine ou, plus précisément, le nombre de processeurs présents.
- n : la taille du problème, c'est à dire le nombre de données à traiter ou le nombre d'opérations à exécuter
- I/O : Le débit des entrée-sorties.
- m : la capacité mémoire.
- c : le temps de communication et de synchronisation.
- T : le temps CPU d'exécution.

On pourrait vouloir définir la scalabilité comme suit : "l'application Y est scalable sur l'architecture X si le speedup obtenu sur une machine de taille p est égale à p, quelle que soit n la taille de l'application." Comme le montrent les exemples

du paragraphe 4.5 cette définition est trop exigeante et pratiquement aucune machine ne serait scalable. On préfère donc une définition plus nuancée.

La scalabilité s'exprime à travers la notion **d'efficacité** E qui est le rapport entre le speedup obtenu pour une application de taille n et le nombre de processeurs p utilisé

$$E(n,p) = \frac{T_s(n)}{pT_p(n)} \tag{5.5}$$

où $T_s(n)$ est le temps d'exécution du meilleur algorithme séquentiel résolvant le problème et $T_p(n)$ est le temps de l'algorithme parallèle considéré. L'efficacité indique quelle fraction des performances potentielles on obtient d'une application parallèle. Par exemple, une efficacité de 0.8 indique que 80% du temps est passé en calculs "utiles" alors que le 20% restant est dépensé en overhead.

Une efficacité de 1 correspond à un speedup de p, ce qui est évidemment idéal. En général, ce n'est pas le cas, mais la question est ici de savoir s'il est possible d'agrandir suffisamment la taille n du problème afin que sur p processeurs, l'efficacité soit acceptable. En effet, de la loi de Gustafson, on s'attend à ce que pour p donné, plus n est grand, plus l'efficacité soit bonne.

La notion de **scalabilité d'application** donne un sens quantitatif à cette observation. On pose la définition suivante :

Un algorithme est scalable sur une architecture donnée s'il est possible d'obtenir une efficacité constante, quel que soit le nombre de processeurs, en ajustant la taille du problème n en fonction de p.

La relation

$$n = F(E,p) \tag{5.6}$$

qui donne la dépendance de n en fonction de p pour avoir une efficacité E constante définit ce qu'on appelle la fonction F **d'isoefficacité**

L'expression de F dépend évidemment de l'algorithme parallèle choisi et des possibilités offertes par l'architecture considérée. Il peut y avoir une autre fonction d'isoefficacité si l'on change d'algorithme ou de machine.

Pour mieux comprendre ce concept de scalabilité d'application, considérons quelques exemples.

Supposons tout d'abord que le speedup soit égal à p, quel que soit n. Si $S = p$, on a $E = 1$. Donc n'importe quelle fonction $n = n(p)$ garantit une efficacité constante et l'application est scalable. Évidemment, comme on l'a remarqué précédemment, on ne peut pas augmenter p arbitrairement sans aussi augmenter n car, sinon, il arrivera un stade où il n'y aura plus assez de données pour alimenter tous les processeurs.

Considérons maintenant les exemples du paragraphe 4.5. Dans le cas de la

solution de l'équation de Laplace, on avait obtenu un speedup

$$S = \frac{p}{1 + RC\left(\frac{p}{n}\right)^{1/3}}$$

où $n = N^3$ est le nombre total de points de grille. Ce speedup implique une efficacité

$$E = \frac{1}{1 + RC\left(\frac{p}{n}\right)^{1/3}}$$

Clairement, si n est assez grand, l'efficacité peut être rendue aussi proche de 1 que désiré. L'isoefficacité est obtenue en extrayant n de la relation ci-dessus

$$n = F(E, p) = \left[\left(\frac{E}{1-E}\right) RC\right]^3 p$$

Pour un nombre de processeurs p donné, on peut donc adapter la taille du problème n pour avoir une efficacité E choisie. La fonction d'isoefficacité ci-dessus indique que la taille du problème doit croître linéairement avec p. Plus l'efficacité exigée est grande, plus le coefficient devant p augmente. Mais ce résultat montre avant tout que la fonction d'isoefficacité est linéaire en p, et que le problème est parfaitement scalable (tout au moins sur un ensemble de processeurs organisés selon une grille tridimensionnelle).

Par contre, avec le mauvais algorithme de sommation parallèle que sous avons vu à la section 5.1, les choses se passent moins bien. Le speedup obtenu par cette méthode vaut (voir equation (5.1))

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

d'où une efficacité

$$E = \frac{1}{1 + \left(1 + \frac{C}{T}\right) \log_2 p}$$

Le point crucial est qu'ici, l'efficacité ne dépend plus de la taille n du problème. Par conséquent, il n'est pas possible de trouver n en fonction de p pour obtenir une efficacité constante. La fonction d'isoefficacité n'existe pas et l'algorithme n'est **pas scalable**. Cela ne signifie pas que cette parallélisation est plus lente que l'exécution séquentielle car, pour des valeurs raisonnables de C/T, on aura de toute façon un gain de temps ($S > 1$ et S augmente avec p). Par contre, l'efficacité diminue si p augmente.

Dans ces exemples, on voit que l'architecture utilisée joue aussi un rôle dans la discussion car l'expression du speedup en dépend, à travers les possibilités de communications offertes (topologie des connexions).

Entre les deux extrêmes que nous venons de voir (isoefficacité linéaire et non définie), il existe toute une gamme de possibilités. Par exemple, pour le bon algorithme de sommation, l'equation (5.3) donne

$$E = \frac{1}{1 + \left(1 + \frac{C}{T}\right) \frac{p}{n} \log_2 p}$$

et pour avoir une efficacité constante il faut que

$$\frac{p}{n} \log_2 p = \text{cte}$$

c'est à dire que

$$n = F(p) \sim p \log_2 p$$

Bien que n doive augmenter plus vite que p, cet algorithme est scalable puisqu'on peut choisir n en fonction de p pour garantir une efficacité donnée.

Le degré de scalabilité dépend de la forme de la fonction d'isoefficacité F. On parle de

— Scalabilité linéaire ou idéale si $F(p) \sim p$.
— Scalabilité poly-logarithmique si $F(p) \sim p \log^k(p)$.
— Scalabilité faible si $F(p) \sim p^k$, où k est un entier.

Il faut remarquer qu'avec la définition de scalabilité ci-dessus, une application dont le speedup S serait égal à p pour tout n ne serait pas scalable puisque la fonction d'isoefficacité ne serait pas définie. Cela parait contraire au bon sens, mais cette situation n'est pas possible car, comme nous l'avons déjà souligné, n doit toujours être au moins supérieur à p, sans quoi les processeurs en excédent seront inactifs et $S = p$ ne pourra pas être maintenu si $p > n$.

5.4.4 Weak scaling et strong scaling

Les concepts de scalabilité d'application qu'on vient de mentionner permettent de réinterprèter les lois d'Amdahl et de Gustafson dans un cadre plus général. Amdahl analyse le cas où, si p augmente, on garde n constant. Par contre, Gustafson considère le cas où n augmente si p augmente.

Pour l'exemple ci-dessus de la résolution de l'équation de Laplace, on avait

$$S = \frac{p}{1 + RC \left(\frac{p}{n}\right)^{1/3}}$$

La figure 5.4 illustre les valeurs de speedup obtenues en fonction de p, soit avec n constant (vision à la Amdahl) ou soit avec $n = p$ (vision à la Gustafson).

Une façon plus moderne de qualifier ces deux approches est d'utiliser les termes **weak scaling** et **strong scaling**. Le «strong scaling» consiste à analyser le speedup en fonction de p mais en gardant n, la taille du problème constant. Le «weak scaling» considère le speedup en fonction de p tout en gardant le travail par processeur constant et proportionnel à n/p.

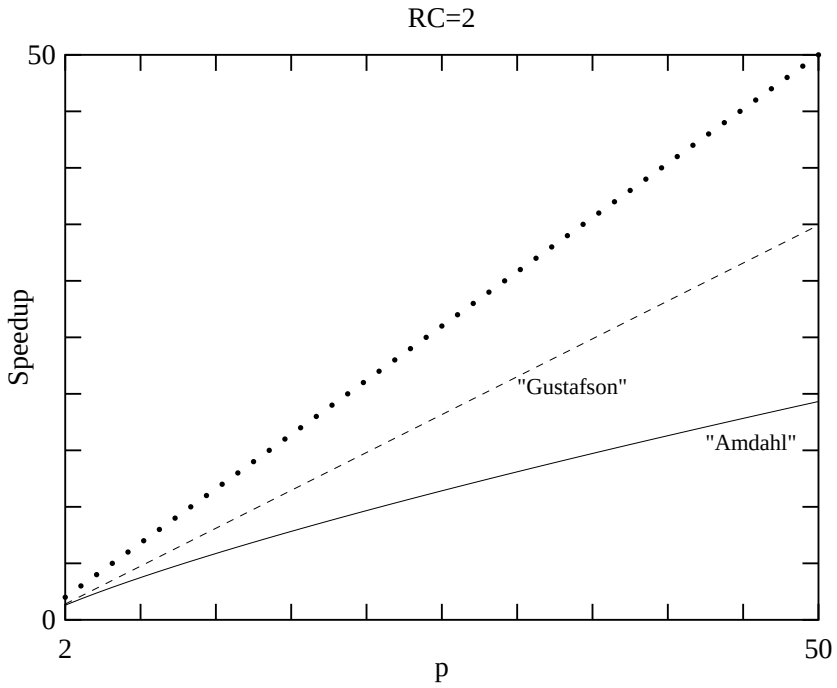


FIGURE 5.4 – Speedup, en fonction du nombre p de processeurs, pour la résolution itérative de l'équation de Laplace. La ligne continue illustre la vision «Amdahl» (n constant) et la ligne traitillée illustre la vision «Gustafson» ($n \propto p$). Les points indiquent un speedup parfait $S = p$.

Overhead fractionnaire : Comme on l'a vu ci-dessus et dans l'équation (4.8), l'efficacité s'écrit souvent sous la forme

$$E = \frac{1}{1 + \Delta(n,p)}$$

On definit alors Δ comme «l'overhead fractionnaire». Par exemple, dans le cas de la résolution de l'équation de Laplace en 3 dimensions, l'overhead fractionnaire est

$$\Delta(n,p) = RC\left(\frac{p}{n}\right)^{1/3}$$

et il tend vers zéro quand $n \gg p$.

5.4.5 Speedups relatifs

Avec la croissance continuelle des machines parallèles, il devient impossible de comparer le temps d'exécution $T_{par}(n,p)$ pour n et p grand, avec $T_{seq}(n)$, car la mémoire de la machine séquentielle est trop petite pour stocker un problème de la taille de ceux considérés en parallèle. Il est donc courant de définir un **speedup relatif**

$$S_{p_0} = \frac{T_{par}(p_0,n)}{T_{par}(p,n)}$$

où p_0 est un nombre de processeurs de référence et $p > p_0$ et le nombre de processeurs pour lequel on compare la performance.

Pour le cas de la résolution de l'équation de Laplace en 3D, on a vu ci-dessus que

$$T_{par}(p,n) = 6\frac{n}{pR} + 6C\left(\frac{n}{p}\right)^{2/3}$$

où p est le nombre de processeur et n le nombre total de points du domaine de calcul. On a aussi obtenu que

$$S = \frac{p}{1 + RC\left(\frac{p}{n}\right)^{1/3}} \qquad E = \frac{1}{1 + RC\left(\frac{p}{n}\right)^{1/3}} \tag{5.7}$$

Si ce même problème de taille n est résolu sur p_0 processeurs, on aura

$$T_{par}(p_0,n) = 6\frac{n}{p_0R} + 6C\left(\frac{n}{p_0}\right)^{2/3}$$

et le speedup relatif à p_0 processeurs sera

$$S_{p_0} = \frac{6\frac{n}{p_0R} + 6C\left(\frac{n}{p_0}\right)^{2/3}}{6\frac{n}{pR} + 6C\left(\frac{n}{p}\right)^{2/3}}$$

ce qui se simplifie en

$$S_{p_0} = \frac{\frac{n}{p_0}[1 + RC\left(\frac{n}{p_0}\right)^{-1/3}]}{\frac{n}{p}[1 + RC\left(\frac{n}{p}\right)^{-1/3}]}$$

ou encore

$$S_{p_0} = \frac{p}{p_0}\frac{[1 + RC\left(\frac{p_0}{n}\right)^{1/3}]}{[1 + RC\left(\frac{p}{n}\right)^{1/3}]} \tag{5.8}$$

On voit donc que le speedup relatif contient le terme attendu, p/p_0, mais avec un facteur inférieur à 1, et qui tend à diminuer quand p augmente.

Dans le cas de «strong-scaling», l'équation (5.8) donne le speedup sous la condition que n reste constant pour toute valeur de p. Pour le «weak-scaling» on aura typiquement $p/n = \gamma$, ou γ est maintenu constant quand p augmente. On peut donc poser $n = p/\gamma$ et $p_0/n = \gamma p_0/p$. L'équation (5.8) devient, dans le cas du weak-scaling,

$$S_{p_0} = \frac{p}{p_0}\frac{[1 + RC\gamma^{1/3}\left(\frac{p_0}{p}\right)^{1/3}]}{[1 + RC\gamma^{1/3}]} \tag{5.9}$$

L'efficacité relative se définit alors comme la fraction du speedup relatif idéal, à savoir

$$E_{p_0} = \frac{S}{\frac{p}{p_0}} = \frac{p_0}{p}S$$

On constate que dans les deux cas de scaling, que E diminue à mesure que p croît.

Dans le cas du weak-scaling, on a donc, avec E donné par l'éq. (5.7), que

$$E_{p_0} = \left(1 + RC\gamma^{1/3}\left(\frac{p_0}{p}\right)^{1/3}\right)E \geq E \tag{5.10}$$

Cela suggère que l'efficacité relative est plus grande que l'efficacité obtenue en comparant l'exécution parallèle à l'exécution séquentielle. Cependant, dans la limite $p \to \infty$, on obtient $E_{p_0} = E$.

On peut généraliser ce résultat avec la relation (4.7) avec $R = 1$

$$T_{par}(p, n) = \frac{W(n)}{p} + \frac{\Delta(p, n)}{p}$$

où W est le travail séquentiel et $\Delta(p, n)$ l'overhead. Il s'en suit que

$$S_{p_0} = \frac{\frac{W(n)}{p_0} + \frac{\Delta(p_0,n)}{p}}{\frac{W(n)}{p} + \frac{\Delta(p,n)}{p}} = \frac{p}{p_0}\frac{\left(1 + \frac{\Delta(p_0,n)}{W(n)}\right)}{\left(1 + \frac{\Delta(p,n)}{W(n)}\right)} = \frac{p}{p_0}E\left(1 + \frac{\Delta(p_0, n)}{W(n)}\right)$$

5.5 Mesures de performances et benchmarks

Comme on vient de le voir, les performances d'une machine parallèle sont très dépendantes de la nature de l'application considérée. Beaucoup de facteurs qui relèvent de l'architecture et du logiciel interviennent : temps de communications, répartition des données, distribution du travail, etc. Souvent, des modifications apparemment anodines d'un programme peuvent conduire à des améliorations ou des détériorations importantes des performances. Certaines machines peuvent être très efficaces pour une classe d'applications donnée et beaucoup moins bonnes pour une autre.

Dans une certaine mesure, cette remarque est aussi vraie pour les performances d'un mono-processeur. Les différentes solutions architecturales utilisées pour augmenter la puissance de calcul donnent des résultats qui dépendent souvent de la nature du programme.

Il est donc insuffisant d'évaluer les performances d'une machine séquentielle par sa puissance de crête théorique, donnée par la fréquence d'horloge multipliée par son degré de superscalarité (voir section 2.6), ou d'une machine parallèle en indiquant seulement sa puissance maximale, obtenue comme le produit de la puissance individuelle de chaque processeur par le nombre p de processeurs. Souvent, les performances réelles ne représentent que le 10% de cette performance de crête.

5.5.1 Benchmarks

Une façon simple d'estimer les performances d'un ordinateur est de le comparer à un autre de référence, dont le comportement est bien connu. On dit ainsi qu'une machine B est n fois plus rapide qu'une machine A pour une application donnée, si le temps d'execution $T_A = n \times T_B$. On exprime volontiers cette augmentation de la puissance par un gain g en %. Le gain de la machine B sur la machine A est calculé ainsi :

$$g = \frac{T_A - T_B}{T_B} = n - 1$$

Par exemple, si la machine B offre un gain de 100%, cela signifie qu'elle est deux fois plus rapide que A pour l'application considérée.

Les applications testées peuvent être choisies par l'utilisateur selon ses besoins. Cependant, une façon plus universelle est de recourir à des suites d'applications (benchmarks) comme les Whetstone, SPEC, NAS, Linpack, Eurobench et d'autres encore.

On appelle *benchmark* la mesure des performances d'un ordinateur, à travers un ensemble de programmes tests. Souvent, les benchmarks consistent à donner le nombre de Mflop/s (ou Gflop/s et Tflop/s) qu'une machine peut atteindre sur

un code donné. Il n'y a cependant pas de règles générales et certains benchmarks indiquent seulement le temps d'exécution.

Un benchmark a pour but de déterminer les performances réelles d'un ordinateur. En général, chaque benchmark se spécialise pour une caractéristique particulière du système : la puissance de calcul (flop/s) les capacités d'entrée-sortie, le speedup ou encore le temps d'accès à la mémoire, au cache ou au disque. En général, un benchmark se restreint aux parties les plus critiques des programmes et, de ce fait, est souvent constitué de noyaux d'applications réelles.

Souvent, la portabilité des programmes de benchmarks est recherchée pour permettre les comparaisons entre plusieurs machines. De ce fait, les benchmarks sont souvent écrits en C ou Fortran.

Un benchmark qui fut proposé dans le passé est le **SPEC** (System Performance Evaluation Corporation) qui est un ensemble de 10 programmes de base sélectionnés par un consortium pour l'évaluations des performances des stations de travail. Ce consortium est composé d'IBM, Intel, SUN, SGI, HP et Compaq. Les programmes de base comportent par exemple des applications scientifiques comme des simulations par la méthode de Monte-Carlo, des algorithmes de tri, la résolution des équations de Maxwell, ou des applications systèmes comme le compilateur `gcc`. En tout, il y a de l'ordre de 150'000 lignes de codes. Souvent, les performances sont séparées entre les opérations en virgule flottante (SPECflp) ou en virgule fixe (SPECint). Les résultats du benchmark sont définies comme le rapport du temps total (wall-clock time) d'exécution par rapport à un temps de référence (donné par une VAX 11/780).

Il y a aussi les **benchmarks synthétiques** qui ne sont pas construits à partir d'applications réelles mais mimiquent leur comportement en reproduisant les structures de données usuelles et les instructions standards dans des proportions et avec des fréquences réalistes. Dans ce type de benchmarks, il y a le Whetstone, proposé en 1976 et qui contient 11 modules testant entre autres les opérations mathématiques élémentaires, les structures de données de base et les appels de fonctions. Dans le même ordre, on peut aussi mentionner le Dhrystone (1984). Notons que les benchmarks synthétiques existent pour les systèmes séquentiels ou parallèles.

La suite NAS est uniquement pour les machines parallèles. Notons que de nombreux benchmarks parallèles existent, tels que la suite NAS (ou NPB), Eurobench, Parkbench qui se spécialise dans le détermination du temps de latence et la bandwidth des réseaux d'interconnexion et SLALOM qui, dans l'esprit de la loi de Gustafson, indique combien de travail peut être fait en 1 seconde.

Plus récemment, un ensemble d'applications a été identifié comme représentatif de ce qu'une machine parallèle doit savoir faire efficacement. Ces applications sont dénommées **«Dwarfs»**, ou «nains» pour refléter le fait que ce sont des problèmes simples, à la base des applications du futur. Il faut donc que le HW, les languages et compilateurs puissent les exécuter rapidement.

Actuellement, 13 dwarfs sont proposés. Ce sont les problèmes suivants

1. Dense Linear Algebra
2. Sparsae Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtracking / Branch & Bound
12. Graphical Model Inference
13. Finite State Machine

5.5.2 LINPACK parallèle

Ci-dessous, nous allons présenter en plus de détails le benchmark Linpack qui est à la fois utilisé pour les ordinateurs parallèles et séquentiels. Dans le cas parallèle la suite LINPACK permet de donner un classement, le top500, des machines parallèles les plus rapides du monde.

Dans le benchmark LINPACK on considère un code pour la résolution d'un système de N équations linéaires à N inconnues (système dense). Parfois, on donne (en Mflops) les performances d'un LINPACK 100 (N=100) ou LINPACK 1000 (N=1000). Mais, pour présenter les machines sous leur meilleur jour, il est fréquent de choisir la taille N du problème qui donne les meilleures performances et d'optimiser le code. On parle alors du HLP (high performance Linpack) et on génère aléatoirement des systèmes denses d'équations linéaires.

Les performances sont alors données de la manière suivante

— p le nombre de processeurs utilisés.
— R_{max} la puissance maximum obtenue (en Gflops, par exemple).
— N_{max} la taille du problème considéré.
— $N_{1/2}$ la taille du problème pour laquelle on aurait une puissance effective de $R_{max}/2$.
— R_{peak} la performance de crête de la machine, c'est à dire p fois la performance de crête des processeurs individuels

Cette manière d'indiquer les performances a donné lieu à un classement des ordinateurs les plus rapides du monde : le Top500 [1]. Cette démarche reflète bien le

1. Ces données sont disponible sur WWW à l'adresse
http :// www.top500.org

soucis des constructeurs à être les premiers à proposer l'ordinateur le plus rapide du monde.

En novembre 1995, le record était détenu par une machine Fujitsu à 140 processeurs offrant $R_{max} = 170$ Gflops pour un problème de taille $N_{max} = 42000$ (alors que $R_{peak} = 206$ Gflops et $N_{1/2} = 14000$). En 96, c'est une machine Hitachi avec 2048 processeurs qui détient le record du meilleur R_{max} au Top500. Cet ordinateur parallèle donne $R_{max} = 368$ Gflops pour un problème de taille $N_{max} = 103680$, avec $R_{peak} = 614$ Gflops et $N_{1/2} = 30720$.

Ce record a été pulvérisé en décembre 96 par une machine d'Intel composée de 7264 Pentium Pro qui a, pour la première fois de l'histoire, dépassé la barrière psychologique du TeraFlop sur un problème réaliste : une inversion de matrice de taille 215000×215000 (le problème est créé pour l'occasion par un générateur de matrices) pour laquelle une performance de 1.07 teraflop est obtenue. (La valeur de $N_{1/2}$ est 53400).

En décembre 1997, c'est de nouveau cette machine d'Intel qui détient le nouveau record, avec $R_{max} = 1.34$ Tflops sur 9152 processeurs Pentium (avec $R_{peak} = 1.8$ Tflop, $N_{max} = 235000$ et $N_{1/2} = 63000$). La machine complète (dans sa configuration finale) contient 9260 processeurs, avec 573 gigabytes de mémoire et 2.25 terabytes d'espace disque. Elle occupe une surface d'environ 12 m^2 pour un poids de 40 tonnes et consomme une puissance de 850kW. Plus de 3 km de câbles sont nécessaires pour interconnecter les noeuds.

Le tableau 5.1 résume l'évolution au cours du temps de la machine la plus rapide.

Sur la figure 5.5, les données de la table 5.1 sont reportées. On constate que la performance R_{max} en fonction du temps t en année est bien approximée par la relation

$$R_{max} = \alpha 10^{\beta t}$$

avec $\beta = 0.244$. Cela signifie un accroissement de puissance d'un facteur $10^{0.246} \approx 1.75$ chaque année, ou encore un facteur $10^{2.5} \approx 275$ tout les dix ans. Cet accroissement de performance sur 10 ans diminue d'année en année. Il n'y a pas si longtemps, la pente de ce graphique indiquait un accroissement d'un facteur 500 en 10 ans.

année	ordinateur	p	R_{peak}	N	R_{max}	$N_{1/2}$	MW
1995	Fujitsu	140	206 Gflop/s	42 000	170 Gflop/s	14 000	
1997	Intel	9152	1.8 Tflop/s	235 000	1.34 Tflop/s	63 000	
2000	IBM ASCI White	8192	12 Tflop/s	430 000	4.9 Tflop/s		
2001	IBM ASCI White	8192	12 Tflop/s	518 096	7.2 Tflop/s	179 000	
2002	NEC Earth Simulator	5120	41 Tflop/s	1 075 200	36 Tflop/s	266 240	
2004	IBM Blue Gene	32'768	91.7 Tflop/s		70.7 Tflop/s		
2005	IBM Blue Gene	131'072	367 Tflop/s	1'769'471	280 Tflop/s		
2007	IBM Blue Gene	212'992	596 Tflop/s		478 Tflop/s		
2008	IBM roadrunner	129'600	1.45 Pflop/s		1.1 Pflop/s		
2009	Cray XT5 Jaguar	224'162	2.33 Pflop/s		1.76 Pflop/s		6.9
2010	Tianhe-1A	186'386	4.70 Pflop/s	3'600'000	2.57 Pflop/s	1'000'000	
2011	K-Computer	705'024	11.28 Pflop/s		10.51 Pflop/s		12.66
2012	Cray Titan	560'640	27.12 Pflop/s		17.6 Pflop/s		8.2
2013	Tianhe-2	3'120'000	54.9 Pflop/s		33.8 Pflop/s		17.8
2014	Tianhe-2	3'120'000	54.9 Pflop/s		33.8 Pflop/s		17.8
2015	Tianhe-2	3'120'000	54.9 Pflop/s		33.8 Pflop/s		17.8
2016	TaihuLight	10'649'600	125.5 Pflop/s		93.0 Pflop/s		15.3
2017	TaihuLight	10'649'600	125.5 Pflop/s		93.0 Pflop/s		15.3
2018	IBM Summit	2'397'824	200.8 Pflop/s		143.5 Pflop/s		9.78
2019	IBM Summit	2'414'592	200.8 Pflop/s		143.6 Pflop/s		10.1
2020	Fugaku (Arm)	7'630'848	537.2 Pflop/s		442 Pflop/s		29.9
2021	Fugaku (Arm)	7'630'848	537.2 Pflop/s		442 Pflop/s		29.9
2022	Frontier (Cray)	8'730'112	1'685 Pflop/s	24'440'832	1'102 Pflop/s		21.1

TABLE 5.1 – *L'ordinateur le plus rapide au cours des années précédentes, selon le classement du top500.*

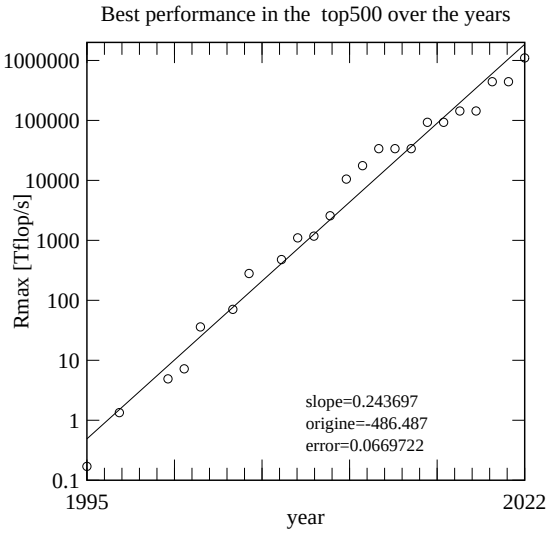


FIGURE 5.5 – Graphique le de performance de la machine la plus rapide du top500 au cours des années passées.

Chapitre 6

Tâches, partitionnement et ordonnancement

6.1 Introduction

Un point essentiel pour obtenir des performances avec un ordinateur parallèle est d'utiliser des algorithmes adaptés au modèle de programmation et à l'architecture sous-jacente. Tout algorithme parallèle repose sur un ensemble de tâches qui traitent un ensemble de données. L'exécution concurrente de ces tâches par plusieurs processeurs implique certains choix quant à la manière de diviser un problème en sous-problèmes, et pour savoir sur quel processeur chacun sera assigné et à quel moment il devra démarrer. Ces questions sont directement liées à ce que l'on appelle le partitionnement, le placement et l'ordonnancement des tâches. Selon ces choix on aura des performances plus ou moins bonnes. Le but de ce chapitre est de mettre ces problèmes en évidence et de les discuter plus en détail.

6.2 Calcul de l'ensemble de Mandelbrot

Pour commencer cette discussion, nous allons illustrer le problème du découpage d'un problème en sous-problèmes dans le cas du calcul de l'ensemble mathématique de Mandelbrot.

Pour construire cet ensemble, il faut itérer la relation $z_{n+1} = z_n^2 + c$ où z_i et c sont des nombres complexes, et regarder pour quelles valeurs de c, $z_n \to \infty$ pour n assez grand. En pratique, on itère cette relation jusqu'à ce que $\|z_n\|$ dépasse un seuil fixé (par exemple, la valeur 100) et on associe la valeur de n au c choisi.

On prend généralement $z_0 = 0$ et on considère un ensemble de N valeurs $c = c_x + ic_y$ également réparties dans une région prédéfinie du plan complexe. En représentant, pour chacune des valeurs de c une couleur correspondant au nombre d'itérations nécessaires pour dépasser le seuil choisi, on génère une image

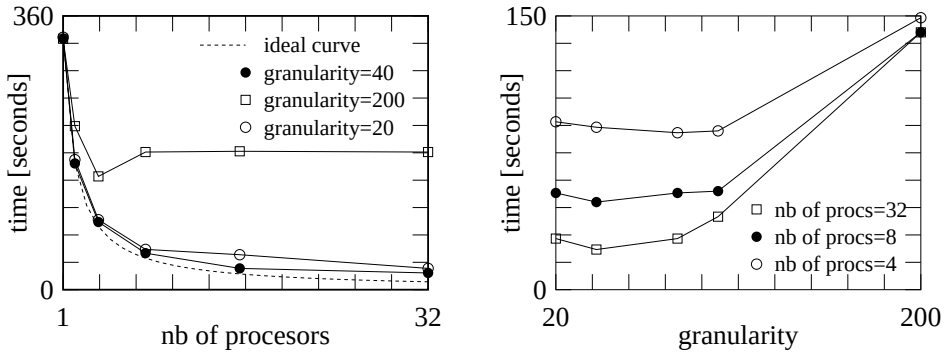


FIGURE 6.1 – *Performances obtenues pour le calcul de l'ensemble de Mandelbrot, en fonction de la granularité de la tâche et du nombre de processeurs. (Cas de la machine Archipel de Volvox, à 32 processeurs, 1992.)*

complexe, très spectaculaire, appelée ensemble de Mandelbrot.

Si l'on dispose d'une machine parallèle, on peut itérer simultanément la relation $z_{n+1} = z_n^2 + c$ pour toutes les valeurs de c considérées. A première vue, ce problème est de type SIMD puisque, pour chaque valeur de c, on effectue la même opération. Cependant, le nombre d'itérations à effectuer est différent d'un point à l'autre : une fois que l'on a dépassé le seuil fixé pour un c donné, on arrête le calcul. Dès lors, pour éviter d'avoir des processeurs inactifs, il est nécessaire qu'une fois son travail terminé, un processeur soit réaffecté à une autre valeur de c.

Ceci est possible avec une machine MIMD. Toutefois, le gain de performance que l'on observe n'est pas toujours optimum non plus, comme le montre la figure 6.1.

Les mesures ont été effectuées en faisant varier le nombre de processeurs collaborant au calcul de l'ensemble de Mandelbrot. Chaque processeur reçoit comme tâche le calcul d'une portion de l'espace des valeurs de c. Lorsqu'un processeur a terminé, un gestionnaire de tâches (ici une station de travail centrale) réalloue cette ressource de calcul à une nouvelle portion de l'image. Chaque tâche est caractérisée par la taille de ces portions d'image. On appelle ceci la **granularité** de la décomposition du travail et on la représente par un nombre l donnant la taille en pixels de chaque bloc d'image. Ainsi, une granularité de 40 signifie que chaque processeur reçoit un carré de 40×40 pixels à calculer. Ce calcul est illustré sur la figure 6.2.

Les courbes obtenues sur la figure 6.1 montrent que les performances dépendent crucialement de la granularité des tâches et que le gain maximum n'est jamais obtenu (idéalement, on espère que le temps d'exécution d'une machine comprenant N processeurs est inférieur d'un facteur N à celui d'une machine ne comprenant qu'un seul de ces processeurs). Bien que ce problème n'implique aucune communication de données entre les tâches (et constitue ainsi un cas de

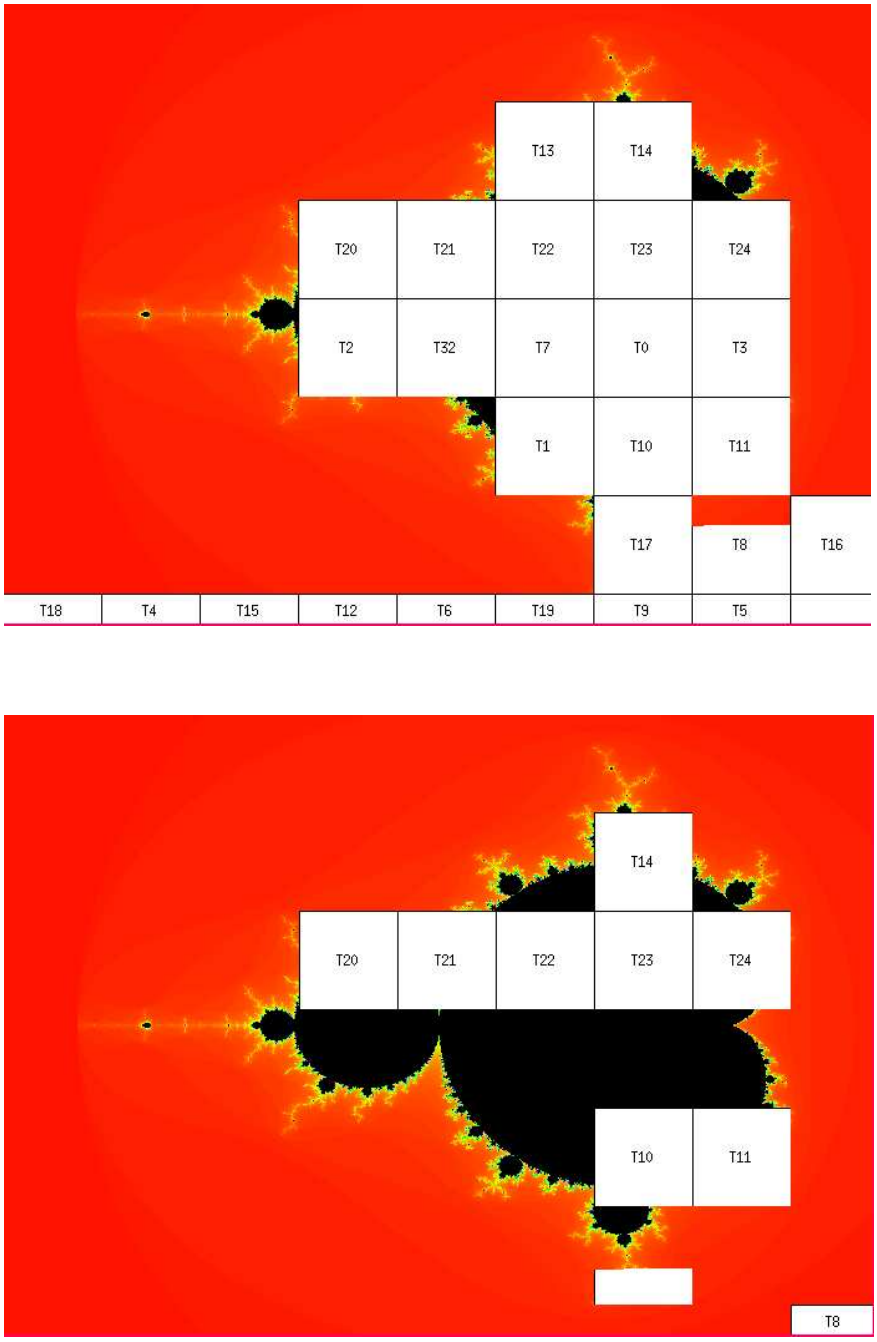


FIGURE 6.2 – *Calcul de l'ensemble de Mandelbrot avec la machine Archipel. Chaque carré désigne la portion de l'espace alloué au processeur indiqué (T0, T1,...,T32). La figure montre le calcul à deux étapes distinctes, illustrant la réaffectation dynamique des processeurs à de nouvelles tâches.*

parallélisme idéal), une saturation est observée, au delà de laquelle il est inutile d'ajouter encore d'autres processeurs. Ces résultats s'expliquent de la manières suivante :

1. Granularité trop fine : Le gestionnaire des tâches passe son temps à réaffecter les processeurs à de nouvelles régions du plan. Le travail de chacun est trop court et la réallocation devient le "goulet d'étranglement" de l'application. Le système n'arrive même pas toujours à utiliser tous les processeurs disponibles (en particulier lorsque très peu d'itérations par point sont suffisantes).
2. Granularité trop grossière : Comme toutes les tâches ne sont pas de la même durée, un petit nombre de processeurs se voit allouer des régions de l'espace nécessitant un long travail, alors que les autres terminent très rapidement le reste de l'image, sans pouvoir être réutilisés. En d'autres termes, une petite fraction des processeurs disponibles font effectivement le travail et on ne gagne rien à ajouter de nouveaux processeurs dans le calcul. Il y a donc une granularité idéale pour ce problème, et selon le graphique, on voit qu'elle vaut environ 40.

Cet exemple suggère plusieurs solutions pour améliorer la vitesse d'exécution et s'approcher de la courbe de performance idéale. On pourrait découper le plan des valeurs de c de manière inégale en un nombre de sous-régions identiques au nombre de processeurs utilisés, de sorte que chaque sous-région contienne la même quantité de travail. On aurait ainsi un équilibre des charges. Cette répartition n'est pas toujours facile à deviner et il peut être nécessaire d'exécuter le programme une fois afin de se rendre compte de la bonne décomposition. Dans le cas de l'ensemble de Mandelbrot, cela peut être une bonne approche au problème car, souvent, on désire en un deuxième temps zoomer une partie plus intéressante de l'image et on peut utiliser les calculs précédents pour améliorer la rapidité d'exécution de la suite du travail.

Une autre solution pour augmenter les performances de calcul est de resubdiviser dynamiquement les tâches les plus longues : lorsqu'il n'y a plus de travail pour un processeur, on peut essayer d'en enlever un peu à celui qui en aurait trop et de le redistribuer. Cela a l'avantage de ne pas nécessiter la connaissance des tâches les plus longues à l'avance, mais le désavantage d'avoir une gestion compliquée et intelligente de la répartition du travail. On reviendra plus loin sur certaines méthodes d'équilibrage dynamique des charges dans le paragraphe 6.7

6.3 Tâches et notion de dépendance

De façon très générale, un programme parallèle peut être vu comme un ensemble de tâches **concurrentes** dont certaines peuvent être exécutées simultanément alors que d'autres doivent être exécutées dans un ordre donné.

Formellement, une tâche T_i est une suite d'opérations exécutée en séquentielle par un seul processeur. Par définition elle ne contient donc aucune communication avec d'autres processeurs. Une tâche est par ailleurs caractérisée par

1. Un ensemble E_i de variables d'entrée qui sont nécessaires à l'exécution de la tâche.
2. Un ensemble S_i de variables de sortie qui sont modifiées lors de l'exécution.

Une tâche peut être vue comme un opérateur de l'ensemble de ses entrées sur l'ensemble de ses sorties. On peut associer à chaque tâche un temps d'exécution t_i.

Deux tâches T_i et T_j sont dites **indépendantes** si elles ne modifient pas leur variables communes :

$$S_i \cap S_j = E_i \cap S_j = E_j \cap S_i = \{\}$$

Des tâches indépendantes peuvent être exécutées en parallèle. Si elles ne sont pas indépendantes, il faut préciser leur ordre d'exécution. Si T_i doit être exécutée avant T_j, on note $T_i < T_j$.

La tâche T_j est **consécutive** à T_i s'il n'existe aucune autre tâche T_k devant être exécutée entre T_i et T_j.

Les dépendances entre tâches impliquent une relation dite de **précédence** entre elles qui permet la construction d'un graphe de tâches : un arc relie la tâche T_i à la tâche T_j si T_j est consécutif à T_i. Le graphe de précédence donne une représentation d'un algorithme et constitue un outil de base pour en discuter la parallélisation.

6.4 Partitionnement

Le partitionnement est la division du problème à résoudre en sous-tâches qui seront assignées à des processeurs différents.

En général, le partitionnement n'est pas unique et la granularité idéale n'est pas connue à l'avance. C'est souvent un problème de compromis. Plus le problème peut être découpé en petits morceaux, plus le parallélisme à disposition sera grand (ce qui est favorable à l'obtention de performances). Par contre, si les tâches sont trop courtes, l'overhead de parallélisation (communications, synchronisations) devient important et les performances globales chutent. Le partitionnement est donc un problème d'optimisation qui revient à chercher le découpage en tâches qui minimise le temps total d'exécution. Ce problème n'est pas simple en général car l'overhead dépend de la manière dont le problème est partitionné.

Cette situation est illustrée dans la figure 6.3, pour le calcul de la somme de N nombres sur p processeurs, avec $N \geq p$. Ce calcul se réalise en répartissant N/p valeurs par processeur. La somme se fait tout d'abord en additionnant ces N/p valeurs sur chaque processeur, afin d'obtenir p valeurs sur p processeurs. Sur

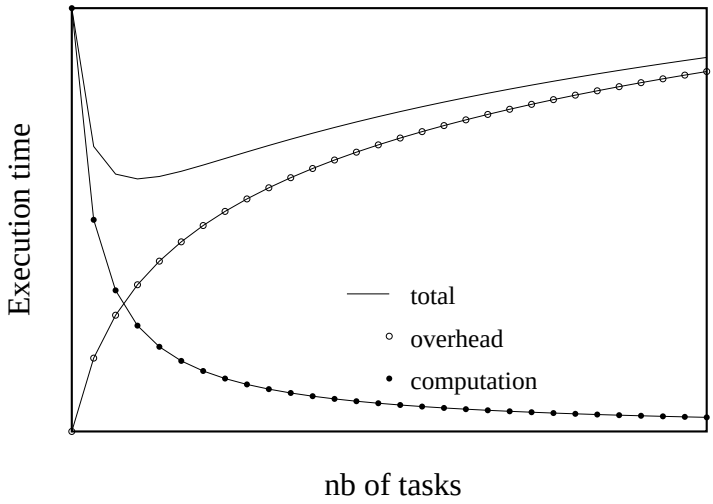


FIGURE 6.3 – *Temps d'exécution en fonction du nombre de tâches. Plus celui-ci est grand, plus le degré de parallélisation est grand et plus l'overhead de parallélisation devient important.*

une architecture en hypercube, il faut ensuite $\log_2 p$ communications et calculs pour sommer les p nombres placés sur les p processeurs. On peut donc séparer le temps total d'exécution en N/p pour le temps de calcul proprement dit et $\log_2 p$ pour l'overhead de parallélisation. Pour N fixé, le temps de calcul diminue si p augmente alors que l'overhead augmente.

6.5 Placement et ordonnancement

Le placement est le choix de quel processeur exécutera quelle tâche. On cherche à distribuer le travail parmi les processeurs pour garantir qu'un maximum d'entre eux soient utilisés à tout moment, tout en minimisant l'overhead de communication. Le placement est un problème difficile même dans des cas simples : supposons qu'on ait des tâches T_1, T_2,...,T_n à distribuer sur p processeurs. Même si ces tâches sont indépendantes, mais de durées différentes, une répartition qui optimise l'équilibrage de charge est difficile à trouver.

L'ordonnancement détermine le moment où chaque tâche débutera. Une fois que le partitionnement et le placement sont décidés, il faut encore assurer que les différentes tâches s'exécutent dans le bon ordre ou encore qu'elles n'attendent pas inutilement longtemps avant de démarrer. Une tâche ne peut débuter que si toutes celles dont elle dépend sont achevées. Comme il est probable que plusieurs de ces tâches soient sur des processeurs différents, il faut des mécanismes de synchronisation inter-processeurs. Il faut notamment éviter les situations d'interblocage (*deadlock*) dues au fait que deux tâches (ou plus) attendent mutuellement

leur achèvement respectif avant de pouvoir se terminer.

Les critères qui définissent un bon placement et ordonnancement sont évidemment les performances obtenues. Il faut en particulier diminuer l'overhead du parallélisme et obtenir un bon **équilibre des charges** (ou **load balancing**) sur tous les processeurs.

Le choix d'un bon partitionnement, placement et ordonnancement pour une application donnée est en général un problème d'optimisation combinatoire NP-complet. En d'autres termes, il n'existe pas d'algorithme simple qui le résolve rapidement. Heureusement, la nature de l'application considérée offre souvent des solutions heuristiques acceptables.

6.6 La méthode des temps «au plus tôt» et «au plus tard»

Dans ce paragraphe, nous allons illustrer le problème du placement et de l'ordonnancement par un exemple simple pour lequel les temps de communications inter-processeurs sont négligeables (parallélisation à gros grain). De plus, nous supposerons connu le temps d'exécution de chaque tâche, ce qui correspond à une situation d'ordonnancement statique car ces temps ne changent pas durant l'exécution du programme.

La méthode dite des temps au plus tôt et au plus tard constitue une technique permettant d'analyser le graphe de précédence d'une application et de construire le placement et l'ordonnancement des tâches (qui n'est pas forcément l'optimum absolu).

Pour illustrer cette technique, considérons l'exemple décrit dans la figure 6.4. Les différentes tâches sont notées T_i et les indications à côté de chaque tâche représentent la durée d'exécution. On commence par déterminer le premier moment où une tâche pourra débuter. Cela s'obtient en parcourant le graphe de haut en bas. La tâche T_1 est supposée commencer au temps $t = 0$. Au plus tôt, les tâches T_2 et T_3 pourront débuter au temps $t = 1$, soit dès que T_1 est terminée.

De même, les tâches T_4 et T_5 ne peuvent pas commencer avant que T_2 soit achevée, c'est-à-dire au plus tôt au temps $t = 3$. En poursuivant ce raisonnement, la dernière tâche T_9 ne pourra commencer qu'au temps $t = 14$. Donc, l'exécution optimum de ce programme sera telle que T_9 débutera à $t = 14$. Si T_9 débute plus tard, les performances ne seront pas les meilleures et il faut essayer d'éviter cela. Dans ce but, on cherche maintenant le dernier moment où chaque tâche peut commencer afin de garantir que T_9 démarre effectivement au bon moment. On observe ainsi que T_7 doit commencer au plus tard 7 secondes avant le temps optimal de T_9, soit au plus tard au temps $t = 7$. De même, la tâche T_8, ne durant que 4 secondes, doit débuter au plus tard au temps $t = 14 - 4 = 10$. En remontant de la sorte jusqu'à T_1 on détermine la table des temps « au plus

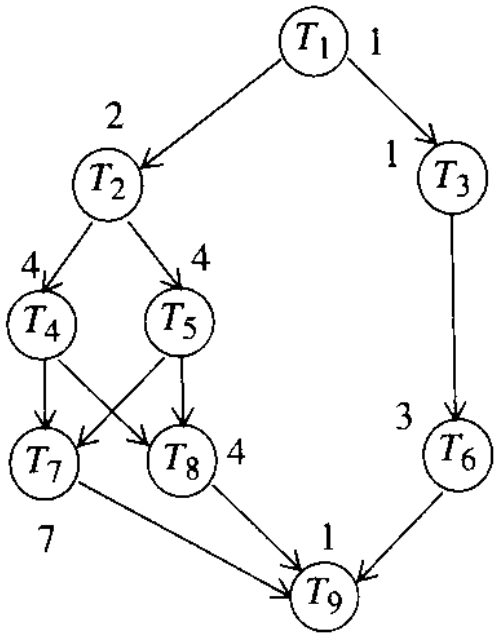


FIGURE 6.4 – *Graphe de précédence d'une application composée de plusieurs tâches (Moldovan, p. 385).*

tôt »et « au plus tard »indiquant le début de chaque tâche, comme le montre le tableau 6.1 (Moldovan, p. 385).

Comme on le voit, certaines de ces tâches ont ces deux temps identiques (T_4, T_7, par exemple), ce qui signifie qu'elles doivent impérativement débuter à un moment précis. Ce temps est d'ailleurs le même pour T_4 et T_5 ce qui indique qu'il faudra au moins deux processeurs si l'on ne désire pas introduire des délais supplémentaires et garder le rythme fixé.

Pour établir le placement final (et l'ordonnancement) correspondant à cette situation, on construit la table 6.2 dont chaque ligne correspond à une unité de temps (ici, on aurait 15 lignes, pour les temps $t = 0$ jusqu'à $t = 14$). Les colonnes correspondent aux processeurs à disposition. On place dans cette table les tâches dont le temps d'exécution est bien défini (earliest_ time=latest_ time), en partant de la première colonne. Si celle-ci est pleine, on remplit la colonne suivante, ce qui signifie qu'un autre processeur devient nécessaire. Une fois que toutes les tâches sont placées (on place en priorité les tâches pour lesquelles la différence latest_time-earliest_ time est minimum), le nombre de colonnes remplies indique le nombre de processeurs nécessaires. Ici, il en faut deux. On constate aussi que le premier processeur est utilisé à 100% alors que l'autre l'est 12 secondes sur 15 (soit 80%). Il y a donc un léger déséquilibre de charge.

Le speedup maximum qu'on peut espérer pour cette situation est facile à ob-

Task	Earliest Time	Latest Time
1	0	0
2	1	1
3	1	10
4	3	3
5	3	3
6	2	11
7	7	7
8	7	10
9	14	14

TABLE 6.1 – *Table des temps au plus tôt et au plus tard pour le graphe de la figure précédente.*

tenir de notre analyse : T_9 termine après 15 secondes avec 2 processeurs (ce qui est l'optimum pour cette décomposition en 9 tâches). Avec un seul processeur, le temps total serait la somme des temps de chaque tâche, soit 27 secondes. Le speedup maximum est donc 27/15=1.8, indépendamment du nombre de processeurs à disposition (cf loi d'Amdahl : le speedup est limité par la quantité de parallélisme à disposition).

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

TABLE 6.2 – *Placement et ordonnancement des tâches. Deux processeurs suffisent. Les cases marquées "idle" signifie que le processeur concerné est inactif.*

6.7 Equilibrage de charge

Comme on l'a vu déjà plusieurs fois, un déséquilibre de charge entre les processeurs est une cause importante de perte d'efficacité dans une application parallèle. On se souvient (paragraphe 4.2) que le speedup est donné par le **degré de parallélisme moyen**. S'il y a un fort déséquilibre de charge (load unbalance), certains processeurs seront inactifs et le degré de parallélisme diminuera. Une telle situation est illustrée dans la section 6.2, pour le calcul de l'ensemble de Mandelbrot. On constate dans cet exemple que seuls un petits nombre de processeurs travaillent pendant la durée complète de l'exécution. Le travail réalisé par de nombreux processeurs est immédiat et ne contribue pas à apporter un speedup intéressant. Dans une telle situation, il faut trouver des stratégies pour

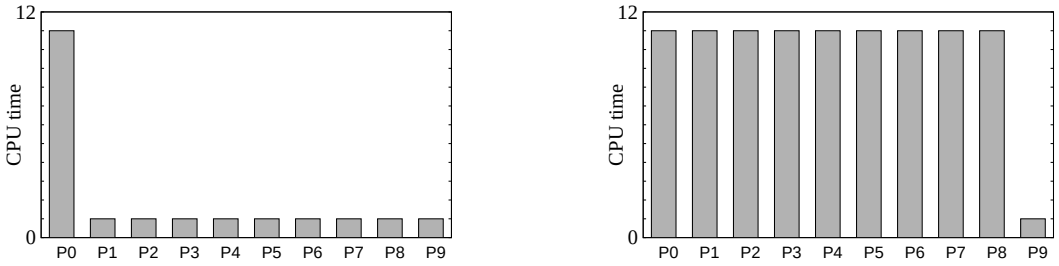


FIGURE 6.5 – Deux situations de déséquilibre de charge, avec les même temps T_{par}, mais un déséquilibre de nature très différente. A gauche, la situation est très mauvaise car presque tous les PE attendent sur le plus lent. A droit, seul un processeur attend. Le nombre de cycles perdus est bien moindre.

mieux partitionner le domaine de calcul ou bien il faut dynamiquement réallouer des parts de l'image aux processeurs qui ont terminé. Dans ce cas précis, un partitionnement basé sur un découpage cyclique (ou modulo) du domaine est une bonne solution.

La figure 6.5 illustre deux cas de déséquilibre de charge, caractéréisés par des temps d'exécution parallèles T_{par} identiques, déterminés par le processeur le plus chargé. On se rend bien compte que la situation de gauche est très mauvaise car un grand nombre de cycles machine sont gaspillés par les 9 PE qui attendent P_0. Par contre, dans l'image de droite, la situation est bien moins dramatique car un seul processeur est trop peu chargé.

6.7.1 Métrique de déséquilibre de charge

Il est donc important de proposer des métriques qui mesurent le déséquilibre de charge et dont la valeur reflétent la sévérité de la situation.

Une mesure de l'équilibre de charge est donnée par la valeur de l'efficacité E lorsque les communications et autres sources d'overhead peuvent être négligées. Une efficacité de 100% implique alors un équilibre parfait alors qu'une efficacité de 20% signifie que, globalement, seuls 20% des processeurs ont travaillé.

Cela peut se voir directement par le raisonnement suivant. Soit T_{seq} le temps d'exécution séquentiel du programme. Idéalement, le temps d'exécution parallèle devrait être T_{seq}/p, si chaque processeur hérite d'une portion identique du travail. On peut donner comme mesure de l'équilibre de charge, l'écart à 1 du rapport entre ce temps idéal et le temps du processeur le plus chargé (qui est d'ailleurs aussi le temps T_{par} de l'exécution parallèle)

$$\Delta = 1 - \frac{(T_{seq}/p)}{T_{par}} = 1 - \frac{T_{seq}}{pT_{par}} = 1 - E$$

où E est l'efficacité définie dans l'équation (5.5). La grandeur Δ ainsi définie varie entre 0 (pas de déséquilibre) et 1 (déséquilibre maximum). Dans le cas de la figure 6.5, en supposant que T_{seq} est la somme de temps de chaque PE, on a respectivement pour les images de gauche et de droite, les valeurs

$$\Delta = 1 - 20/(10 \times 11) = 0.818 \qquad \Delta = 1 - 100/(10 \times 11) = 0.09$$

ce qui reflète bien la différence des deux cas.

Il y a de nombreuses autres mesures, qui quantifient le déséquilibrage de charge indépendamment d'une référence au temps séquentiel, ce qui est raisonnable sachant qu'il y a en général un overhead de parallélisation (en particulier les communications).

Par exemple si T_i dénote le temps de calcul du processeur P_i pour réaliser sa part d'un travail W_{par} donné, la variance des T_i

$$V = \frac{\sum_{i=1}^{p}(T_i - \mu)^2}{p} \in [0, \infty]$$

est une mesure du déséquilibre, où

$$\mu = \frac{1}{p}\sum_{i=1}^{p} T_i$$

est la moyenne des temps T_i et p le nombre de processeurs. On notera que $\mu = T_{moy}$ donne le temps CPU d'une situation parfaitement équilibrée.

Si la variance V est nulle, l'équilibrage est parfait. Mais plus V est grand, plus le déséquilibre est important. Cependant, cette métrique n'est pas toujours très selective. Sur la figure 6.5 de gauche on a $T_{moy} = \mu = 2$ et $V = \mathrm{Var(T_i)} = 9.0$. A droite on a $T_{moy} = \mu = 10$ mais aussi $V = \mathrm{Var(T_i)} = 9.0$. La variance des temps d'exécution ne détecte donc pas la sévérité du déséquilibre de charge. Une amélioration serait de normaliser V par μ.

Une autre mesure de déséquilibre de charge qu'on va utiliser par la suite est donnée par

$$u = m - \mu \in [0, \infty] \tag{6.1}$$

où

$$m = \max_i(T_i) \qquad \text{et, comme avant,} \qquad \mu = T_{moy} = \frac{1}{p}\sum_{i=1}^{p} T_i$$

Une valeur nulle de u indique que $\max(T_i) = T_{moy}$, ce qui exprime un équilibrage parfait. A l'opposé, une valeur élevée indique un déséquilibrage substantiel. Pour les cas de la figure 6.5, on obtient respectivement

$$u = 11 - 2 = 9 \qquad u = 11 - 10 = 1$$

ce qui indique bien que le premier déséquilibre est plus grave que le deuxième.

En supposant comme ci-dessus qu'on peut négliger l'overhead dû aux communications, on peut poser que $m = \max_i(T_i) = T_{par}$ et $\mu = T_{moy} = (\sum_i^p T_i)/p = T_{seq}/p$. Il en résulte que u vaut dans ce cas

$$u = T_{par} - T_{seq}/p = T_{par}\,(1 - E) = \Delta \times T_{par}$$

Dans le reste de ce chapitre, on va considérer différentes situations de déséquilibre de charge et présenter des stratégies de répartition du travail permettant d'assurer que chaque processeur reste occupé le plus longtemps possible. Nous commencerons par le cas statique et étudierons ensuite le cas dynamique, plus compliqué. Il n'y a pas de méthode universelle qui résoud tous les cas. Souvent, le problème d'équilibrage de charge revient à résoudre un problème d'optimisation difficile pour lequel seules des heuristiques sont utilisables. Par conséquent, la suite de ce chapitre a seulement pour but d'illustrer les situations courantes et de proposer des solutions possibles.

6.8 Equilibrage de charge statique

.

Dans ce paragraphe, on suppose que le temps de calcul de chaque partitionnement possible est calculable et qu'il ne varie pas durant l'exécution du programme.

Le problème de l'équilibrage des charges est alors un problème de partitionnement et de placement qui peut, en principe, se résoudre avant exécution. La méthode des temps au plus tôt et au plus tard présentée au paragraphe 6.6 est un exemple de méthode de solution pour les problèmes à granularité plutôt grossière et un graphe de tâche irrégulier. Ci-dessous, nous décrivons d'autres approches liées à la recherche d'une répartition des données équitables.

Le cas le plus courant dans un programme SPMD est que le partitionnement est équivalent à un découpage du domaine de calcul en sous-domaines ("domain decomposition", en anglais). Souvent le temps de calcul est proportionnel au nombre de données du sous-domaine. Pour avoir équilibrage de charge, il suffit donc d'avoir un découpage en sous-domaines de taille égale. Une première difficulté est que souvent le domaine ne se divise pas exactement par le nombre de processeurs, à l'instar de l'exemple donné au paragraphe 5.2.

Mais si on néglige ce problème d'arithmétique, la prochaine difficulté est que, souvent, le domaine de calcul est géométriquement irrégulier. On peut penser à un plan de ville où les calculs ne doivent se faire que dans les régions entre les bâtiments. C'est notamment le cas lorsqu'on étudie comment le vent souffle dans les rues d'une ville ou encore si l'on modélise le trafic des véhicules.

6.8.1 «Space filling curves»

Il est en principe possible de numéroter tous les points du domaine de calcul, de les parcourir séquentiellement et d'en attribuer un nombre égal à tous les processeurs. Le parcours séquentiel d'un domaine en deux ou trois dimensions est un problème standard de l'informatique. On le fait couramment lorsqu'on décrit un tableau 2D par un indice unique qui parcourt les lignes du tableau les unes après les autres.

Un tel mapping s'exprime facilement si on considère un domaine à deux dimensions, par exemple de taille $2^b \times 2^b$ dont les coordonnées (x, y) sont exprimées en binaires

$$x = x_b x_{b-1} \dots x_1 \qquad y = y_b y_{b-1} \dots y_1$$

On peut construire

$$z = y_b y_{b-1} \dots y_1 x_b x_{b-1} \dots x_1 \tag{6.2}$$

par concaténation des expressions de y et x. Si on parcourt z de 0 à $2^{2b} - 1$ et qu'on ré-exprime x et y à partir de z on traverse le domaine ligne par ligne.

Mais il est souvent souhaitable de choisir un parcours qui préserve au mieux la localité des données au sens que des données proches dans la représentation z soient aussi proches dans les coordonnées (x, y). Cela permet d'exploiter au mieux la mémoire cache et de créer des domaines ayant un meilleur rapport surface sur volume dans le cas d'un partitionnement sur plusieurs processeurs. Un parcours linéaire qui «remplit» efficacement un domaine en D-dimensions est appelé en anglais une *Space filling Curve*.

La courbe dite **de Morton** est une modification de l'équation (6.2) qui permet un parcours unidimensionnel dans domaine à d-dimention et qui préserve mieux la localité que le choix précédent. En deux dimensions, la courbe de Morton est construite en alternant les bits de y et ceux de x

$$z = y_b x_b y_{b-1} x_{b-1} \dots y_2 x_2 y_1 x_1 \tag{6.3}$$

Le tableau suivant illustre cette construction pour un domaine 4×4.

	00	01	10	11
00	0000	0001	0100	0101
01	0010	0011	0110	0111
10	1000	1001	1100	1101
11	1010	1011	1110	1111

La figure 6.6 montre le parcours ainsi obtenu sur des domaines plus grands. On observe l'invariance d'échelle du parcours à mesure que le domaine croît en taille. On constate aussi que, le plus souvent, la localité spatiale est préservée.

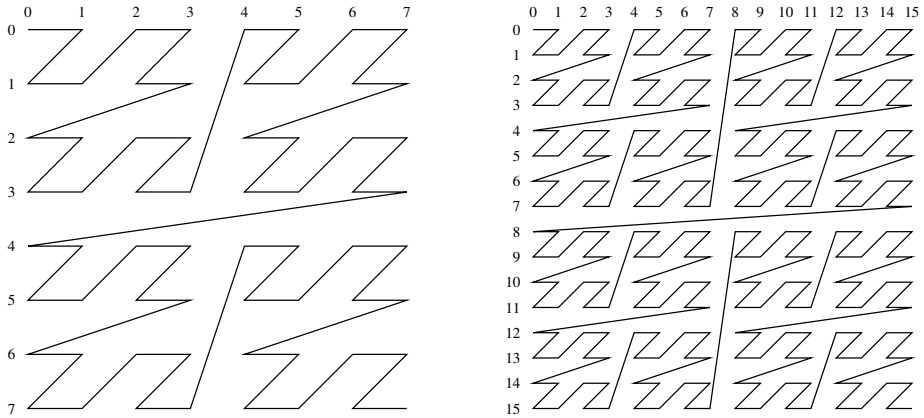


FIGURE 6.6 – Le parcours correspondant à la courbe dite de Morton dont le mapping est donné par l'équation (6.3).

Cependant, un des exemples le plus célèbre de «space filling curve» est donné par la **courbe de Hilbert** [1] qui est illustrée sur la figure 6.7. On notera cependant que la relation (voir https ://en.wikipedia.org/wiki/Hilbert_curve) entre la position sur la courbe et la coordonnée (x, y) correspondante n'est pas si simple que dans les exemples précédents. Cette relation est aussi difficile à inverser.

On peux aussi utiliser une courbe de Hilbert pour partitionner un domaine irrégulier. Sur la figure 6.8 un partitionnement équilibré des points à calculer en quatre sous-domaines est indiqué par les quatre couleurs. L'irrégularité du domaine est ici obtenu en supposant que la résolution de la grille de calcul varie en certains points. En conséquence, sur cet exemple, il y a 238 points à répartir sur quatre processeurs, à raison d'environ de 60 points pour chacun d'eux.

6.8.2 Bissection récursive

Une autre façon de partitionner un domaine de calcul est d'utiliser la technique dite de **bissection récursive** qui consiste à diviser par une ligne (ou un plan) le domaine de calcul en 2 parties contenant chacune un travail identique et de procéder récursivement sur chacun des morceaux ainsi obtenus, en altérnant les découpage dans les d dimensions du domaine. On obtient alors un partionnement contenant une puissance de 2 de sous-domaines. Cette procédure est illustrée sur la figure 6.9 où l'on voit un domain 2D irrégulier dont une partie est exclue du calcul (le rectangle gris). Ce domaine est itérativement divisé en deux parties de taille égale (au mieux de ce que la géométrie permet), par des découpes verticales alternant avec des découpes horizontales. L'exemple de la figure comprend au final quatre étapes et produit ainsi 16 sous-domaines. La tailles des sous-domaines est

1. `https://en.wikipedia.org/wiki/Hilbert_curve`

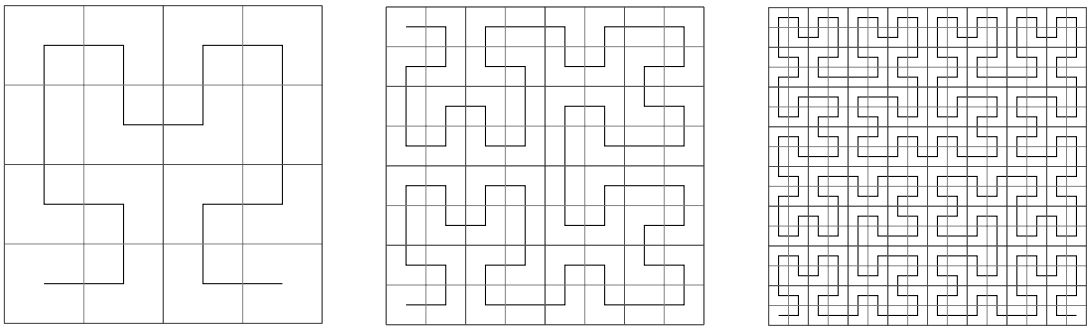


FIGURE 6.7 – Courbe de Hilbert qui parcourt un domain de calcul à deux dimensions, dont la résolution $m \times m$ augmente de gauche à droite, avec $m = 4, 8, 16$, respectivement. Le mapping entre l'indice i qui parcourt la courbe de Hilbert et les points (x, y) du domaine est donné sur `https://en.wikipedia.org/wiki/Hilbert_curve`. Ce mapping suppose que le nombre total de points est une puissance de 2.

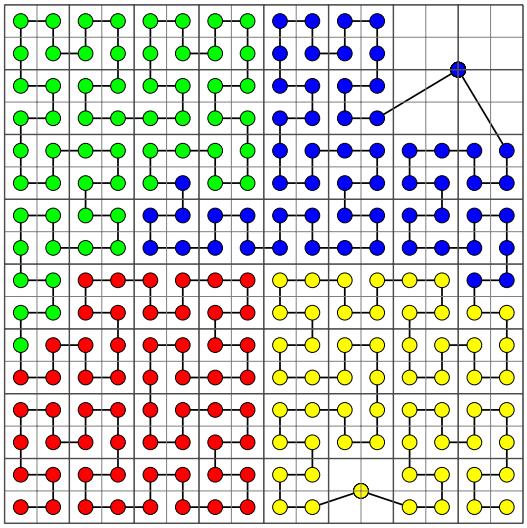


FIGURE 6.8 – Courbe de Hilbert dans un domaine dont la résolution varie d'une région à l'autre. Il y a dans cet exemple 238 points à calculer. Ce nombre est ici divisé en 4 pour réaliser une partition équilibrée sur 4 processeurs. Les couleurs indiquent l'appartenance des point à un même processeur.

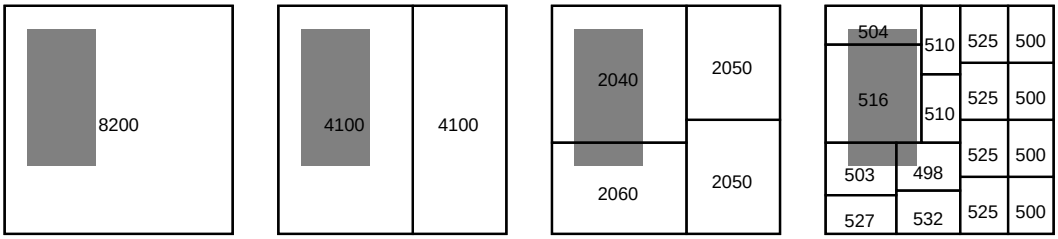


FIGURE 6.9 – Exemple de bissection récursive sur un domaine de calcul irrégulier, caractérisé par le rectangle grisé dans lequel aucun calcul n'est nécessaire. Le nombre total de point à calculer est 8200, au lieu de 10'000 pour le domaine complet. De gauche à droite, on voit l'effet des découpages successifs et la taille des sous-domaines obtenus,

donnée par les nombres indiqués sur la figure. Il y a des variations de tailles inévitables mais globalement le déséquilibre sera moindre que si on avait découpé le domaine de façon régulière.

6.8.3 Partitionnement de graphe

La méthode de la bissection récursive génèrent souvent des sous-domaines dont les longueurs de frontière sont très inégales. Donc, il n'y a pas de garantie que le découpage ainsi obtenu soit optimal en terme de communications, d'autant que certains processeurs auront peut-être leurs sous-domaines voisins éclatés parmi beaucoups d'autres processeurs.

De façon générale, le problème de construire des partitions de tailles identiques dont les frontières sont les plus petites possibles est équivalent à celui du **partitionnement de graphe**. Chaque point de calcul est connecté à d'autres selon les spécificités de la méthode numérique utilisée (p. ex. les 4 voisins nord, sud, est, ouest dans la résolution de l'équation de Laplace). Cela permet de représenter le domaine de calcul selon un graphe que l'on cherche ensuite à partitionner en sous-graphes, contenant tous le même nombre de sommets (pour obtenir l'équilibrage de charge) et minimisant le nombre d'arcs qui relient les sous graphes entre eux (pour avoir des communications efficaces). Des bibliothèques comme *Metis*, accessible sur le web, permettent d'obtenir des partitionnement de domaines irréguliers très satisfaisant. La figure 6.10 illustre le résultat obtenu par Metis pour le découpage d'un domaine irrégulier en huit morceaux

Finalement, dans le cas où aucune communication inter-processeur n'est nécessaire, une technique très simple peut-être utilisée, même si on ne connait pas les temps de calculs associés à chaque point du domaine. On peut alors utiliser une répartition des données **cyclique** (ou modulo), comme discutée au paragraphe 7.3. Dans ce partitionnement, chaque processeur contient des points provenant de

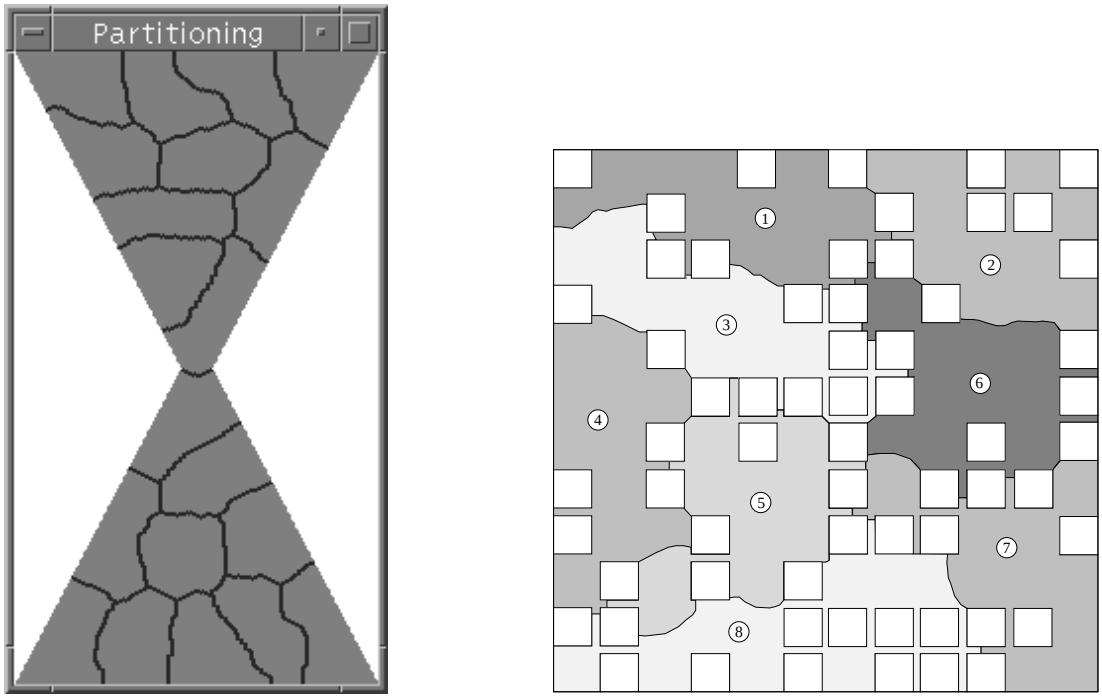


FIGURE 6.10 – *Partitionnement (avec Metis) de domaines irréguliers en sous-domaines afin d'équilibrer les charges et minimiser les communications. A droite, les carrés blancs indiquent les régions absentes du domaine de calcul. (Images : Alexandre Dupuis.)*

toutes les régions du domaine de calcul et, statistiquement, cela assure que le travail et bien réparti. Dans le cas de la recherche de motif dans une image, cette approche simple est souvent efficiace, à condition que chaque processeur ait une copie complète de l'image.

6.9 Equilibrage de charge dynamique

6.9.1 Problématique

Le cas de l'équilibrage de charge dynamique est plus compliqué car il se fait en cours d'exécution. Il est nécessaire lorsqu'on ne sait pas estimer les temps de calcul associés aux différentes tâches du problème, ou lorsque le partionnement doit être modifié dans un calcul itératif car la durée des tâches changent d'une itération à l'autre.

On peut illustrer une telle situation par le problème du trafic routier. Statiquement, on peut découper un réseau routier de façon que chaque processeur ait la même longueur de route ou le même nombre d'intersections. Mais le calcul du trafic dépend aussi du nombre de véhicules sur chaque tronçon de route, ce qui change au cours du temps et aussi en fonction des bouchons qui peuvent potentiellement se former, obligeant alors les véhicules à choisir d'autres parcours.

Les surcharges possibles induites dans un processeur par un autre utilisateur sont un autre facteur qui cause un déséquilibre de charge temporel dans une système parallèle multi-utilisateur.

Par rapport au cas statique, il y a maintenant de nouveaux enjeux :

— Il faut **détécter** un possible déséquilibre de charge lors de l'exécution. Il y a donc une partie du programme qui doit mesurer les charges respectives de chaque processeur ou les temps de calculs entre chaque points de synchronization.

— Il faut **décider** selon un critère donné si le déséquilibre de charge mesuré doit être traité ou non. Il peut être trop faible pour justifier un repartitionnement qui sera forcément coûteux en performance et pourrait masque le benéfice attendu. Ou bien, le déséquilibrage de charge est fluctuant est il n'est pas clair comment le corriger de façon durable.

— Finalement, il faut un mécanisme de **migration** de tâches ou de **re-répartition** des données entre les processeurs qui permet de diminuer le déséquilibre mesuré. Ceci s'obtient en appliquant les partitionnements proposés dans la section 6.8.

Pour chacun des trois éléments ci-dessus, on peut choisir une **stratégie globale** (ou centralisée) qui fait jouer un rôle particulier à un processeur donné qui se charge de comparer les temps d'exécution, de les analyser selon certains critères heuristiques et ensuite de repartitionner tout le calcul. C'est la solution la plus simple car toute l'information est rassemblée en un seul processeur. Mais c'est

aussi la moins scalable et celle qui va créer le plus de congestion.

Une **stratégie locale** est aussi envisageable pour diminuer les coût d'overhead de la détection et du traitement du déséquilibre de charge. Dans une telle situation, les proceseurs se comparent à leur voisins immédiats et décident entre eux qui est le plus chargé. Comme ces processeurs doivent de toute façon communiquer aux points de synchronization, l'overhead d'échanger en plus un temps de calcul ainsi qu'éventuellement une partie du sous-domaine de calcul reste petit. Malheureusement, une telle stratégie peut être lente à converger si un processeur a des contraintes contradictoires entres ses differents voisins. De plus, si la charge varie dans le temps plus vite que l'algorithme ne converge, il n'y aura jamais d'équilibre de charge global.

6.9.2 Exemples dans le cas itératif

A titre d'exemple de cette approche, la figure 6.11, gauche illustre un cas de résolution sur 4 processeurs de l'équation de la chaleur sur un domaine irrégulier, dû à des régions (les rectangles blancs) où la température est imposée et pour lesquelles aucun calcul n'est nécessaire.

L'équilibrage dynamique de la charge s'obtient par un mouvement progressif des frontières entre les sous-domaines, au cours des premières itérations du calcul. Dans le cas présent, ceci est facile à mettre en oeuvre car à chaque itération, les processeurs ont besoin des valeurs de températures des colonnes du bord des sous-domaines voisins. Il est donc possible, lors de la communication de ces colonnes d'échanger aussi la charge des processeurs respectifs. Cette charge est simplement estimée par le nombre de points de grille associés à chaque processeur. Si l'un de ces processeur a une charge inférieure à sont voisin, il gardera la colonne que ce dernier lui aura transmise dans la phase de communication, agrandissant ainsi son sous-domaine au profit de son voisin.

Sur la figure 6.11 (gauche) on voit, en couleur, le champ de température après convergence du calcul. On voit aussi le partitionnement initial en quatre sous-domaines de taille égale (lignes blanches verticales) et le partitionnement final (lignes noires) Sur la partie droite de la figure, on voit l'évolution des 3 frontières séparant les 4 sous-domaines, selon l'algorithme d'équilibrage dynamique décrit ci-dessus. On constate une convergence rapide de la position idéale de ces frontières. Pour ce résultat, il faut que la tolérance de différence de travail entre deux sous-domaines soit au moins de 100 sites, correspondant à la taille d'une colonne. Si on veut forcer un écart plus faible, les frontières se mettront à osciller comme le montre la figure 6.12 où un échange de colonne se fait dès que la différence de taille des sous-domaines adjacents dépasse 50 points de grille.

Notons qu'on peut évidemment combiner les approches globales et locales ce qui, selon les problèmes, permet d'exploiter le meilleur de chacune des stratégies.

On va maintenant présenter un exemple dans lequel le domaine de calcul s'agrandit au cours du temps. Il s'agit ici de calculer le front d'injection d'un

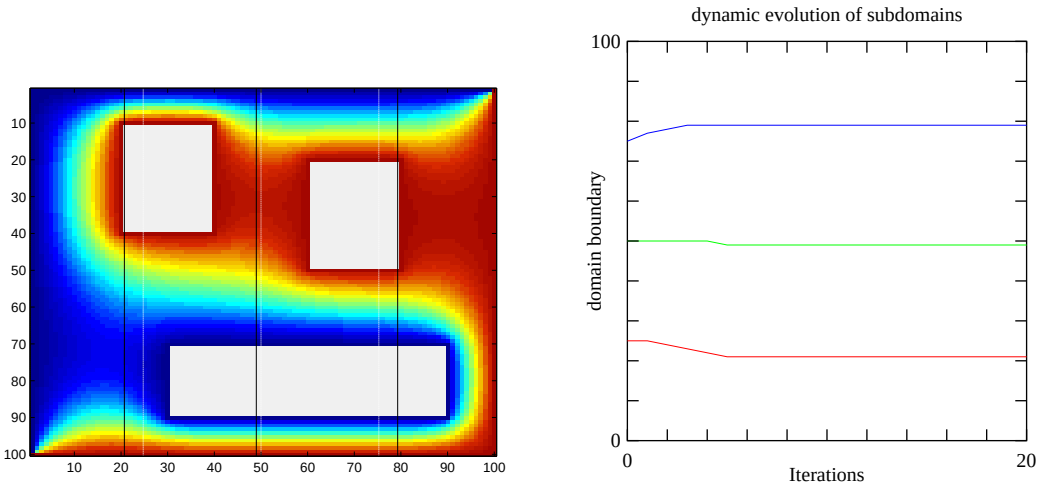


FIGURE 6.11 – Equilibrage dynamique de la taille des sous-domaines dans le cas de la résolution de l'équation de la chaleur. Les sous-domaines s'agrandissent ou se rétrécissent après chaque communication, en gardant ou non la colonne de sites communiquée par ses voisins (Simulations : Christophe Charpilloz).

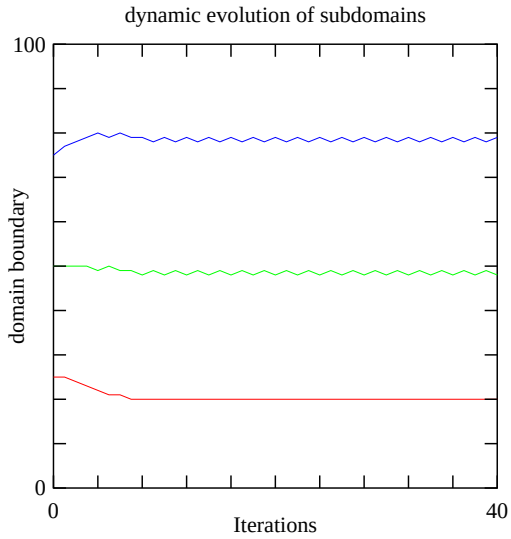


FIGURE 6.12 – Situation identique au cas de la figure précédente, mais avec une exigence trop forte sur l'égalité de la taille des sous-domaines (Simulations : Christophe Charpilloz).

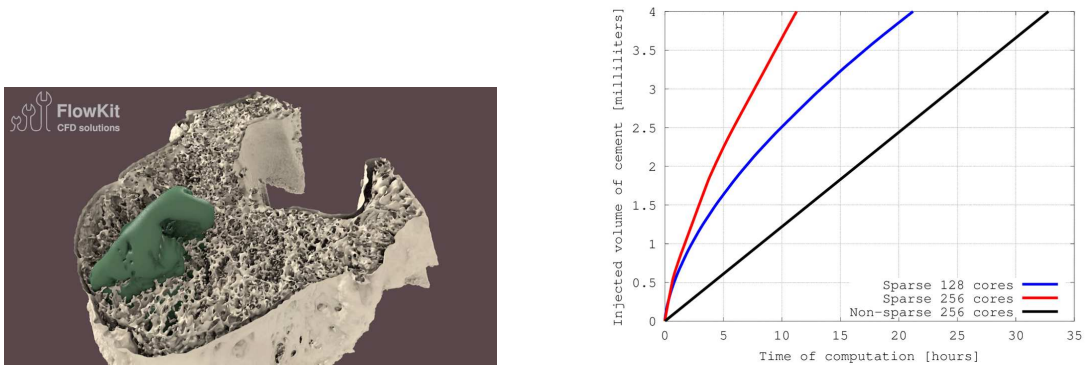


FIGURE 6.13 – **Gauche :** Simulation de l'injection de ciment dans une vertèbre fracturée. **Droite :** performance avec (noté *sparse* sur la figure) et sans (noté *non-sparse* sur la figure) équilibrage de charge dynamique. (Simulations et image : Jonas Latt, logiciel Palabos).

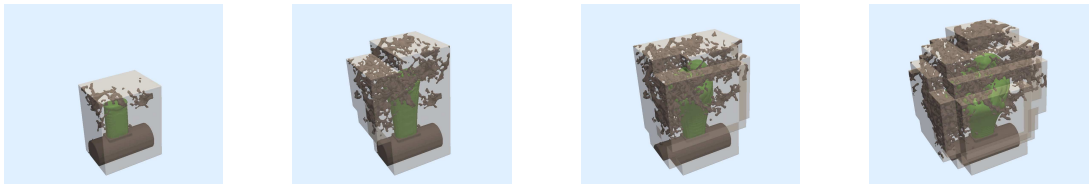


FIGURE 6.14 – A mesure que la simulation avance, on élargit la région sur laquelle les calculs ont lieu (Simulations : Jonas Latt, Palabos).

ciment dans une vertèbre fracturée, comme sugéré à la Fig. 6.13 (gauche).

La question à laquelle les médecins souhaitent répondre est la quantité exacte de ciment qu'il faut injecter dans la vertèbre. S'il n'y an a pas assez, la réparation ne se fera pas, s'il y en a trop, le ciment pourrait déborder de la vertèbre et causer des dommages aux vertèbres voisines. La simulation numérique permet donc d'évaluer au mieux la quantité idéale de ciment. Il faut une image de la vertèbre endommagée et connaître le point d'injection. Au début de la simulation, seule la région proche du point d'injection change. Les processeurs qui sont chargés de calculer les zones éloignées de ce point d'injection, n'ont rien à faire avant que le front de ciment ne les atteignent. Pour cette raison, un équilibrage de charge dynamique est souhaitable. Ici on repartitionne le domaine à mesure que le calcul avance, comme l'illustre la Fig. 6.14. A chaque étape d'équilibrage de charge, les processeurs se partagent un domaine qui grandit car il suit le front d'injection. Sur la Fig. 6.13 (droite) on peut voir le gain de temps de calcul qui découle de cet équilibrage de charge. En comparant les courbes rouge et noir, on voit que l'équilibrage de charge dynamique a permis un gain de temps d'un facteur 3 environ.

Dans le paragraphe suivant, nous discutons plus en détail le cas d'une ap-

plication **non-itérative** dans laquelle chaque proceseur qui a terminé le travail qui lui était initialement alloué essaye d'obtenir un supplément de la part des processeurs encore occupés.

6.9.3 Critère de rééquilibrage

Comme discuté précédemment, une partie importante du rééquilibrage de charge dynamique est de décider si un re-partitionnement est nécessaire à un moment de l'exécution. On considère encore ici une application itérative et on va dériver un critère, basé sur le temps CPU des itérations précédente, qui indiquera qu'il est opportun de rééquilibrer les charges. Ce critère est obtenu par un calcul de minimisation du temps T_{par} qui suppose un **principe de persistence**, à savoir que la cause de déséquilibre de charge va continuer suffisamment longtemps tout au long de l'exécution.

Soit $m(t)$ le temps CPU de l'itération t de notre application itérative. Ce temps est, comme on l'a souvent répété, le temps CPU du processeur le plus lent. Si on exécute N iterations, on a que le temps total sera

$$T_{par} = \int_0^N m(t)\ dt$$

où, pour des simplifications d'algèbre, on a remplacer la somme des $m(t)$ par une intégrale. On peut aussi écrire cette équation en y ajoutant et retranchant les temps moyen $\mu(t)$ de chaque itération

$$T_{par} = \int_0^N m(t)\ dt = \int_0^N [m(t) - \mu(t)]ds + \int_0^N \mu(t)dt \tag{6.4}$$

On va maintenant supposer qu'aux itérations $t_i = t_{i-1}+\tau_{i-1}$, $i = 1, \ldots, n$, on effectue un réquilibrage de charge dont le coût CPU vaut C_i pour le re-partitionnement des données. Selon l'équation (6.1), on introduit

$$u_i(t) = m_i(t) - \mu_i(t)$$

le déséquilbre de charge observé durant les itérations $t \in [t_{i-1}, t_i]$. La figure 6.15 illustre un scénario d'équilibrage aux itérations t_i. Après chaque équilibrage, on s'attend à ce que u_i soit nul ou presque, selon que la nouvelle répartition de charge soit paraite ou non. On écrit alors

$$T_{par} = \int_0^N \mu(t)\ dt + \sum_{i=1}^{n} \left(\int_0^{\tau_i} u_i(t)dt + C_i \right) \tag{6.5}$$

On notera que la fonction u_i dépend à priori du moment auquel l'équilibrage se fait. Cela rend très difficile la découverte des t_i qui minimisent T_{par}. Il y a un cas où l'équation (6.5) peut être minimisée. C'est en supposant que tous les intervalles

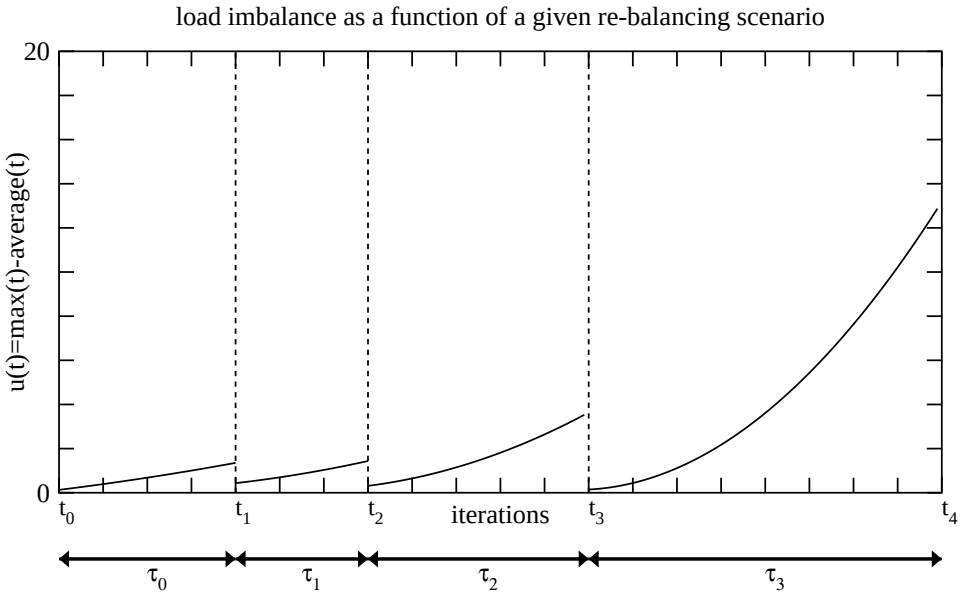


FIGURE 6.15 – Illustration des fonctions u_i qui décrivent l'augmentation du déséquilibre de charge suite aux re-partitionnements effectués aux itération t_i.

entre rééqquilibrage sont identiques et correspondent à $\tau_i = \tau$ itérations, et que de plus la fonction $u_i = u$ est toujours la même (principe de persistence) et que $C_i = C$. On obtient donc que

$$T_{par} = \int_0^N \mu(t)dt + \frac{N}{\tau}\left(\int_0^\tau u(t)\ dt + C\right) \tag{6.6}$$

La valeur optimale de τ s'obtient en posant

$$\frac{\partial T_{par}}{\partial \tau} = 0$$

On a que

$$\frac{\partial T_{cpu}}{\partial \tau} = -\frac{N}{\tau^2}\left(\int_0^\tau u(t)dt + C\right) + \frac{N}{\tau}u(\tau) = 0 \tag{6.7}$$

dont la solution est

$$\tau u(\tau) - \int_0^\tau u(t)dt = C \tag{6.8}$$

Cette relation est implicite et tant que u n'est pas connu, on ne peut pas déterminer τ a priori. Cependant, en pratique il faut considérer l'équation (6.8) comme des termes à calculer à la volée, à chaque itération, en utilisant la valeur de u mesurée. Comme l'indique la figure 6.16, l'itération τ pour laquelle la surface du

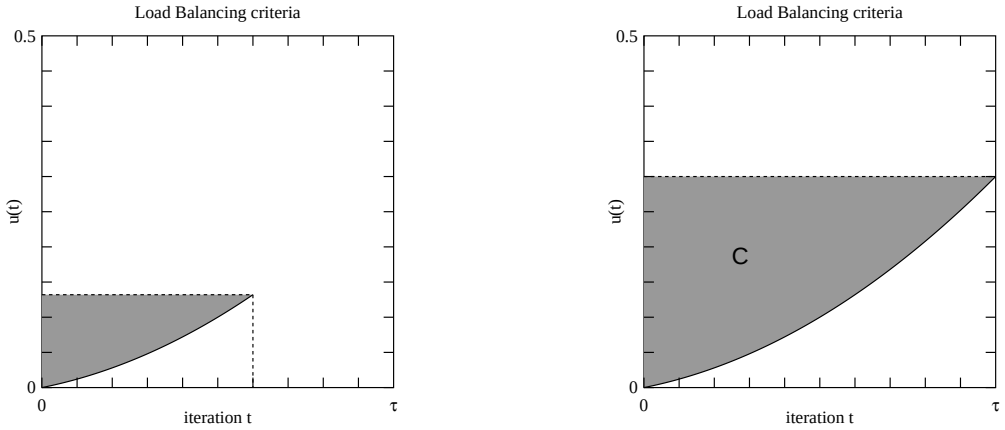


FIGURE 6.16 – Illustation du critère de rééquilibrage de charge donné par l'équation (6.8). A gauche, la surface hachurée n'atteint pas encore la valeur critique C. Il ne faut donc pas rééquilibrer à ce stade. A droite, on atteint une itération τ pour laquelle le critère est vérifié. Il faut don rééquilibrer les charges.

rectangle $\tau \times u(\tau)$ soustraite de la surface sous la courbe $u(t)$ atteint la valeur C, il vaut la peine de rééquilibrer les charges.

On remarque aussi que, même si l'équation (6.8) a été dérivée sous des hypothèses précises, elle s'applique de façon identique dans tous les cas. Cependant, dans le cas général, il n'y aura pas de garantie que la valeur de τ obtenue sera obtimale.

6.9.4 Cas non-itératif

Dans le cas d'applications non-itératives, chaque sous-domaine n'est calculé qu'une seule fois et la stratégie est différente que dans le cas itératif où l'on peut ajuster progressivement la taille de ces derniers entre les itération consécutives.

Pour commencer, nous allons envisager une situation symétrique (ou presque) telle que tous les processeurs jouent un rôle identique. Chacun reçoit initialement un ensemble de tâches (task pool), selon un critère quelconque et en commence l'exécution.

Lorsqu'un processeur a terminé le travail qui lui était initialement attribué et qu'il devient inactif, on obtient un déséquilibre de charge parmi les processeurs. Cela conduit à une dégradation des performances. Une répartition dynamique du travail (*dynamic load balancing*) doit être envisagée : un processeur inactif envoie une requête à un autre processeur du système pour obtenir du travail. Si ce dernier a suffisamment de travail, il en distribura la moitié au demandeur. S'il ne dispose pas assez de travail, il retournera une réponse négative car l'overhead

de transfert de travail peut être important et ne pas être justifié si la quantité de travail restante est trop petite. Dans ce cas, le processeur demandeur envoie une nouvelle requête auprès d'un autre processeur.

Le pseudo-code ci-dessous illustre cette stratégie locale de demande de travail. Les processeurs actifs sont continuellement à l'écoute des autres chaque fois qu'ils ont réalisé une fraction ΔW du travail qui leur a été attribué.

Active PEs	requesting PEs
	loop
	check messages :
loop	if response== "no-work" : mark PE as asked
execute ΔW	if response== "work" : exit loop and become active
Check all n messages requesting work	if response== "finished" : mark PE as finished
Respond with yes/no, and if yes, send $1/(n-1)$ part of the remining work	
	if not all PE are finished :
If no more work : be a requesting PE	if still unasked PEs : ask one PE for work
end loop	else : become finished
	else : exit loop
	end loop
	MPI_finalize()

En terme de performance, la situation la plus défavorable arrive si seul le processeur P_k détient la totalité du travail restant, alors que tous les autres sont devenus inactifs. La question est combien de requêtes faut-il envoyer pour être sûr que le processeur P_k reçoive au moins une demande. On définit dans ce but la quantité $V(p)$ comme le nombre total de requêtes nécessaires pour qu'au moins une requête ait atteint chacun des p processeurs présents.

Cette fonction $V(p)$ exprime en partie l'overhead résultant de la répartition dynamique et va dépendre de la stratégie d'équilibrage de charge choisie (c'est à dire comment et à qui les requêtes sont envoyées). On distingue trois stratégies courantes d'équilibrage : l'*asynchronous round robin*, le *global round robin* et le *random polling* (sondage aléatoire).

Asynchronous Round Robin (tour de rôle)

Pour cette stratégie, chaque processeur détient une variable indiquant auprès de quel autre processeur il faudra adresser la prochaine requête. Cette variable contient typiquement l'adresse ou le numéro du processeur auquel la demande sera envoyée.

Initialement, cette variable pointe sur le processeur voisin (`local_address+1`) et elle est incrémentée de 1 modulo P après chaque envoi de requête. Les différentes requêtes sont donc faites de manière non concertée entre les processeurs et deux d'entre eux peuvent très bien envoyer une requête à un même troisième

processeur.

La situation la plus défavorable pour le calcul de $V(p)$ est obtenue si le processeur P_0 est le seul occupé et que tous les autres pointent leur requête vers P_1. Dans ce cas, chacun des $p-1$ processeurs inactifs doit envoyer $p-2$ requêtes avant qu'une atteigne P_0. Il s'en suit que $V(p) = (p-1)(p-2)+1 = O(p^2)$.

En général, cependant, la situation est moins défavorable et $V(p)$ se situe entre p et p^2.

Global Round Robin

Dans le cas du global round robin, une variable unique détenue par le processeur P_0, par exemple, contient l'adresse du prochain donneur potentiel. Chaque fois qu'un processeur devient inactif, il demande et reçoit de P_0 l'information à qui s'adresser. En communiquant cette donnée, P_0 incrémente du même coup la variable pour pointer sur le donneur suivant. Le processeur demandeur envoie alors une requête de travail au donneur dont il a reçu le nom.

Cette stratégie a l'avantage de répartir équitablement les requêtes parmi tous les processeurs. Par contre, le processeur P_0 est un point de congestion du réseau, à moins qu'on utilise une primitive telle que fetch-and-add pour l'interroger.

L'évaluation de $V(p)$ est simple : après exactement p requêtes, on aura reçu exactement une demande. Donc $V(p) = p$.

Random Polling

Le random polling et une méthode simple pour faire de l'équilibrage dynamique des charges en sondant aléatoirement les autres processeurs pour obtenir du travail. Les processeurs inactifs sélectionnent au hasard un donneur, ce qui conduit à une distribution homogène des requêtes. La fonction $V(p)$ est plus difficile à estimer. En fait, elle peut être arbitrairement grande dans le cas improbable où un processeur n'est jamais choisi comme donneur. On posera donc que $V(p)$ est le nombre moyen de requêtes nécessaires pour être sûr d'avoir atteint tout le monde.

Pour calculer cette valeur, on va chercher la probabilité $P_{p-i,j}$ d'atteindre les $p-i$ processeurs restant en j requêtes. Si i processeurs sont déjà "marqués", la prochaine requête peut soit marquer un nouveau processeur avec une probabilité $W_{i,i+1} = (p-i)/i$, soit marquer un processeur déjà atteint, avec probabilité $W_{i,i} = i/p$. Donc, on peut écrire la relation de récurrence

$$P_{p-i,j} = P_{p-(i+1),j-1} W_{i,i+1} + P_{p-i,j-1} W_{i,i}$$

C'est à dire que la probabilité de marquer $p-i$ processeurs en j essais, c'est la probabilité d'en marquer un pour commencer et de marquer les $p-(i+1)$ restant en $j-1$ essais ou bien de ne pas en marquer un et de tous les marquer en $j-1$ essais.

On a évidemment $\sum_j P_{p-i,j} = 1$ et $P_{p-i,j} = 0$ si $j < (p-i)$. Le nombre moyen d'essais pour marquer les $p-i$ cases restantes est

$$f(i) \equiv \sum_j j P_{p-i,j}$$

et $V(p)$ s'exprime comme $V(p) = f(0)$. En multipliant la relation de récurrence pour $P_{p-i,j}$ par j et en sommant sur tous les j, on obtient une relation pour $f(i)$:

$$f(i) = \frac{p-i}{p}(f(i+1)+1) + \frac{i}{p}(f(i)+1)$$

d'où

$$f(i) = f(i+1) + \frac{p}{p-i}$$

et finalement

$$f(0) = p\sum_{k=0}^{i} \frac{1}{(p-k)} + f(i+1) = p\sum_{l=1}^{p} 1/k$$

Comme

$$\sum_{l=1}^{p} 1/k \sim \ln(p)$$

on obtient que $V(p) = O(p\ln(p))$.

Troisième partie

Modèles de programmation

Chapitre 7

Modèle de programmation à mémoire distribuée

Dans une architecture à mémoire distribuée, l'espace mémoire est fragmenté entre les différents processeurs. Afin de coopérer, les processeurs doivent se transmettre des informations tout au long du calculs. L'architecture à mémoire distibuée impose que ces échanges se fassent à travers des messages inter-processeurs, dans le même esprit que l'on envoie une lettre par la poste pour communiquer avec quelqu'un qui est éloigné.

Cette façon de communiquer donne lieu à un modèle de programmation spécifique, appelé **modèle par échange de messages**. Il est mis en oeuvre grâce à des appels à des bibliothèques d'échange de messages spécialisées, à partir de langages de programmation classiques tels que Fortran, C, C++ ou Java.

Les bibliothèques d'échange de messages les plus courantes sont MPI (Message Passing Interface) et PVM (Parallel Virtual Machine). Le but de ce chapitre n'est pas d'expliquer comment utiliser MPI ou PVM mais plutôt d'indiquer la philosophie de ce modèle de programmation, ses principes de base, son implémentation possible et des applications typiques.

7.1 Principes de base

Dans le modèle à échange de message, il n'y a pas de variables globales. Chaque processeur n'a accès qu'à sa propre mémoire et le programmeur est responsable de gérer la répartition des données et les transferts d'information.

En général, un programme utilisant le modèle à échange de message est du type SPMD (Single Program, Mutilple Data). Chaque processeur exécute ainsi le même programme, de façon asynchrone. Cependant, le comportement de chaque programme peut être différencié sur la base de l'identité du processeur qui l'exécute.

En effet, un ingrédient fondamental de la programmation par échange de

message est que chaque processeur se voit attribuer une adresse (ou encore une identité ou un rang) qui le distingue des autres. Par exemple dans MPI, les p processeurs [1] participants à une exécution parallèle sont numéroté de 0 à $p-1$. Ainsi, chaque processeur peut avoir accès au rang qui lui a été attribué par l'environnement d'exécution (`mpirun` par exemple) en appelant une fonction adéquate de la bibliothèque. Typiquement, dans une pseudo-syntaxe, la fonction

```
my_rank(id,nb_procs)
```

retourne, pour chaque processeur, une valeur différente entre 0 et $p-1$ dans la variable locale `id`. Le nombre de processeurs p choisi pour l'exécution en cours est aussi accessible à travers la variable `nb_procs`.

L'esprit de la programmation par échange de message est donc de paramétrer le programme en fonction de cette adresse `id` et du nombre de processeurs p.

Le deuxième élément essentiel du modèle de programmation par échange de message est justement sa capacité d'envoyer et recevoir des messages entre n'importe quelles paires de processeurs. Il y a donc deux primitives fondamentales de communication `SEND` et `RECEIVE`.

La fonction `SEND` prend en argument le rang du destinataire souhaité du message, ainsi que la nom de la variable contenant la valeur à communiquer à ce destinataire. De façon analogue, `RECEIVE` précise l'identité de l'expéditeur en provenance duquel un message est attendu, ainsi que le nom d'une variable où sera stocké l'information reçue.

L'exemple ci-dessous illustre l'envoi du contenu de la variable `a` contenue dans le processeur de rang 3 au processeur de rang 8. Ce dernier stocke la valeur reçue dans sa variable `b`

```
my_rank{id,nb_procs}
if(id==3){ a=100; send(a,8)}
if(id==8){receive(b,3)}
```

En pratique, d'autres arguments sont nécessaires pour les fonctions d'envoi et de réception. De même, il y a de nombreuses variantes de `SEND` et `RECEIVE` qui permettent de préciser plus finement les modes d'envoi ou de réception (bloquant ou non-bloquant,...) Nous renvoyons le lecteur au paragraphe 7.4 ou à un guide de programmation détaillé pour en savoir plus (p.ex. Parco, MPI doc,..).

Les bibliothèques d'échange de messages offrent aussi la plupart des primitives de communications collectives décrites au paragraphe 3.1.3, ainsi qu'une **barrière de synchronisation**, qui est une forme de communication collective, permettant de bloquer l'avancement des processeurs jusqu'à ce qu'ils aient tous effectué l'appel à cette primitive barrière. Cela peut être nécessaire pour gérer

1. Plus généralement, on devrait dire les p processus car le nombre de processeurs demandés peut excéder le nombre de processeurs physiquement disponibles.

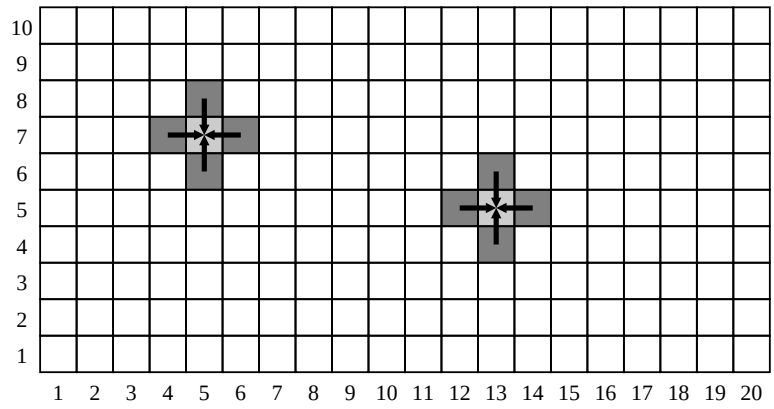


FIGURE 7.1 – *Domaine de calcul sur lequel on applique la relation itérative 7.1. Graphiquement, ce calcul peut s'illustrer par l'échange d'information indiqué par les flêches. Le point en gris clair est le point dont on calcul la nouvelle valeur et, ceux en gris foncé, sont les points dont la valeur est nécessaire.*

l'accès à une ressource partagée comme par exemple un fichier de données, ou mesurer le temps d'exécution d'un programme parallèle.

En général, les bibliothèques d'échange de messages permettent de diviser l'ensemble des processeurs disponibles en plusieurs groupes. Les opérations collectives sont alors restreintes à un groupe donné, identifié par `group_id` ou, dans MPI, son *communicator*. Des opérations inter-groupes sont évidemment aussi possibles.

7.2 Un exemple typique

Le modèle de programmation par échange de message est particulièrement bien adapté à la parallélisation du calcul itératif sur une grille. Par exemple, on considère un domaine de calcul en 2D, de taille $n \times m$ et on suppose que le but du calcul est d'itérer k fois la transformation :

$$s_{ij} = f(s_{i-1,j}, s_{i+1,j}, s_{i,j-1}, s_{i,j+1}) \tag{7.1}$$

où f est une fonction donnée et s_{ij} des valeurs numériques associée à chaque point de grille de coordonnée (i, j). On suppose que des valeurs initiales pour s_{ij} sont fixées et que le domaine de calcul global est périodique c'est à dire les les indices i et j sont pris respectivement modulo n et modulo m. La figure 7.1 illustre le domaine de calcul. Pour paralléliser un tel calcul, on commence par

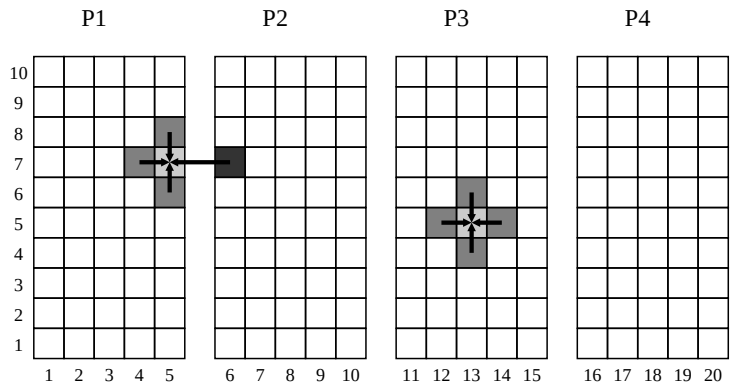


FIGURE 7.2 – *Partitionnement du domaine de calcul de la figure 7.1 en sous-domaines et illustration de la communication inter-processeur qui est necessaire pour calculer les points frontières.*

fractionner (on dit en pratique **partitionner**) le domaine de calcul en p tranches de taille $n \times (m/p)$, chacune assignée à un processeur différent. Dans chaque processeur, un tableau 2D est définit pour contenir ces valeurs, comme illustré dans la figure 7.2. Le calcul des s_{ij} qui sont strictement à l'intérieur de chaque sous-domaine se fait sans difficulté par chacun des processeurs en parallèle. Par contre, pour les points sur les fontières des sous-domaines le calcul demande une information non-locale, détenue par les processeurs voisins. Ces derniers ont d'ailleurs eux aussi besoin de l'information contenue chez leur propres voisins. Il faut donc que chaque processeur envoie aux autres les valeurs qui leur sont utiles et attendent de ces derniers la réciproque. La figure 7.3 montre comment les colonnes fontières de chaque sous-domaine sont envoyées par un processeur à son voisin. Une fois cet échange d'information terminé, les points frontières peuvent être calculés et on peut passer aux itérations suivantes. Le pseudo-code parallèle réalisant une telle itération de calcul est par exemple :

```
// compute internal nodes
 for(i=0,i<n,i++){
    for(j=1,j<m/p-1,j++){
        s_new(i,j)=f(s(i-1,j),s(i+1,j),s(i,j-1),s(i,j+1))
        }
     }

// exchange boundary nodes
 send(array=s(:,0),size=n,dest=id-1)
 send(array=s(:,m/p-1),size=n,dest=id+1)
 receive(array=left_column,size=n,source=id-1)
```

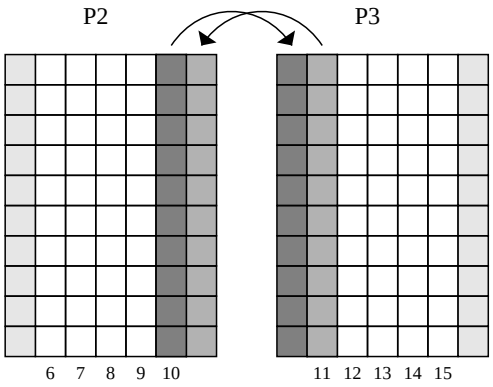


FIGURE 7.3 – *Illustration de la communication entre processeurs voisins afin de réaliser le calcul demandé par la relation 7.1. Pour des raisons d'efficacité de la communication interprocesseur, il est préférable d'envoyer les colonnes frontières en une seul fois et de les recevoir dans des colonnes que l'on adjoint aux sous-domaines de calcul.*

```
 receive(array=right_column,size=n,source=id+1)

// compute boundary nodes
 J=m/p-1
 for(i=0,i<n,i++){
        s_new(i,0)=f(s(i-1,0),s(i+1,0),left_column(i),s(i,1))
        s_new(i,J)=f(s(i-1,J),s(i+1,J),s(i,J-1),right_column(i))
        }
     }
```

Dans cet exemple on a implicitement supposé que `i`$\pm$`1` est correctement calculé modulo n, de même que `id`$\pm$`1`.

7.3 Décomposition en tranche, en morceaux et cyclique

Dans le modèle à échange de message, la répartition des données sur les processeurs est laissée au soin du programmeur. Ce paragraphe illustre les partitionnements courrants lorsque le domaine de calcul est régulier.

Nous considérons ici le cas de 16 valeurs a_{ij}, $i, j = 1, ..., 4$, à répartir sur un réseau carré de 2×2 processeurs. Le découpage ou répartition en tranche discuté ci-dessus est illustré dans la figure 7.4.

Mais on peut aussi choisir un découpage par **morceau** ou **blocs**. Dans notre

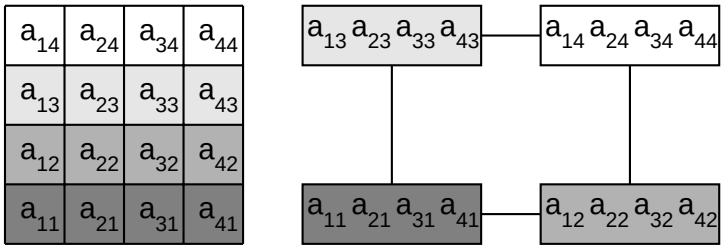


FIGURE 7.4 – *Répartition en tranches (ici en lignes) de la structure de données de gauche sur quatre processeurs (partie droite de l'image) interconnectés selon une grille* 2×2.

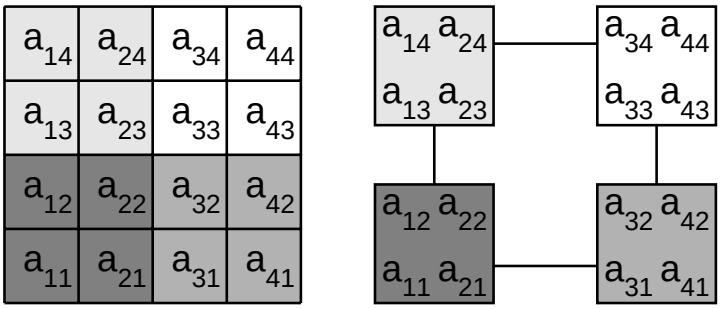


FIGURE 7.5 – *Répartition en morceaux (blocs) de la structure de données de gauche sur quatre processeurs (à droite) interconnectés selon une grille* 2×2.

exemple, on divisera le tableau original par blocs 2×2 comme indiqué sur la figure 7.5.

On peut aussi envisager une distribution des données dite **cyclique** ou **modulo**, comme illustré sur la figure 7.6.

On remarque que cette répartition fait perdre la contiguité des sous-domaines, ce qui peut générer des communications supplémentaires, par exemple dans la situation d'échange avec les voisins nord, sud, est et ouest discutée au paragraphe 7.2. Ceci est illustré sur la figure 7.7 où l'on voit un partitionnement cyclique en bande avec 4 processeurs. Le processeur P_i reçoit les colonnes x telles que $x \bmod 4 = i$. On voit aussi dans cet exemple que, dans le cas d'un schéma de communication comme celui de la résolution de l'équation de Laplace, chaque PE doit communiquer l'ensemble de ses données à ses voisins, ce qui est très peu efficace quand on se souvient que l'overhead de la parallélisation va comme le rapport du volume de données communiquées sur le volume des données modifiées.

Par contre, un avantage de la distribution cyclique est qu'elle permet un bon équilibrage de charge dans certaines applications (par exemple le calcul de l'ensemble de Mandelbrot) où la quantité de calcul varie d'un point à l'autre de la structure de données. C'est donc un paritionnement qui peut être très intéressant

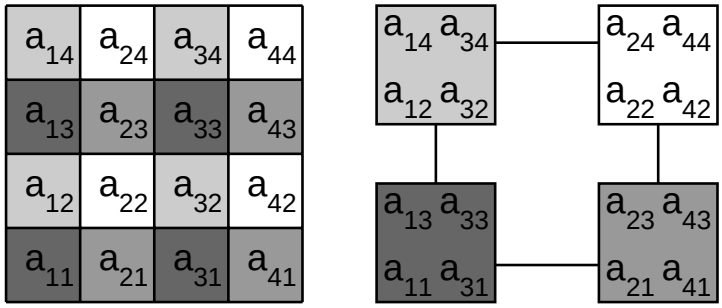


FIGURE 7.6 – *Répartition cyclique (ou modulo) de la structure de données de gauche sur quatre processeurs (à droite) interconnectés selon une grille* 2×2.

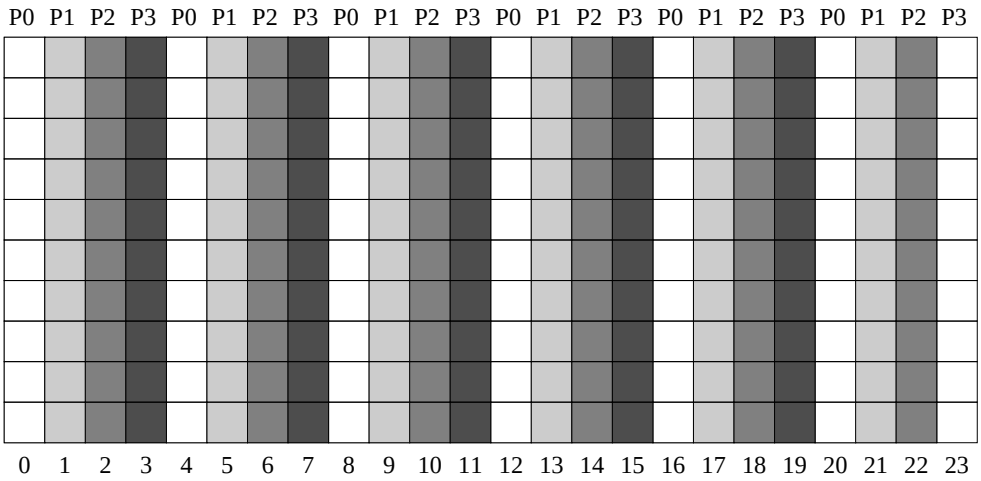


FIGURE 7.7 – Paritionnement cyclic (ou modulo) avec 4 PE, chacun recevant une colonne sur 4. On voit que les frontières cumulées de chaque processeur est très grande ce qui rend ce partitonnement peu adapté à des situations où les processeurs communiquent régulièrement.

pour l'équilibrage de charge dynamique, ainsi que discuté dans la section 6.9.

En fonction des communications souhaitées, la manière de répartir les données sur l'ensemble des processeurs a des conséquences sur l'efficacité du calcul. Il faut trouver un schéma optimum qui tienne compte du réseau à disposition et des communications qui seront requises pour résoudre le problème considéré. Cette tâche se révèle parfois difficile, comme dans le cas des transformées de Fourier rapides où l'interaction entre les données est compliquée. Le stockage de matrices creuses (contenant une majorité d'éléments nuls) est un autre exemple où la distribution des données s'avère difficile. C'est en général le cas de tous les problèmes qui utilisent des structures de données irrégulières.

Pour terminer ce paragraphe, il est important de remarquer que dans les architectures modernes (par exemple les réseaux dynamiques multiétages), la topologie des liens entre processeurs est de plus en plus transparente à l'utilisateur. En fait, du point de vue du temps d'accès à une donnée, ce qui est déterminant c'est de savoir si la donnée est *locale* (c'est à dire dans le processeur même) ou distante (*"remote"*) (c'est à dire dans la mémoire d'un autre processeur). Ceci est principalement dû au fait que le temps de propagation d'un long message est proportionnel à sa longueur plutôt qu'au nombre de noeuds qu'il a à traverser. On pourrait alors penser que la distribution des données est secondaire puisque, en première approximation, chaque processeur est voisin de tous les autres. Cependant, si toutes les données sont trop éparpillées à travers les processeurs, un trafic **non local** important en résultera et pourra produire des congestions du réseau néfastes aux performances de l'application.

7.4 Les primitives SEND et RECEIVE

La forme générale de l'instruction SEND est typiquement

```
SEND(message_out, msg_size, destination, type)
```

où `message_out` est la position mémoire qui contient l'ensemble des données à envoyer, `msg_size` est la taille en octets du message, `destination` est l'adresse du processeur qui recevra le message et `type` est un identificateur permettant de distinguer des messages de types différents (par exemple, on peut alternativement envoyer des entiers ou des flottants et, à la réception, il faut pouvoir retrouver la nature des données reçues).

La structure de la primitive RECEIVE est similaire :

```
RECEIVE(message_in, msg_size, source, type)
```

où `message_in` est cette fois la position mémoire où le message sera stocké après réception et `source` est l'adresse du processeur ayant expédié le message.

En pratique, après un SEND, le système d'exploitation copie le message dans une zone mémoire réservée, le **buffer de communication**, et y inclut une entête

avec des informations de routage. Il déclenche ensuite le départ du message qui, à l'arrivée, est stocké dans le buffer de communication du processeur destinataire. Une variable système est activée pour indiquer l'arrivée du message. La primitive RECEIVE prend le contenu du buffer de communication et le copie dans la zone mémoire choisie par l'utilisateur.

Il n'est pas nécessaire de connaître la source d'un message pour le recevoir (parfois, on ne sait pas d'avance de qui on va recevoir quelque chose). Il est possible d'utiliser des *wildcards* qui permettent que n'importe quel message présent soit reçu, quelle que soit son origine. De même, le type d'un message peut aussi être donné comme wildcard. Lorsque le message est reçu, les variables `source` et `type` sont en général mises à jour par le système et contiennent alors la source réelle du message lu et son type.

Il existe aussi des primitives qui permettent de tester l'arrivée d'un message et d'en connaître ses paramètres, sans pour autant le lire. Cela est très utiles lors d'applications avec des échanges de données non structurées et peu prévisibles.

Finalement, on distingue encore les communications bloquantes et non bloquantes. Les communications sont dites *bloquantes* si la tâche est interrompue jusqu'à ce que le transfert soit terminé. Elles sont *non-bloquantes* si le message est transmis dans le réseau et la tâche poursuivie. Ainsi, un *blocking receive* est une instruction qui arrête l'exécution jusqu'à la réception du message attendu. Le *nonblocking receive* permet de ne pas perdre de temps et d'accomplir une autre tâche en attendant l'arrivée du message. Il est utile lorsque plusieurs messages venant de processeurs différents sont attendus, mais dans un ordre imprévisible. Lorsqu'on utilise une telle primitive, il est nécessaire de pouvoir tester ultérieurement, avec des primitives adéquates, le succès et l'accomplissement de la communication.

Selon les systèmes, le *blocking send* interrompt l'exécution en cours jusqu'à ce que le buffer de communication soit rempli. Cela garantit que la variable dont le contenu est envoyé n'est pas modifiée entre le moment où la primitive SEND est appelée et le moment où le buffer de communication reçoit la valeur.

Les communications non bloquantes sont asynchrones. Par contre, une primitive telle que

```
bsendrecv(msg_out,msg_in,msg_size,source,destination)
```

est une communication bloquante synchrone où le send et le receive sont condensés en une seule instruction. Ce type de SEND-RECEIVE couplé est très utile dans les problèmes où les processeurs échangent des données de façon régulière et répétitive (par exemple, chaque fois qu'un processeur envoie un message à son voisin de droite, il attend aussi le message correspondant envoyé par son voisin de gauche). C'est aussi une opération qui optimise les temps d'exécution et la place mémoire dans les buffers de communication.

Les communications asynchrones (et en particulier le RECEIVE non-bloquant) sont aussi très utiles dans la mesure où elles permettent de faire se chevaucher

communications et calculs. On peut ainsi (si le problème le permet) profiter de faire certains calculs pendant que les données qu'on attend transitent encore dans le réseau.

Messages actifs : Les messages actifs (*active messages*) sont une autre catégorie de messages. Comme leur nom l'indique, ils ne véhiculent pas seulement des données passives mais peuvent aussi déclencher, dans le processeur destinataire, un appel à une fonction et lui en fournir les paramètres adéquats.

Le résultat d'un message actif est d'interrompre le processeur destinataire immédiatement dès réception, afin qu'il exécute la fonction dont il a reçu l'adresse. Lorsque cette dernière est exécutée, le processeur retourne à la tâche interrompue.

7.5 Implémentation

La mise en oeuvre d'un protocole d'échange de message peut différer notablement d'un constructeur à l'autre, ou d'une bibliothèque de primitives à l'autre. Il s'en suit que certaines opérations sont possibles dans un cas mais pas dans l'autre. Par exemple, certaines bibliothèques d'échange de messages permettent de superposer des communications collectives avec des communications point à point alors que d'autres pas. De même certains systèmes ne peuvent recevoir plusieurs messages venant d'un même expéditeur que dans l'ordre dans lequel ils ont été envoyés. La taille des buffers de communication peut aussi varier selon les implémentations et certaines stratégies sont parfois nécessaires pour garantir un transfert important de données.

Dans ce paragraphe, nous présentons le modèle de mise en oeuvre utilisé sur la Connection Machine CM-5. Les principes de bases sont génériques mais pas certains détails. La figure 7.8 illustre la façon dont les différents composants sont organisés. L'accès des processeurs au réseau se fait à travers des chips spécialisés, appelés le *Network Interface* (NI) dans la CM-5 (et adaptateurs sur l'IBM SP2). Leur but est d'offrir aux processeurs une vue unifiée du réseau et, au réseau, une vue unifiée des processeurs. L'avantage est de pouvoir découpler ces deux côtés complémentaires d'une architecture parallèle. Indépendamment, processeurs ou réseaux peuvent évoluer et être améliorés selon les nécessités spécifiques ou les progrès technologiques du moment.

Grâce aux NI, les communications se font sans recours au système d'exploitation et sont directement contrôlées par le programme utilisateur. Le principe de fonctionnement des NI est le suivant : les processeurs voient l'interface comme une collection d'adresses mémoire chacune représentant les destinations possibles du réseau. Les processeurs peuvent lire ou écrire dans ces positions mémoire (buffer de communication), ce qui revient, respectivement, à recevoir ou envoyer un message à travers le réseau. L'envoi proprement dit des messages est généré automatiquement par le NI après écriture, par un processeur, d'un message dans une

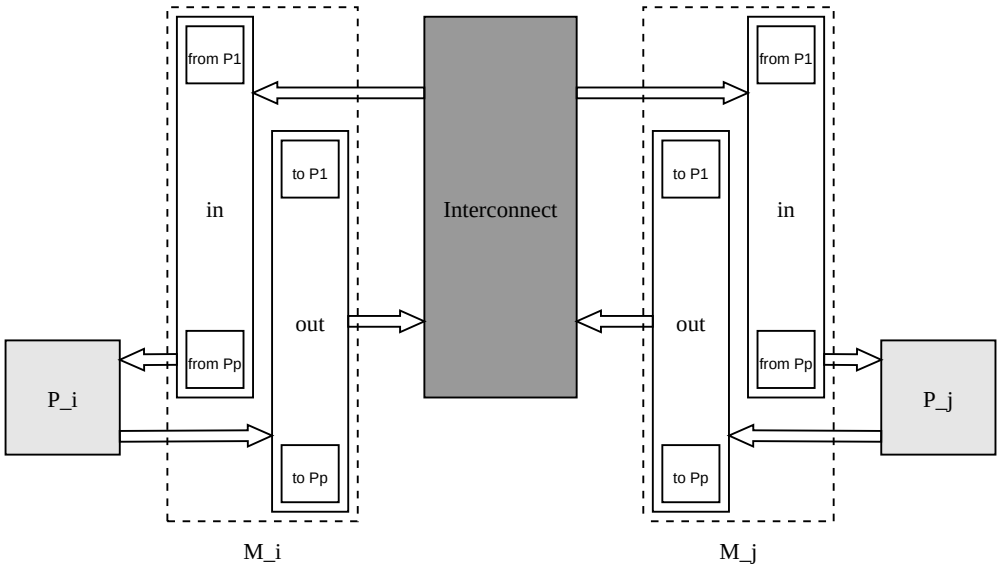


FIGURE 7.8 – *Implémentation schématique du processus d'échange de message entre deux processeurs P_i et P_j qui écrivent et lisent dans des buffers de communications IN ou OUT selon qu'ils envoient ou reçoivent un message. Le réseau d'interconnexion se charge de faire transiter les messages.*

position mémoire correspondante. De cette manière, l'accès au réseau est transparent tout en ne nécessitant pas d'appels système. Les communications sont générées par le code utilisateur, ce qui permet un parallélisme à grain fin avec peu d'overhead. Le NI permet en outre de protéger efficacement les utilisateurs les uns des autres car les adresses correspondant à des processeurs d'une autre partition peuvent être interdites par le système d'exploitation.

En pratique, le NI est constitué de piles FIFO qui sont entrantes (in) ou sortantes (out) selon qu'elles génèrent des messages dans le réseau ou qu'elles en reçoivent. Il existe une telle paire in/out de ces piles pour chaque fonctionnalité du réseau (une pile pour les communications point à point, une pour les opérations de synchronisation, une pour les opérations combinant des données (SCAN), etc). Ces piles sont gérées par le NI selon un jeu de priorités établies. Schématiquement, le mécanisme est le suivant :

— processeur → out-FIFO → réseau ;
— réseau → in-FIFO → processeur.

Ainsi, la vue que l'utilisateur a du réseau est indépendante de sa topologie réelle. La topologie physique du réseau n'est d'ailleurs pas garantie puisque, en cas de panne telles ou telles parties du réseau peuvent être désactivées et se trouver absentes. La topologie physique peut aussi, selon les machines, changer selon la partition qui est allouée à un utilisateur.

Dans la CM-5, un mode timesharing est possible. Pour se faire, le NI garde les données qu'il envoie jusqu'à la fin du transit (un signal de fin de routage est envoyé par le réseau de contrôle). En cas d'interruption de la tranche de temps d'un utilisateur, ces données font partie intégrante de l'état associé à chaque utilisateur et sont sauvegardées pour que le transfert puisse recommencer plus tard.

Contrôle du transfert : cela désigne la manière dont les échanges se font et les techniques permettant de confirmer la bonne réception des messages. En règle générale, les protocoles utilisés sont fiables et on peut compter sur le fait qu'un message envoyé arrive toujours à destination. Cela est assuré par des mécanismes **d'accusé de réception**, transparents à l'utilisateur, qui, si le message n'est pas arrivé, provoque la répétition de l'envoi.

En général, les messages sont asynchrones, ils sont envoyés dès que l'instruction `SEND` (ou son équivalent) est exécutée et que le réseau permet le transfert. Il n'y a pas de concertation préalable avec le destinataire chez lequel les messages sont stockés dans des buffers jusqu'à ce qu'ils soient lus. Si les buffers sont pleins, le message est renvoyé.

On peut aussi envisager des protocoles d'échange par **rendez-vous** où l'expéditeur et le destinataire se mettent d'accord sur un transfert synchrone d'un message. Cependant, cette technique est plus lourde à mettre en oeuvre (nécessitant plusieurs aller-retours de messages pour établir la communication) et moins

adaptée à une architecture naturellement asynchrone.

7.6 MPI et PVM

Parmi les nombreuses bibliothèques d'échange de messages, deux d'entre elles s'imposent actuellement comme des standards. Il s'agit de MPI (Message Passing Interface) et PVM (Parallel Virtual Machine). Toutes deux offrent des primitives similaires à celles décrites ci-dessus, avec bien sûr certaines variations de fonctionnalité, de paramètres d'appel et de noms.

Toutefois, certaines différences méritent d'être soulignées, bien que les avantages et désavantages d'un système par rapport à l'autre soient actuellement en train de se combler.

PVM a été initialement développé pour un ensemble hétérogène de machines UNIX (par exemple des stations de travail de différents types connectées avec des super-ordinateurs, etc) alors que MPI a d'abord été conçu pour une machine MPP homogène (par exemple une SP2 avec plusieurs processeurs identiques). MPI a tout de suite été pensé dans l'esprit d'applications parallèles fortement couplées et offre de nombreuses primitives de communications collectives qui manquaient dans les premières versions de PVM.

PVM est un peu plus lourd à utiliser que MPI, en raison de l'inhomogénéité possible des plate-formes. En effet, PVM doit d'abord coder les messages selon un format propre avant de les envoyer car la représentation interne des nombres n'est pas forcément la même sur toutes les architectures. De plus, différents protocoles sont utilisés selon les plate-formes (TCP/IP pour les stations de travail ou des protocoles natifs dans les MPP).

Par contre, PVM offre des possibilités de gestion de processus que MPI n'a pas. Par exemple, PVM permet de créer et arrêter un processus avec la primitive `pvm_spawn`. L'ensemble d'une application PVM est gérée par des démons qui tournent sur chacune des machines hôtes. PVM peut varier dynamiquement le nombre de machines hôtes utilisées alors que MPI travaille avec un nombre fixe de processeurs.

Notons encore que dans la version de MPI-2.0, des primitives d'écriture et de lecture directes dans la mémoire des autres processeurs seront disponibles. Cette fonctionnalité NUMA s'appelle les communications "one-sided" puisqu'elle n'implique pas un comportement symétrique des deux processeurs impliqués. De plus, MPI-2.0 prévoit aussi des primitives d'entrées-sorties parallèles.

7.7 Avantage et désavantage du modèle

Au contraire des architectures à mémoire partagée qui imposent des contraintes difficiles à réaliser, les architectures à mémoires distribuées sont plus simples de

conception mais impliquent un plus grand effort de la part de l'utilisateur pour les programmer. En particulier, l'espace des adresses n'est pas uniforme et le modèle de programmation par échange de messages est plus contraignant et moins naturel que le modèle à mémoire partagée. Il est souvent "do-it-yourself" pour le programmeur qui doit gérer explicitement les échanges de données, la division en sous-tâches et le placement des données sur les processeurs (**le partitionnement** ou la **décomposition du domaine**).

Dans le modèle de programmation par échange de messages, l'utilisateur est directement responsable de la gestion des communications entre processeurs à l'intérieur de son programme. Notamment, il doit veiller que les données envoyées soient effectivement lues quelque part, au risque de saturer les buffers de communications. De même, un message qui est attendu par un processeur mais jamais envoyé par un autre cause un deadlock.

Les données sont purement locales et aucune ressource n'est partagée. Une donnée qui doit être utilisée par plusieurs processeurs doit être soit dupliquée localement (ce qui est avantageux si sa valeur ne change pas en cours d'exécution), soit recopiée explicitement d'un processeur qui en serait le propriétaire.

Dans une architecture à mémoire distribuée, il est difficile de faire migrer une tâche (la faire exécuter par un autre processeur) car cela nécessite de recopier toute la mémoire d'un processeur à l'autre. En revanche, une telle architecture est scalable et on peut ajouter des processeurs sans diminuer les capacités d'accès mémoire. Cela est dû au fait que, pour les applications scalables (dominées par des communications locales, par exemple), lorsqu'un processeur est ajouté avec sa propre mémoire, les échanges de données générées par ce nouveau noeud vont essentiellement utiliser les nouveaux liens de communication ajoutés au système. Cela découle du principe de localité qui requière que les données les plus utilisées par un processeur soient locales à ce processeur.

Chapitre 8

Modèle de Programmation à Mémoire Partagée

Dans un modèle de programmation à mémoire partagée, tous les processeurs ont un accès uniforme à la mémoire et peuvent lire ou écrire sur n'importe quelle variable commune ou globale. C'est ainsi que les processeurs communiquent, un peu comme si un groupe de personnes collaborait en modifiant des informations sur un tableau noir visible de tous.

Le fait d'avoir une mémoire unique allège de façon significative la programmation car il n'est pas nécessaire de se soucier de l'emplacement d'une donnée ni de gérer son mouvement entre des processeurs différents.

Par contre, l'accès simultané par plusieurs processeurs à des ressources partagées ne peut se faire sans règles strictes, sous peine de générer des incohérences. On cherche en particulier à garantir le principe de **cohérence séquentielle** qui signifie que, quelle que soit la manière dont l'exécution parallèle se fait, le résultat du programme est identique à celui d'une exécution séquentielle suivant l'ordre des instructions du programme. La coordination et synchronisation des processeurs est donc un point crucial de l'approche à mémoire partagée et nous les discuterons en détail dans le paragraphe 8.2.

8.1 Multithreading

D'un point de vue pratique la mise en oeuvre du parallélisme dans une architecture à mémoire partagée se fait avec le concept de **microtasking** plutôt que celui de **macrotasking**. Le macrotasking correspond à du parallélisme à gros grain et utilise des primitives comparables aux primitives de UNIX permettant de créer de nouveaux processus et de les terminer (spawn, fork et join).

Le microtasking est plus adapté à un parallélisme à grain fin, comme par exemple à celui qui peut exister au niveau des variables d'une boucle. Le microtasking utilise le concept de **multithreading** qui est une notion de base dans un

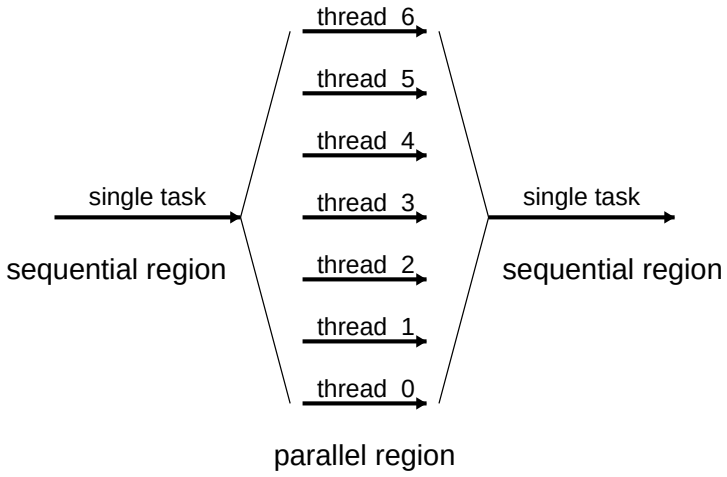


FIGURE 8.1 – *Exemple de multithreading d'une tâche dans une région parallèle.*

modèle de programmation à mémoire partagée. Il permet la création de plusieurs flots de contrôle séparés qui pourront être exécutés sur des processeurs différents.

La création de processus enfants comme le permet le système d'exploitation UNIX avec la primitive *fork* n'est pas adaptée au parallélisme à grain fin en raison de la lourdeur d'un tel processus. La structure de données nécessaire à la description d'un processus UNIX standard est considérable car elle contient toutes les informations relatives à l'état de la machine (mémoire virtuelle, privilèges d'accès, état de fichiers, etc).

Toute cette structure doit être créé ou détruite chaque fois qu'un processus apparaît ou disparaît. Cela implique un temps d'overhead important, inacceptable en parallélisme où un flot de contrôle doit pouvoir se créer rapidement. De plus, l'état complet d'un processus est souvent inutile dans le cas d'une exécution parallèle.

Pour cette raison, on introduit le concept de **threads** qui sont des processus légers qui *partagent* l'état du processus parent (souvent noté **tâche** dans ce contexte). Les threads ont été d'abord mises en oeuvre dans le système d'exploitation MACH puis la terminologie est restée.

Une tâche peut ainsi se subdiviser en plusieurs threads, chacun assigné à un autre processeur (en général, il n'y a qu'un thread par processeur). Un thread est beaucoup plus éphémère qu'une tâche. Il peut être créé et détruit avec très peu d'overhead. Le modèle d'exécution par thread est illustré sur la figure 8.1. On parle de **région séquentielle** là où il n'y a qu'une tâche et de **région parallèle** lorsqu'il y a plusieurs threads concurrents. Souvent le système d'exploitation se charge de la création des threads. Leur nombre n'est pas fixe d'une exécution à l'autre, ce qui permet d'exploiter le parallélisme à disposition et de donner lieu à des programmes portables : le même code peut s'exécuter indifféremment sur $1, 2, ..., p$ processeurs. Bien que les différents threads peuvent prendre un temps

plus ou moins long, la fin de la section parallèle implique que tous aient terminé avant de débuter la section séquentielle qui suit. Cela constitue donc un point de synchronisation.

8.1.1 OpenMP

Il est possible de créer des threads parallèles directement avec les primitives offertes par POSIX à partir d'un programme écrit dans un langage quelqconque. Cependant, les machines à mémoire partagée actuelles (par exemple les SMP) offrent une approche un peu plus simple pour le programmeur.

En utilisant le standard OpenMP on peut, à partir de directives de compilation ajoutées comme commentaires à des programmes écrits en C, C++ ou Fortran, spécifier la création de threads parallèles pour la partie du code concernée par la directive. Par exemple, avec la syntaxe Fortan, on aurait

```
!$omp parallel
  print *, omp_get_thread_num()
!$omp end parallel
```

Dans cet exemple, les lignes commentaires, qui commencent par `!$omp`, sont interprêtés par OpenMP qui, en l'occurence, crée une section parallèle où chaque thread ne fait rien d'autre qu'écrire son identificateur. En C ou C++, les directives s'appellent des *pragma* et la syntaxe est

```
#pragma omp parallel
```

Ces directives sont typiquement placées avant une boucle pour indiquer la possibilité de répartir les itérations sur plusieurs processeurs. Pour indiquer une telle situation en C++, on aurait par exemple

```
#pragma omp parallel
  for(i=0,i<n,i++){
    z[i]=a*x[i] + b
  }
#pragma end omp parallel
```

ce qui, dans ce cas, se parallélise très bien car chaque itération est indépendante des autres. On voit par ces exemples que la programmation multithread, ou plus spécifiquement OpenMP est du type SPMD : il n'y a qu'un seul programme qui décrit l'ensemble des threads.

Cepandant, pour bénéficier des performances que plusieurs coeurs peuvent offrir, il faut que chaque thread ait suffisament de travail entre deux accès mémoire. Sinon, les threads sont en compétition pour l'accès aux données et l'exécution parallèle peut être plus lente que l'exécution séquentielle.

Le code ci-dessous, qui ne fait pas grand chose d'utile, illustre néanmoins l'importance d'avoir une grande intensité arithmétique. Pour chaque donnée `v[i]`

on calcule un résultat `w[i]` avec une itération arbitraire nécessitant 1 million d'étapes pour chacune des $N = 10'000$ éléments du vecteur.

L'instruction `omp_set_num_threads(n)` permet de choisir le nombre de threads. Ici on pris $n = 4$ en vue d'une exécution sur les 4 coeurs d'un laptop de 2020. Avec $n = 1$, on a une exécution séquentielle. On observe alors un speedup de 3.3 sur 4 coeurs. Par contre, si on réduit la taille de la boucle sur `j`, le speedup chute jusqu'à des valeurs qui peuvent devenir inférieure à 1.

Le temps d'exécution est mesuré grâce à la fonction `omp_get_wtime()`. Certaines autres fonctions de mesure de temps peuvent donner des résultats éronés. Ce code utilise les threads posix et nécesssite la bibliothèque et les «header files» appropriés.

```
//  g++  iter.cc  -fopenmp -lpthread

#include <iostream>
#include <pthread.h>
#include <omp.h>
#include <vector>
#include <numeric>  // for acumulate

using namespace std;

int main(){
  int N=1e4;
  double s=0;
  double t0,t1;
  auto v=vector<double>(N);
  auto w=vector<double>(N);

#pragma omp parallel for
  for(int i=0;i<N;i++)v[i]=i;

  omp_set_num_threads(4);  // change it to 1 for a seq execution
  // 40 sec in sequential   12 sec with 4 threads

  t0=omp_get_wtime();

#pragma omp parallel for
  for(int i=0; i<N; i++){
    for(int j=1;j<=1000000;j++)w[i]+=v[i]/(j*j);}

  t1=omp_get_wtime();
  s=accumulate(w.begin(), w.end(), 0);
  cout<<"s="<<s<<"  execution time="<<t1-t0<<" [s]"<<endl;
}
```

Des directives semblable au `parallel for` permettent de spécifier plus finement comment le partage de la boucle devra être réalisé (par bloc d'indices consécutifs ou plutôt de façon modulo). Nous renvoyons le lecteur à la documentation OpenMp pour plus de détails.

Finalement, soulignons que bien que l'espace mémoire soit accessible par tous les processeurs, certaines directives OpenMP (par exemple `!$omp private ma_variable`) permettent de définir des variables locales dont l'accès n'est possible que par un seul processeur.

Finalement notons que la fonction

```
omp_set_num_threads(p)
```

qui permet d'imposer le nombre de threads souhaité ne garantit pas forcément une distribution équilibrée des threads sur le processeurs disponibles. S'il n'y a pas assez de processeurs, ou bien si certains sont trop chargés aux yeux du système d'exploitation, plusieurs threads pourrant être lancés sur le même processeur.

8.2 Coordination et Synchronisation des processeurs

Plusieurs processeurs qui travaillent ensemble dans le but de résoudre un même problème doivent **coordonner** leurs actions pour garantir l'exécution correcte d'un programme. La *coopération* entre les processeurs se double obligatoirement d'une certaine forme de *compétition* qui, si elle n'est pas strictement contrôlée, conduit à une exécution erronée. Par exemple, plusieurs processeurs peuvent vouloir écrire en même temps des valeurs différentes en une même position mémoire ou entrer en compétition pour l'accès à des ressources communes.

Le temps d'exécution d'un processus peut fluctuer de manière imprévisible d'un processeur à l'autre. Par conséquent, l'ordre dans lequel les opérations prennent place n'est pas répétitif d'une fois à l'autre, ce qui peut conduire à un comportement non déterministe si un schéma de synchronisation n'est pas clairement défini. Si par exemple le processeur P_1 contient l'instruction `A=A+B` et le processeur P_2 l'instruction `B=B+A`, la valeur finale de `A` et `B` dépend crucialement de l'ordre dans lequel ces deux instructions sont exécutées. En supposant qu'initialement `A=3` et `B=5`, le résultat final est `A=8`, `B=13` si P_1 passe avant P_2, et `A=11`, `B=8` dans le cas contraire.

Une situation, comme celle expliquée ci-dessus, où un résultat dépend du premier processeur qui atteint une instruction donnée s'appelle une *race condition* (condition de course).

La mise au point d'un programme et les raisonnements qui s'y rapportent peuvent ainsi s'avérer trompeusement faciles. Le contrôle de la coordination et de la compétition entre les processeurs se résume au contrôle de l'accès à la mémoire, et au contrôle de l'ordonnancement (synchronisation) des différentes tâches.

8.2.1 Contrôle d'accès

Un exemple simple permet d'illustrer le problème du contrôle de l'accès à une variable partagée par tous les processeurs. Supposons que l'on désire calculer la somme d'un grand nombre de données et que chaque processeur se charge

d'évaluer une partie de cette somme. Le résultat final est obtenu en récoltant les sommes locales contenues dans chaque processeur et en les additionnant dans la variable `Global_sum`, par exemple. Chaque processeur aura donc typiquement à exécuter l'instruction

```
Global_sum=Global_sum+local_sum
```

où `local_sum` a une valeur différente en chaque processeur. Comme tous les processeurs travaillent à leur propre rythme, il est impossible de prévoir comment l'accès à `Global_sum` va se faire. Il se pourrait très bien que le processeur P_1 lise la valeur de `Global_sum` et, avant d'avoir eu le temps de la modifier et de la récrire en mémoire, un autre processeur, disons P_2 lise de même `Global_sum`. Lorsque la valeur de `Global_sum` sera récrite en mémoire par P_2, mettons juste après P_1, la contribution de P_1 sera oubliée et le résultat de la somme totale incorrect. L'instruction

```
Global_sum=Global_sum+local_sum
```

doit donc être exécutée de manière séquentielle, un processeur après l'autre, ou, autrement dit, il ne faut permettre l'accès à certaines variables qu'à un processeur à la fois. C'est ce qu'on appelle l'**exclusion mutuelle**.

Un autre exemple classique illustrant l'importance d'une exclusion mutuelle est donné par l'accès simultané par plusieurs personnes à un même compte en banque. On peut imaginer que des titulaires communs se présentent au même moment à différents guichets automatiques et essayent tous de retirer la totalité du montant disponible. Si chaque guichet automatique ne bloque pas l'accès à toutes les personnes présentes, sauf une, chacun pourra effectuer ce retrait et tromper la machine en prenant en tout plus d'argent qu'il n'y en a sur le compte.

Pour assurer cette exclusion mutuelle, des mécanismes de contrôle d'accès sont disponibles dans les machines à mémoire partagée. Ils s'expriment typiquement par les primitives `LOCK` et `UNLOCK`, avec lesquelles l'exemple précédent s'écrirait

```
LOCK(Global_sum)
Global_sum=Global_sum+local_sum
UNLOCK(Global_sum)
```

Le premier processeur qui rencontre le `LOCK` pénètre dans la section de code en bloquant l'accès à `Global_sum`. Les autres processeurs qui rencontrent ensuite le `LOCK` doivent attendre que la variable soit débloquée par le `UNLOCK` avant de pouvoir continuer et exécuter l'instruction de somme.

Ce mécanisme présente en principe un risque d'*interblocage* ou *deadlock* qui apparaît lorsque deux processeurs attendent mutuellement que l'autre débloque la situation. Par exemple, P_1 bloque la variable `A` et essaye de lire la variable `B` déjà bloquée par P_2. Si P_2 a aussi besoin de `A` avant de débloquer `B`, le système part dans une boucle infinie dans l'attente d'un événement qui n'aura jamais lieu.

Pour éviter ce type de complication et dans le cas d'une programmation SPMD, il est préférable de définir **sections critiques** pour protéger les parties de code qui ne doivent être effectuées que par un seul processeur à la fois. Une section critique est donc, par sa nature même, une entité séquentielle, qui empêche le parallélisme de prendre place. Si on ne peut pas éviter les sections critiques dans certaines applications parallèles, il faut en revanche essayer de réduire au maximum l'attente qu'elles provoquent dans les autres processeurs.

Avec OpenMp, on déclare une section critique de la façon suivante :

```
!$omp critical
  list of critical instructions
!$omp end critical
```

8.2.2 Contrôle de séquence

Le but du contrôle de séquence est de définir l'ordonnancement des différentes tâches, afin de garantir la validité d'un programme s'exécutant sur un multiprocesseur. Comme on l'a déjà mentionné, on ne peut pas compter sur les vitesses d'exécution relatives d'un processeur à l'autre pour prédire l'ordre dans lequel les instructions d'un programme auront lieu.

Le besoin de synchronisation est particulièrement évident dans le cas de calculs itératifs sur un grand nombre de données. Si les données sont réparties sur plusieurs processeurs et que le calcul implique des communications entre les processeurs, il est indispensable de synchroniser les actions de chacun : avant de passer à l'itération suivante, il faut être sûr que la précédente est totalement achevée et que toutes les données sont correctement mises à jour par les processeurs qui en ont la charge.

En pratique, on réalise ce but en incluant dans le programme des **barrières** de synchronisation. Une barrière de synchronisation est un point du code qui ne peut être franchi avant que tous les processeurs l'aient atteint.

Par exemple, si on a le code multithread suivant qui calcule un tableau `b` à partir d'un tableaux `a` de même taille `n`, selon le code

```
for i=1 to n:  b[i]=0.5*(a[i-1] + a[i+1])
```

Puis le tableau `a` est mis à jours avec les nouvelles valeurs calculée dans `b`

```
for i=1 to n:  a[i]=b[i]
```

et le tout est itéré plusieurs fois. La figure 8.2 illustre la nécessité de mettre une barrière de synchronisation après chacune des boucles, afin d'éviter que certains threads utilisent des valeurs déjà modifiée par d'autre threads.

Le code ci-dessous propose une implémentation des barrières dans le cas d'un code Fortran utilisant openMP.

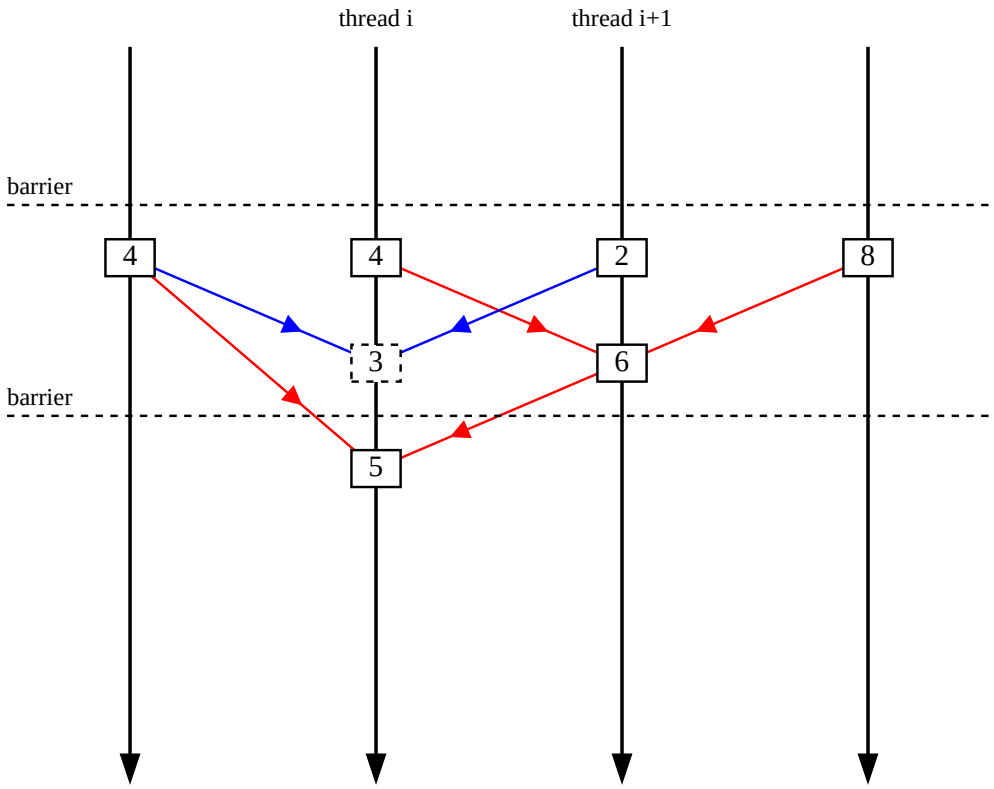


FIGURE 8.2 – Illustration d'un problème d'incohérence de résultat qui peut arriver en l'absence d'une barrière de synchronisation. Dans cet exemple le thread i est en retard sur le thread $i+1$ et utilise la valeur de $a[i+1]$ de l'itération suivante pour calculer $b[i]$.

```
!$omp parallel private me, p, i, i0, i1
  omp_set_num_thread(p)    ! p est le nombre the thread choisi
  me=omp_get_thread_num()
! compute the local limits of the loop
  i0=(n/p)*me
  i1=(n/p)*(me+1)

  do i=i0, i1
    b(i)=(1/2)*(a(i-1) + a(i+1) )
  enddo

!$omp barrier

  do i=i0, i1
    a(i)=b(i))
  enddo

!$omp end parallel
```

La directive `!$omp barrier` indique que la boucle sur `i` doit se terminer pour tous les processeurs avant de passer à la boucle suivante suivante. Sans quoi, certains processeurs pourraient déjà avoir modifié certaines valeurs de `a` avant que celles-ci ne soit utilisée par d'autres.

Une façon simple de construire une barrière de synchronisation est d'utiliser une variable partagée (appellons-la "`barrier`"), dont la valeur initiale est le nombre de tâches à synchroniser. Chaque fois qu'un processeur arrive à la barrière (parce qu'il a terminé), il décrémente de 1 cette variable. Lorsque la valeur atteint zéro, cela veut dire que tous les processeurs ont fini la tâche qui leur était assignée et que la suite du programme peut être exécutée.

A un tel point de synchronisation, les processeurs attendent habituellement que la barrière se lève en contrôlant sans cesse la valeur de la variable `barrier`. Les processeurs les plus rapides attendent les plus lents et il vaut mieux avoir des tâches de durée à peu près égales sur chaque processeur pour ne pas voir se dégrader le bénéfice du parallélisme (load balancing).

Que cela soit à une barrière de synchronisation ou à un `LOCK`, le fait de "boucler" continuellement sur l'instruction qui bloque les processeurs s'appelle **busy-waiting**. C'est la même chose qu'attendre au téléphone que son correspondant soit prêt à répondre. Pendant ce temps, on ne fait rien et on gaspille des cycles CPU.

Remarquons encore qu'un *busy-waiting* mis en oeuvre de manière trop naïve peut conduire à une situation de deadlock si le nombre de tâches est supérieur au nombre de processeurs. Par exemple, dans le cas d'une barrière de synchronisation, tous les processeurs pourraient se bloquer en attendant la fin de la dernière

tâche. Mais celle-ci n'aura jamais lieu car la dernière tâche doit être exécutée par un des processeurs déjà en attente.

8.3 Primitives de Synchronisation

Une grande partie des primitives de synchronisation utilisées dans les multiprocesseurs proviennent des systèmes d'exploitation multitâches développés pour les monoprocesseurs. Bien que la compétition pour des ressources communes soit de même nature, les solutions inspirées par ces techniques ne sont pas toujours adaptées au cas de systèmes comportant beaucoup de processeurs. Le parallélisme inhérent au système d'exploitation séquentiel multitâche est de grande granularité et intervient à un degré beaucoup plus faible que dans le cas d'une machine véritablement parallèle.

La clé des primitives de coordination les plus simples est l'utilisation d'instructions dites **indivisibles** ou **atomique** qui sont des instructions non interruptibles qui ne peuvent être exécutées que par un processeur à la fois. Cela signifie qu'il existe un dispositif hardware qui "sérialise" les appels provenant des différents processeurs pour exécuter ces instructions indivisibles. Dans ce sens là, ces primitives permettent de contrôler et gérer le parallélisme en ayant recours à une technique d'exécution séquentielle et ne sont ainsi pas vraiment satisfaisantes pour des systèmes avec beaucoup de processeurs. Comme on le verra, il existe tout de même une primitive de coordination véritablement parallèle (la primitive `fetch-and-add`), mais qui nécessite un réseau d'interconnexion actif.

8.3.1 La primitive test-and-set

Une des primitives de base est le **test-and-set** qui utilise une variable de blocage ou flag (`S`, dans l'exemple qui suit). La définition de `test-and-set` peut s'exprimer par les instructions suivantes

```
test-and-set(S)
   atomic{ tmp=S; S=0}
   return tmp
```

Les accolades { et } délimitent la partie indivisible de l'instruction `test-and-set`. Le fonctionnement est le suivant. La variable `S` est initialisée à 1. Lorsque le `test-and-set` est exécuté pour la première fois, en une étape indivisible, `S` est copié dans `tmp` et mis à zéro. Le processeur qui exécute le `test-and-set` en premier reçoit en retour la valeur 1, alors que les suivants reçoivent la valeur 0.

La primitive `test-and-set`, par le biais de la variable `S`, réalise la mise en oeuvre de **sémaphores** qui permettent d'assurer une exclusion mutuelle ou un `LOCK`. Lorsque `S` est trouvé à 0, l'accès est bloqué, alors que si `S=1`, le processeur

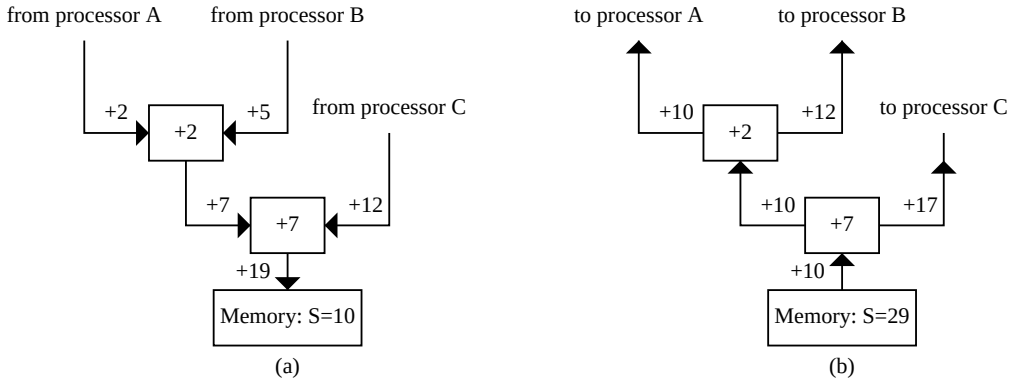


FIGURE 8.3 – *Mise en oeuvre du Fetch-and-Add sur un réseau d'interconnexion dont les switches permettent des opérations arithmétiques et le stockage d'information : (a) Les données provenant des processeurs sont respectivement 2, 5 et 12. Chaque switch conserve la valeur venant de gauche et renvoie vers le bas la somme des valeurs reçues. La valeur de S est incrémentée et passe à 29. (b) L'ancienne valeur de S (10) est renvoyée dans le réseau. Les switches renvoient à gauche la valeur reçue par le bas et à droite la somme de la valeur reçue et la valeur stockée. Le processeur A reçoit ainsi l'ancienne valeur de S.*

peut continuer le programme. A la fin de la section critique (ou région d'exclusion), le processeur "ayant la main" remet `S` à 1 (`UNLOCK`), et les autres peuvent à leur tour exécuter le code réservé.

Parfois, les sémaphores sont accessibles à l'utilisateur sous forme de primitives de haut niveau, généralement notée `P` et `V`, dont l'usage est le suivant :

```
P(S); section critique; V(S)
```

8.3.2 La primitive Fetch-and-add

Les primitives de synchronisation que nous venons de voir sont dites **bloquantes**, car un seul processeur à la fois peut les exécuter. Il peut arriver, si le nombre de processeurs est important, que chaque tâche passe un temps considérable simplement à attendre de pouvoir exécuter la primitive en question ou tester un sémaphore.

Il y a donc un besoin pour une primitive de coordination véritablement parallèle qui puisse être exécutée simultanément par plusieurs processeurs. Il se trouve que beaucoup d'opérations sur des variables communes sont indépendantes de l'ordre dans lequel elles sont exécutées. C'est le cas, notamment, de la somme de plusieurs données ou encore d'une file d'attente. Le `fetch-and-add` est aussi une opération de ce type.

La primitive `fetch-and-add` peut être exécutée par plusieurs processeurs en

même temps, sans créer de blocages ni nécessiter d'essais réitérés. Elle permet de protéger une section critique sans elle-même en constituer une. Cependant, elle demande un réseau d'interconnexion élaboré qui permet de combiner plusieurs données devant être stockées au même endroit et de conserver des résultats intermédiaires. Le `fetch-and-add` retourne en chaque processeur une valeur différente permettant de décider de la suite des opérations.

On verra plus loin comment on utilise `fetch-and-add` pour mettre en oeuvre des sémaphores et des barrières de synchronisation en parallèle. Mais, avant cela, étudions son fonctionnement. L'instruction `fetch-and-add` peut s'exprimer ainsi :

```
fetch-and-add(S,I)
  atomic{ tmp=S; S=S+I}
   return tmp
```

`S` est la variable partagée et `I` un incrément propre à chaque processeur. Lors de plusieurs appels simultanés, le résultat du `fetch-and-add` est identique à celui qui serait obtenu si ces appels étaient faits de manière séquentielle, mais dans un ordre quelconque non spécifié. De plus, la variable `S` n'est lue qu'une seule fois.

La mise en oeuvre du `fetch-and-add` sur une machine parallèle disposant d'un réseau capable d'assurer les opérations intermédiaires est illustrée dans la figure 8.3. Les incréments `I` sont respectivement 2, 5 et 12. La variable `S` (initialement égale à 10) en mémoire vaut 29 après le `fetch-and-add` et les valeurs retournées au processeurs sont 10, 12 et 17, ce qui correspond aux valeurs intermédiaires de `S`. Ces mêmes valeurs seraient retournées par 3 appels consécutifs de `fetch-and-add` avec incréments `I=2,5,12`, respectivement.

Le `fetch-and-add` est une construction très puissante qui limite les contentions de réseau et de mémoire. Par contre, le prix du réseau d'interconnexion est élevé par rapport à un bus.

Une utilisation naturelle du `fetch-and-add` est la mise en oeuvre d'une boucle partagée par plusieurs processeurs :

```
!$omp parallel do
  do i=1,n
    compute a(i)
  enddo
!$omp parallel do
```

Si l'indice de boucle commun `i` est incrémenté avec `fetch-and-add(i,1)`, il y aura une gestion efficace du partage du travail. On peut donc imaginer l'implémentation suivante

```
  i=1
  repeat
```

```
    j=fetch-and-add(i,1)  // j est une variable locale
    if(j>n)exit
    compute a(j)
  end repeat
```

8.3.3 Réalisation d'un sémaphore parallèle avec fetch-and-add

Un sémaphore pouvant être simultanément testé par plusieurs processeurs et protégeant une section critique peut être construit comme suit, avec une variable partagée `S` initialisée à 1.

```
while fetch-and-add(S,-1)<1
   do fetch-and-add(S,1)
end while
CRITICAL SECTION
fetch-and-add(S,1)
```

La boucle `while` fait office d'instruction `LOCK`. En effet le "premier" processeur qui exécute le `fetch-and-add` entre dans la section critique. Les autres $p-1$ processeurs incrémentent et décrémentent continuellement `S` mais, avec $p-1$ processeurs, `S` ne peut varier qu'entre $p-1$ et 0. Les $p-1$ processeurs seront en "busy waiting" dans le `while` tant que le premier processeur n'aura pas incrémenté à nouveau `S` avec le `fetch-and-add` qui suit la section critique (le `UNLOCK`). Dès lors, `S` pourra repasser à 1 et le processeur suivant sera admis dans la section critique.

8.3.4 Réalisation d'une barrière de synchronisation avec fetch-and-add

La manière la plus simple de construire une barrière avec un `fetch-and-add` est de considérer une variable globale `X` initialisée à zéro. Pour synchroniser N tâches, il suffit d'écrire

```
fetch-and-add(X,1)
wait for X=N
```

à la fin de chaque tâche. La variable `X` sera mise à N lorsque la Nème tâche sera terminée, débloquant ainsi la barrière. Remarquons, entre parenthèse, que tester la valeur de `X` peut se faire de façon non-blocante avec la valeur retournée par `fetch-and-add(X,0)`.

En principe, on pourrait se contenter de l'implémentation ci-dessus pour une barrière. Mais en pratique, un programme contient beaucoup de telles barrières et

on veut éviter d'avoir une nouvelle variable dans chaque cas. Il faut donc remettre `X` à zéro avant la prochaine barrière. La solution consistant à écrire

```
X=0
...
if fetch-and-add(X,1)=N-1 then X=0
wait for X=0
```

n'est pas satisfaisante non plus car les processeurs peuvent être à des phases imprévisibles dans l'exécution des instructions. On pourrait avoir le cas défavorable suivant : la barrière est levée par le processeur A. Avant que le processeur B n'exécute le test pour savoir si `X=0`, le processeur A pourrait avoir déjà rencontré une autre barrière et remis `X` à 1. A nouveau, on tombe sur une situation de deadlock car B ne quittera jamais la première barrière et pourra encore moins ouvrir la deuxième. La bonne solution est

```
local_flag=(X<N)
if fetch-and-add(X,1)=2N-1 then X=0
wait for (X<N)≠ local_flag
```

Avant la première barrière, `X` est mis à 0 et `local_flag` (qui est une variable différente pour chaque processeur) est initialisé à la valeur `true`. Lorsque tout le monde a atteint la première barrière, `X=N`, et la barrière est levée. On arrive à la deuxième barrière avec toujours `X=N` mais `local_flag=false`. Comme `X<N` est faux aussi, la barrière est fermée jusqu'à ce que `X=2N` (c'est à dire que tous les processeurs aient terminé et `fetch-and-add(X,1)` ait retourné la valeur `2N-1`). Alors, `X` passe à zéro et `(X<N)` devient vrai. On passe ainsi la deuxième barrière, avec `X=0` à nouveau. Avec cette technique, on alterne, d'une barrière à l'autre, les conditions d'ouverture `X=N` et `X=2N`.

La raison fondamentale qui fait que cette approche marche est que la première barrière reste ouverte jusqu'à ce que le dernier processeur atteigne la deuxième.

En conclusion, la primitive **fetch-and-add** offre un moyen élégant et flexible pour gérer la coordination entre processeurs dans une architecture à mémoire partagée. Du point de vue fonctionnelle, elle permet de réaliser des barrières de synchronisation, des contrôles d'accès (lock/unlock) et permet aussi d'obtenir un équilibrage de charge dynamique. Du point de vue hardware, elle peut en principe être réalisée de façon non-blocante.

8.4 Avantages et désavantage du modèle

Dans ce paragraphe, nous discutons brièvement les avantages et désavantages du modèle à mémoire partagée par rapport au modèle à mémoire distribuée.

8.4.1 Avantages

Un système à mémoire partagée est plus simple du point de vue de l'utilisateur qui est en présence d'un modèle de programmation proche de celui des machines séquentielles.

Les communications entre les différentes tâches se font en modifiant des données dans des locations mémoires communes, ce qui constitue une extension naturelle du modèle de von Neumann. Le réseau d'interconnexion est ainsi caché à l'utilisateur et il en résulte une facilité de programmation qui évite au programmeur le besoin de gérer des mouvements de données entre les processeurs.

Notons que les mémoires partagées permettent par ailleurs d'émuler facilement des modèles de programmation différents, comme les échanges de messages. En conséquence, il est en général plus facile d'adapter, sur une architecture à mémoire partagée, un code développé pour une machine séquentielle car les modifications à y apporter sont de nature moins profondes.

Finalement, il n'est pas nécessaire de dupliquer des données communes en mémoire, ce qui optimise la gestion de l'espace mémoire global unique. Par ailleurs, cette gestion est facilitée par les techniques sophistiquées inspirées de l'expérience acquise pendant de nombreuses années avec les machines séquentielles. Les systèmes multi-utilisateurs sont ainsi plus faciles à réaliser que dans le cas des machines à mémoire distribuée pour lesquelles cette gestion n'atteint pas le même degré de maturité.

8.4.2 Désavantages

La souplesse des architectures à mémoire partagée se solde cependant par plusieurs problèmes. Du point de vue conception, elles sont plus compliquées en raison de la complexité de la gestion de la coopération entre les processeurs, notamment des conflits qui peuvent survenir. En pratique les systèmes à mémoire partagée se caractérisent par un nombre modeste de processeurs (quelques dizaines au maximum). Ces architectures ne sont pas extensibles ou scalables (on ne peut pas augmenter le nombre de processeurs arbitrairement) en raison, soit de la saturation du bus d'interconnexion, soit de l'augmentation des conflits mémoire. L'accès simultané de plusieurs processeurs à une même mémoire est limité et implique un temps de latence important.

De plus, les architectures à mémoire partagée sont peu compatibles avec un principe de **localité de référence** qui s'applique au parallélisme de la manière suivante : statistiquement, les données ne sont pas utilisées de manière uniforme par tous les processeurs. Certains utilisent des données plus fréquemment que d'autres et il faudrait que ces dernières soient accessibles rapidement par les processeurs qui les consomment. Dans une architecture à mémoire partagée, il n'y a pas de concept de proximité puisque l'on veut la mémoire uniforme. Si les données sont effectivement réparties de façon homogène dans la mémoire,

un accroissement du nombre de processeurs va engendrer un excès de trafic non local à travers le réseau d'interconnexion et causer des risques de congestion. Les architectures NUMA ou COMA permettent de remédier à ce problème.

Chapitre 9

Parallélisme de données

9.1 Concepts de base

Le parallélisme de données (ou “data-parallel”) exploite le fait que, dans de nombreuses applications, la même opération doit être répétée sur un grand nombre de données. L'exemple le plus simple est l'addition de deux matrices A et B qu'on écrira dans un langage data-parallèle

```
C=A+B
```

et qui, en fait, implique une addition en parallèle de tous les éléments a_{ij} et b_{ij}.

Ce modèle de programmation est l'apanage des architectures SIMD en raison de leur granularité fine et de leur synchronisme, qui en général correspondent bien à la nature du problème traité. Mais le principe du modèle de programmation data-parallel va au delà des architectures et une machine MIMD peut tout à fait le mettre en oeuvre. Si ce modèle s'applique très bien aux problèmes réguliers, sa généralisation à des données irrégulières et peu structurées est difficile et les performances risquent de ne pas être au rendez-vous.

Parmi les langages qui mettent en oeuvre le parallélisme de données on peut citer le «High Performance Fortan» (HPF) qui a eu un certain succès à la fin des année 90, en offrant des instructions qui portent directement sur des structures de données, comme par exemple un vecteur ou un tableau. Des directives au compilateur permettent de spécifier la façon dont les données doivent être distribuées sur les processeurs.

Plus récemment, le C++17 et versions suivantes permettent une exécution parallèle automatique, en particulier sur des vecteurs C++ qui sont transformés par ce qu'on appelle un algorithme en C++17. Cette parallélisation se fait sur des machines multicoeurs à mémoire partagée, ainsi que sur des GPU, et ceci simplement en spécifiant le mode d'éxécution (execution policy) des algorithmes choisis. Des exemples sont présentés au paragraphe 9.3.

9.2 Exemples d'instructions «Data-parallel»

Dans un modèle à parallélisme de données on est en présence **d'un seul flot d'instructions** et on peut compter sur la **synchronisation** des processeurs après chaque instruction. De plus, ce modèle offre une vision globale de l'espace mémoire puisque les structures de données sont définies "à travers" les processeurs. Le langage HPF (High Performance Fortran) est un bon exemple de la mise en oeuvre de ce modèle de programmation.

Les instructions qui agissent sur des structures de données correspondent, à un niveau plus bas et de façon transparente à l'utilisateur, à une opération exécutée en parallèle par les processeurs virtuels, sur chacun des éléments de la structure. Les opérations arithmétiques ou logiques sont des exemples simples de telles instructions parallèles. D'autres instructions, plus élaborées, transforment une structure de données en une autre, permettent des réarrangements des données dans une même structure ou réalisent sur elles des opérations complexes comme des SCAN.

Par exemple, l'opération SCAN_ADD peut être vue comme le calcul de la "primitive" discrete d'un tableau f :

$$\text{SCAN} : f \to F$$

tel que

$$F(x) = \int_0^x f = \sum_{i=1}^{x} f(i)$$

Cette opération est utile for formuler plusieurs algorithmes parallèle, en particulier le «radix-sort» dans lequel on construit les indices de position des éléments d'une liste en les «scannant» de gauche à droite.

Une utilisation simple du SCAN_ADD est la réalisation de la fonction PACK qui regroupe, dans un vecteur `W`, des éléments dispersés d'un vecteur `V`

```
V= (0 0 a 0 b c 0 0 d)
I= (0 0 1 0 1 1 0 0 1)   // on construit I qui vaut 1 là où
                         //                    V est non-nul
J= (0 0 1 1 2 3 3 3 4)   // J=scan_add(I)
W= (a b c d 0 0 0 0 0)   // where(I != 0)W(J)=V
```

(voir ci-dessous pour l'explication détaillée de l'instruction `where(I !=0)W(J)=V`).

On a aussi à disposition des instructions spécifiant le **contexte** et qui permettent de distinguer certains processeurs virtuels parmi d'autres, en fonction des données qu'ils contiennent, de manière à différencier les opérations qui y seront effectuées. En d'autres termes, ces instructions réalisent le concept de branchement en parallèle. L'instruction `where` est un exemple. Supposons qu'on ait

```
d=(0 0 0 0)
```

```
a=(1 2 0 1)
b=(2 3 1 2)
c=(4 2 3 3)
```

alors l'instruction `where(a==1){d=b+c}` signifie que la somme n'est faite que pour les éléments de `a` valant 1. Donc `d=(6 0 0 7)`.

Le modèle data-parallel permet d'exprimer le parallélisme de façon compacte, élégante et efficace. Par exemple, supposons qu'on dispose d'un tableau bidimensionnel `a` contenant des valeurs numériques. Une opération courante en calcul scientifique est de remplacer chaque élément du tableau par sa somme avec les valeurs comprises dans les 8 éléments voisins (nord, nord-est,...). Ce résultat est obtenu dans le tableau `c` par le programme suivant

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

où `b` est un tableau temporaire et l'instruction `cshift` indique une translation de `shift` unités selon la dimension `dim` du tableau (avec, ici, des conditions de bord périodiques). On remarque que quatre communications sont suffisantes alors qu'on accède huit voisins.

Des communications non-régulières, type SEND-RECEIVE, peuvent aussi s'exprimer dans ce mdèle de programmation. Un tableau d'indice `i` indique, pour chaque position d'une structure de données source, la position cible dans une autre structure de donnée. Par exemple, l'instruction

```
C=A(B)
```

où A, B et C sont des vecteurs, signifie, qu'en parallèle, pour chaque indice i dans le vecteur on exécute

```
C(i)=A(B(i))
```

Ainsi pour les valeurs

```
A=(17 34 12 78)   B=(2 4 3 1)
```

on a `C=(34 78 12 17)`. Evidemment, si `B` n'est pas une permutation des indices de 1 à 4 il y a un conflit dont la résolution échappe au programmeur.

Un autre exemple pratique qui illustre bien le modèle de calcul data-parallel est la solution d'équations aux dérivées partielles. Ce problème est à la base de nombreux calculs scientifiques, comme en particulier les prévisions météorologiques. Cette application est d'ailleurs en partie à l'origine du développement des architectures parallèles en raison de l'ampleur des calculs qui sont nécessaires pour obtenir des prévisions acceptables.

Pour illustrer le modèle de programmation data-parallel, revenons au cas simple de l'équation dite de la diffusion, ou encore, de la chaleur, dans un espace à deux dimensions

$$\frac{\partial \rho}{\partial t} = \nabla^2 \rho \tag{9.1}$$

En général, on cherche la fonction $\rho(x, y, t)$ en se donnant la condition initiale $\rho(x, y, 0)$ et certaines conditions aux bords. En remplaçant les dérivées par rapport au temps et à l'espace par des différences finies, on obtient l'équation suivante

$$\begin{aligned} \rho(x, y, t+\tau) = \rho(x, y, t) \quad &+ \quad \frac{\tau}{\lambda^2}(\rho(x+\lambda, y, t) + \rho(x-\lambda, y, t) \\ &+ \quad \rho(x, y+\lambda, t) + \rho(x, y-\lambda, t) - 4\rho(x, y, t)) \end{aligned} \tag{9.2}$$

L'opération de discrétisation ci-dessus revient à considérer le problème sur une grille dont la maille est λ et avec une résolution en temps de valeur τ.

Sous la forme (9.2), l'équation de diffusion (ou équation de Laplace)se prête parfaitement bien à une solution numérique faisant appel au parallélisme. On considère un ensemble de processeurs connectés en un réseau bi-dimensionnel (une grille), chacun pouvant communiquer avec ses 4 plus proches voisins (nord, sud, est et ouest). Chaque processeur correspond à un point (x, y) de la grille, et contient la valeur $\rho(x, y, t)$, à un temps t donné. La valeur de ρ au temps $t+\tau$ est obtenue *simultanément* pour tous les point (x, y) en faisant exécuter par chaque processeur la *même* opération, en l'occurrence celle spécifiée dans le membre de droite de l'équation (9.2). En une première étape, chaque processeur lit la valeur de ρ contenue dans les processeurs voisins (ce qui nécessite des communications). Ensuite, les mêmes additions et multiplications sont effectuées indépendemment en chaque site, sur des valeurs localement différentes. On conçoit bien que dans ce cas un seul jeu d'instructions synchrones permet de mettre en oeuvre le parallélisme nécessaire à la résolution du problème. Les conditions aux bords constituent par contre une difficulté car elles impliquent en général une opération différente que celle donnée en (9.2). Dans le modèle data-parallel, les bords du domaine de calcul sont pris en compte en ajoutant en chaque point de la grille une variable de contexte indiquant si le point en question fait ou non partie du bord. Si c'est le cas, les processeurs ainsi marqués, n'effectuent pas l'opération spécifiée par (9.2). La construction `WHERE` mentionnée ci-dessus permet cette exécution conditionnelle par un PE, en fonction de la valeur de la variable de contexte.

Des instructions de réduction, comme les primitives `SUM` ou `MAXVAL` de HPF, permettent d'avoir une information globale (un scalaire) sur l'ensemble des valeurs de ρ. Ceci peut s'utiliser notamment pour avoir un critère d'arrêt dans la méthode itérative ci-dessus, par exemple en détectant qu'entre deux itérations successives aucune valeur de ρ n'a varié de plus qu'une valeur seuil prédéfinie.

L'exemple de la résolution itérative de l'équation de Laplace (pour $A = \rho$) peut donc se formuler très simplement dans le modèle data-parallèle

```
A=0      // initialisation du domaine: A est une matrice nxn
e=1e-6   // critere de convergence
Repeat
  where(bord==false)
     B=A    // ancienne valeur
     A=0.25*(cshift(A,dim=1,shift=1) + cshift(A,dim=1,shift=-1) +
             cshift(A,dim=2,shift=1) + cshift(A,dim=2,shift=-1)  )
  elsewhere
     A=1   // condition au bord imposée
  endwhere
until( sum( (A-B)*(A-B)< e) ) // * est la multiplication
                              //     element par element
```

9.3 Parallélisme de données en C++17

Dans ce paragraphe nous donnons quelques exemple de code C++17 qui sont parallélisés sur une machine à 48 coeurs. Les constructions C++ nécessaires sont les suivantes

— vectors
— lambda functions
— algorithms
— execution policy

L'exemple proposé ci-dessous calcule numériquement une intégrale I avec une somme de Riemann, comme indiqué sur la figure 9.1

$$I = \int_{x_0}^{x_n} f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx = \sum_{i=0}^{n-1} \text{Integral}(x_i)$$

Cette intégrale peut être calculée en C++ par l'algorithme `transform_reduce()` qui modifie chaque élément d'un vecteur x par la fonction `Integral` qui calcul la somme de Riemann de x_i à x_{i+1}, et qui somme chaque valeur retournée avec la fonction `plus`

$\texttt{I} = \texttt{transform\_reduce}(\texttt{execution} :: \texttt{par}, \texttt{x.begin}(), \texttt{x.end}(), \texttt{0.}, \texttt{plus}\{\}, \texttt{Integral})$

Le code suivant implémente ce calcul. Il définit la fonction à intégrer comme

$$\text{my\_function} = \frac{\sqrt{|\sin(x^2)|}}{2 + \cos(\sqrt{x})}$$

puis il définit le domaine d'intégration de 0 à `Lx`, le nombre n d'intervales, qui peut s'interprèter comme le nombre de threads vu que C++ va paralléliser l'algorithme `transform_reduce` sur les éléments du vecteur x.

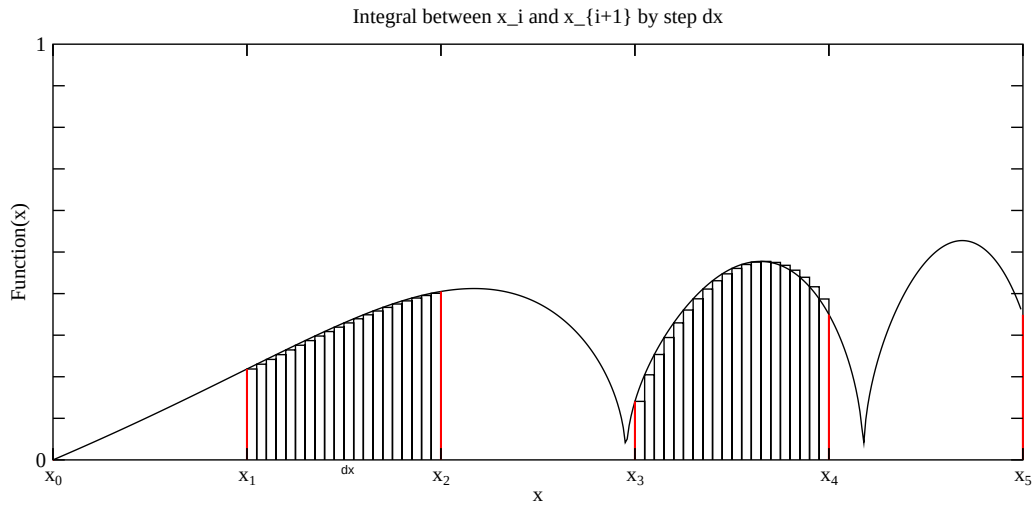


FIGURE 9.1 – L'intégrale se décompose ici en n morceaux, définis par les intervales $[x_i, x_{i+1}]$, pour $i = 0, 1, \ldots, n-1$, où n est la taille du vecteur x qui sera utilisé dans un algorithme C++. L'axe horizontal est divisé en rectangle de taille dx. En fonction de n il y a plus ou moins de ces rectangles dans les intervales, ce qui permet d'ajuster la quantité de travail associée à chaque élément x_i du vecteur x.

.

```
//g++ -O3 -std=c++17 programme.cpp -o programme -ltbb
#include <iostream>
#include <vector>
#include <execution>
#include <cmath>

using namespace std;


double my_fonct(double x){
  return sqrt(abs(sin(x*x))/(2+cos(sqrt(x))));
}


int main(int argc, char **argv){

  auto Lx=double{48};  // we will integrate from 0 to 48
  auto n=int{4800};      // on sub-intervals starting at 0,1,..,n-1
  auto Dx=double{Lx/n}; // size of a sub-interval
  auto dx=double{1e-7};  // accuracy

  auto m=(long) (Dx/dx); // each sub-interval is divided in m points

  auto x=vector<double>(n);
  // x contains the lower limit of th sub-intervals
  for(int i=0;i<n;i++)x[i]=i*(Lx/n);

  auto integral=[&m, &dx](auto x){
  auto s=double{0.};
  for(int j=0;j<m;j++)s+=my_fonct(x+j*dx)*dx;
  return s;
};

  // ----------  parallel execution------------

  auto r=transform_reduce(execution::par, begin(x), end(x), 0.0, plus{},integral);
  cout<<"Parallel: r="<<r<<endl;
}
```

La figure 9.2 donne les performance obtenues sur une machine à 48 coeurs. On constate que le temps d'exécution dépend de la taille du vecteur sur lequel la paralléliszation s'applique. Bien qu'il soit difficile de savoir combien de threads sont utilisés, on peut faire l'hypothèse que les éléments de x sont distribués équitablement entre les threads et que, tant que disponible chaque thread est exécuté par un coeur différent. Ainsi, on peut supposer que la taille n du vecteur est un indicateur du nombre de coeurs utilisés, tout au moins pour $n \leq 48$, le nombre maximum de coeurs de la machine considérée.

D'autre part, si on demande une exécution séquentielle avec

```
r=transform_reduce(execution::seq, begin(x), end(x), 0.0, plus,integral);
```

on obtient un temps de calcul essentiellement indépendant de n, comme on pouvait s'y attendre. On peut alors comparer le temps d'exécution séquentiel, T_{seq} avec le temps d'exécution parallèle, T_{par}, en fonction de n. Empiriquement on trouve que la relation suivante

$$T_{par} = \frac{T_{seq}}{\left(\frac{2}{3}n - 1\right)^{0.68}} \qquad n \leq 96 \qquad T_{par} = \text{const} \quad \text{if } n \geq 480$$

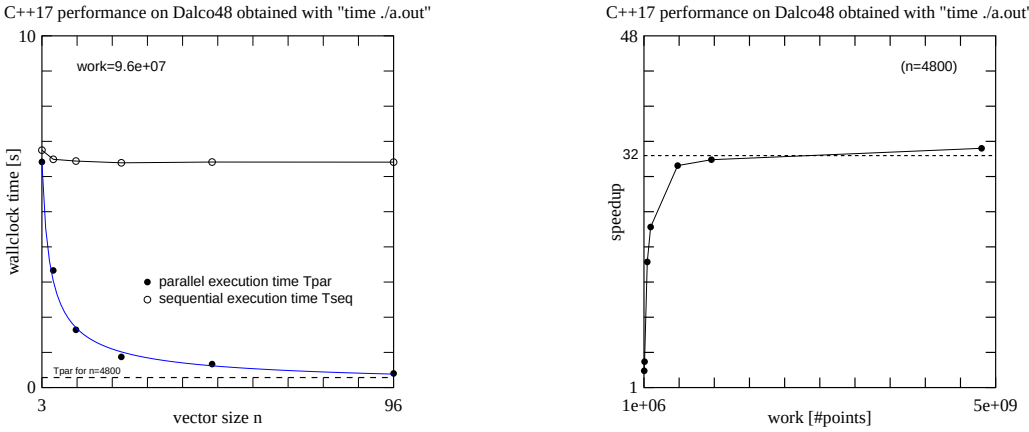


FIGURE 9.2 – Performance de l'execution parallèle du calcul d'une intégrale, en fonction de la taille du vecteur x (qui peut s'interpréter come le nombre de threads), et la quantité de travail, $W = (x_n - x_0)/dx$. A gauche, le temps d'exécution séquentiel et parallel en fonction de n. A droite, le speedup, avec $n = 4800$, en fonction du travail, à savoir en fonction de la précision dx.

explique bien les temps mesurés. Si on comprend bien que le temps T_{par} diminue avec n (le nombre supposé de coeurs actifs), les autres facteurs sont difficiles à interprèter. Ils reflètent certainement des sources d'overhead, comme la gestion des threads et les conflits mémoire, mais les quantifier plus précisément n'est pas possible avec les informations dont on dispose.

Parmi les modes d'exécution possibles, on trouve

— `execution::seq` c'est une exécution séquentielle.
— `execution::par` c'est une exécution «multithread», avec plus qu'un thread, mais le nombre exact n'est pas spécifié.
— `execution::par_unseq` c'est un exécution multithread, avec «vectorisation», ce qui utilise une unité SIMD qui permet d'agir simultanément sur quelques éléments du vecteur.
— `execution::unseq` c'est une exécution vectorisée au sens ci-dessus, mais avec un seul thread.

On notera que le parallélisme du C++ n'est pas forcément «thread-safe» et compatible avec le principe de cohérence séquentielle. Les threads pouvent s'exécuter «out of order». Des conditions de course (*race conditions*) peuvent se produire si on utilise mal les instructions, et des comportement non-déterministes peuvent en résulter.

Le code suivant donne un example ou plusieurs threads vont écrire une valeur différente dans une même variable s. Le code se compile et s'exécute, mais son résultat est non deterministe. La valeur de s n'est pas égale à 9999 comme une exécution séquentielle le produirait, mais contient en fin d'exécution des valeurs

imprévisibes, qui changent à chaque fois.

```
#include <iostream>
#include <execution>
#include <vector>

using namespace std;

int main()
{
  auto a=array<int,10000>{};
  iota(a.begin(), a.end(),0); // a[i]=i

  auto s=int{0};
  vector<int> v;

  for_each(std::execution::par, std::begin(a), std::end(a), [&](int i)
   {
    s=i; // data race
   });

  cout<<"s="<<s<<endl;
}
```

Les sites suivants donnent des informations supplémentaires.

`https://www.geeksforgeeks.org/execution-policy-of-stl-algorithms-in-modern-cpp/`
`https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t`

Le code ci-dessous propose une implémenation de la résolution de l'équation de Laplace, en 1D, en itérant la relation

$$u[i] = 0.5(u[i-1] + u[i+1])$$

On utilise une arithmétique de pointeurs pour obtenir les valeurs de $u[i-1]$ et $u[i+1]$ dans l'algorithme `transform()`, et ceci grâce à la capture du pointeur `p` sur le premier élément de u. Sur une machine à 48 coeurs, un spedup de 28 est observé entre les modes `execution::par` et `execution::seq`.

```
#include <iostream>
#include <execution>
#include <vector>

using namespace std;


int main()
{
  int n=1000000;
  vector<double> u(n);
  double* p=&u[0];
  double* p2=&u[n/2];

  //  u[i]=1e-6*(i-n/2)^2
  transform(execution::par, u.begin(), u.end(), u.begin(),
    [p2](double& u_from){ int i= (&u_from-p2);
   return 1e-6*i*i;});
  cout<<"u[n/2]="<<u[n/2]<<endl;

  vector<double> uu(n);
```

```
  int tMax=10000;
  for(int t=0;t<tMax;t++){
    transform(execution::par, u.begin(), u.end(), uu.begin(), [p,n](double& u_from)
     {int index=&u_from-p;
      return 0.5*(p[(index+n-1)%n]+p[(index+1)%n]);
     });

    transform(execution::par, uu.begin(), uu.end(), u.begin(), [](double& uu_from)
    {return uu_from;});

  } // end for loop

  cout<<"u[n/2]="<<u[n/2]<<endl;

  return 0;
}
```

Finalement, la figure 9.3 donne une liste des algortihmes C++17, et marque ceux qui acceptent un mode d'exécution parallèle.

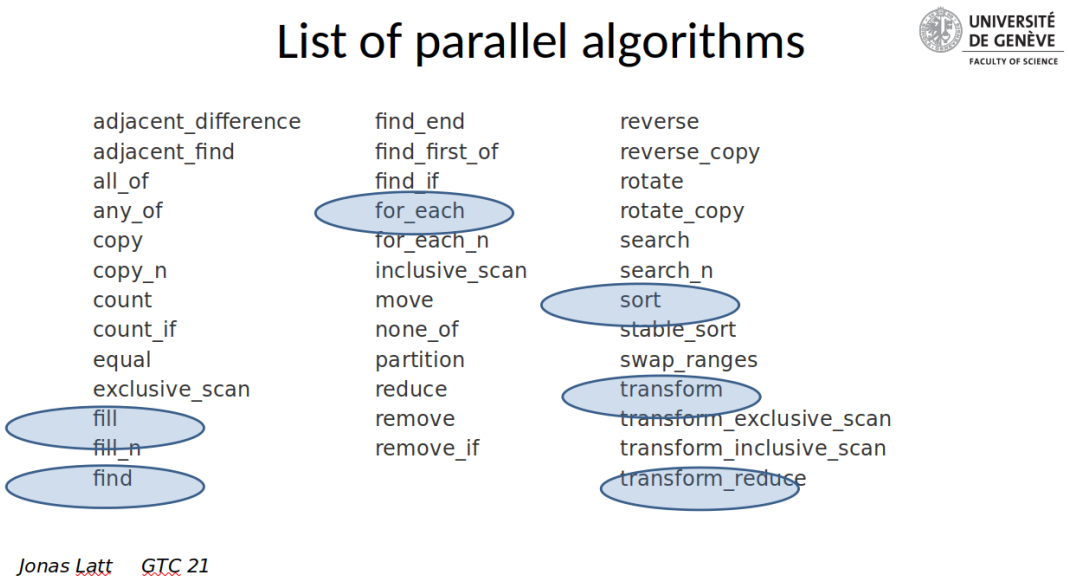


FIGURE 9.3 – Liste des algortithmes C++17 qui se parallélisent.

Bibliographie

[1] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, 1989.

[2] G. Blelloch. *Vector models for Data-parallel Computing*. The MIT Press, 1990.

[3] Michel Cosnard and Denis Trystram. *Algorithmes et Architectures Parallèles*. Collection IIA. InterEditions, 1993. ISBN 2 7296 0426 x.

[4] Hesham El-Rewini and Ted Lewis. *Distributed and Parallel Computing*. Manning, Greenwich, 1998.

[5] Ian Foster. *Designing and Builiding Parallel Programs*. Addison Wesley, 1995.

[6] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Serie in Computer Science. Prentice-Hall International, 1992.

[7] Kai Hwang. *Advanced Computer Architecture : Parallelism, Scalability, programmability*. Mc Graw Hill, 1993.

[8] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing : design and analysis of algorithm*. Benjamin/Cummings, 1994.

[9] F. Thomson Leighton. *Introduction to parallel algorithms and architectures : array, trees, hypercubes*. Morgan Kaufmann, San Mateo, California, 1992.

[10] T.G. Lewis and H. El-Rewim. *Introduction to Parallel Computing*. Prentice-Hall International Editions, 1992.

[11] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation*. Clarendon Press, Oxford, 1988.

[12] Dan. I. Moldovan. *Parallel Processing : From application to Systems*. Morgan Kauffmann, San Mateo, California, 1993.

[13] H. Stephen Morse. *Practical Parallel Computing*. AP Professional, 1994.

[14] Michael J. Quinn. *Parallel Computing : Theory and Practice*. Series in Computer Science. Mc Graw Hill, 1994.

[15] W. Schröder-Preikschat. *The logical design of parallel operating systems*. Series in Innovative Technology. Prentice Hall, 1994.

[16] D. Sima, T. Fountain, and P. Kasuk. *Advance computer architecture : a design space approach*. Addison-Wesley, 1997.

[17] William Stallings. *Computer Organization and Architecture*. Prentice Hall, 2000.

[18] H.S. Stone. *High Performance Computer Architecture*. Addison-Wesley, 1987.

[19] Arthur Trew and Greg Wilson. *Past, Present and Parallel*. Springer, 1991.

[20] Michael Wolfe. *High Performance Compiler for Parallel Computing*. Addison Wesley, 1996.

Index