Chapitre 1

Introduction au parallélisme et aux machines parallèles

Sur un serveur HPC doté de processeurs modernes, un code simple de produit matriciel écrit en C :

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

n'atteint même pas $0,05\%$ de la capacité de calcul théorique de la machine. Une bibliothèque de calcul scientifique lourdement optimisée (le BLAS fourni par Intel dans leur produit MKL) peut atteindre 66%. Le but de ce cours consiste à *i*) apprendre à mieux exploiter un serveur de calcul, et *ii*) apprendre à en exploiter plusieurs à la fois.

Pour réaliser de gros calculs, il est nécessaire de faire travailler simultanément plusieurs processeurs pour résoudre le même problème. Le *parallélisme* est donc un aspect essentiel du HPC. Depuis le début, les machines dédiées au calcul scientifique sont des machines parallèles.

Ceci dit, HPC et parallélisme ne s'identifient pas. Des programmeurs compétents peuvent, en connaissant bien l'architecture matérielle du processeur (réputé séquentiel) sur lequel leurs programment s'exécutent, en tirer de bien meilleurs performances que des programmeurs « normaux ». En outre, un processeur « séquentiel » est en fait lui aussi une « machine parallèle ».

S'il fallait un slogan pour résumer, on pourrait dire que le HPC c'est : *i*) du matériel (parfois) spécialisé que les programmeurs connaissent intimement, *ii*) des techniques de programmation parallèle, et *iii*) des algorithmes adaptés aux architectures matérielles visées.

Bien sûr, le problème de faire collaborer différents processeurs qui partagent une mémoire ou communiquent par un réseau ne sont pas spécifiques au HPC. C'est aussi l'objet d'autres « disciplines » telles que le calcul distribué (mode client-serveur, fonctionnement du web, bittorrent, architecture de Google, connections réseau lentes et/ou non-fiables, machines hétérogènes et physiquement éloignées, ...) ou la conception des systèmes d'exploitation (partage équitable des ressources entre processus concurrents, mécanismes de synchronisation, isolation des tâches, simulation de « machines virtuelles », ...).

Une partie de ces problématiques se retrouvent dans le HPC, mais ce dernier a généralement une « saveur » spécifique. En général, dans le cadre du HPC, on considère qu'on a affaire à un ou des ordinateurs tous plus ou moins identiques, physiquement proches, qui communiquent vite et de manière fiable, et qui ne tombent pas en panne.

En réalité, la *tolérance aux pannes* est un sujet essentiel du HPC : quand on met dans la même pièce des dizaines de milliers de processeurs, de composants mémoire, etc. il y en a qui tombent en panne chaque jour. Les plus gros calculs doivent pouvoir survivre à de telles pannes, sinon ils ne seront jamais réalisés jusqu'au bout.

1.1 La programmation parallèle a de beaux jours devant elle

Plusieurs arguments tendent à démontrer que la programmation parallèle est une tendance d'avenir.

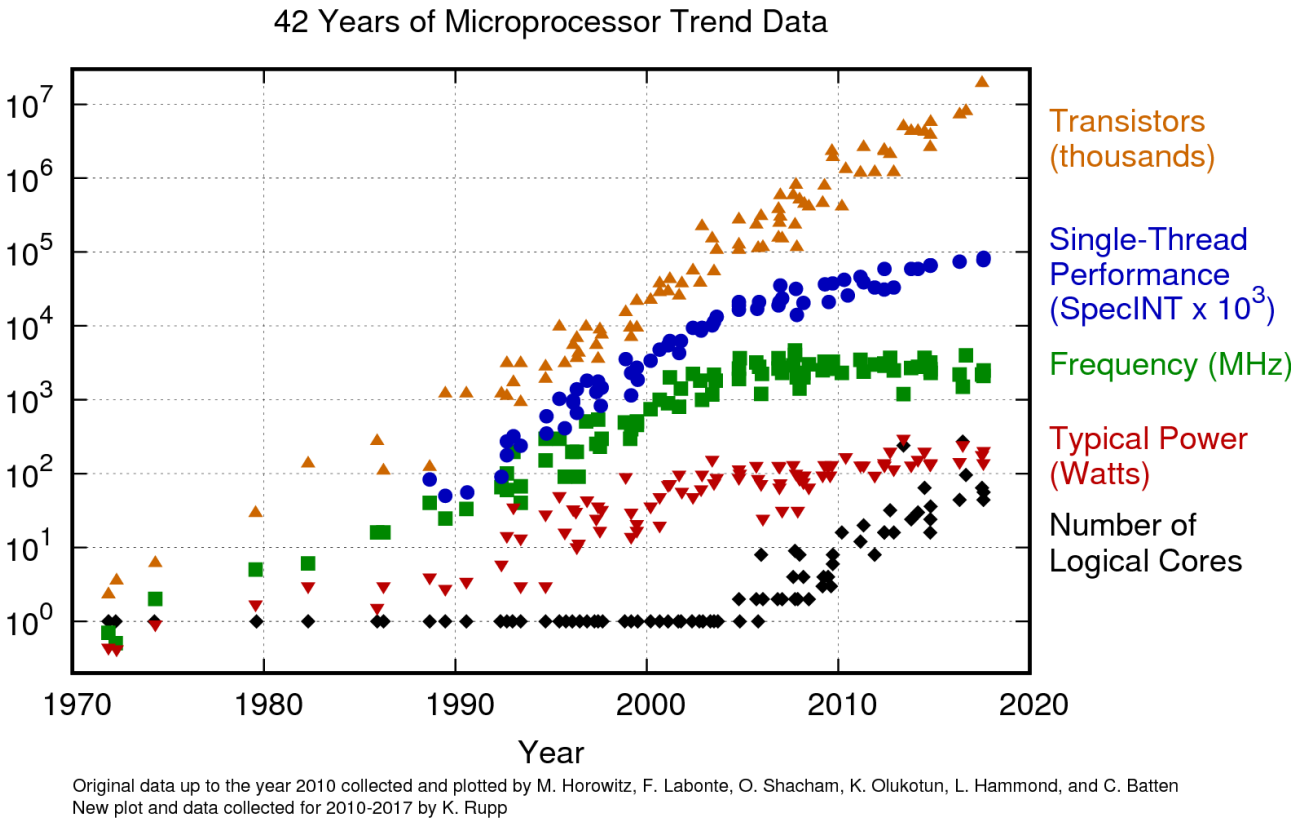


FIGURE 1.1 – Le « *Dennard Scaling* et la loi de Moore illustrées (crédit image : `https://www.karlrupp.net`).

La fin du « *Dennard Scaling* » La « loi de Moore » est bien connue : elle affirme que le nombre de transistors dans un CPU double tous les 18 mois, à coût (relativement) constant. Pendant plusieurs décennies, ceci a pu être obtenu grâce à une stratégie formulée en 1974 par un ingénieur d'IBM, Robert N. Dennard [1] (1932–), qui porte du coup le nom de « *Dennard Scaling* ». L'idée, c'est qu'à chaque génération technologique, si on parvient à réduire la taille des transistors de 30%, alors on peut doubler le nombre de transistors *et* augmenter la fréquence de 40% (à surface et à puissance dissipée constante).

La puissance électrique dissipée par un CPU est la somme de deux composantes : $P_{cpu} = P_{dyn} + P_{leak}$. La partie « dynamique » est la puissance consommée par le changement d'état des transistors qui composent le CPU pendant les calculs. P_{leak} désigne la puissance absorbée par les « courants de fuite » qui parviennent à passer entre deux matériaux censés être isolés, indépendamment de l'activité réelle du CPU. On peut estimer que $P_{dyn} = \mathcal{O}\left(CV^2 f\right)$, où V désigne la tension du courant qui alimente le CPU, C désigne la capacité des conducteurs et f désigne sa fréquence d'horloge.

L'idée de Dennard, c'est que si on multiplie la dimension des transistor par $\lambda < 1$, alors la surface qu'ils occupent est multipliée par λ^2, et les longueurs des conducteurs sont multipliées par λ. La réduction des taille multiplie la capacité des conducteurs par λ. On multiplie volontairement la tension V appliquée au processeur par λ, ce qui multiplie mécaniquement les intensités I par λ aussi (loi d'Ohm approximative). Du coup, la charge électrique qu'il faut transférer pour faire basculer un transistor est multipliée par λ^2 ($q = CV$). Le temps qu'il faut pour transférer cette charge est multiplié par λ ($t = q/I$). Ceci permet de multiplier la fréquence par un facteur $1/\lambda > 1$, et la puissance dynamique dissipée est multipliée par λ^2. Le « *Dennard Scaling* » consiste à parvenir à $\lambda = \sqrt{2}/2$ tous les 18 mois.

Le problème, c'est que le *Dennard Scaling* a cessé en 2005-2006 (cf. fig 1.1) — avec la miniaturisation accrue, les courants de fuite deviennent trop importants. La fréquence des processeurs stagne au-dessous de 4Ghz, à de rares exceptions près. L'augmentation de la puissance de calcul des ordinateurs les plus puissants du monde a légèrement ralenti en conséquence (fig 1.2). Par contre, la loi de Moore, elle, continue à être vérifiée, grâce à l'augmentation du nombre de coeurs dans les processeurs.

Mais du coup, le parallélisme est une tendance de long terme : les ordinateurs, et surtout ceux dédiés au calcul scientifique, sont déjà et seront de plus en plus des machines parallèles, avec beaucoup de coeurs. De nos jours, les smartphone haut-de-gamme ont déjà 6 ou 8 coeurs.

1. Dennard a également inventé la DRAM...

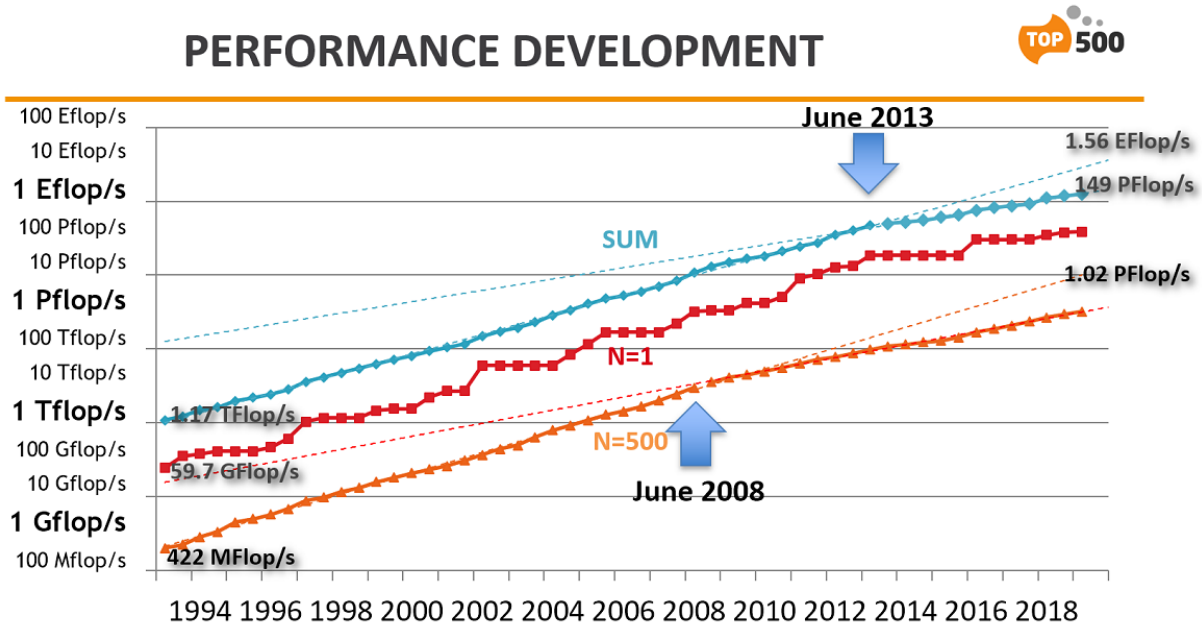


FIGURE 1.2 – Répercussion de la fin du « *Dennard Scaling* sur la puissance des ordinateurs du TOP 500. L'effet se fait d'abord sentir sur le 500-ème système du classement, avant de se répercuter sur la puissance de calcul cumulée des 500 machines.

Limites énergétiques La consommation énergétique des ordinateurs prend de plus en plus d'importance. On estime que les prochains ordinateurs « exascale » consommeront environ 20MW voire plus. L'évacuation de la chaleur produite par unités de calcul est un problème majeur de leur conception. Les processeurs modernes ont beaucoup de coeurs, mais ils ne peuvent pas les faire tous fonctionner à la fréquence maximale sous peine de surchauffe.

Sur les processeurs Intel les plus communs, cette caractéristique porte le nom commercial de « *turbo boost* ». L'idée c'est que lorsqu'un seul coeur est utilisé, il peut être « overclocké », mais lorsque tous les coeurs sont utilisés, c'est impossible. Par exemple, les processeurs Intel Xeon Gold 6248 qui équipent le calculateur « Jean-Zay » du CNRS ont 20 coeurs. Chaque coeur est capable de fonctionner à 3.9Ghz ! Mais tous ensemble, ils sont limités 2.5Ghz (la dissipation de chaleur serait trop importante s'ils étaient tous à fond).

En réalité, c'est encore plus compliqué que ça. Sur les processeurs Intel « Skylake » et suivants, la fréquence maximale d'une part et « minimum garantie » d'autre part dépendent de la nature des calculs effectués : s'ils sont scalaires (fréquence la plus élevée), riches en AVX2 (fréquence moins élevée) ou riches en AVX-512 (encore moins élevée).

Dans le HPC, la prise en compte de la consommation énergétique des calculs est principalement le problème des ingénieurs qui conçoivent les composants matériels. En résumé, la répercussion de ces contraintes sur les applications parallèle est à peu près la suivante : il est plus efficace, énergiquement parlant, d'avoir beaucoup de processeurs simples lents que peu de processeurs complexes rapides, or ceci nécessite de faire des calculs largement parallèles.

En effet, il découle des raisonnements ci-dessus que si on veut augmenter la fréquence, il faut que les conducteurs se chargent plus vite. Pour cela, il faut augmenter l'intensité des courants. Et pour cela, il faut augmenter la tension. Du coup, la puissance dynamique dissipée, qui est asymptotiquement $V^2 f$, est en fait en f^3. On a tout intérêt à garder f assez faible pour minimiser la puissance.

Par exemple, dans [1], les promoteurs de l'architecture libre `RISC-V` démontrent un processeur capable d'aller jusqu'à 1.3Ghz, mais dont l'efficacité énergétique est maximale à 250Mhz : la puce réalise alors 16.7 GFlops/W. Pour aller dans le même sens, dans l'article [4], les auteurs exécutent un petit code de calcul sur un processeur de smartphone. Il s'agit d'un code qui permute de manière déterministe les éléments d'un tableau, et qui est notamment utilisé dans le calcul de la transformée de Fourier rapide (FFT). Les auteurs observent que la consommation d'énergie divisée par la taille du tableau a un optimum (vers 600-700MHz) qui n'est pas la fréquence maximale (de 1.6Ghz) (cf. fig 1.3).

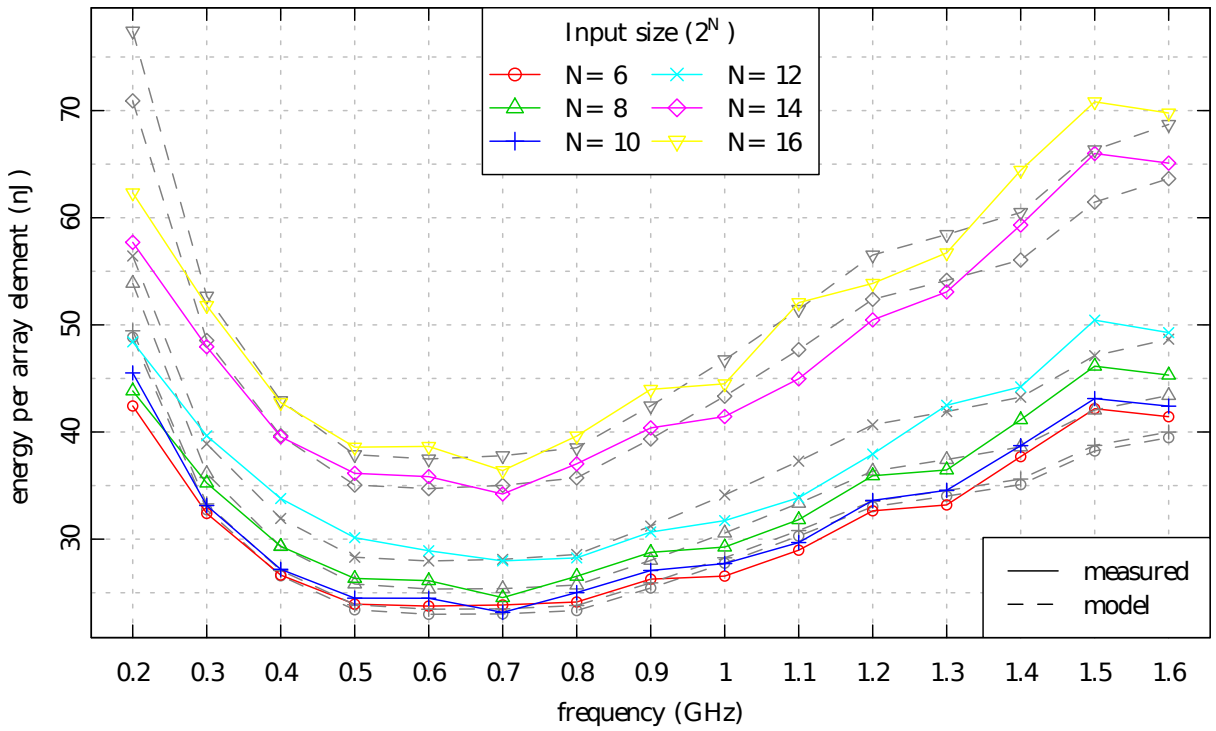


FIGURE 1.3 – Efficacité énergétique d'un calcul (en nJ par élément traité) en fonction de la fréquence d'horloge du CPU. L'optimum énergétique est atteint pour une fréquence donnée, qui n'est ni la minimale, ni la maximale. Le CPU est un processeur de smartphone (Cortex-A9) (image : [4]).

Autre exemple : l'ordinateur massivement parallèle IBM BlueGene/Q. Les ingénieurs d'IBM ont conçu une machine massivement parallèle en assemblant un grand nombre de processeurs « lents ». Ils ont pris un processeur simple (RISC, *in-order*, le PowerPC A2), et ils ont délibérément choisi de faire le fonctionner à 1.6Ghz alors qu'il aurait supporté 2.4Ghz : ce choix améliore l'efficacité énergétique. Avec 98304 tels processeurs (à 16 coeurs), ils ont obtenu simultanément la meilleure place au Top500 (17.1 PFLOPS en 2011) et au Green500 (2.1 GFLOPS/W pour l'ensemble de la machine).

Aujourd'hui, les microprocesseurs « normaux » sont gros (beaucoup de transistors), rapides (fréquence élevée), complexes, avec beaucoup de parallélisme à l'intérieur pour masquer les latences (*out-of-order*), etc. À l'avenir, les contraintes énergétiques seront plus dures. Une stratégie possible consiste à utiliser des processeurs petits, simples, plus lents, avec peu de parallélisme à l'intérieur (*in-order*). C'est d'ailleurs *déjà* largement ce qui se passe à l'intérieur des « accélérateurs » de calculs de type GPU, Xéon Phi et autres *many-cores*. C'est notamment pour cette raison qu'ils sont plus efficaces d'un point de vue énergétique (plus de calculs réalisés par Joule dissipé).

On y reviendra plus tard, mais il se trouve que les programmeurs peuvent jouer un rôle dans l'amélioration de la consommation énergétique : le moyen le plus sûr d'écrire des programmes économes en énergie consiste à écrire des programmes... « haute-performances ».

1.2 Le HPC a de beaux jours devant lui

(cette section n'est pas rédigée)

Quasiment tous les domaines du calcul scientifique sont désormais concernés.

Pour ne donner qu'un exemple en géophysique (recherche de gisements pétroliers en vue de futurs forages) : d'après des représentants de Total (en 2018) : le coût d'un forage en eau peu profonde est de ≈ $30 millions. Le coût d'un forage en eau profonde est de ≈ $100 Millions. Mieux vaut investir dans une machine de calcul scientifique et faire des simulations que de rater un forage !

Autres domaines problématiques : simulation d'écoulements turbulents (par exemple les combustions), simulation de l'interaction entre des molécules complexes (protéines) en simulant les forces élémentaires entre les atomes, etc.

Voir aussi les grands challenge scientifiques (d'après Quinn) : Levin. E. "Grand challenges to computational

science." Communications of the ACM 32(12) : 1456-1457, December 1989.

1.3 Qu'est-ce qu'une machine dédiée au HPC ?

On peut bien sûr faire de la programmation parallèle et viser un bon niveau de performance sur des ordinateurs « normaux », ou bien avec la « machine parallèle du pauvre » (des PCs de bureau reliés par un réseau ethernet classique). Mais les machines dédiées au calcul scientifique sont aux ordinateurs normaux ce qu'une Formule 1 est à une petite citadine. Les processeurs n'ont rien à voir, le réseau n'a rien à voir et le stockage n'a rien à voir.

Tout d'abord, voici un petit glossaire :

machine l'ensemble des composants : *noeuds* de calcul, réseau, stockage, etc.

noeud (*node*) un « ordinateur » indépendant (dans le cadre d'une machine parallèle qui en contient plusieurs), constitué d'un ou plusieurs processeurs, d'une mémoire et d'une interface le connectant au réseau.

baie (*rack*) une armoire destinée à recevoir des boîtiers, typiquement les noeuds, des commutateurs réseaux, des modules d'alimentation électriques, etc. Leur taille est souvent standardisée : 46.5 cm de large (19 pouces) et 1.8m de haut.

multiprocesseur (*Symmetric Multi Processing* / SMP) se dit d'un noeud qui contient plusieurs processeurs qui ont accès à la même mémoire.

processeur (*CPU*) un circuit intégré qui constitue un objet unique physique. Il contient au moins un *coeur* (cf. ci-dessous), ainsi que potentiellement du cache, des contrôleurs mémoire, éventuellement des interfaces à d'autres périphériques (réseau, PCI express, etc.).

coeur (*core*) un ensemble de circuits capables d'exécuter du code de façon autonome. En particulier, un coeur contient en principe une unité arithmétique (ALU), une unité de gestion de la mémoire (MMU), bien souvent un cache de données/d'instructions, ainsi qu'au moins un *thread matériel* (cf. ci-dessous).

multicoeur se dit d'un processeur qui contient plusieurs coeurs.

thread matériel (*hardware thread*) un contexte d'exécution autonome à l'intérieur d'un coeur. Constitué au moins d'un banc de registres (*register file*) et un pointeur d'instruction, etc. Parfois appelé « coeur logique ».

Simultaneous Multi-Threading se dit d'un coeur qui héberge plusieurs threads matériels. Les unités d'exécution (arithmétique et logique, flottante, mémoire, cache éventuel, etc.) sont *mutualisées* entre les threads. L'intérêt de cette technique est que si l'un des threads est bloqué, les autres peuvent continuer à exploiter le matériel.

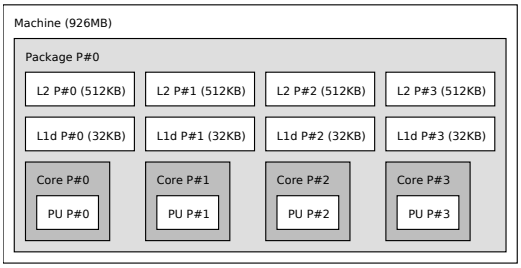
Il est très facile de confondre processeur/coeur/thread. Grosso-modo : un processeur est un objet (qui s'insère dans un « *socket* ») qui contient au moins un coeurs (ainsi que d'autres bricoles) ; un coeur héberge au moins un thread matériel. Pour fixer les idées, la figure 1.4 contient deux exemples. Depuis le milieu des années 2000, les processeurs les plus communs ont deux threads matériel par coeur (chez Intel, ceci porte la dénomination commerciale « HyperThreading »). Certains processeurs peuvent en avoir plus.

On dit parfois « processeur » par abus de langage, pour dire « coeur » ou « unité de traitement ». Au CNRS, les « CPU-hours » désignent le nombre d'heure de calcul multiplié par le nombre de coeurs utilisés. Sur le cloud Amazon, un vCPU est un thread matériel, etc. Ce qui complique les choses, c'est que sous la plupart des systèmes d'exploitations, les threads matériels apparaissent comme des processeurs indépendants.

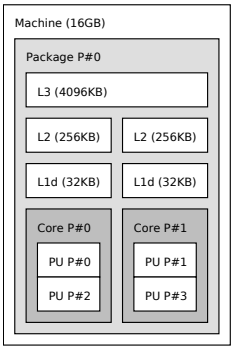
Il est aussi très facile de confondre thread matériel/thread logiciel. Un thread logiciel est un contexte d'exécution au sein du système d'exploitation. L'ordonnanceur de l'OS place les threads logiciels sur les threads matériels disponibles [2]. En général, dans un OS moderne, des dizaines de processus/threads logiciels s'exécutent de manière concurrente, et l'ordonnanceur de l'OS choisit lesquels affecter aux (quelques) threads matériels pendant le prochain quantum de temps. Des changements de contexte périodiques et suffisamment rapides peuvent donner l'illusion à l'utilisateur d'une exécution simultanée de tous ces processus.

— Le CNRS possédait, de 2011 à 2019 une machine IBM Bluegene/Q. C'est une « petite » installation avec seulement 6 rack. Chaque rack contient 1024 noeuds, chaque noeud contient un processeur PowerPC A2 à 16 coeurs et 16Go de RAM, cadencé à 1.6Ghz (cf. fig. 1.6). Il y a donc 98 304 coeurs en tout et 98To de RAM. La puissance de crête est 1.2 PetaFLOPS et la machine consomme 600kW. Chaque processeur contient une interface réseau « maison » avec 10 liens à 2Go/s dans chaque sens vers 10 autres processeurs (tore 5D). Chaque coeur héberge de 1 à 4 threads matériel (au choix du programmeur). Refroidissement par eau. Les noeuds exécutent une version *light* de Linux (les utilisateurs ne peuvent pas se connecter aux noeuds). Coût : ≈ 20 millions d'euros.

2. la surcharge du mot « thread » est pénible.



(a) Un processeur (ARM Cortex A53), quatre coeurs. Chaque coeur possède 32Ko de cache L1 et 512Ko de cache L2.



(b) Un processeur (Intel Core i7 6600U), deux coeurs, deux threads matériels par coeur. Chaque coeur possède 32Ko de cache L1 partagé entre les deux threads ainsi que 256Ko de cache L2. Le processeur contient un cache L3 de 4Mo partagé entre les deux coeurs.

FIGURE 1.4 – À Gauche, un Raspberry Pi modèle `3B+`. À droite, le laptop du prof.

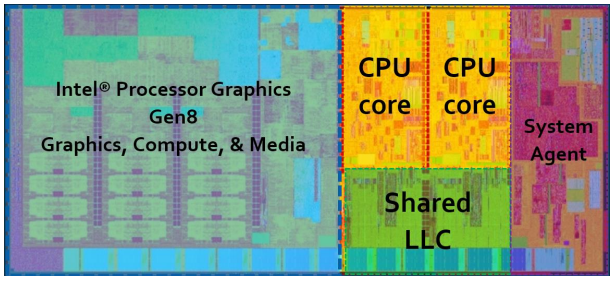


FIGURE 1.5 – Un processeur contemporain (Intel core i5 « Skylake ») à deux coeurs habituel dans les laptops. 3Mo de cache et GPU intégré. (image : Intel)

— Le 12ème ordinateur le plus puissant du monde, `sequoia`, est une BlueGene/Q de 96 racks, donc 1 572 864 coeurs, 1.5Po de RAM et 10MW de consommation électrique.

— Le CNRS vient (en 2019) de faire l'acquisition d'une machine plus moderne (`jean-zay`) construite par HPE. Il est composé de 1528 noeuds « scalaires » contenant 2 processeurs (Intel Xeon « Cascade Lake » 6248) à 20 coeurs, 2.5Ghz et 192Go de RAM. Il y a aussi 261 noeuds « convergés » qui contiennent aussi 4 GPU NVIDIA V100 SXM2 avec 32Go de RAM chacun. Il y a donc 61 120 coeurs CPU, 1044 GPUs et 343To de RAM en tout. La puissance de crête est 14 PetaFLOPS. Refroidissement par eau chaude. Réseau Intel OmniPath 100Gb/s.

— À l'échelle mondiale, la machine la plus puissante est `summit`, construite par IBM. Composée de 4608 noeuds avec chacun 2 processeurs Power9 à 22 coeurs, 6 GPU NVIDIA V100 et 600Go de RAM. Consommation électrique : 13MW. Coût : ≈ $500 millions. Puissance de crête : 200 PetaFLOPS. Réseau Infiniband 100Gb/s. Total : 202 752 coeurs, 27 648 GPUs, 2.7Po de RAM.

— En position numéro 5, on trouve `Frontera`, construit par Dell, la machine la plus puissante installée dans une université. Composée de 8008 noeuds, chacun avec 2 processeurs (Intel Xeon platinum 8280 « Cascade Lake », cf. fig. 1.7) à 28 coeurs, 2.7Ghz et 192Go de RAM. Réseau Infiniband 100Gb/s. Puissance de crête : 39 PetaFLOPS. Total : 448 448 coeurs et 1.5Po de RAM. Coût : ≈ $60 millions.

— En cours de construction, mais qui va bientôt dominer le classement, `fugaku` construit par Fujitsu. Composée de 150 000+ noeuds, chacun avec un processeurs (Fujitsu A64FX, cf. fig. 1.8) à 48 coeurs, 2Ghz et 32Go de RAM HBM (très rapide). Réseau maison « Tofu 3 » (tore 3D de tore 3D). Puissance de crête : 400+ PetaFLOPS. Total : 7 200 000+ coeurs et 4.7Po de RAM. ≈ 900 millions d'euros.

— Et bien sûr, la machine parallèle du pauvre : les 16 PCs à 4 coeurs d'une salle de TP reliés par un switch Gigabit ethernet (dans le meilleur des cas).

1.3.1 Petit historique

(cette section est incomplète)

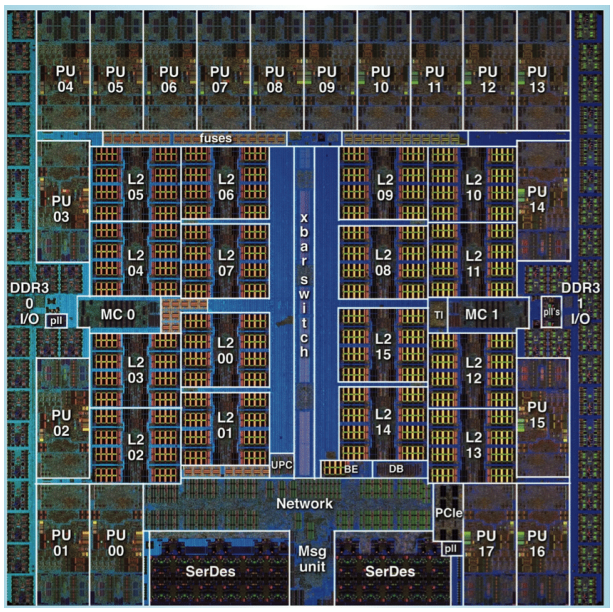


FIGURE 1.6 – Un processeur PowerPC A2 de 2011–2012, utilisé dans les machines IBM Bluegene/Q. 18 Coeurs (16 pour les applications, 1 réservé pour l'OS, 1 spare), 32Mo de cache, 2 contrôleurs mémoire DDR3 et 11 contrôleurs réseau 2Go/s intégrés. (image : IBM)

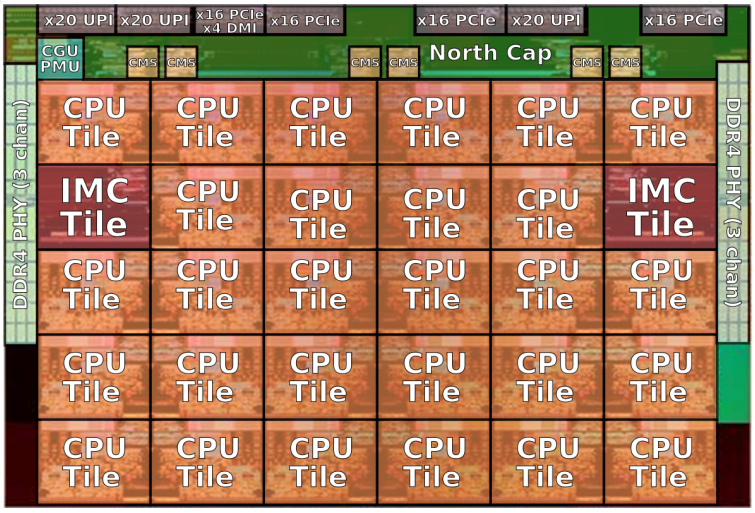


FIGURE 1.7 – Un processeur Intel Xeon « Skylake » de 2019 à 28 coeurs. Contrôleur réseau (OmniPath) et 6 contrôleurs mémoire DDR4 intégrés. (image : Intel)

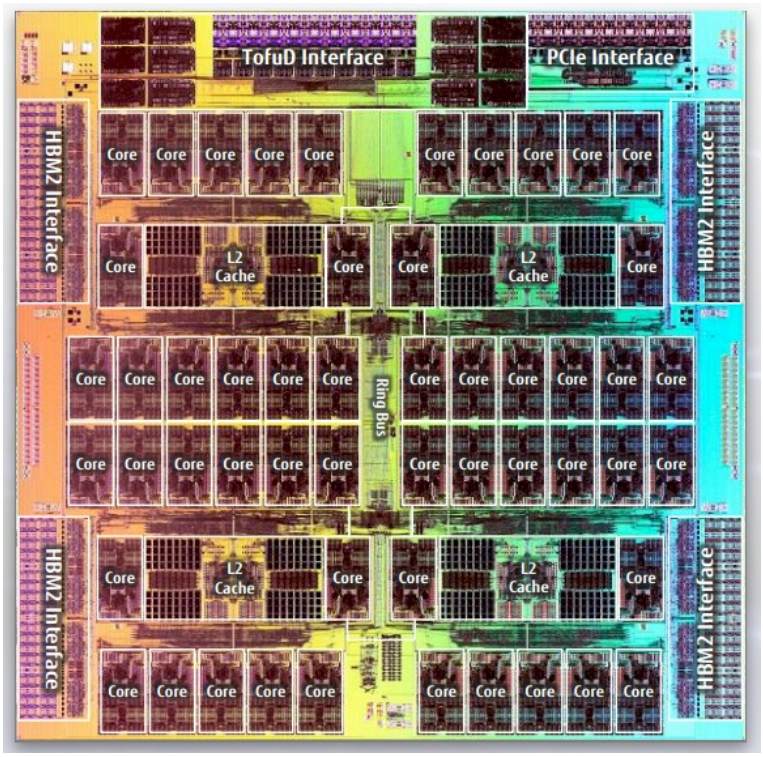


FIGURE 1.8 – Un processeur A64FX de 2019 (jeu d'instruction ARM) conçu par Fujitsu pour la machine « Fugaku ». 52 coeurs (48 pour les applications, 4 pour l'OS), 32Mo de cache, contrôleur réseau (TofuD) et contrôleur mémoire HBM2 intégré. (image : Fujitsu)

En 1997, Intel livre l'ASCII red, avec plus de 9000 processeurs Pentium Pro. C'est la première machine à atteindre 1 TeraFLOPS.

En 2008, `road runner` conçu par IBM atteint 1 PetaFLOPS. Composé de 296 racks, il contient 12 960 processeurs IBM PowerXCell 8i (contenant 1 gros coeur Power PC et 8 mini-coeurs) et 6 480 AMD Opteron à deux coeurs. Coût : ≈ $100 millions.

En 2011, le `K computer` japonais atteint 10 PetaFLOPS. Il contient 88 128 processeurs SPARC64 VIIIfx à 2Ghz et 8 coeurs, répartis dans 864 racks (ça fait 705 024 coeurs).

1.4 Architectures parallèles

Les sections précédentes démontrent qu'il existe une grande variabilité dans l'architecture des machines parallèles. Différents niveaux de parallélisme existent : à l'échelle de la machine (plusieurs noeuds), à l'intérieur d'un noeud (plusieurs processeurs), à l'intérieur d'un processeur (plusieurs coeurs), à l'intérieur d'un coeur (plusieurs threads et/ou plusieurs unités d'exécutions).

Pour chaque niveau, il convient de savoir combien d'éléments de calcul sont présent ? Quelle est leur puissance ? Sont-ils homogènes ? De quelle mémoire disposent-ils ? Comment sont-ils reliés les uns aux autres ? Comment synchronisent-ils leurs efforts ?

Classification de Flynn

En 1966, Michal J. Flynn (1934–) a proposé une classification des architectures parallèles qui est restée depuis. Grosso-modo, il s'agit de distinguer le parallélisme de contrôle d'une part (SI = Single Instruction / MI = Multiple Instruction) d'une part, et le parallélisme de données d'autre part (SD = Single Data / MD = Multiple Data).

		Flot de données	
		unique	multiple
Flot d'instructions	unique	SISD	SIMD
	multiple	MISD	MIMD

Les architectures SISD sont « purement séquentielles » : un processeur exécute un flot d'instruction, et chaque instruction opère sur une seule donnée.

Dans les architectures SIMD, les unités de traitement (« *Processing Units* », PU) exécutent simultanément la même opération sur des données différentes. Par exemple, la même instruction effectue plusieurs (2, 4, 8, 16, ...) additions flottantes simultanément sur des « vecteurs ». Ceci est en fait très répandu : un GPU haut-de gamme (style NVIDIA V100) contient 80 « coeurs », qui sont en fait de grosses unités SIMD capable de traiter 32 `double` / 64 `float` simultanément. Les processeurs habituels contiennent aussi des unités SIMD : instructions SSE (128 bits, 4× `float`, puis AVX2 (256 bits, 8× `float`) et maintenant AVX-512 (16× `float`). Les processeurs de smartphone ont aussi des instructions SIMD NEON (128 bits, 4× `float`). Tout ceci avait été précédé par des machines dites « vectorielles » (CRAY, NEC) qui dominaient le monde du HPC dans les années 1970–1990 : les processeurs pouvaient traiter des vecteurs entiers en une seule instruction. Après être un peu tombé en désuétude, cela revient à la mode !

Les architectures MISD (plusieurs instructions différentes appliquées aux mêmes données) sont assez rare, surtout sous cette forme pure. Mais les *systolic arrays* peuvent être vu de la sorte.

Les architectures MIMD sont très courantes : des processeurs indépendants peuvent effectuer différentes opérations sur différentes données simultanément. Ils ont un fonctionnement asynchrone (chaque processeur est autonome et gère son propre flux de donnée). Par exemple : les processeurs multicoeur, un cluster de PC, etc.

Quand on se pose la question en terme de « technique de programmation », on est amené à envisager les différents types de systèmes en deux grandes catégories :

machines à mémoire partagée
: Les différents unités de traitement ont accès à la même mémoire (elles sont probablement connectées à un même bus, et sont synchronisées). Un seul système d'exploitation fonctionne sur l'ensemble des processeurs, alloue la RAM, etc.

  Les différents processus d'une même application parallèle possèdent un espace mémoire commun, des variables globales. Ceci leur permet (ou leur évite, selon le point de vue) de communiquer explicitement.

machines à mémoire distribuée
: Les unités de traitement possèdent chacune leur propre mémoire séparée, et exécutent probablement chacune leur propre système d'exploitation.

  Le seul moyen de communication disponible est le réseau. Il est géré en partie par l'OS (UDP, TCP/IP, etc.) et en partie par des librairies qui appartiennent à l'utilisateur (*middleware*). La synchronisation, l'accès aux ressources, la répartition de la charge, ne sont pas gérées par l'OS et doivent être assurées par l'application parallèle.

Machines à mémoire partagée

Par bien des aspects, les machines à mémoire partagée sont plus faciles à programmer. Pas besoin de s'embêter avec une gestion du réseau, des adresses IP, des liens réseaux qui tombent en panne, des machines qui ne vont pas toutes à la même vitesse, qui ne sont pas toutes à l'heure (!), etc.

Par contre, si plusieurs processeurs ont accès à la même mémoire, cela ouvre la porte à des conflits d'accès (*race conditions*) : lorsqu'au moins deux processeurs tentent d'accéder simultanément à la même adresse mémoire et qu'au moins un de ces accès est une écriture, le résultat est mal défini. Afin d'éviter ces conflits, la synchronisation entre les différents processus doit est assurée par des mécanismes explicites, typiquement des verrous ou des sections critiques qui sont en partie gérés par l'OS.

L'inconvénient des machines à mémoire partagée, c'est qu'il est difficile de les faire grossir au-delà d'une certaine limite (la plus grosse dont j'ai entendu parler, une Silicon Graphics Altix UV, aurait 2048 processeurs — c'est-à-dire rien du tout face aux dizaine/centaines de milliers de processeurs des plus grosses machine de HPC). En effet, tous les processeurs sont « en compétition » pour l'accès aux données en RAM.

Pourquoi les machines à mémoire partagée ne passent pas à l'échelle ? L'accès des processeurs à la RAM fonctionne *grosso modo* de la façon suivante. Le processeur est relié au contrôleur de la RAM par un bus. Lorsqu'il veut accéder au contenu d'une adresse spécifique en RAM, le processeur écrit l'adresse requise sur le bus, et envoie un « ping » au contrôleur sur le bus. Le contrôleur lit l'adresse, récupère le contenu, écrit ensuite la valeur lue en RAM sur le bus, et envoie un signal « pong » sur le bus. A la réception du signal, le processeur lit la valeur sur le bus.

Si toute la RAM et tous les processeurs sont reliés au même bus, alors seul un processeur peut utiliser le bus à chaque instant, et tous les autres doivent attendre qu'il ait fini. Plus le nombre de processeur est élevé, et plus le temps d'attente pour la RAM augmente. Cette technique est donc limitée à un petit nombre de processeurs (typiquement de 2 à 4).

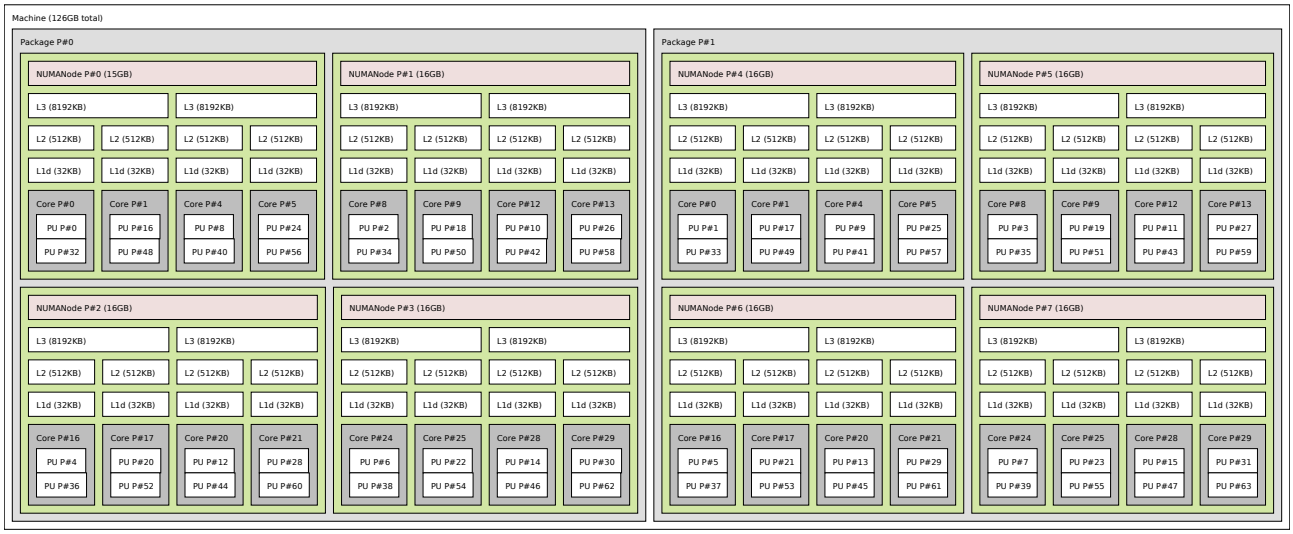


FIGURE 1.9 – Un noeud (du cluster chiclet de GRID5000) contenant deux processeurs AMD EPYC 7301. Chaque processeur contient quatre « paquets » de quatre coeurs adossés à un contrôleurs mémoire indépendant, connecté à une partie des barrettes de RAM. Chaque processeur contient 4 « zones NUMA » et l'ensemble de la machine en contient 8. Ci-dessous, la matrice des « distances » entre zones NUMA (plus la distance est grande, plus l'accès est lent).

node	0	1	2	3	4	5	6	7
0	10	16	16	16	28	28	22	28
1	16	10	16	16	28	28	28	22
2	16	16	10	16	22	28	28	28
3	16	16	16	10	28	22	28	28
4	28	28	22	28	10	16	16	16
5	28	28	28	22	16	10	16	16
6	22	28	28	28	16	16	10	16
7	28	22	28	28	16	16	16	10

Archuitecture NUMA Une autre solution consiste à relier directement chaque processeur à une petite partie de la RAM, par un bus qui lui est propre. Cette partie de la RAM est alors entièrement sous son contrôle. Pour accéder au reste de la RAM, il doit en « négocier » l'accès avec les autres processeurs, via des bus ou des connexions directes. L'accès au reste de la RAM est alors plus lent que l'accès à sa « région propre ». Cette architecture est appelée « *Non-Uniform Memory Access* (NUMA) ». Elle est très répandue actuellement (un des avantages est que le contrôleur mémoire peut être incorporé au processeur, rendant la communication très rapide). Cf. fig. 1.9. Mais en fait, si on regarde ça de loin, on se rend compte que les architectures NUMA sont un moyen de *simuler une machine à mémoire partagée avec une machine à mémoire distribuée* (et un réseau très, très rapide).

Pour de bonnes performances, il est alors critique que l'OS soit conscient de la topologie de cette hiérarchie mémoire, qu'il ne déplace pas un processus d'un CPU à l'autre (sinon le processus se trouve éloigné de « ses » données), et qu'il alloue de mémoire de manière judicieuse (ne pas donner à un processus de la mémoire « lointaine »). Cela pose des problèmes d'ordonnancement des tâches plus compliqués. Certains algorithmes peuvent être optimisés pour ces architectures. Mais de toute façon, la *localité* des accès mémoire est un aspect important du HPC, mémoire uniforme ou pas.

En 2020, un noeud SMP est quasi-systématiquement de type NUMA.

Partager la mémoire entre plusieurs processeurs pose des problèmes assez compliqués au niveau du matériel, notamment car il faut plus ou moins synchroniser leurs caches : si le processeur A modifie ce qui se trouve à une certaine adresse en mémoire, et que ces données étaient répliquées dans le cache du processeur B, alors il faut que B mette à jour (ou invalide) son cache — mais pour ça il faut qu'il soit conscient de l'écriture du processeur A !

Si un bus mémoire est partagé, alors les uns et les autres peuvent « espionner » les écritures qui ont lieu sur le bus et invalider automatiquement leur cache. Mais dans les machines NUMA, c'est plus compliqué ! Il faut que les processeurs communiquent pour s'informer de leurs écritures mutuelles dans certains cas.

Machines à mémoire distribuée

Dans les machines à mémoire distribuée (toutes les « grosses » machines parallèles décrites précédemment), le réseau joue un rôle indispensable pour relier entre eux les différentes unités de traitement, leur permettre de coopérer et les synchroniser.

Les caractéristiques (débit, latence) et la topologie du réseau jouent donc un rôle central dans le niveau de performance atteint *in fine* par les applications parallèle : communiquer prend du temps, or ce temps est un surcoût qu'il faut réduire au minimum. Les machines de HPC ont des réseaux « haut-de-gamme » (Infiniband, OmniPath, etc., alors TCP/IP et ethernet sont bons pour le « commun des mortels »...), avec des débits élevés mais surtout avec des latences très faibles.

On rencontre des réseaux en arbre (« fat tree »), des hypercubes, des grilles (ou des tore) à plusieurs dimensions (5 dimensions pour IBM BlueGene/Q, 6 dimensions pour `K computer` et `Fugaku`). Avantage du mesh/tore : pas de point central de congestion. Inconvénient : nécessité d'un protocole de routage. Sur `turing`, la BlueGene/Q du CNRS, on fait le test suivant : 4096 noeud envoient chacun 1.8Mo à chacun des autres noeud (27.5To sont donc transmis en tout). Temps total : 18.7s, donc le réseau en tore 5D assure le transport de ≈ 1.5To/s.

Comme les performances du réseau sont très critiques, sur les machines les plus sophistiquées les contrôleurs réseaux sont parfois placés directement sur les processeurs (IBM BlueGene, Fujistu « K computer » et Fugaku, ... Les processeurs Intel Xeon Platinum ont un contrôleur OmniPath intégré). Les données partent et arrivent directement des mémoires les plus rapides (les caches), en « tâche de fond » pendant que le reste des calculs ont lieu.

La programmation de machines à mémoire distribuées se fait principalement par l'utilisation de bibliothèques de passage de messages (par ex. MPI) ou par l'utilisation d'extensions des langages (par ex. les `coarray` de Fortran) qui masquent une partie de la complexité.

Parallélisme au niveau des instructions

Un processeur réputé séquentiel contient en réalité un certain degré de parallélisme, même si les programmeurs ne peuvent pas le contrôler directement. Un exécutable (« binaire »), contient une liste (ordonnée) d'instructions exécutées par le processeurs. Par exemple (en assembleur PowerPC) :

```
slwi 10,9,3
add 8,11,10
lwzx 10,11,10
lwz 7,4(8)
or. 10,10,7
bne 0,.L146
addi 5,5,8
stw 3,0(8)
stw 4,4(8)
cmplw 7,6,5
bne 7,.L24
lwz 9,144(19)
li 10,1
stw 10,20704(31)
addi 9,9,1
stw 9,144(19)
```

Les processeurs offrent la garantie que tout se passe comme si ces instructions étaient exécutées les unes à la suite des autres. Mais en fait, pour des raisons de performance, des formes de parallélisme existent à l'intérieur de ce processus en apparence séquentiel.

Sans rentrer dans les détails, les processeurs contemporains, même les plus rudimentaires, ont un *pipeline* (par exemple, le pipeline RISC classique à 5 étapes : Instruction fetch ; Instruction decode ; Execute ; Memory access ; Writeback. Les processeurs modernes ont parfois des pipelines plus long, avec 10+ étapes). Une instruction met plusieurs cycles à s'exécuter, mais le processeur peut en « commencer » une à chaque nouveau cycle, et la faire avancer dans le pipeline. À un instant donné, il peut y avoir plusieurs instructions « en vol » dans le pipeline, donc plusieurs instructions en cours d'exécution.

Un processeur est « scalaire » s'il ne peut exécuter qu'une seule instruction à chaque cycle. Les processeurs « superscalaires » peuvent en exécuter plusieurs (si elles sont compatibles). Les unités d'exécutions sont alors parfois répliquées. Le processeur affecte dynamiquement les instructions aux différentes unités d'exécutions

(dont le nombre n'est pas connu à l'avance par le programmeur et qui varie d'un modèle de processeurs à l'autre), et le tout est donc portable. Dans le monde de l'informatique grand public, les processeurs habituels sont superscalaires depuis l'Intel Pentium de 1993.

Par exemple, sur un coeur Intel Skylake il y a 4 ALU (*Arithmetic and Logic Unit*) capable d'exécuter les opérations simples sur les entiers : et, ou, xor, addition, comparaison. Dans l'ensemble, le coeur est capable d'exécuter jusqu'à 6 μ-instructions par cycle dans le meilleur des cas.

Il peut par ailleurs y avoir plusieurs pipelines. Par exemple, sur le processeur PowerPC A2 (qu'on trouve dans l'IBM BlueGene/Q), il y a deux pipelines parallèles : un pour les opérations entières, un pour les flottantes.

Pour obtenir les performances maximales des unités d'exécutions qui sont présentes dans le processeur, il faut être capable de leur fournir du travail en permanence, et de les utiliser simultanément (donc de faire des choses en parallèle), malgré le fait que le flux d'instructions à exécuter soit « séquentiel ». Une des difficultés potentielle est que certaines instructions peuvent ne pas être « prêtes » lorsque c'est leur tour, parce qu'il faut lire la mémoire (ce qui est lent) ou parce qu'elles dépendent du résultat d'une instruction précédente qui n'est pas encore terminée, etc. Pour éviter ces temps morts, les processeurs modernes peuvent exécuter les instructions du programme dans un ordre différent de celui qui est spécifié, tout en garantissant la sémantique du code fourni. C'est la *Out of Order Execution*. Ceci repose essentiellement sur un algorithme [3] inventé en 1967 par Robert Tomasulo (1934–2008), un ingénieur d'IBM. Un coeur Intel Skylake peut gérer jusqu'à 224 μ-instructions simultanément dans un *Reorder Buffer* qui peut les permuter ; de là elles sont transmises vers un ordonnanceur qui peut en gérer 97 simultanément et qui a la charge de les envoyer, potentiellement en parallèle, vers l'un des sept « ports » auxquels sont reliées les unités d'exécution.

Exploiter à fond l'ILP nécessite d'ordonnancer soigneusement les instructions soumises au processeur pour exploiter à fond les unités d'exécutions. Ceci implique d'éviter les blocages éventuels (instructions incompatibles simultanément, dépendances de données, etc.). C'est plutôt la tâche des compilateurs, mais il faut parfois modifier un peu les algorithmes pour en tirer partie.

Certains processeurs (les Intel Itanium) utilisent une technique de parallélisme explicite où le compilateur doit explicitement former des groupes d'instructions qui peuvent être exécutées simultanément, pour augmenter l'ILP. Ceci place la complexité sur les compilateurs et en retire aux processeurs — ça n'a pas été un succès commercial.

Dans les processeurs superscalaires, le nombre d'unité d'exécution est augmenté (pour pouvoir traiter plusieurs instructions simultanément). Mais du coup, elles ne sont pas forcément toutes actives simultanément. Le *simultaneous multi-threading* (SMT) permet éventuellement d'augmenter leur degré d'utilisation. L'idée consiste à mutualiser les unités d'exécution entre plusieurs flux d'instructions parallèles (=threads matériels). Il peut y avoir un gain, ou pas. Si un seul thread sature les multiplicateurs, par exemple, en rajouter un autre qui fait lui aussi des multiplications ne sert à rien. Par contre, si l'autre thread fait d'autres opérations, les deux peuvent éventuellement progresser en parallèle. D'autre part, si un thread est bloqué (attente de données depuis la RAM par exemple), alors l'autre peut progresser. Avec 2 threads matériels, on observe souvent des gains de 0–30%.

Le SMT peut également être utilisé pour amortir la latence de certaines opérations. Un coeur du processeur PowerPC A2 gère 4 threads matériels. Il n'est pas superscalaire : il ne peut exécuter qu'une seule instruction par cycle, mais elle peut venir de n'importe lequel des 4 threads (deux instructions peuvent être exécutées en réalité : une entière et une flottante car ce sont deux pipeline distincts). Les threads progressent équitablement. Par contre, lire des données depuis le cache L1 bloque un thread pendant 4 cycles. Mais du coup, pendant ce temps-là, les autres threads matériels peuvent progresser. Il suffit d'exécuter une seule instruction sur chacun des trois autres pour que le premier puisse reprendre et que la latence de l'accès mémoire soit « masquée ». Ainsi, avec 4 threads, on peut atteindre 1 instruction / cycle (le maximum possible), même avec des accès mémoire qui prennent 4 cycles.

1.5 Parallélisme dans les applications

Autant exploiter le parallélisme au niveau des instructions est l'affaire des processeurs et des compilateurs, autant exploiter les autres niveaux de parallélisme (SIMD, entre threads, entre noeuds) est l'affaire des programmeurs et des concepteurs d'algorithmes.

Une application parallèle doit définir des « tâches » qui ont vocation à être pouvoir être exécutées simultanément. Il faut que les outils de programmation permettent aux programmeurs de définir ces tâches et que les environnements d'exécution les déploient et les exécutent de manière efficace.

Il faut donc parvenir à découper un problème initial (« simuler l'évolution du climat ») en tâches plus ou moins indépendantes. Parfois on peut choisir le nombre et le temps de traitement des tâches (beaucoup de toutes

petites tâches ou peu de tâches très grosses). En pratique, il faut faire des compromis : il faut créer assez de tâches pour occuper l'ensemble des ressources de calcul disponibles ; mais gérer un nombre astronomique de tâches entraîne un surcoût (par exemple, créer un thread logiciel n'est pas gratuit) ; les tâches doivent être synchronisées et communiquer entre elles, donc plus il y en a, plus ce sera coûteux ; enfin, si les tâches sont de tailles irrégulières et qu'il y en a trop peu, la charge de travail risque d'être mal répartie sur les différentes unités de calcul, etc.

Bref, il faut choisir soigneusement la *granularité* du calcul parallèle. Pour reprendre l'exemple du produit de matrice :

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

On voit que toutes les itérations de la première boucle `for` pourraient être effectuées « en parallèle » (les lignes de C peuvent être calculées indépendamment les unes des autres). Ceci donnerait n tâches de taille $\approx n^2$ opérations. Mais en fait, les itérations des *deux* boucles externes pourraient être accomplies en parallèle (chaque coefficient C_{ij} peut être calculé indépendamment). Ceci donnerait alors n^2 tâches de $\approx n$ opérations. On note enfin que les itérations de la boucle interne accèdent toutes à C_{ij}, donc il faut modifier l'algorithme si on veut pouvoir les exécuter en parallèle elles aussi (ce qui est possible).

Découper un gros calcul en tâche repose généralement sur la combinaison de deux types de parallélisme : le parallélisme de données et le parallélisme de contrôle.

Parallélisme de données Il s'agit d'appliquer le même traitement sur des données différentes d'une tâche à l'autre. C'est ce qu'on a fait implicitement dans l'exemple du produit de matrice ci-dessus : le même code peut calculer C_{ij} pour tout (i, j), mais en lisant des données différentes qui dépendent de i, j (la i-ème ligne de A et la j-ème colonne de B).

Certaines techniques de rendu d'image de synthèse (le ray-tracing par exemple) permettent de calculer la couleur de chaque pixel de l'image finale séparément. L'image finale peut alors être découpée en petits carrés, et le rendu de chaque carré constitue une tâche indépendante.

Très souvent, lorsqu'il s'agit de simuler numériquement un phénomène physique complexe, on peut partitionner l'espace (par exemple, l'atmosphère terrestre en petits cubes). Ceci permet de découper la simulation globale en un grand nombre de tâches qui consistent chacune à effectuer la simulation à l'intérieur des partitions de l'espace. Il faut généralement alors que les tâches de synchronisent avec leurs « voisines », pour partager leurs états respectif aux frontières communes de leur petites zones. Pour cette raison, le parallélisme de donnée est parfois appelé « *domain decomposition* ».

Parallélisme de contrôle Il s'agit de tâches indépendantes effectuant des traitements différents. Par exemple, dans un jeu vidéo on peut imaginer qu'il faut préparer la bande son, calculer les prochaines actions des personnages virtuels, simuler des phénomènes physiques et effectuer un rendu graphique. Toutes ces tâches, qui doivent être faites dans des délais contraints, sont de nature différentes.

Pour donner un autre exemple, plus « académique » :

```
1   u = 1;
2   v = 2;
3   w = sqrt(u*u + v*v);
4   z = cos(u) - sin(v);
5   r = z / w;
```

Les lignes 3 et 4 peuvent constituer deux tâches exécutables en parallèle. Il s'agit pourtant d'opérations différentes.

Calculs intrinsèquement séquentiels Il y a aussi des problèmes calculatoires qui se prêtent mal, voire pas du tout, à la parallélisation.

Sur un plan théorique, les problèmes qui sont complets pour la classe de complexité **P** en font partie. Pour ceux-là, l'humanité ne connaît pas à ce stade d'algorithme dont le temps d'exécution soit polylogarithmique (c.a.d. un polynôme en $\log n$), même avec autant de processeurs qu'on veut.

Un exemple bien connu de problème « fortement séquentiel » est le problème du flot dans un graphe : étant donné G dont les arêtes sont étiquetées par des capacités positives, peut-il circuler au moins t unités de flot du sommet u au sommet v ? Un autre exemple est le calcul du PGCD : en effet, dans l'algorithme d'Euclide, il peut y avoir $\Omega(n)$ iterations dans le pire des cas, où n désigne le nombre de bits des arguments (ceci se produit notamment si on calcule le PGCD de deux nombres de Fibonacci successifs).

Ces problèmes intrinsèquement séquentiels ont par exemple des applications en cryptographie (ce sont les « *time-lock puzzles* [2]) ! En effet, posséder des ordinateurs massivement parallèles *n'aide pas* à les résoudre. Ceci permet de garantir qu'ils ne seront pas résolus dans le futur proche. Sans rentrer dans les détails, calculer $x^{2^t} \bmod n$ nécessite t étapes successives avec l'algorithme standard d'exponentiation modulaire rapide, donc c'est très long si t est très grand — la feinte, c'est que si on connaît la factorisation de n, alors on peut faire le calcul en temps $\mathcal{O}(\log t)$ au lieu de $\mathcal{O}(t)$.

On peut quantifier le *degré de parallélisme* disponible dans un algorithme donné : c'est le nombre maximum de tâches que l'on peut effectuer en parallèle.

1.5.1 Évaluation des performances

Dans le monde séquentiel, on compare les algorithmes entre eux sur la base du nombre d'opérations élémentaires qui sont nécessaires à leurs exécutions. Le tri-fusion est « plus rapide » que le tri à bulle car le premier nécessite $\mathcal{O}(n \log n)$ opérations pour trier un tableau de taille n, tandis que le second nécessite $\mathcal{O}\left(n^2\right)$. Dès que n devient assez grand, le nombre d'opérations est plus faible dans le premier cas. De même, il est bien connu que le « bon » algorithme séquentiel pour calculer le maximum d'un tableau nécessite $\mathcal{O}(n)$ opérations (et a vrai dire on ne peut pas descendre sous cette borne, car il faut bien *lire* au moins une fois chacune des n cases du tableau).

La complexité d'un algorithme séquentiel est donc la fonction $T(n)$ qui donne le nombre d'opérations nécessaires au traitement d'une entrée de taille n. Dans le monde séquentiel (et si on tolère des approximations), le *nombre d'opérations* s'identifie à peu près au au *temps* nécessaire à l'exécution de l'algorithme.

Ceci n'est malheureusement plus vrai si on dispose de plusieurs processeurs, car le temps d'exécution dépend alors du nombre de processeurs disponibles. On aimerait bien qu'un algorithme dont l'exécution nécessite 1000 opérations sur un seul processeur puisse être exécuté en 50 unités de temps si on disposait de 20 processeurs. Dans le monde de l'algorithmique parallèle, le juge de paix est l'*horloge murale* (« *wall-clock time*) : on prend un chronomètre et on mesure le temps que la machine met à effectuer le calcul.

Ce qui complique tout, c'est que rien n'interdit que le nombre d'opérations effectué par un algorithme parallèle dépende du nombre de processeurs disponibles. C'est d'ailleurs presque toujours le cas en pratique, vu la façon dont les programmes parallèles sont exécutés.

On s'intéresse donc au *gain* réalisé par le passage à un algorithme parallèle. La question du *passage à l'échelle* (*scalability*) est donc : l'algorithme (ou le programme) reste-t-il efficace lorsque le nombre de processeurs augmente ?

Cependant, on peut dire des choses simples. Notons $T_1(n)$ le temps d'exécution du meilleur algorithme séquentiel pour résoudre un problème donné, et notons $T_p(n)$ le temps d'exécution d'un algorithme parallèle sur p processeurs pour des entrées de taille n (temps mesurés en secondes avec un chronomètre). L'*accélération* (*Speedup*) obtenue par la parallélisation est :

$$S(n,p) = \frac{T_1(n)}{T_p(n)}.$$

L'*efficacité* (*efficiency*) obtenue par la parallélisation est :

$$E(n,p) = \frac{S(n,p)}{p}.$$

1.5.2 « *Strong Scaling* » et loi d'Amdahl

Partant d'un code ou d'un algorithme séquentiel, et qu'on souhaite paralléliser, on peut se donner plusieurs objectifs. Le plus ambitieux d'entre eux, le « *strong scaling* (« extensibilité forte ») consiste à obtenir une accélération aussi proche que possible de p (donc une efficacité proche de 1), lorsque p augmente et que n reste fixé (« avec p processeurs, ça va p fois plus vite »)

C'est souvent difficile à atteindre, et généralement on a $S(n,p) \ll p$ (et donc $E(n,p) \ll 1$), car d'une part la parallélisation entraîne des surcoûts (communications, synchronisation, opérations supplémentaires, attente). et d'autre part il peut être difficile de maintenir tous les processeurs occupés à 100% tout le temps.

De plus, lorsque p augmente, l'accélération ne peut pas augmenter indéfiniment : A l'extrême, ça ne sert à rien d'avoir un million de processeurs pour calculer le maximum de deux entiers. Au bout d'un moment, les processeurs excédentaires vont devenir inutiles.

Le nombre total d'opération effectué par un algorithme parallèle est nommé le *travail* $W(n)$ — on admet un instant que ceci ne dépend pas de p. On peut se dire que l'exécution sur p processeurs nécessitera *au moins* $W(n)/p$ unités de temps. Le temps d'exécution d'un algorithme parallèle avec *autant de processeurs qu'on veut* est appelé sa *profondeur* $T(n)$. Son exécution nécessitera au moins $T(n)$ unités de temps, quel que soit le nombre de processeurs disponibles. Dans de très nombreux cas, on aura $T(n) \geq \log_2 n$.

Sur une machine à mémoire partagée, il est plus ou moins possible d'exécuter un algorithme parallèle de travail $W(n)$ et de profondeur $T(n)$ en $\mathcal{O}(T(n) + W(n)/p)$ unités de temps. Cette observation découle du raisonnement suivant, le « *principe d'ordonnancement de Brent* ». Si la profondeur de l'algorithme est $T(n)$, alors l'algorithme se décompose en $T(n)$ « étapes », au sein desquelles toutes les opérations peuvent être effectuées en parallèle. Disons que la i-ème étape contient $W_i(n)$ opérations faisables en parallèle. Le travail total de l'algorithme est donc : $W(n) = W_0(n) + W_1(n) + \cdots + W_{T(n)}(n)$. Si on dispose de p processeurs, et qu'on répartit convenablement les $W_i(n)$ opérations à effectuer sur les processeurs, il va falloir $W_i(n)/p$ unités de temps pour accomplir cette i-ème étape. Comme la somme des $W_i(n)$ donne le nombre total d'opérations à effectuer, on obtient ce qui a été annoncé.

Si $T(n) \approx \log_2 n$, alors utiliser plus de $n/\log n$ processeurs ne peut pas améliorer l'accélération. Toute la difficulté, en pratique, consiste à « répartir » les opérations de chaque étape sur les processeurs disponibles.

Un autre problème, c'est que la plupart des programmes parallèles ont des portions séquentielles, où un seul processeur est actif. Ceci limite forcément le *strong scaling*.

Considérons un algorithme séquentiel, et notons f la fraction intrinsèquement séquentielle (non parallélisable) des calculs effectués par l'algorithme. La *loi d'Amdahl* affirme que l'accélération maximale sur p processeurs de l'algorithme parallèle correspondant est :

$$S(n,p) \leq \frac{1}{f + (1-f)/p}.$$

Par exemple : si 20% d'un algorithme n'est pas parallélisable, l'accélération est alors limitée à 5, avec autant de processeurs qu'on veut.

Démonstration

(Cette preuve est tirée du livre de Quinn). Notons $\sigma(n)$ la portion du calcul inhéremment séquentielle et $\phi(n)$ la portion parallélisable.

Un programme séquentiel va prendre un temps $\sigma(n) + \phi(n)$. Dans le cas le plus favorable possible, la partie parallélisable du calcul peut être parfaitement répartie entre les p processeurs. Le temps de calcul parallèle est donc minoré par $\sigma(n) + \phi(n)/p$. Notons $f = \sigma(n)/(\sigma(n) + \phi(n)$ la fraction inhéremment séquentielle du calcul. On note que $\phi(n) = \sigma(n)(1/f - 1)$. Alors :

$$\begin{aligned} S(n,p) &\leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n)/p} \\ &\leq \frac{\sigma(n)/f}{\sigma(n) + \sigma(n)(1/f-1)/p} \\ &\leq \frac{1/f}{1 + 1(1/f-1)/p} \\ &\leq \frac{1}{f + 1(1-f)/p}. \end{aligned}$$

La loi d'Amdahl, due à Gene Amdahl (1922–2015), ingénieur chez IBM, est en réalité *très optimiste* car elle néglige le surcoût éventuel lié à la parallélisation (temps passé dans les communications, les synchronisations, ...). En réalité l'accélération observée sera inférieure à la borne que la loi fournit.

1.5.3 « *Weak Scaling* » et loi de Gustafson-Barsis

Le *strong scaling* contient l'idée que le but de la parallélisation est de réduire le temps d'exécution : on considère la taille du problème comme une constante et on observe comment le temps d'exécution diminue quand le nombre de processeurs augmente.

Mais la parallélisation peut avoir d'autres buts, et notamment celui de pouvoir traiter des problèmes plus gros dans un temps donné. C'est typiquement le cas des simulation climatiques qui visent à obtenir des prévisions

météorologiques : elles doivent s'exécuter en un temps contraint (probablement moins de 24h). Avec l'augmentation de la puissance de calcul, on ne cherche pas vraiment à raccourcir cette durée, mais surtout à augmenter la finesse du modèle et la précision des résultats obtenus.

On observe empiriquement que, à nombre de processeurs fixés, l'accélération augmente généralement avec n : c'est l'*effet d'Amdahl*. L'idée sous-jacente c'est qu'augmenter n fait augmenter la portion parallélisable du calcul plus vite que la fraction séquentielle et les éventuels surcoûts.

Le *Weak Scaling* (« extensivité faible ») consiste à faire augmenter n et p ensemble, avec l'objectif de traiter des problèmes plus gros tout en maintenant un temps d'exécution constant.

Considérons cette fois un programme parallèle, et notons s la fraction de son temps d'exécution passé dans du code séquentiel. Alors la *loi de Gustafson-Barsis* affirme que la meilleure accélération possible est :

$$S(n,p) \leq p + (1-p)s.$$

(la loi d'Amdahl borne l'accélération à partir d'un algorithme séquentiel, tandis que celle de Gustafson-Barsis part d'une exécution d'un algorithme parallèle et estime à quel point ceci *serait théoriquement* plus rapide que d'utiliser un seul processeur, si c'était possible).

Par exemple, si un programme s'exécute en 220s sur 64 processeurs, et qu'on mesure qu'il passe 5s dans une portion séquentielle, alors on peut estimer qu'il sera 60.85 fois plus lent sur un seul processeur — à condition que ce soit possible de l'exécuter sur un seul processeur, car avec plus de processeurs on a aussi plus de mémoire.

Démonstration

(Cette preuve est tirée du livre de Quinn) Par définition, s est la fraction du temps de calcul parallèle passée dans la section séquentielle, et $1-s$ est la fraction du temps de calcul passé à faire des opérations parallèles. On a donc :

$$s = \frac{\sigma(n)}{\sigma(n) + \phi(n)/p},$$
$$1 - s = \frac{\phi(n)/p}{\sigma(n) + \phi(n)/p}.$$

Par conséquent :

$$\sigma(n) = s(\sigma(n) + \phi(n)/p),$$
$$\phi(n) = (1-s)p(\sigma(n) + \phi(n)/p).$$

et

$$\begin{aligned} S(n,p) &\leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n)/p} \\ &\leq \frac{(\sigma(n) + \phi(n)/p)(s + p(1-s)}{\sigma(n) + \phi(n)/p} \\ &\leq s + p(1-s) \\ &\leq p + s(1-p). \end{aligned}$$

Dans des cas précis, on peut calculer à quelle vitesse n doit augmenter en fonction de p pour maintenir un niveau donné d'efficacité parallèle (on parle alors de « fonction d'iso-efficacité »).

1.6 Comment écrire des programmes parallèles ?

Une fois qu'on a conçu des algorithmes parallèle, il faut bien les coder. Plusieurs possibilités s'offrent au programmeur.

La parallélisation automatique On écrit l'algorithme dans un langage séquentiel « familier » (C, Fortran, ...) puis on fait confiance au compilateur pour 1) détecter qu'il y a des portions parallélisables, 2) « extraire » le parallélisme (définir des tâches exécutables simultanément et de tailles comparables), 3) introduire le code qu'il faut pour lancer ces tâches en parallèle, les synchroniser, etc. tout en maintenant la sémantique du code original.

En pratique, ça peut marcher sur des cas simples. La parallélisation est alors *implicite*.

Ajouter une couche parallèle à un langage séquentiel On écrit l'algorithme dans un langage séquentiel « familier » (C, Fortran, ...), et on ajoute des *directives* qui indiquent au compilateur ce qui est parallélisable et comment. C'est le compilateur qui génère tout seul le code qui sert à déployer les différentes tâches, mais c'est au programmeur de décider ce qui est parallèle ou pas. Par exemple, Cilk étend le langage C avec des boucles `parallel for` (toutes les itérations peuvent être faites en même temps), et un système de création de « tâches ». Les `co-array` qui ont été intégré à Fortran peuvent aussi être vus comme ceci.

En C et en Fortran, OpenMP est un standard industriel bien supporté par les compilateurs actuels, qui permet de paralléliser assez facilement des programmes même compliqués, en ajoutant de telles directives en commentaire.

Du coup, en ignorant ces directives on obtient un programme séquentiel. Il incombe au programmeur de bien comprendre comment paralléliser son programme, car le compilateur « ne fera qu'obéir aux ordres », même si ça donne un programme parallèle incorrect.

Utiliser des bibliothèques On écrit l'algorithme dans un langage séquentiel « familier », et on utilise des *fonctions externes* ou des bibliothèques pour exprimer le parallélisme. Par exemple, on peut utiliser les fonctions de la librairie `pthread` (standard POSIX) pour créer et synchroniser des threads logiciels. Les bibliothèques MPI (*Message-Passing Interface*) constituent un standard industriel universellement déployé dans les centres de calcul et universellement utilisé par les applications de HPC. Elles existent nativement en C et en Fortran.

Le programmeur doit non seulement faire apparaître explicitement le parallélisme, mais il doit aussi le mettre en oeuvre, donc il y a un surcoût de développement important. L'avantage, c'est que ça repose sur des compilateurs courants et éprouvés pour des langages séquentiels normaux.

Inventer des langages nouveaux Mais jusque-là, aucun ne s'est vraiment imposé et n'a réussi à simplifier la vie des programmeurs. Notons que `go`, le langage développé par Google, contient des fonctionnalités « natives » pour le parallélisme (canaux de communications, « goroutines » parallèles, etc.).

1.6.1 Le paradigme « *Single Program Multiple Data* »

Une manière répandue, et assez intuitive, d'exécuter un programme en parallèle sur plusieurs processeurs consiste à exécuter le même code partout (Single Program). Chaque « copie » du programme est identique. Par contre, chacune à accès à son *rang* (un numéro) qui est différent sur chacune d'entre elle, et le code peut donc adapter son comportement à son rang. Ceci permet d'exprimer aisément à la fois le parallélisme de données et le parallélisme de contrôle. Par exemple :

```
if (rank == 0) {
  <<Portion purement séquentielle>>
} else {
  <<Ne fait rien>>
}
```

Cette technique simple est largement utilisée dans le calcul scientifique avec MPI, OpenMP et sur les GPU.

Bibliographie

[1] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanovic. A 45nm 1.3ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference, Venice Lido, Italy, September 22-26, 2014*, pages 199–202. IEEE, 2014.

[2] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, USA, 1996.

[3] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1) :25–33, Jan 1967.

[4] Karel De Vogeleer, Gérard Memmi, Pierre Jouvelot, and Fabien Coelho. The energy/frequency convexity rule : Modeling and experimental validation on mobile devices. In Roman Wyrzykowski, Jack J. Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics - 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part I*, volume 8384 of *Lecture Notes in Computer Science*, pages 793–803. Springer, 2013.