Chapitre 8

Le « *Memory Wall* »

L'expression « *Memory Wall* » a été utilisée pour la première fois au milieu des années 1990 dans un article de recherche nommé : "*Hitting the Memory Wall : Implications of the Obvious*" [16]. Il commence par ces quelques mots :

> This brief note points out something obvious — something the authors "knew" without really understanding. With apologies to those who did understand, we offer it to those others who, like us, missed the point.

L'expression *Memory Wall*, qui a eu un grand succès, mets un nom sur un phénomène simple : la puissance de calcul des CPUs (typiquement le nombre d'opérations flottantes par seconde) augmente plus vite que le débit de la mémoire (le nombre de flottants qui peuvent être transférés de/vers la mémoire par seconde). La figure 8.1 montre la tendance depuis les années 1990.

En fait, l'écart augmente exponentiellement au cours du temps. Et l'augmentation du nombre de coeurs ne fait qu'aggraver le problème, car les coeurs se trouvent en concurrence les uns avec les autres pour accéder à la mémoire. Evidemment, ceci pose un problème : les CPUs modernes risquent de passer une fraction sans cesse plus grande de leur temps à *attendre* des données, plutôt qu'à faire des calculs utiles.

Ceci est encore aggravé par le fait que l'écart entre le nombre de FLOPS réalisable par les processeurs et la *latence* des accès mémoire augmente encore plus vite que l'écart avec le *débit* de la mémoire (cf. fig. 8.2).

Du coup, on est amené à distinguer des algorithmes qui sont *compute-bound* (les CPUs ne peuvent pas faire de FLOPs assez vite) de ceux qui sont *memory-bound* (les données n'arrivent pas assez vite de la mémoire pour alimenter les CPUs). Dans ces derniers, on pourrait chercher à distinguer ceux qui sont limités par le débit de la mémoire d'un côté, ou par sa latence de l'autre.

8.1 Le Matériel sous-jascent

Dans les ordinateurs contemporains, la mémoire dite « vive » (*Random-Access memory*, RAM) est caractérisée par le fait que ce n'est pas un stockage pérenne (quand on coupe le courant, les données sont perdues), que les opérations de lecture-écriture sont rapides et ne nécessitent pas de « préparation » coûteuse (comme le fait de pré-positionner une tête de lecture sur un disque, etc.)

Si on cherche « 16Go RAM » dans le moteur de recheche du site `amazon.fr`, on a le droit à 50 000+ suggestions, parmi lesquelles se trouvent les cinq produits suivants, tous de la même marque :

- Produit x 16Go (DDR4, 2666 MT/s, PC4-21300, Dual Rank ×4, ECC, Registered, DIMM, 288-Pin)
- Produit y 16Go (DDR4, 2666 MT/s, PC4-21300, Dual Rank ×8, Unbuffered, DIMM, 288-Pin)
- Produit z 16Go (DDR4, 2666 MT/s, PC4-21300, Single Rank ×4, ECC, Registered, DIMM, 288-Pin)
- Produit u 16Go (DDR4, 2666 MT/s, PC4-21300, Dual Rank ×8, ECC, Unbuffered, DIMM, 288-Pin)
- Produit v 16Go (DDR4, 2666 MT/s, PC4-21300, Dual Rank ×8, ECC, Registered, DIMM, 288-Pin)

Difficile de faire son choix ! Ce document n'est pas un guide d'achat, mais il décrit en partie à quoi correspondent toutes ces spécifications techniques et quelles influences elles peuvent exercer sur le domaine du calcul haute-performance.

Plusieurs familles de technologies permettent de réaliser de la RAM, notamment la SRAM (*Static* RAM, plus chère, plus rapide, moins dense) et la DRAM (*Dynamic* RAM, moins chère, plus lente, meilleure capacité, meilleure densité). La SRAM est typiquement utilisée dans les caches les plus rapides des processeurs (cf.

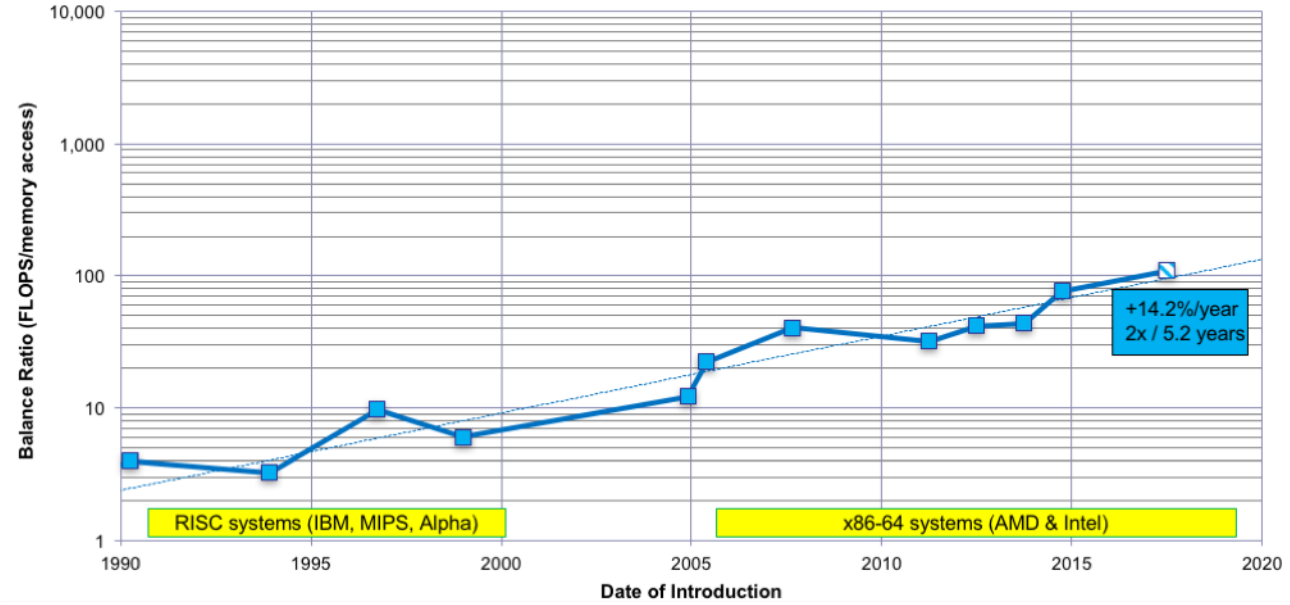


FIGURE 8.1 – Le « *Memory Wall* », partie 1. La courbe montre le nombre d'opérations sur des flottants que les processeurs ont le temps d'effectuer à chaque fois qu'un flottant peut être lu depuis la mémoire : cette quantité ne fait qu'augmenter. Autrement dit : la puissance de calcul augmente plus vite que le *débit* de la mémoire (image : John McCalpin).

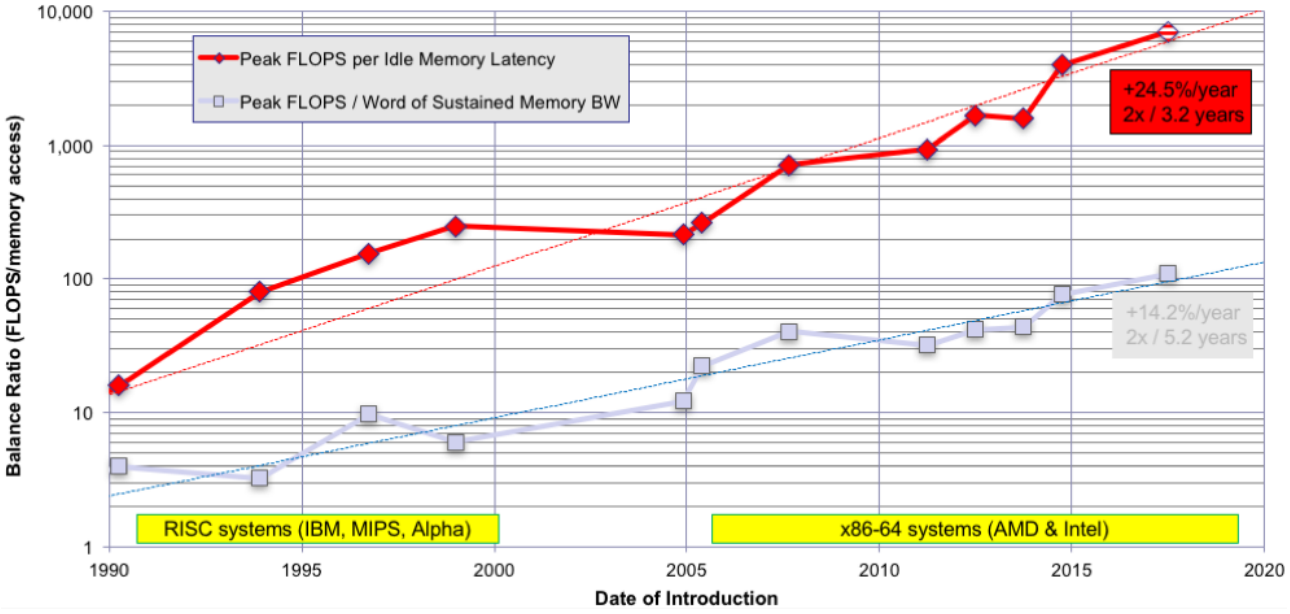


FIGURE 8.2 – Le « *Memory Wall* », partie 2. La puissance de calcul augmente *encore plus vite* que la *latence* de la mémoire. Et encore, il s'agit de la latence « au repos » (*idle latency*), et pas de la latence « en pleine charge » (*loaded latency*), qui est typiquement 3× ou 4× plus importante... (image : John McCalpin).

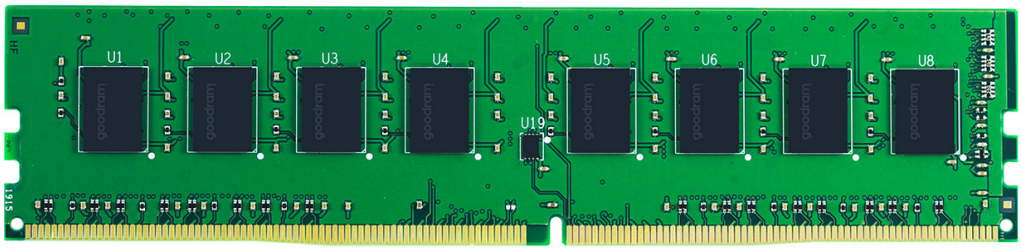


FIGURE 8.3 – Un DIMM de mémoire SDRAM DDR4 (largeur=13.5cm). On distingue 8 DRAM chips.

infra), tandis que la DRAM est utilisée pour former le gros de la RAM des ordinateurs normaux. Toute la RAM moderne est en fait constituée de SDRAM (« *Synchronous ...* ; elle est pilotée de manière synchrone par le contrôleur mémoire). Plus récemment, la mémoire HBM et HBM2 (« *High-Bandwidth Memory* ») s'est développée ; il s'agit de piles de puces DRAM empilées en 3D. Il y a aussi de la eDRAM (*embedded Dynamic* RAM, DRAM directement *dans* un processeur) qui est utilisée pour réaliser certains « gros » caches dans des processeurs tels que celui de la Playstation 2 (4Mo), des processeurs haut de gamme IBM, certains processeurs Intel avec carte vidéo intégrée, etc.

Les composants matériels qui composent la RAM (hors HBM et eDRAM) se présentent typiquement sous la forme de « barrettes » : ce sont les DIMMs (*Dual In-line Memory Module* — le « dual » vient du fait que les n connecteurs connecteurs du devant ne sont pas reliés à ceux du derrière, ce qui fait $2n$ connecteurs en tout). Voire fig. 8.3. Un DIMM contient des circuits intégrés, les *DRAM chip*s. Sur un DIMM donné, ils sont en principes tous identiques. Dans les ordinateurs portables, on met des SO-DIMM (*Small Outline DIMM*) plus compacts, mais c'est la même chose.

Avant les DIMMs, il y avait les SIMMs (Single ...), avec seulement 144 contacteurs répliqués des deux côtés. Ces derniers ont disparu dans les années 2000.

Les spécifications des modules de DRAM sont établies par le JEDEC (*Joint Electron Device Engineering Council*), un organisme de standardisation.

DDR Tous les produits mentionnés ci-dessus sont de la DRAM « normale ». Depuis l'an 2000, toute la mémoire DRAM est à « *Double Data Rate* » (DDR), c'est-à-dire que sur le *bus* qui la relie au contrôleur mémoire, il peut y avoir *deux* « transfert » par cycle (un lorsque le signal d'horloge « monte » et l'autre lorsqu'il « descend »). Un transfert est un paquet de données de 64 bits (c'est constant sur toutes les générations de SDRAM).

Il y a plusieurs générations technologiques successives : DDR, DDR2, DDR3, DDR4, ... Elles sont incompatibles, et les DIMMs ont des détrompeurs qui empêchent de les insérer dans des connecteurs d'une autre génération. Au sein d'une même génération, il y a des niveaux de performances standardisés (cf. fig. 8.4). Grosso-modo, un module de DDRx-y est capable d'effectuer y Mega-tranferts par seconde, pour une une bande passante de $8y$ Mo/s (puisqu'un « transfert » c'est 64 bits, donc 8 octets). Parfois, la dénomination commerciale PCx-z est utilisée, où $z = 8y$ est la bande passante (en Mo/s) et y désigne la nomenclature « standard » (les deux sont données dans la *shopping list* ci-dessus, ce qui rend le tout encore plus confus).

Une différence qui n'est pas immédiatement apparente entre les générations est la notion de « *Burst* ». Les transferts se font forcément par paquets de x à des adresses consécutives, où x est le *Burst*. Autrement dit, en 2020, quand on veut lire un octet dans un module de RAM, il y a forcément 8 transferts de 64 bits, donc 64 octets qui circulent sur le bus mémoire. On verra section 8.4.1 que ceci est logique vu l'organisation des caches.

UDIMM, RDIMM Un module de RAM peut être *Unbuffered* (on dit aussi *Unregistered* et on parle de UDIMM) ou bien *Registered* (on dit aussi *Buffered* et on parle de RDIMM). Dans le deuxième cas, un *registre* (c.a.d. un tampon) est présent entre les DRAM chips et le contrôleur mémoire. Ce tampon joue le rôle de relais et il réduit la quantité de courant qui circule sur le bus (le contrôleur n'a pas à « piloter » directement les DRAMs chips). Ceci permet d'augmenter la quantité de DIMMs qu'on peut brancher sur le même bus et améliore la stabilité globale du système. On trouve typiquement des UDIMMs dans les ordinateurs personnels et des RDIMMs dans les serveurs et les machines de HPC. Les RDIMMs sont généralement plus chers, et le *buffer* ajoute généralement une pénalité d'un cycle aux latences d'accès à la mémoire. Il existe aussi des LRDIMM (*Load Reduced ...*), Une variante améliorée de DIMM à registre (plus récente, depuis 2013, généralement de plus grande capacité).

Une autre caractéristique que les DIMM peuvent avoir ou pas est la capacité de correction d'erreur : on parle alors de DIMM ECC (*Error-Correcting Code*). En effet, lors du stockage ou du transfert de 64 bis, un code

Génération	Année	Burst	Standard	Freq. chips (Mhz)	Freq. bus (Mhz)	Mo/s
DDR	2000	2	DDR-200	100	100	1600
			DDR-266	133	133	2133
			DDR-233	166	166	2667
			DDR-400	200	200	3200
DDR2	2003	4	DDR2-400	100	200	3200
			DDR2-533	133	266	4267
			DDR2-667	166	333	5333
			DDR2-800	200	400	6400
			DDR2-1066	266	533	8533
DDR3	2007	8	DDR3-800	100	400	6400
			DDR3-1066	133	533	8533
			DDR3-1333	166	667	10667
			DDR3-1600	200	800	12800
			DDR3-1866	233	933	14933
			DDR3-2133	266	1067	17067
DDR4	2014	8	DDR4-1600	200	800	12800
			DDR4-1866	233	933	14933
			DDR4-2133	266	1066	17067
			DDR4-2400	300	1200	19200
			DDR4-2666	333	1333	21333
			DDR4-2933	366	1467	23467
			DDR4-3200	400	1600	25600
DDR5	2020 ?	?	DDR5-6400	?	3200	51200
DDR6	?	?	?	?	?	

FIGURE 8.4 – Standard pour les modules de DRAM.

correcteur d'erreur permet la correction de n'importe quelle erreur affectant un seul bit ainsi que la détection d'erreurs affectant deux bits. Le prix à payer pour cette capacité est qu'il faut stocker et transférer 72 bits au lieu de 64. Un module ECC doit donc avoir une capacité réelle 12.5% plus élevée qu'un module non-ECC de la même capacité nominale. Les modules ECC se retrouvent typiquement dans les serveurs et les machines de HPC, rarement dans les ordinateurs personnels. La gestion du code correcteur peut augmenter un peu la latence d'accès à la mémoire.

Bien que cela puisse paraître surprenant, la RAM peut en effet exhiber des erreurs. Deux phénomènes distincts apparaissent :

1. Les erreurs *passagères* (*transient*, *soft error*). Le contenu d'un bit de la mémoire est modifié ponctuellement sous l'effet d'interférences electro-magnétiques (« raysons cosmiques » ou exposition à la radioactivité), mais à part ça les circuits fonctionnent normalement.
2. Les erreurs *permanentes* : le circuit est endommagé et ne stocke plus rien correctement.

Les erreurs passagères sont plus fréquentes qu'on ne pourrait le croire. Les ingénieurs de Google ont surveillé l'ensemble de leurs machines pendant 2.5 ans [10] : environ un tiers machines est affectée par une « erreur mémoire récupérable » (par le code correcteur) chaque année, et un peu plus de 1% par une erreur irrécupérable qui fait planter la machine et nécessite son remplacement. L'étude note que dans plus de 85% des cas, une erreur récupérable est suivie d'au moins une autre erreur récupérable dans le mois qui suit (ce qui laisse supposer que le composant matériel va bientôt faillir complètement).

Le problème est considérablement aggravé dans des environnements radioactifs (ou dans l'espace ; l'atmosphère et le champ magnétique terrestre bloquent une partie de la radioactivité).

Le code correcteur utilisé est construit à partir du code de Hamming $[127, 120, 3]$, qui est un code parfait. L'entrée est tronquée à 64 bits, ce qui donne un code $[71, 64, 3]$. Un bit de parité additionnel (le XOR de tous les bits d'entrée) est ajouté, ce qui donne un code $[72, 64, 4]$.

Canaux La DRAM fonctionne essentiellement avec des condensateurs : chargés, ils contiennent le bit UN ; déchargés, ils contiennent le bit ZÉRO. Il y a cependant des fuites de courant : ces condensateurs perdent naturellement leur charge au cours du temps, et il faut donc périodiquement les *rafraichir* en « lisant » le bit contenu puis en le « ré-écrivant ». Il faut typiquement faire ceci toutes les 64ms.

La gestion du bon fonctionnement de la DRAM est donc typiquement déléguée à un contrôleur matériel. Ce dernier gère le rafraîchissement de la DRAM, mais aussi l'ordonnancement des requêtes, la concurrence, etc. Bref c'est encore très compliqué.

En 2020, les contrôleurs mémoire sont généralement *dans* les CPUs, mais ça n'a pas toujours été le cas. Dans du matériel hau-de-gamme, c'est le cas depuis plus longtemps, mais pour le matériel « grand public », c'est depuis ≈ 2008 (AMD K8 et Intel « Nehalem »).

Un contrôleur mémoire pilote un ou des canaux (« *channel* »), sur lequel sont connecté un ou des DIMMs. Avant ≈ 2000, un contrôleur mémoire avait un seul canal. Mais, pour augmenter la bande passante de la mémoire, le nombre de canaux a inexorablement augmenté : en effet, la bande passante d'un seul DIMM est limitée (cf. fig. 8.4), mais avec n DIMMs sur n canaux, on peut obtenir n fois la bande passante d'un seul DIMM. Aujourd'hui, les processeurs « entrée de gamme » ont 2 canaux (tous les Intel Core i3 et Intel Core i5 par exemple). En 2008, les Intel Core i7 920 (« Bloomfield ») ont trois canaux. En 2012, les Core i7-3820 (« Série x, Sandy Bridge ») en ont quatre. En 2017, les Xeon SP (« Skylake ») en ont 6 (deux contrôleurs de trois canaux), tandis que les AMD Epyc en ont huit (4 contrôleurs à deux canaux), tout comme les IBM Power9.

Il faut noter que ce parallélisme de données permet d'augmenter la bande passante de la mémoire, mais il ne diminue pas la latence. La façon dont les adresses physiques sont réparties sur les différents canaux n'est pas fixée. Elle est parfois configurable dans le micrologiciel de la machine.

Par contre, les DIMMs qui sont branchés sur le même canal ne peuvent pas être lus/écrits en parallèle. En effet, sur *un* canal il y a *un* bus partagé entre tous les DIMMs. Un seul peut lire/écrire sur ce bus partagé à la fois. Les performances maximales sont donc en principe atteintes avec un seul DIMM par canal.

Au passage, il y a aussi des limites physiques qui contraignent le nombre de DIMMs par canal. Les processeurs Intel et AMD récents ne supportent que deux DIMMs par canal, ce qui fait 12 ou 16 DIMMs en tout par processeur.

Dans les processeurs AMD Epyc, un canal peut fonctionner jusqu'à 2666MHz avec *un seul* DIMM, mais la fréquence baisse à 2133MHz si on met *deux* DIMMs. Sur les processeurs Intel récents, cette diminution n'est pas systématique mais elle peut avoir lieu quand même.

Donc : a) si on veut vraiment beaucoup de RAM, il faut utiliser beaucoup de processeurs, et b) plus on veut de RAM, plus elle est lente.

Rangs Sur l'image de la figure 8.3, on voit un DIMM composé de plusieurs *DRAM chips*. Ces DRAM chips sont groupés par *rangs*. Un DIMM peut comporter un seul rang, ou bien deux, ou bien quatre (traditionnellement, les DIMMs à deux rangs ont des *DRAM chips* des deux côtés : côté pile, ils forment le rang zéro, et côté face le rang 1 ; mais il n'y a pas une correspondance stricte entre rangs et côtés). Depuis la DDR4, il est possible d'avoir huit rangs par DIMM. En pratique, c'est rare en dehors des LRDIMM de la plus grosse capacité (64/128Go en 2020). Au sein d'un rang, tous les *DRAM chips* sont lus/écris simultanément, en parallèle. En effet, chaque rang doit être capable de lire/écrire 64 bits de données « d'un coup », et pour cela les 64 bits de données sont répartis sur plusieurs *DRAM chips*. La configuration la plus courante pour les UDIMM est de 8 chips qui lisent/écrivent 8 bits par cycle (on parle alors de « ×8 »). La configuration la plus courante dans les RDIMM avec correction d'erreur est de 18 chips qui lisent/écrivent 4 bits par cycle (on parle alors de ×4). Là encore, c'est une forme de parallélisme de données qui augmente le débit, mais pas la latence.

À ce sujet, les *DRAM chips* fonctionnent à une fréquence moindre que celle du bus qui relie le DIMM au contrôleur. On voit sur la figure 8.4 que leur fréquence n'a que peu évolué depuis l'an 2000 (multiplication par deux), alors que la bande passante totale des DIMMs, elle, a augmenté de manière plus importante. Or, la latence des accès à la mémoire est inversement proportionnelle à la fréquence des *DRAM chips*.

Au passage, sur un canal, le contrôleur ne peut lire/écrire/commander qu'un seul rang à la fois par cycle d'horloge du bus, car tous les rangs sont branchés « en parallèle » sur le canal. Mais, comme verra plus bas, les commandes envoyés par le contrôleur mémoire aux *DRAM chips* mettent un certain temps à être exécutées (plusieurs cycles). Pendant qu'un des rangs du canal est « occupé », le contrôleur mémoire peut communiquer avec les autres, ce qui augmente sa capacité à satisfaire des requêtes d'accès à la mémoire. Des expériences suggèrent qu'avec deux rangs, la latence « en plein charge » des accès à la mémoire est diminuée de 30% sur certaines machines par rapport à des DIMMs comparables à un seul rang. Au-delà de 4 rangs, ce bénéfice est compensé par le trafic additionnel qui a lieu sur le bus.

Certains processeurs imposent une limite au nombre de rangs qui peuvent coexister sur le même canal, par exemple 8 pour certains processeurs Intel.

Standard	Freq. chips (Mhz)	Mo/s	CL min.	Latence (ns)
DDR4-1600	200	12800	10	12.5
DDR4-1866	233	14933	12	12.857
DDR4-2133	266	17067	14	13.125
DDR4-2400	300	19200	15	12.5
DDR4-2666	333	21333	17	12.75
DDR4-2933	366	23467	19	12.96
DDR4-3200	400	25600	20	12.5

FIGURE 8.5 – Standard pour les modules de DRAM.

Banques, lignes, colonnes Au bout de la chaine, nous trouvons les *DRAM chips*. Chaque norme de DDR spécifie des tailles de chips standardisées. Dans la norme DDR4, il peut y en avoir de 2, 4, 8 ou 16 Gigabits.

À l'intérieur d'un DRAM chip, les données sont stockées selon un schéma à trois dimensions : il y a 16 *banques*, chacune constituée d'un tableau à deux dimensions, avec des *lignes* et des *colonnes*. À la fin, chaque case contient 4 bits ou 8 bits (selon que c'est un ×4 ou un ×8). Dans la norme DDR4, les lignes sont toujours de taille 1024 (c.a.d. qu'il y a 1024 colonnes).

> Si on prend un DIMM « standard » de 16Go, (unbuffered, ×8, *dual-ranked*), possédant 16 *DRAM chips*, possédant chacun 16 banques, où chaque « case » contient 8 bits, on trouve qu'il y a nécessairement 65536 lignes.

Chaque banque est indépendante. Les commandes adressées par le contrôleur indiquent quelle banque est visée. Lire ou écrire les données est un processus en plusieurs étapes ; pour lire ou écrire une adresse donnée, il faut :

1. Activer la bonne ligne (command ACTivate).
2. Lire ou écrire une colonne de la ligne active (commandes READ/WRITE).
3. Désactiver la ligne active (commande PREcharge).

Une seule ligne peut être active simultanément par banque. Il est nécessaire de désactiver la ligne active avant d'en pouvoir en lire une autre. Les trois opérations prennent un certain nombre de cycles du *DRAM chip* : pendant qu'une banque effectue une opération, cette banque est *occupée* un certain temps. Mais les autres banques, elles, peuvent être libres (prêtes à recevoir des commandes). Le contrôleur mémoire doit jongler avec ces contraintes. Dans le cas de la commande de lecture, le délai est imposé entre l'envoi de la commande et le moment où les données sont disponibles sur le bus du canal.

Pour simplifier un peu, les choses, dans la norme DDR4, chaque commande nécessite toujours le même nombre de cycles, et il est généralement noté CL (dans les normes précédentes, la latence de chaque opérations était différente). Et ce qui simplifie encore plus les choses, c'est que ça prend *toujours à peu près le même temps*, quelle que soit la fréquence d'horloge des *DRAM chips*, comme le montre la figure 8.5. Ceci dit, des DIMMs de moins bonne qualité peuvent avoir un CL un peu plus élevé que le minimum légal indiqué dans cette figure.

Une conséquence du mécanisme en trois étapes décrit ci-dessus est que le temps nécessaire pour lire la mémoire est *variable* : si le contrôleur veut lire/écrire la ligne i sur une certaine banque, alors trois situations peuvent se produire :

— La ligne i est déjà active. Il envoie READ/WRITE. Durée : CL.
— Aucune ligne n'est active. Il envoie ACTivate puis READ/WRITE. Durée : 2× CL.
— Une ligne $j \neq i$ est active. Il envoie PREcharge, ACTivate puis READ/WRITE. Durée : 3× CL.

Par conséquent, les contrôleurs mémoire essaye de minimiser les pénalités dues au second et au troisième cas, en *réordonnant* les requêtes d'accès à la mémoire pour maximiser le nombre de *hits* sur les lignes déjà actives. Il y a 16 banques par *DRAM chip* à gérer en parallèle (qui peuvent toutes avoir des lignes actives différentes), ce qui donne des problèmes d'ordonnancement assez compliqués pour obtenir de bonnes performances. Le « rafraîchissement » qui doit être effectué périodiquement par le contrôleur revient en fait à activer une ligne, effectuer une lecture quelconque, puis désactiver la ligne. Ceci doit être effectué pour chaque ligne de chaque banque toutes les 64ms, ce qui rajoute encore des contraintes !

En tout cas, lire des adresses *physiques* contiguës est a priori légèrement plus favorable que lire des adresses complètement aléatoires, car on a plus de chance de tomber dans la même ligne. En réalité, ceci est inexploitable dans les applications, car ces dernières manipulent des adresses *logiques* qui ne correspondent absolument pas aux adresses physiques (et que l'OS randomise parfois).

Mais ce qu'on peut dire, c'est que *dans le meilleur des cas*, une requête de lecture/écriture adressée au contrôleur mémoire ne peut être satisfaite que 12.5ns plus tard, ce qui correspond à 37.5 cycles d'un CPU à 3GHz. Et

nom	opération	octet/itération	FLOP/itération
COPY	$A[i] \leftarrow B[i]$	16	0
SCALE	$A[i] \leftarrow \alpha \cdot B[i]$	16	1
SUM	$A[i] \leftarrow B[i] + C[i]$	24	1
TRIAD	$A[i] \leftarrow B[i] + \alpha \cdot C[i]$	24	2

FIGURE 8.6 – Les tests effectués par le STREAM benchmark. Pour tout $0 \leq i < N$, l'opération est effectuée. Le nombre α est un flottant double précision fixé.

dans le pire des cas, c'est au moins trois fois plus. Et en réalité, la latence des accès mémoire est encore plus importante.

Sous linux, les programmes `lshw` et `dmidecode` permettent d'apprendre des informations intéressantes sur la mémoire installée.

8.2 STREAM

En 1995, John D. McCalpin a conçu un petit programme de test pour mesurer la bande-passante de la mémoire des machines « vectorielles » qui existaient à l'époque. Son petit programme, STREAM [7], a eu beaucoup de succès et il est toujours utilisé aujourd'hui. Il se compose de quatre tests décrits figure 8.6. Il alloue trois grands tableaux A, B et C de N flottants double précision, avec N suffisamment grand pour que les données ne tiennent pas dans les caches de la hiérarchie mémoire (il est recommandé de choisir 10× la taille du plus grand cache). Ceci oblige la machine à transférer depuis/vers la RAM et cela permet donc de mesurer le débit de la mémoire. Les opérations sont vectorisables, et leur intensité arithmétique est faible (peu d'opérations arithmétiques par flottant transféré de la mémoire).

La figure 8.7 montre le résultat du STREAM benchmark sur quelques machines. Cela met en évidence le « *memory wall* » : sur un laptop conventionnel ou un Raspberry Pi, un seul coeur suffit à saturer la bande-passante de la mémoire.

Sur les machines dédiées au caclul scientifique, qui sont mieux équilibrées, ce n'est pas le cas. Ceci dit, sur un noeud de BlueGene/Q (une machine qui date de 2011), un quart des coeurs disponibles suffit à saturer la RAM, tandis que sur un noeud d'un cluster récent, équipé de processeurs Xeon Gold, la moitié des coeurs suffit à saturer la RAM.

Cela signifie concrètement que sur des machines contemporaines, si les calculs nécessitent l'accès à de grande quantité de données, alors les unités d'exécution des processeurs vont passer (au moins) la moitié de leur temps à attendre l'arrivée des données !

Voici un commentaire un peu plus approfondi des résultats. Le laptop possède un SO-DIMM de DDR4-2133 (unbuffered), 2 rangs ×8. La bande passante maximale théorique est donc 17Go/s. On atteint 80% du maximum.

Le `raspberry Pi 3B+` possède 1Go de LPDDR2-800 (*Low-Power DDR2*), donc possède une bande passante théorique maximale de 6.4Go/s. On plafonne à 42% du maximum théorique, mais il s'agit d'une machine à 35€...

L'IBM BlueGene/Q possède deux contrôleurs à deux canaux, sur lesquels sont branchés des chips de DDR3-1333 avec correction d'erreur renforcée. La bande passante maximale théorique est donc $4 \times 10.6 = 42.7 Go/s$. On atteint 65% du maximum.

Le *cluster node* possède deux processeurs. Le test a été effectué en forçant le système d'exploitation à n'en utiliser qu'un seul, pour éviter les effets NUMA (la machine a 192Go de RAM, mais chaque processeur n'est directement relié qu'à 96Go) — ceci peut se faire avec la commande `numactl --cpunodebind=0`. Chaque processeur a deux contrôleurs mémoire de trois canaux chacuns. Chaque canal est équipé d'un RDIMM de DDR4-2666 de 16Go, *dual-rank*. La bande passante maximale théorique pour un processeur est donc de $6 \times 21.3 = 128$Go/s. On voit qu'on atteint 65% du maximum.

Au passage, dans le cas du *cluster node*, le processeur (un Intel Xeon Gold 6130 CPU @ 2.10GHz) possède 16 coeurs physiques et l'hyper-threading. Faire le STREAM benchmark avec 16 ou 32 threads ne change rigoureusement rien : c'est le débit de la mémoire qui est saturé, point final. On peut aussi essayer de faire le test avec 16 threads logiciels placés sur 16 coeurs physiques différents, ou bien 16 threads logiciels placés sur 8 coeurs physiques différents seulement : la différence n'est pas énorme (82.8Go/s vs 75.8Go/s).

En essayant sur les deux processeurs à la fois, on obtient des résultats variables en utilisant seulement un thread par coeur physique (de 85Go/s à 160Go/s selon les exécutions). Par contre, en utilisant tous les hyper-threads, on obtient bien systématiquement 155Go/s, donc un quasi-doublement par rapport l'utilisation d'un seul CPU. Mais il est alors nécessaire d'utiliser tous les hyper-threads...

Machine	Threads	COPY	SCALE	SUM	TRIAD	TRIAD speedup w.r.t. 1 thread
Laptop	1	13.8	9.7	10.9	10.9	-
Laptop	2	13.8	9.7	10.9	10.9	1
Raspberry	1	2.7	2.4	2.2	1.8	-
Raspberry	2	2.4	2.3	2.3	2.3	1.3
Raspberry	4	2.1	2.0	2.0	2.0	1.1
BlueGene/Q	1	5	5.5	7.5	7.5	-
BlueGene/Q	2	10.5	11.2	15	15	2
BlueGene/Q	4	21.4	22.6	26.8	26.8	3.6
BlueGene/Q	8	26.3	25.9	27.6	27.9	3.7
BlueGene/Q	16	26.2	26.2	28.0	28.0	3.7
Cluster node	1	10.5	12.2	12.8	12.7	-
Cluster node	2	20.5	23.8	24.8	24.6	1.9
Cluster node	4	39.7	45.6	48.1	47.8	3.7
Cluster node	8	75.3	60.0	67.7	67.6	5.3
Cluster node	16	83.2	65.4	73.5	73.4	5.7

FIGURE 8.7 – Quelques résultats du STREAM benchmark : débit de la RAM en Go/s. Le « laptop » et le « Raspberry » sont décrits dans la figure 8.10. La machine « BlueGene/Q » est décrite figure 8.11. Le « cluster node » est décrit figure 8.13.

8.3 Débit et latence

En fait, la situation est pire que ce qui est suggéré ci-dessus. Le *débit* (ou encore la « bande-passante ») de la mémoire est une chose, mais sa *latence* en est une autre, comme en témoigne un examen attentif des deux figures 8.1 et 8.2. Même si on accède à peu de données, on risque de devoir les attendre quand même. Les algorithmes de parcours de graphe forment un exemple typique de ce problème : il faut en permanence lire de petits paquets de données éparpillés dans toute la mémoire, et donc payer à chaque fois la latence des accès.

Le modèle RAM est une simplification Lorsqu'on étudie un algorithme (séquentiel), on considère généralement que cet algorithme va être exécuté sur une machine dont le modèle abstrait est la *Random-Access Machine* : une machine qui a accès à une mémoire, c'est-à-dire un un tableau (généralement illimité) de cases de w bits chacune. Les « opérations élémentaires » de cette machine sont la lecture et l'écriture d'une case en mémoire ainsi que les opérations arithmétiques usuelles sur des mots de w bits (on peut prendre $w = 64$ de nos jours).

La complexité d'un algorithme est généralement décrite comme le nombre d'opérations élémentaires qu'une telle machine doit effectuer pour l'exécuter. Ce modèle abstrait a l'avantage d'être simple, mais l'inconvénient de ne pas décrire fidèlement la réalité, en tout cas pas tout le temps. Plus précisément, il n'est pas pas faux de dire que, sur une machine donnée, exécuter l'instruction :

```
double x = A[i];
```

prend un temps qui, dans le pire des cas, est borné par une constante indépendante de i. Mais la réalité, c'est que le temps au bout duquel la valeur de x sera disponible va dépendre de i car la mémoire est organisée de manière *hiérarchique*. Pour obtenir de bonnes performances, il faut absolument exploiter cette structure pour éviter de se trouver dans le pire cas, c'est-à-dire le cas où la valeur de `A[i]` doit être transférée depuis une barette de RAM jusqu'aux unités d'exécution du CPU.

Limites physiques D'un point de vue asymptotique, l'idée qu'on peut accéder à une quantité arbitraire de mémoire en temps constant se heurte de toute façon aux réalités du monde matériel. Comme l'information ne peut (*a priori*) pas se déplacer plus vite que la vitesse de la lumière, et qu'il ne peut y avoir qu'une quantité finie de bits de mémoire dans un volume donné, alors plus on a de mémoire et plus une partie de cette dernière est « loin », donc plus ça prend de temps d'y accéder. Si M bits de mémoire étaient disposée dans une boule autour d'une unité de calcul, alors dans le pire des cas cette dernière ne pourrait accéder aux bits « externes » qu'en attendant un temps proportionnel au rayon de la boule, donc à $\sqrt[3]{M}$. De nos jours, les ordinateurs sont plutôt disposés « à plat », donc en réalité ce serait plutôt $\sqrt{M}$. Tout ça pour dire que le modèle RAM, qui affirme que c'est constant, ne peut prétendre au réalisme.

En pratique, on observe empiriquement que plus une machine a de RAM, plus la latence de cette mémoire est élevée.

Un micro-benchmark Pour illustrer ceci, on va se pencher sur un autre micro-benchmark qui stresse la mémoire, le *pointer-chasing*. On alloue un tableau T de taille N, et on l'initialise préalablement avec une permutation pseudo-aléatoire des entiers $\{0, 1, \ldots, N-1\}$ [1]. Ensuite, on mesure le temps d'exécution du bout de code suivant :

```
int x = 0;
for (int i = 1000000000; i != 0; i--)
        x = T[x];
```

Ce bout de code ne fait rien d'autre qu'accéder à la mémoire, mais contrairement au STREAM benchmark qui accède à zones contiguës de la mémoire, de manière régulière, ici on charge des adresses *imprévisibles*. De plus, le programme ne peut pas progresser tant que la valeur de `T[x]` n'est pas disponible.

Dans le modèle RAM, le temps d'exécution ne devrait pas dépendre de N. Or, dans la pratique, il en dépend très sensiblement. C'est d'ailleurs facile à mesurer. Sur un Raspberry Pi modèle `3B+`, le compilateur `gcc 8.3.0` produit un code assembleur de 3 instructions seulement pour la boucle critique :

```
; r4 == adresse de T, r9 == x, r5 == i
loop:
        ldr     r9, [r4, r9, lsl #2]        ; x := *(T + 4*x)
        subs    r5, r5, #1                  ; i := i - 1
        bne     loop                        ; if i != 0 then GOTO loop
```

Le processeur Cortex A53 du Raspberry Pi 3B+ est *in-order* et il exécute une instruction par cycle d'horloge dans le meilleur des cas. Si tout se passait bien, une itération de la boucle nécessiterait donc 3 cycles d'horloge. Voyons ce qu'il en est sur la figure 8.8 : on voit que c'est bien le cas tant que les données occupent moins de 32Ko. Par contre, la latence des accès mémoire est multipliée par ≈ 130 lorsque la taille des données passe de de 32Ko à 800Mo.

On observe sensiblement le même résultat sur des machines dédiées au calcul scientifique (cf. fig. 8.9). En outre, sur ces machines qui ont beaucoup de RAM, on observe un doublement progressif de la latence au-delà de 1Go de mémoire à laquelle on accède aléatoirement. La forme des courbes obtenue résulte de plusieurs phénomènes qu'on va analyser séparément.

8.4 Hiérarchie mémoire : les caches

Pour accélérer les accès à la mémoire, les ordinateurs disposent de *cache* : de petites mémoires rapides (et chères) placées plus près des unités d'exécution. Il peut y en avoir plusieurs montés en cascade : dans ces cas-là on parle du cache de « niveau 1 » (*Level 1* ou L1, celui qui est le plus proche des unités d'exécution), puis de niveau 2, puis de niveau 3, etc. (les caches de niveau 4 sont rares de nos jours). Généralement, les caches les plus proches sont plus rapides et plus petits que les caches les plus éloignés, pour des raison de coût et de complexité des circuits.

La taille des caches et leurs caractéristiques varient sensiblement d'une machine à l'autre. Le programme `lstopo`, qui fait partie de la bibliothèque `hwloc` qui fait elle-même partie de OpenMPI permet d'afficher la hiérarchie mémoire. Les figures 8.10, 8.13 et 8.14 montrent la hiérarchie mémoire de plusieurs machines assez différentes : on voit qu'il y a une grande variabilité.

Une partie du fonctionnement de la hiérarchie mémoire des processeurs n'est pas, ou mal documenté. De toute façon, c'est très compliqué dès qu'on rentre dans les détails et ce document ne prétend d'ailleurs pas à l'exhaustivité !

Pour obtenir le contenu d'une adresse mémoire, les unités d'exécution du processeur adressent une requête au cache L1. Si l'adresse demandée est présente dans le cache, il y a un *cache hit* et la valeur demandée est renvoyé rapidement. Sinon, il y a une « faute de cache » (*cache miss*) et le cache L1 s'adresse à l'étage du dessus pour obtenir lui-même la valeur présente en mémoire à l'adresse voulue. Une fois qu'il la reçoit, il la stocke. Tout ceci entraîne un temps d'attente supplémentaire.

S'il y a un cache L2, le même mécanisme se répète dans le cache L2, puis éventuellement dans le cache L3. Si toutes les tentatives ont donné lieu à des fautes de cache, alors la valeur est finalement lue depuis la RAM.

Le tableau de la figure 8.15 montre, pour quelques processeurs, les latences des caches en cas de *hit*. Par exemple, sur le processeur Cortex A53, la latence du cache L1 est de 3 cycles en cas de *hit* (c.a.d. que la valeur demandée

1. On vérifie que le cycle de la permutation qui commence en zéro est au moins de taille $N/3$, sinon on change la permutation. Ça n'arrive pas très souvent : la longueur moyenne du cycle auquel appartient un élément aléatoire dans une permutation aléatoire de N éléments est $(N+1)/2$ (cf. [3, §1.3.3, exercice 24]).

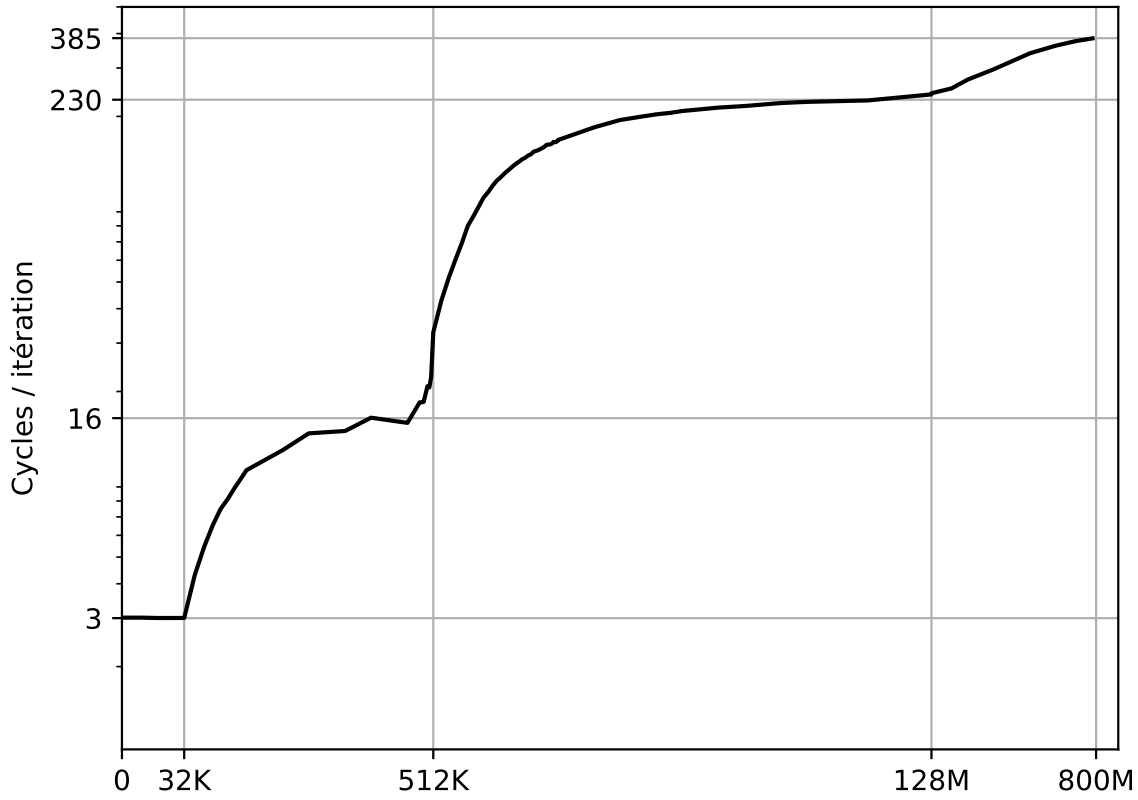


FIGURE 8.8 – Résultat du *pointer chasing* benchmark sur un Raspberry Pi `3B+`. L'axe des abscisses représente la taille du tableau T en octets.

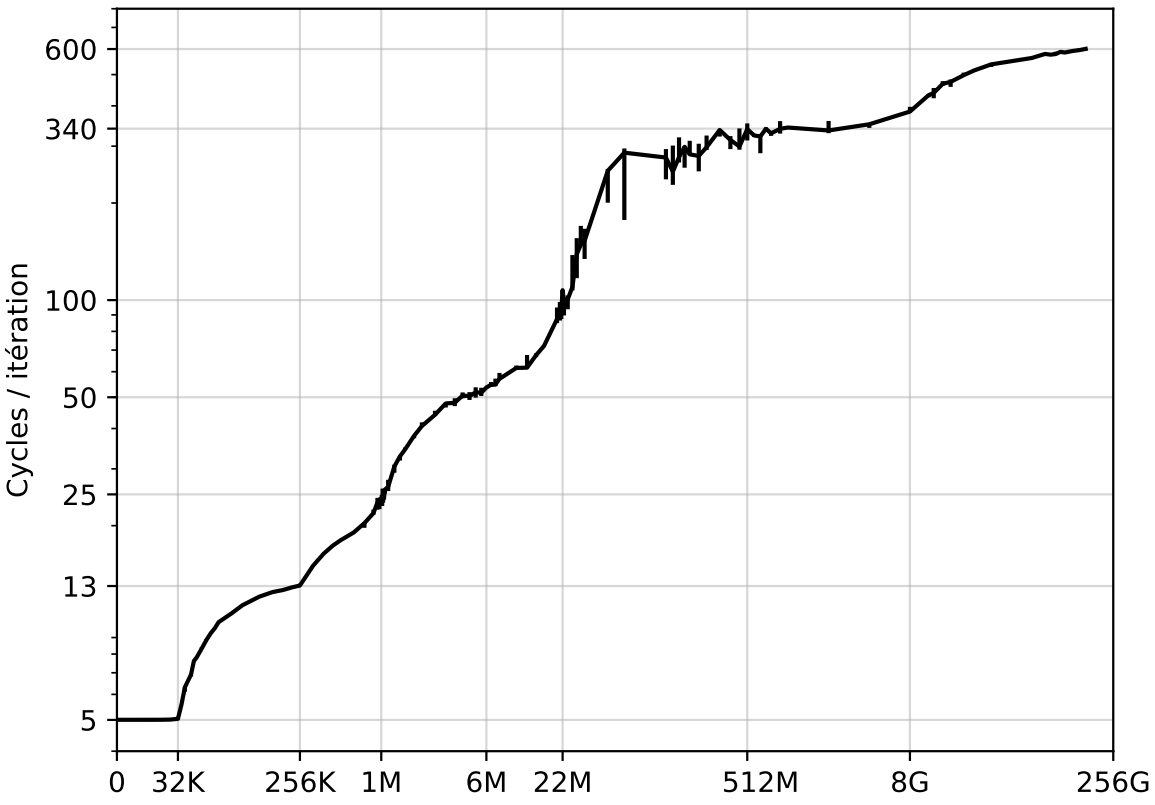
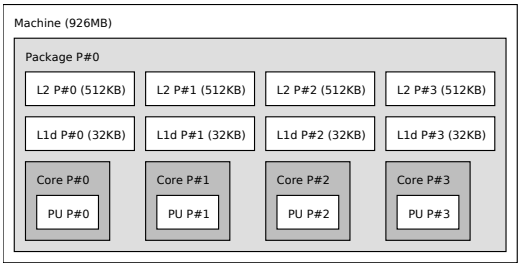
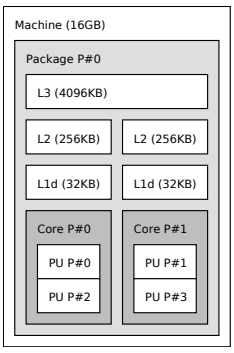


FIGURE 8.9 – Résultat du *pointer chasing* benchmark sur le « *cluster node* » (décrit fig. 8.13).



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

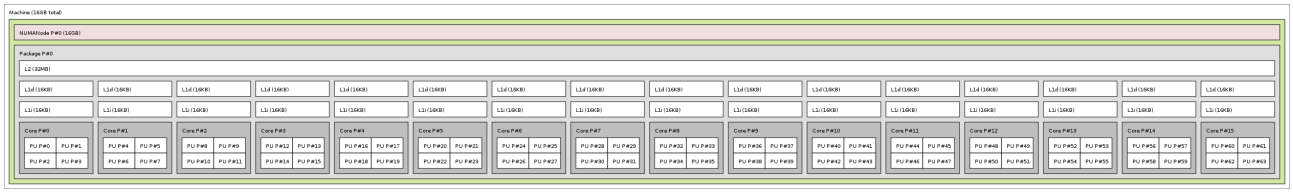


FIGURE 8.11 – Un noeud de calcul d'une machine massivement parallèle IBM BlueGene/Q. Un processeur (PowerPC A2) de 16 coeurs, 4 threads SMT par coeur. Chaque coeur possède 16Ko de cache L1, partagé entre les 4 threads (c'est peu). Un énorme cache L2 de 32Mo est partagé entre tous les coeurs.

FIGURE 8.12 – Un noeud de calcul d'un cluster récent (2018). Deux processeurs (Intel Xeon Gold 6130), 16 coeurs par processeur, deux threads SMT par coeur. Chaque coeur possède 32Ko de cache L1 (partagé entre les 2 threads) et un gros cache L2 de 1Mo. Un cache L3 de 22Mo est partagé entre tous les coeurs. Chaque processeur possède son propre controlleur mémoire et accède de manière privilégiée à « ses » 96Go de RAM. Il y a donc deux noeuds NUMA.

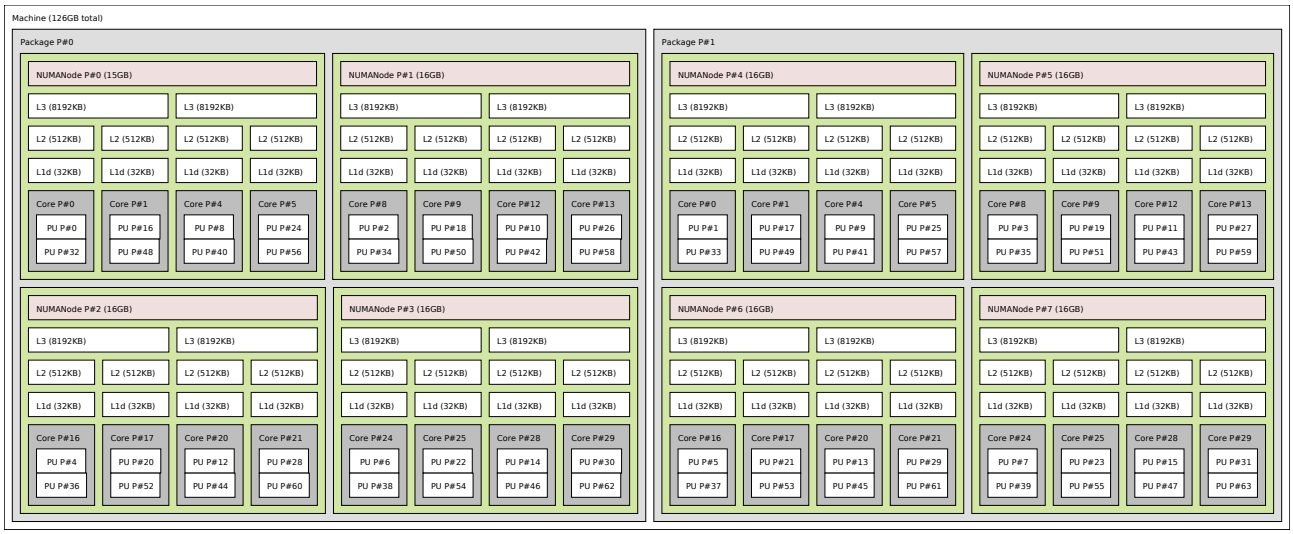


FIGURE 8.13 – Un autre noeud de calcul d'un autre cluster récent (2018). Deux processeurs (AMD EPYC 7301), 16 coeurs par processeur, deux threads SMT par coeur. Chaque coeur possède 32Ko de cache L1 (partagé entre les deux threads) et 512Ko de cache L2. Chaque *paire* de coeurs partage un cache L3 de 8Mo. Chaque processeur est divisé en quatre « chipplets » qui ont chacun leur propre controleur mémoire et qui accèdent de manière privilégiée à « leurs » 16Go de RAM. Chaque processeur contient donc 4 noeuds NUMA, et il y a donc huit en tout dans la machine.

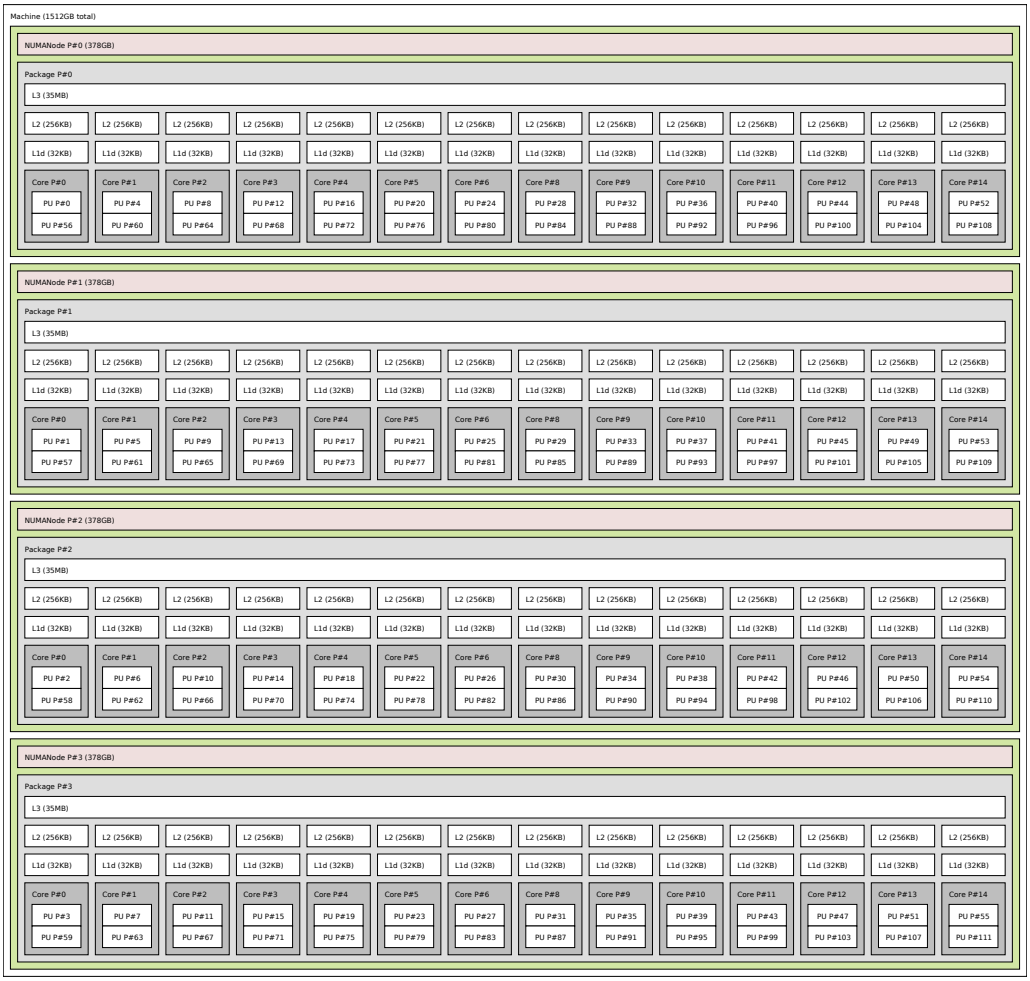


FIGURE 8.14 – Un « fat node » pour les calculs qui demandent beaucoup de mémoire. Quatre processeurs (Intel Xeon E7-4850 v3), 14 coeurs par processeur, deux threads SMT par coeur. Chaque coeur possède 32Ko de cache L1 (partagé entre les deux threads SMT) et 256Ko de cache L2 (cette configuration était courante pendant des années sur les processeurs Intel). Chaque processeur a un cache L3 de 35Mo partagé entre tous les coeurs. Chaque processeur contrôle « ses » 384Go de RAM, il y a donc 4 noeuds NUMA.

CPU	L1	L2	L3	RAM
Cortex A53	3	15	n.a.	≈ 200
PowerPC A2	5	82	n.a.	≈ 350
Xeon Gold	5	14	50	≈ 300

FIGURE 8.15 – Latence des cache en cas de *hit* (en cycles d'horloge).

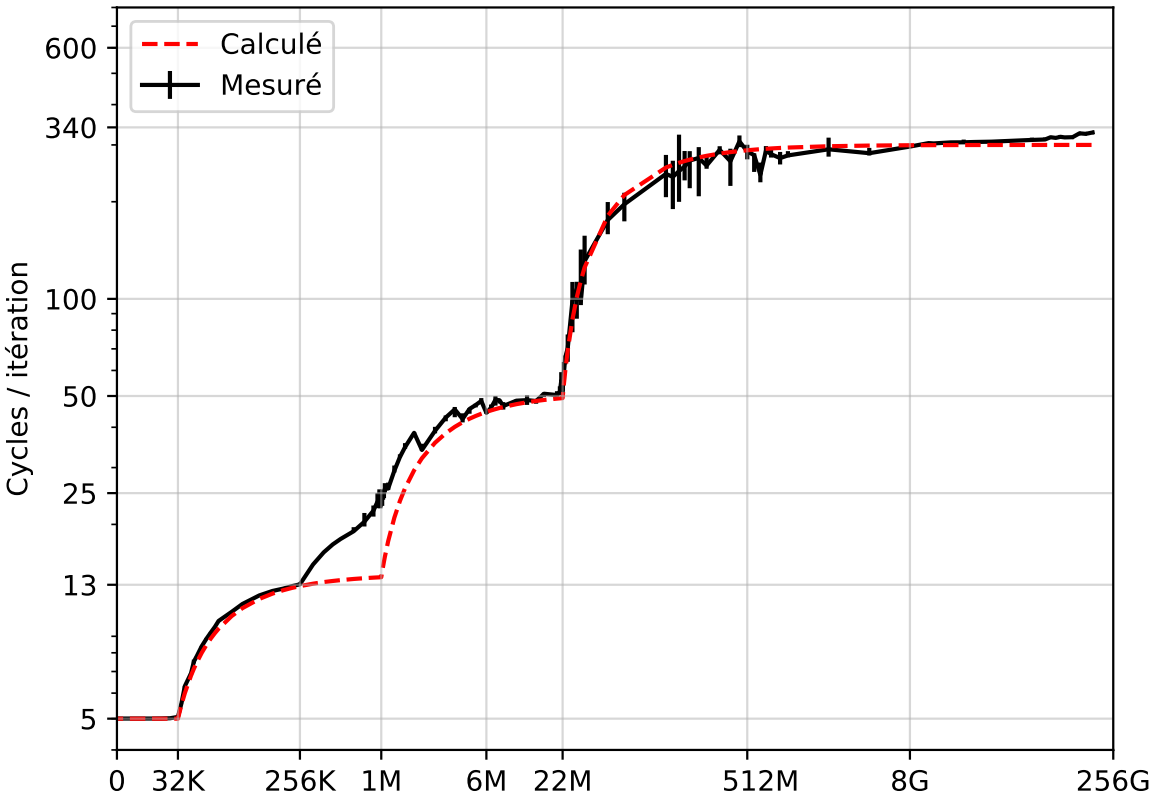


FIGURE 8.16 – Résultat du *pointer chasing* benchmark sur le « *cluster node* » (décrit fig. 8.13), avec simulation.

est disponible 3 cycles après la requête ; si on essaye de l'utiliser avant, le pipeline d'exécution est stoppé jusqu'à ce que la valeur soit disponible). Dans notre cas, on utilise la valeur de x exactement 3 cycles plus tard, donc tout va bien.

Il faut noter qu'on doit pouvoir déduire ces valeurs des courbes de performance obtenues par le *pointer chasing* benchmark.

Muni de ces informations, on peut même modéliser le comportement du *pointer-chasing* benchmark. On considère une hiérarchie de cache L1, L2 L3 dont les tailles (en octet) sont notées L_1, L_2 et L_3 et dont les latences (en cycles) sont notées λ_1, λ_2 et λ_3. Comme la permutation que contient le tableau T est choisie uniformément au hasard, alors on peut estimer qu'en régime stationnaire chaque accès à la mémoire a une probabilité $\min(1, L_1/N)$ d'aboutir à un *hit* dans le cache L1 (idem pour L2, L3). On pourrait alors calculer la latence moyenne de chaque accès à la mémoire :

$$\begin{aligned}\mathbb{E}\left[\text{Latence}\right] &= \mathbb{P}\left[\text{L1 hit}\right]\lambda_1 + \mathbb{P}\left[\text{L1 miss}\right]\Big(\mathbb{P}\left[\text{L2 hit}\right]\lambda_2 + \mathbb{P}\left[\text{L2 miss}\right]\left(\mathbb{P}\left[\text{L3 hit}\right]\lambda_3 + \mathbb{P}\left[\text{L3 miss}\right]\lambda_{\text{ram}}\right)\Big) \\ &= \frac{L_1}{N}\lambda_1 + \left(1 - \frac{L_1}{N}\right)\left(\frac{L_2}{N}\lambda_2 + \left(1 - \frac{L_2}{N}\right)\left(\frac{L_3}{N}\lambda_3 + \left(1 - \frac{L_3}{N}\right)\lambda_{\text{ram}}\right)\right)\end{aligned}$$

On obtient ainsi une approximation raisonnable de la réalité (cf. fig. 8.16).


La théorie correspond mal à la réalité entre 256K et ≈ 22M. Les raisons n'en sont pas claires pour moi. Une enquête est en cours.

8.4.1 Organisation des caches

Une *ligne de cache* est un bloc de mémoire de taille B dont l'adresse est un multiple de B (le plus souvent, $B = 64$ octets, en tout cas c'est le cas sur tous les processeurs cités). Le cache stocke en réalité des *lignes de cache*, c'est-à-dire des blocs de 64 octets consécutifs en RAM. Ainsi, les caches L1 de 32Ko stockent généralement 512 lignes de 64 octets. Lorsqu'une faute de cache a lieu, c'est que l'adresse demandée n'appartient pas à une des lignes contenues dans le cache. Dans ce cas-là, une des lignes présente dans le cache est évincée (*evicted*) pour faire de la place, puis la ligne contenant l'adresse demandée (qui a causé la faute) est lue depuis l'étage du dessus et elle est stockée toute entière.

Ainsi, une faute de cache sur un mot de 32 ou 64 bits entraine le transfert de 512 bits vers le cache ! C'est notamment pour cette raison que les DIMMs de DDR3 et DDR4 transfèrent les données par paquets de 64 octets : c'est la taille d'une ligne de cache.

Par ailleurs, un cache de taille N avec des lignes de taille B est capable de gérer $\leq N/B$ emplacements arbitrairement différents de la mémoire... mais pas plus.

Lorsqu'une faute de cache a lieu, une ligne actuellement présente dans le cache doit être évincée afin de faire de la place pour la ligne entrante. Mais laquelle évincer ? Plusieurs « politiques de remplacement » sont possibles ; la plus commune consiste sans doute à évincer la ligne de cache qui a été « touchée » pour la dernière fois il y a le plus longtemps. Cette stratégie gloutonne (*Least Recently Used* — LRU) possède une garantie : on peut démontrer qu'elle conduit dans le pire des cas à seulement deux fois plus de fautes de cache que le choix optimal (qu'on ne peut faire qu'en connaissant les requêtes à venir, ce qui n'est pas le cas de la stratégie LRU).

Une variante largement utilisée (dans tous les caches L1 des processeurs commerciaux habituels) se nomme PLRU (*Pseudo-LRU*) : elle consiste à utiliser une approximation de l'âge de dernier accès à chaque ligne de cache, plutôt qu'une valeur exacte. Elle est légèrement moins précise mais moins coûteuse en hardware.

8.4.2 Études de cas simples

Recopie d'un tableau 2D Considérons une procédure très simple qui recopie un tableau à deux dimensions. Du point de vue du cache et de la localité des données, il y a une « bonne » manière de faire et une « mauvaise » manière de faire.

```
/* Mauvais */
for (int i = 0; i < N; i++)
    for (int j = 0; j <N; j++)
        dst[j][i] = src[j][i];
```

```
/* Bon */
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        dst[i][j] = src[i][j];
```

Le mauvais code effectue des accès en mémoire non-contigus. En effet, le langage C spécifie explicitement le placement en mémoire des tableaux multi-dimensionnels, et c'est *row-major* (contrairement à Fortran, Matlab, R, Julia, ...). Avec une déclaration comme `double A[N][M];` on trouve que `A[i][j]` se trouve en mémoire à l'adresse `A + i*M + j`. Les adresses varient de 1 lorsque le dernier indice varie, mais de M lorsque le premier indice varie.

Par conséquent, dès que $N \geq 8$, alors chaque accès à `src[j][i]` ou `dst[j][i]` est un accès à une ligne de cache différente (on bouge d'au moins 64 octets lorsque j augmente de 1). Et du coup, lorsque $N > 512$, chaque accès à l'un des deux tableaux provoque une faute de cache L1 (si le cache L1 fait 32Ko, ce qui courant) avec la politique de remplacement LRU.

A contrario, le « bon » code lit et écrit les données de manière contiguë, donc chaque fois qu'une faute de cache a lieu, on a la certitude que les 7 prochains accès seront servis depuis le cache.

Bilan : lorsque N est suffisamment grand, le mauvais code fait N^2 fautes de cache tandis que le bon en fait $\leq N^2/8$.

Produit de matrices naïf On considère le code suivant, qui effectue le produit de matrice $C \leftarrow A \times B$. En fait, il pose une variante du problème précédent.

```
for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
                double x = 0;
```

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

Dans ce code, les coefficients de la i-ème ligne de A sont tous lus, dans l'ordre, N fois consécutives (pour chaque j). Même s'ils ne tiennent pas tous dans le cache L1, alors une faute de cache sur `A[i * N + k]` va charger dans le cache une ligne de cache contenant les 8 prochains coefficients de la i-ème ligne de A, donc il n'y aura pas de faute de cache sur A pendant les 7 prochaines itérations.

Le cas de B est beaucoup plus pénible : B est lu par colonne, et il y a N coefficients de distance entre deux coefficients de B lus de manière consécutive. Par conséquents, dès que $N \geq 8$, les coefficients de B appartiennent à des lignes de cache différentes. Et dès que $N \geq 512$, alors un cache L1 standard de 32Ko n'a pas assez de lignes de cache pour stocker une colonne entière de B. Par conséquent, *chaque accès à B provoque alors une faute de cache L1* avec la politique de remplacement LRU.

Le produit matriciel naïf est donc une *très* mauvaise idée. En fait, ce serait bien mieux de calculer d'abord de $B' \leftarrow B^t$ (la transposée de B) puis d'effectuer le calcul $C \leftarrow A \times B'^t$ pour éviter ce problème (et pouvoir lire B' ligne par ligne sans causer de faute de cache systématique) — ceci aurait en outre l'avantage de permettre une forme intéressante de vectorisation.

Le problème, c'est que calculer la transposée prend du temps, et que ça risque d'en faire perdre plus que ça n'en fait gagner. Pour de petites valeurs de N, ça ne vaudra sûrement pas le coup. Mais pour $N = 3200$, par exemple, la combinaison « transposée + produit avec la transposée » met 63s (0.25 faute L1 / itération), tandis que le produit direct met 177s (une faute L1 / itération). On y gagne donc vraiment.

8.5 Caches : considérations avancées

8.5.1 Écritures dans les cache

Les écritures peuvent elles aussi causer des fautes de cache, mais là plusieurs solutions sont possibles. Dans tous les cas, si l'adresse écrite appartient à une ligne présente dans le cache, alors le cache est mis à jour. C'est ce qui se passe ensuite qui peut varier.

La première solution (« *write-through*) consiste à relayer systématiquement les opérations d'écriture à l'étage du dessus. Si une écriture concerne une adresse qui n'est pas dans le cache, alors on ne fait rien d'autre. C'est ce qui se passe dans le PowerPC A2. Avantages : les lignes de cache peuvent être évincées sans précautions particulières ; des écritures vers des adresses qui vont être abandonnées ne provoquent pas le chargement dans le cache de lignes inutiles. Inconvénient : les étages supérieurs reçoivent toutes les écritures.

La deuxième solution (« *write-back*) consiste à ne pas relayer les écritures aux étages du dessus en temps réel. Lorsqu'une ligne présente dans le cache est modifiée, elle est marquée comme « sale » (*dirty*). Lorsqu'une ligne de cache est évincée, elle est transmise aux étages du-dessus si et seulement si elle est sale. Si le cache reçoit une opération d'écriture pour une adresse qu'il ne contient pas, alors cela déclenche une faute : la ligne en question est d'abord chargée dans le cache depuis les étages du dessus, puis modifiée et marquée comme sale. Cette stratégie est la plus répandue. Avantages : des écritures pour des adresses consécutives sont gérées localement et non transmises au-dessus. Inconvénients : si on écrit une adresse qu'on ne lit plus jamais après, alors a) 512 bits sont lus, puis 512 bits sont écrits dans le niveau du dessus et b) l'écriture « polue » le cache.

Dans le cas d'un cache « habituel » (*write-back*), une optimisation est possible. Quand on *sait* qu'on écrit des données qu'on ne va pas chercher à lire bientôt, alors plutôt que de faire une écriture qui va polluer le cache avec des choses inutiles, on peut utiliser des instructions spéciales d'écriture ou de lecture dites « non-temporelles ». Ce sont des instructions de type `load/store`, mais qui permettent de contourner tout ou partie de la hiérarchie des caches. Sur les processeurs x86, le jeu d'instruction AVX contient des instructions d'écriture non-temporelle (sur 256 bits à la fois) ; son extension AVX2 contient en outre une instructions de lecture non-temporelle (sur 256 bits à la fois).

8.5.2 Problème du « *False Sharing* »

L'explication donnée ci-dessus sur les politiques d'écriture dans les caches néglige un détail dans le cas des machines multi-coeurs. En effet, la plupart (mais il y a des exceptions...) maintiennent la *cohérence des caches*. Le problème est le suivant : un thread matériel écrit une valeur en RAM alors que d'autres unités d'exécution possèdent la même adresse dans leurs caches. Lorsque l'écriture en RAM a lieu, les caches des autres processeurs ne reflètent plus la bonne valeur en mémoire. Pour éviter cette situation, plusieurs solutions sont possibles ; l'une des plus commune consiste à instaurer un protocole de communication entre toutes les unités d'exécution :

# Coeurs	Temps (s)	Faute de cache / itération / coeur		
		L1	L2	L3
1	3.1	0.0	0.0	0.0
2	20.3	1.4	2.9	0.0
4	29.9	2.1	5	0.0
8	52.3	2.5	7.4	0.0

FIGURE 8.17 – Résultat de l'exécution du programme qui démontre le *False Sharing* sur le *cluster node* décrit figure 8.13.

lorsque l'une d'entre elle effectue une écriture, elle publie l'adresse où l'écriture a eu lieu ; tous les autres processeurs éjectent de leur cache la ligne qui contenait cette adresse si elle était présente (en effet son contenu est maintenant périmé). On parle alors d*invalidation* de la ligne de cache (en effet, elle n'est pas « éjectée » mais marquée comme « invalide »).

Ce protocole de maintien de la cohérence de cache a un coût, surtout sur les machines SMP, où des coeurs placés sur des processeurs physiquement distincts doivent communiquer sur une distance plus grande.

Une conséquence de cette politique de cohérence de cache, c'est que même si deux threads matériels accèdent à des adresses en mémoire disjointes (pas de conflit), ils peuvent malgré mutuellement invalider leur cache respectif de manière répétée s'ils écrivent tous les deux des données qui correspondent à la même ligne de cache. On parle alors de « *False Sharing* » : ils ne partagent pas de données, mais ils partagent une ligne de cache.

Voici un petit programme qui illustre ce problème.

```
double A[512][8] __attribute__((aligned(64)));
assert(omp_get_max_threads() < 8);
#pragma omp parallel
{                                   /* all threads execute this */
    int k = omp_get_thread_num();
    unsigned int j = 1 + k;         /* different pseudo-random sequence for each thread */
    for (int i = 0; i < 1000000000; i++) {
        A[j % 512][k] *= 3.14;
        j ^= j << 13;               /* randomize j (XorShift PRNG) */
        j ^= j >> 17;
        j ^= j << 5;
    }
}
```

Le tableau A est dimensionné de telle sorte qu'il occupe 32Ko (chaque ligne de la matrice occupe 64 octets, donc une ligne de cache complète — en plus on force le compilateur à aligner le tout sur une adresse qui est un multiple de 64, donc ça commence pile au début d'une ligne de cache). Le tableau tient donc tout entier dans un cache L1 courant de 32Ko.

Le thread t accède à $A[j][t]$ pour des valeurs arbitraires de j. Les écritures des threads ne sont donc pas conflictuelles (pas besoin de `#omp atomic` ni de synchronisation). Les différents threads ont des séquences d'indices j différentes. Mais on assiste à un phénomène de *False Sharing* : une valeur de j est associée à une seule ligne de cache, donc chaque fois qu'un coeur traite une valeur de j donnée, ça invalide la ligne de cache correspondante chez tous les autres. La figure 8.17 démontre que ça a un effet catastrophique sur les performances.

Avec un seul thread, il n'y a pas de fautes de cache : le tableau A tient tout entier dans le cache L1 (qui fait 32Ko). Sur la machine où le test a été réalisé, chaque coeur possède son propre cache L1 et son propre cache L2, tandis que le cache L3 est partagé. Il n'y a donc pas de *False Sharing* au niveau du cache L3. Par contre, on le voit apparaître au niveau du cache L1 et L2 dès qu'on utilise deux threads ou plus. En effet, chaque écriture par le thread t provoque l'invalidation de la ligne correspondante du cache L1 *et* du cache L2 de tous les autres coeurs.

Les chiffres précis de la figure 8.17 sont difficiles à interpréter [2], mais on voit bien que le problème s'aggrave avec le nombre de coeurs. Le système qui mesure le nombre de fautes de cache mesure *plus* d'une faute par itération, alors qu'il n'y a qu'une lecture/écriture à la même adresse.

2. pour l'auteur de ce document en en tout cas...

CPU	niveau	N	B	Associativité
Cortex A53	1	32K	64	4
	2	512K	64	16
PowerPC A2	1	16K	64	8
	2	32M	128	16
Xeon E5-2695 v4 (Broadwell)	1	32K	64	8
	2	256K	64	8
	3	45M	64	20
Core i7-6600U (Skylake)	1	32K	64	8
	2	256K	64	4
	3	4M	64	16
Xeon Gold 6130 (Skylake)	1	32K	64	8
	2	1M	64	16
	3	22M	64	11

FIGURE 8.18 – Associativité de quelques caches.

8.5.3 Associativité

Un cache de taille N formé de lignes de lignes de taille B possède donc N/B « emplacements ». Dans l'idéal, il devrait être capable de stocker n'importe quel bloc de la mémoire dans n'importe lequel de ses emplacements. S'il en est capable, on dit qu'il est *fully associative*. En réalité, c'est rarement le cas, car cela aurait un coût matériel prohibitif.

En effet, à chaque accès mémoire, il faut déterminer si l'adresse demandée est présente dans le cache ou non. Ceci commence par déterminer l'adresse du début du bloc de RAM (dont l'adresse est un multiple de B) qui contient l'adresse demandée. Ensuite, il faut déterminer si le bloc en question est présent dans le cache. Dans un cache *fully associative*, comme un bloc de mémoire peut potentiellement être stocké dans n'importe lequel des N/B emplacements, il est *grosso modo* nécessaire de tous les examiner. C'est donc coûteux, complexe, énergivore, etc.

À l'autre extrême, on pourrait imaginer un cache simple, qui à chaque bloc de la mémoire affecte *un* emplacement potentiel. Par exemple, si le cache contient 2^k emplacements, alors il suffit d'extraire k bits de l'adresse du début du bloc, et ça désigne un des emplacement du cache. L'algorithme qui teste si un bloc est présent dans le cache est alors particulièrement simple et efficace : il n'y a qu'un emplacement à tester ! Un tel cache, rudimentaire, est dit *direct-mapped*. L'inconvénient de cette stratégie c'est qu'elle augmente le nombre de fautes de cache, car si deux adresses sont accédées alternativement et sont « en compétition » pour le même emplacement, alors chaque accès causera une faute de cache, même s'il est complètement vide par ailleurs.

Entre le *direct-mapped* (simple, peu performant) et le *fully-associative*, on trouve toute une gamme de compromis avec les caches *set-associative*. Dans un tel cache, chaque bloc de RAM peut être stocké dans un *sous-ensemble* (de taille fixe) des emplacements. Le cache est découpé en « *sets* » de petite taille et chaque bloc de RAM est affecté d'office à un *set* ; par contre, au sein d'un *set* il peut occuper n'importe quel emplacement.

Un cache *k-associatif* possède des *sets* de taille k : il peut stocker un bloc arbitraire dans seulement k emplacements possibles (sur les N/B). L'avantage, c'est que pour tester si un bloc est présent dans le cache, il suffit de tester seulement k emplacements possibles. Une des manières dont ceci peut être réalisé consiste à extraire une partie des bits de l'adresse du bloc à stocker pour désigner le *set*.

La quasi-totalité des caches sur les processeurs commerciaux sont associatifs. La figure 8.18 donne quelques exemples. Là encore, on voit qu'il y a beaucoup de variabilité d'une machine à l'autre. La seule espèce de constante est la taille des lignes de cache fixée à 64 octets.

L'associativité des caches pose parfois des problèmes en causant des fautes de caches inattendues, en particulier quand on travaille avec des tableaux dont la taille est une puissance de deux. Par exemple, considérons le petit bout de code suivant, qui transpose (naïvement) une matrice carrée.

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

Si on l'exécute sur de petites valeurs de N (avec les deux matrices qui tiennent largement dans un cache L2 de 256Ko, par exemple), on observe un comportement pathologique pour $N = 2^7$ et $N = 2^8$, ainsi que le montre la figure 8.19. Lorsque les deux matrices ne tiennent pas dans le cache L1 (ce qui est le cas dès que $N \geq 46$),

N	Temps (ms)	Faute L1 / `double` (lecture)	Faute L1 / `double` (écriture)
31	1.3	0	0
32	1.3	0	0
33	1.3	0	0
63	3.5	0.12	0.11
64	3.5	0.22	0.12
65	4	0.11	0.10
127	8	0.13	0.12
128	16	1	0.12
129	9	0.23	0.12
255	36	0.13	0.11
256	119	1	0.12
257	37	0.14	0.11

FIGURE 8.19 – Mise en évidence problèmes d'associativité (transposition naïve). Les deux colonnes de droites indiquent le nombre de fautes de cache par `double` copié de A vers B (mesure expérimentale).

on s'attend à ce qu'une lecture de A (resp. écriture de B) sur huit cause une une faute de cache L1 `double` : chaque faute aboutit au transfert des 8 prochains éléments de A (resp. B) dans le cache. C'est bien ce qu'on observe ($1/8 = 0.125$).

On voit que ce chiffre est sensiblement dépassé pour les puissances de deux, puisqu'on a alors une faute de cache par lecture de A ! Le temps d'exécution est lui aussi sensiblement augmenté. Ceci s'explique par le fait que les adresses lues (pour A) dans deux itérations successives diffèrent d'une puissance de deux (suffisamment grande). Dans le cas de ce cache L1 de 32Ko avec des lignes de $B = 64$, on peut parier que le *set* d'un bloc de RAM contenant une adresse x est déterminé en prenant les bits $[6:12]$ de l'écriture de x en binaire.

Supposons que c'est vrai : ceci impliquerait que deux adresses distantes de $k \cdot 2^{12}$ octets tombent forcément dans le même *set* (pour n'importe quel k), qui ne contient que 8 emplacements. Donc, avec $N = 64$, lors d'un tour de la boucle externe, les adresses lues dans A ($8 \times 64j + 8i$) ne sont pas contiguës, et elles appartiennent toutes à des lignes de cache différentes. Or, ces lignes de cache ne peuvent tomber que dans un sous-ensemble de 8 *sets* sur 64 (par exemple, pour $i = 0 : 0, 8, 16, \ldots, 48, 56$).

En tout état de cause, 64 lignes de cache à répartir dans 8 sets de capacité 8, ça peut tenir, mais tout juste ! En pratique on observe un début de dégradation des performances pour $N = 64$. Par contre, à partir de $N = 128$, ça craque : les lignes transférées vers le cache L1 pour servir les lectures de A ne peuvent atterrir que dans 4 *sets* : le nombre total d'emplacements disponibles n'est plus alors que de 32, alors qu'il en faudrait 128. En pratique, chaque nouvelle lecture de A provoque une faute.

Le programme le plus simple qui exhibe le comportement problématique est le suivant :

```
double A[4608];
...
double x = 0;
for (int k = 0; k < N; k++)
    for (int i = 0; i < 9; i++)
        x += A[512 * i];
```

Avec un cache 8-associatif, il effectue une faute de cache par itération de la boucle externe, alors qu'il n'accède qu'à 9 adresses différentes, pour un total de 72 octets !