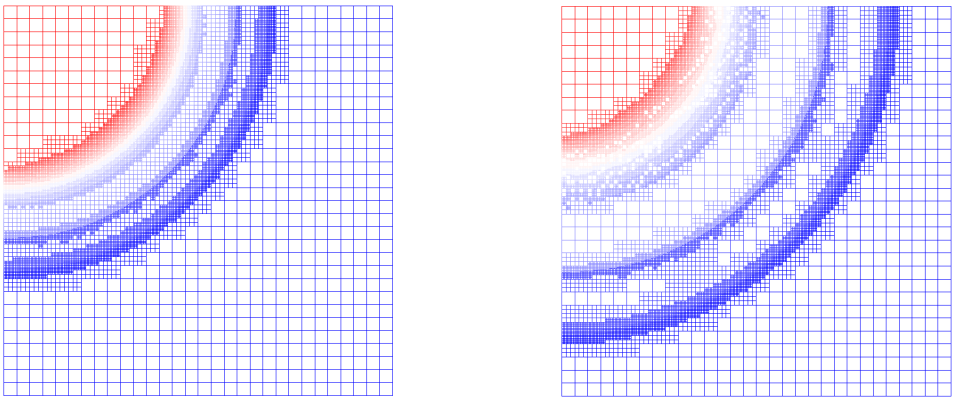


(a) Cycle 171. (b) Cycle 337.

FIGURE 1.1 – Exemple d'évolution de maillage A.M.R.

par exemple, le maillage n'est raffiné qu'au niveau de l'onde de choc, le reste du domaine est simulé par un maillage grossier. Cependant, il arrive que ces portions changent au cours du temps, lorsqu'une onde de choc se propage par exemple. Il est donc courant d'utiliser un *maillage adaptatif* (ou A.M.R.), c'est-à-dire de faire évoluer le raffinement du maillage au cours de la simulation, pour « suivre » le phénomène physique simulé. On voit ainsi sur la figure 1.1(b) que le maillage a été raffiné à la nouvelle position de l'onde de choc, tandis qu'il a été regrossi à son ancienne position. La précision de calcul au niveau du phénomène intéressant peut ainsi être très grande sans que le coût explose : l'occupation mémoire et le temps de calcul sont dépensés essentiellement aux endroits utiles.

Par ailleurs, il est courant d'utiliser un *couplage de codes* : lorsqu'une simulation met en jeu des éléments de natures très différentes (liquide / solide par exemple), il est préférable d'utiliser pour chaque élément un code de simulation qui y est adapté, et de coupler les codes entre eux au niveau des interfaces entre éléments. On se retrouve alors avec plusieurs codes de natures éventuellement très différentes, à faire exécuter de concert sur une même machine.

1.1.2 Un comportement irrégulier

La conséquence de tels raffinements est que le comportement des applications de calcul scientifique devient irrégulier. Il l'est d'abord au sens où la charge de calcul et d'occupation mémoire n'est pas homogène (c'est d'ailleurs l'objectif visé !). On ne peut donc pas se contenter d'utiliser des solutions de répartition de travail triviales. Le comportement est de plus irrégulier au sens où il évolue au cours du temps, selon un schéma qui n'est souvent *pas prévisible a priori* : il dépend des résultats intermédiaires obtenus lors de l'exécution. Dans le

cas de calculs matriciels creux, il peut être possible de prévoir l'évolution des blocs nuls, mais dans le cas d'un maillage A.M.R., le raffinement effectué est très vite difficilement prévisible. Les numériciens parviennent dans une certaine mesure à prévoir dynamiquement comment le raffinement évolue, mais cette prévision ne peut évidemment pas être parfaite (cela voudrait dire que l'on connaît déjà le resultat de la simulation !). Les solutions de répartition de travail doivent donc en plus être *dynamiques*.

1.1.3 Des besoins en ordonnancement particuliers

Enfin, les besoins en ordonnancement de telles applications sont souvent particuliers. En effet, elles utilisent souvent pour leurs calculs des routines BLAS. De telles routines sont « calées » pour exploiter au maximum un processeur et son cache. Il est donc préférable de ne pas interrompre de telles routines pendant leur exécution, pour que la réutilisation du cache soit maximale. Or en général, pour préserver l'*équité* entre les tâches, les systèmes d'exploitation *préemptent* régulièrement la tâche en cours pour en laisser exécuter une autre, afin que toutes les tâches progressent globalement de manière équitable. Dans le cas d'une application de calcul scientifique, c'est inutile : l'objectif est surtout de donner le résultat le plus vite possible. C'est pourquoi les numériciens ont tendance à faire désactiver les mécanismes de préemption. Par contre, il est courant de décomposer les calculs pour réduire la taille des données manipulées par chaque opération à la taille du cache. Les nombreuses tâches résultant de cette décomposition peuvent alors très bien être exécutées (chacune entièrement) dans un ordre quelconque et en parallèle, tant que les dépendances de données sont respectées.

Ainsi donc les applications de calcul scientifique ont des besoins particuliers. Elles doivent pouvoir exprimer la finesse asymétrique de leur structure (pour permettre une distribution de charge fine), et garder un certain contrôle sur l'ordonnancement effectué (pouvoir désactiver toute préemption par exemple).

1.2 Vers une hiérarchisation des machines multiprocesseurs

Le grand public a récemment découvert de nouveaux termes tels que *HyperThreading* et *multicore*, et il lui devient de plus en plus difficile d'acheter une machine qui ne soit pas parallèle. Ceci n'est que la partie émergée d'une tendance profonde du calcul scientifique vers des machines toujours complexes, où le parallélisme s'instaure à tous les niveaux. Dans cette section, nous présentons les caractéristiques des technologies principales de parallélisme actuelles, dont la combinaison produit des machines très hiérarchisées.

1.2.1 Accès mémoire non uniformes

La forme la plus simple de machine parallèle est l'architecture *Symmetric MultiProcessing* (**SMP**), telle que représentée à la figure 1.2 : plusieurs processeurs sont connectés à une mémoire commune via un bus ou un commutateur parfait (*crossbar*). Cependant, avec un nombre croissant de processeurs, une telle implantation devient rapidement problématique :

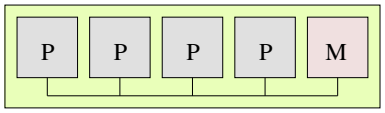


FIGURE 1.2 – Architecture SMP.

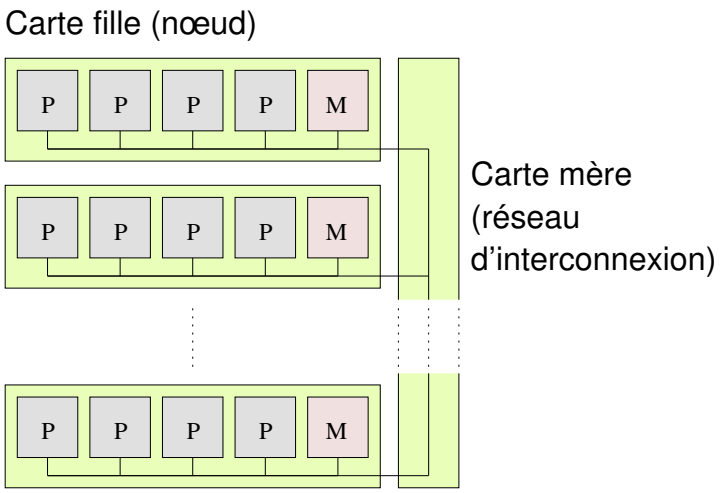


FIGURE 1.3 – Architecture NUMA.

un bus est limité par sa bande passante, et le coût d'un commutateur devient très vite prohibitif. Une solution courante est alors d'utiliser une architecture à accès mémoire non uniforme (*Non-Uniform Memory Access*, **NUMA**), comme représentée sur la figure 1.3. La mémoire y est distribuée sur différents **nœuds**, qui sont reliés par un réseau d'interconnexion pour que tous les processeurs puissent tout de même accéder de manière transparente à toute la mémoire. Typiquement, les nœuds sont des cartes filles enfichées sur une carte mère. Dans une telle machine, les accès mémoire dépendent alors de la position relative du processeur qui effectue l'accès et de l'emplacement mémoire accédé. La latence d'accès à une donnée *distante* peut être par exemple 2 fois plus grande que celle pour une donnée *locale*, car l'accès doit transiter par le réseau d'interconnexion. C'est le **facteur NUMA**, qui dépend fortement de la machine et peut typiquement varier de $1,1$ à 10 ! En pratique, il est généralement de l'ordre de 2, et est légèrement plus grand pour une lecture que pour une écriture.

Ce type d'architecture était traditionnellement plutôt réservé aux super-calculateurs, or comme le montre le classement Top500 [top], ceux-ci tendent depuis quelque temps à laisser la place aux grappes de simples P.C., si bien que l'architecture NUMA était quelque peu tombée dans l'oubli. Cependant, dans ses puces OPTERON le fondeur A.M.D. intègre désormais un contrôleur mémoire directement dans le processeur pour pouvoir y connecter directement la mémoire et ainsi augmenter fortement la bande passante disponible. Cela induit en fait une architecture NUMA, puisque la mémoire se retrouve alors répartie sur les différents processeurs. Ces puces étant devenues à la mode dans le grand public, l'architecture NUMA s'est en fait fortement démocratisée, et on la retrouve donc naturellement dans les grappes de P.C.

Le problème posé par ces machines est bien illustré par Henrik Löf *et al.* au sein de leurs

	Séquentiel	Parallèle	*Speedup*
SUN ENTERPRISE 10 000	220 s	8 s	27,5
SUN FIRE 15 000	45 s	9,5 s	4,7

TABLE 1.1 – Temps d'exécution d'un solveur PDE.

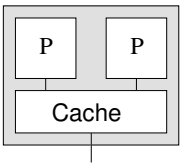


FIGURE 1.4 – Puce multicœur.

travaux sur un solveur PDE [LH05]. Ils disposaient d'une machine SMP SUN ENTERPRISE 10 000 composée de 32 processeurs cadencés à 400MHz avec une latence d'accès mémoire de 550ns, et ont acquis une machine NUMA SUN FIRE 15 000 composée de 32 processeurs également, cadencés à 900MHz avec une latence d'accès mémoire local de 200ns et une latence d'accès mémoire distant de 400ns (soit un facteur NUMA égal à 2). Comme le montre la première colonne du tableau 1.1, l'augmentation de la fréquence des processeurs et la diminution du temps de latence d'accès mémoire permet au temps d'exécution séquentiel d'être divisé par presque 5. Cependant, le temps d'exécution parallèle est plus que décevant : la nouvelle machine se révèle *« plus lente »* que l'ancienne ! La raison à cela est que leur solveur initialise toutes les données séquentiellement sur le premier processeur, et le système SOLARIS les alloue alors dans la mémoire du nœud correspondant, le premier. Lorsque l'exécution parallèle commence réellement, tous les processeurs tentent alors d'accéder à la mémoire du premier nœud, qui se retrouve engorgée. Il apparaît donc essentiel, sur de telles machines, de bien contrôler l'emplacement effectif des allocations mémoire par rapport aux processeurs.

1.2.2 Puces multicœurs

Avec une finesse de gravure toujours plus poussée, les fondeurs disposent de plus en plus de place sur les puces. Pendant longtemps, cela a permis de fabriquer des processeurs de plus en plus sophistiqués avec des opérations flottantes avancées, une prédiction de branchement, etc. De nos jours cependant, la sophistication est devenue telle qu'il est devenu plus simple, pour gagner toujours plus en performances, de graver plusieurs processeurs sur une même puce : ce sont les puces multicœurs, telle qu'illustrée figure 1.4. Tous les grands fondeurs proposent désormais des déclinaisons biccœurs, quadricœurs, voire octocœurs de leurs processeurs, tels le NIAGARA 2 de SUN, ou le CELL d'IBM popularisé par la PLAYSTATION 3 de SONY. Les puces avec 16 cœurs sont déjà sur les agendas, et Intel prévoit même de fabriquer un jour des puces embarquant des centaines de cœurs [Rat] !

Une caractéristique intéressante de ces puces est que, puisque les processeurs sont gravés sur une même puce, il est courant de leur donner un accès en commun à un seul et même cache. En effet, lorsque les deux processeurs exécutent une même application ou une même bibliothèque par exemple, il serait dommage de garder une copie du programme au sein de chacun des processeurs. Par ailleurs, si les deux processeurs ont besoin de communiquer

entre eux, ils peuvent le faire très rapidement par mémoire partagée *via* ce cache. Au-delà de deux cœurs, il est fréquent que plusieurs niveaux de caches soient partagés de manière hiérarchique, tel que le conseillent Hsu *et al.* [HIM+05].

Cependant, la bande passante d'accès vers l'extérieur de la puce est elle aussi partagée. On le constate effectivement expérimentalement : nous avons mesuré sur une puce bicœur Opteron que si un seul processeur écrit en mémoire, il dispose d'une bande passante d'environ $4,4\,\text{Go/s}$, alors que lorsque les deux processeurs écrivent en même temps, ils disposent chacun d'environ $2,1\,\text{Go/s}$.

On est donc déjà ici face à un dilemme : selon les applications, il vaudra mieux placer des tâches sur une même puce multicœur pour qu'elles bénéficient du partage du cache, ou bien au contraire les répartir sur différentes puces pour que chaque tâche puisse bénéficier de la bande passante mémoire maximale et qu'elles évitent d'empiéter l'une sur l'autre dans le cache. Il faudra donc que l'ordonnanceur puisse prendre en compte les contraintes applicatives pour effectuer des choix appropriés.

1.2.3 Processeurs multithreadés

Le parallélisme est en fait au cœur même des processeurs grand public depuis longtemps. Les premiers processeurs Pentium par exemple disposaient déjà de plusieurs unités de calcul pour exécuter en parallèle deux instructions successives d'un programme séquentiel, quand les contraintes de données le permettent. Ce niveau de parallélisme n'est cependant pas vraiment « visible » pour le programmeur : le programme fourni reste séquentiel. Les compilateurs s'efforcent en général, dans le code généré, de disperser les instructions dépendant les unes des autres pour permettre une exécution parallèle, mais cela reste limité. Des instructions spéciales (du type MMX ou SSE du côté des Pentiums) permettent de fournir explicitement une série d'instructions flottantes à effectuer, que le processeur peut traiter facilement en parallèle. Cependant, cela nécessite soit d'écrire le code explicitement en assembleur[1], soit d'inclure un module de vectorisation dans le compilateur. Un tel module n'est cependant bien souvent pas à même de parvenir à extraire de grandes portions de code de calcul vectoriel. Par ailleurs, les unités de calcul sont de plus en plus fortement pipelinées. Lorsqu'une donnée n'est pas disponible immédiatement dans le cache, ces longs pipelines deviennent progressivement inactifs, en attente de l'accès mémoire (ce qu'on appelle *bulle*). Pour mieux profiter de ce parallélisme et tenter de combler les bulles dans les pipelines, certains processeurs contiennent de quoi exécuter plusieurs programmes en parallèle, c'est le *Simultaneous MultiThreading* (**SMT**, aussi appelé *HyperThreading* par INTEL [MBH+02], représenté figure 1.5). Les différents programmes (en général un nombre fixe pour un processeur donné) progressent selon l'occupation courante du processeur. Les détails d'ordonnancement varient d'un processeur à un autre, mais le processeur peut par exemple à chaque cycle prendre une instruction d'un ou plusieurs programmes (tour à tour) selon les disponibilités des unités de calcul et du type de l'instruction en cours des programmes. Cela permet ainsi en général d'alimenter toutes les unités de calcul, et lorsqu'un programme est bloqué en attente d'une lecture mémoire, les autres peuvent progresser plus « souvent ».

[1] On peut citer le cas du codec Xvid qui exploite au maximum cette possibilité, si bien que la version parallèle, lancée sur un processeur hyperthreadé, fonctionne exactement à la même vitesse : la version séquentielle profite en fait *déjà* au maximum de toutes les unités de calcul.

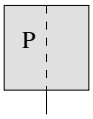


FIGURE 1.5 – Processeur multithreadé 2 voies.

	Flottant	Entier	Inactif
Flottant	280 / 280	120 / 281	- / 282
Entier	281 / 120	110 / 110	- / 210

TABLE 1.2 – Confrontation de deux types de calculs sur processeur hyperthreadé, en itérations par seconde.

Le gain obtenu n'est cependant pas forcément évident. En effet, nous avons essayé de combiner des boucles de calcul entier et flottant très simples sur les puces hyperthreadées d'INTEL qui fournissent deux *processeurs virtuels*. Les résultats sont représentés dans le tableau 1.2, où l'on peut lire la vitesse d'exécution des deux boucles, pour chaque combinaison possible. On constate que la boucle de calcul flottant n'est que très peu ralentie lorsqu'on l'exécute en parallèle sur les deux processeurs virtuels. La boucle de calcul entier, par contre, voit sa vitesse presque divisée par deux. On peut en déduire que ce processeur possède au moins deux unités de calcul flottant, permettant vraiment d'exécuter nos deux boucles en parallèle, mais qu'il ne possède qu'une unité de calcul entier, que les deux processeurs virtuels doivent se partager. La combinaison Entier/Flottant est beaucoup moins évidente à interpréter : la boucle de calcul flottant semble fortement ralentir la boucle de calcul entier.

De manière générale, le comportement des combinaisons de programmes non triviaux est assez difficile à prévoir. Intel, pour sa part, annonce un gain de performances qui peut atteindre 30%. Bulpin et Pratt [BP04] ont essayé de combiner les différents programmes de la suite SPEC CPU2000, et obtiennent effectivement parfois un gain de 30%, corrélé avec un taux de défauts de cache élevé. À l'inverse, lorsque le taux de défauts de cache est faible, ils observent parfois une perte de performances ! C'est pourquoi bien souvent, pour le calcul scientifique, l'*HyperThreading* est désactivé...

1.2.4 De véritables poupées russes

Nous avons jusqu'ici étudié différentes technologies indépendamment les unes des autres. En pratique, elles sont très souvent combinées, ce qui produit des machines très hiérarchisées, telles que les machines SUN WILDFIRE [HK99], SGI ALTIX [WRRF03] et ORIGIN [LL97], BULL NOVASCALE [Bul] ou l'IBM P550Q [AFK+06]. La figure 1.6 montre un exemple d'une machine combinant une architecture NUMA avec des puces multicœurs dont chaque cœur est lui-même hyperthreadé. On peut remarquer que le réseau d'interconnexion est ici lui-même hiérarchisé. Sur d'autres machines cependant, il a parfois une topologie moins symétrique. La figure 1.7 montre par exemple l'architecture d'une des machines de test utilisées pour le chapitre 6 : les 8 nœuds NUMA y sont reliés d'une manière assez particulière. Le

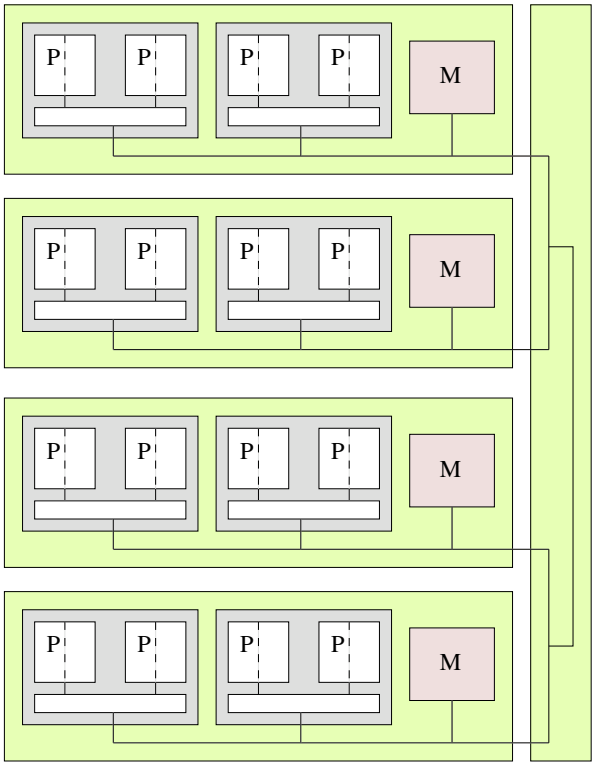


FIGURE 1.6 – Architecture très hiérarchisée.

facteur NUMA varie ainsi beaucoup selon les positions respectives du processeur effectuant l'accès et l'emplacement mémoire accédé. En pratique, nous avons pu mesurer des facteurs variant d'environ $1,1$ pour des nœuds voisins à environ $1,4$ pour des nœuds extrêmes. Il faut noter que ce n'est pas forcément seulement ce facteur qui est pénalisant, mais aussi les bandes passantes qui sont limitées sur chaque lien.

Il devient ainsi bien difficile d'arriver à exploiter de telles machines. L'écart se creuse de plus en plus entre les performances de crête censées être disponibles et les performances réellement obtenues. Par exemple, la machine TERA10, acquise par le CEA pour obtenir une puissance de calcul de 10 TeraFlops pour ses simulations, est en réalité une machine « Tera65 » : sa puissance de crête est de 65 TeraFlops, que cependant aucun code de calcul réel n'est à même d'atteindre...

La variété de ces machines étant par ailleurs très grande, exploiter ces machines de manière *portable* est d'autant plus un défi. Certains constructeurs tentent de palier ces problèmes avec certains artefacts. Par exemple, il est souvent possible de configurer la carte mère pour fonctionner en mode *entrelacé*. En principe sur une machine NUMA, l'espace d'adressage mémoire physique est divisé en autant de fois qu'il y a de nœuds, le système d'exploitation pouvant ainsi allouer une page sur un nœud donné en choisissant simplement son adresse à l'intérieur de l'espace d'adressage correspondant au nœud. En mode entrelacé, les pages mémoire des différents nœuds sont distribuées dans l'espace d'adressage mémoire physique de manière cyclique : avec 2 nœuds par exemple, les pages de numéro pair proviennent du nœud 0 tandis que les pages de numéro impair proviennent du nœud 1. Ainsi, en moyenne

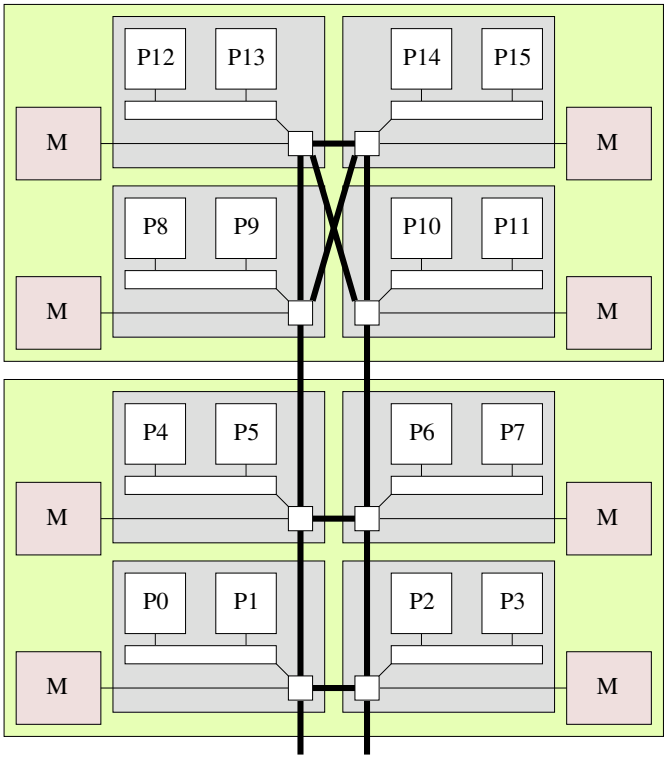


FIGURE 1.7 – Hagrid, octo-bicœur Opteron.
Chaque nœud NUMA est composé d'une puce bicœur Opteron cadencée à $1,8$ GHz et de 8 Go de mémoire. Ces nœuds sont reliés par un réseau de liens HyperTransport. Le facteur NUMA varie selon la position relative de l'emplacement mémoire accédée et du processeur effectuant l'accès. En pratique, nous avons observé grossièrement trois valeurs typiques de facteurs NUMA pour cette machine : environ $1,1$, $1,25$ et $1,4$. Nous en avons alors déduit la topologie du réseau montrée ci-dessus, puisque les facteurs minimum correspondent à un seul saut dans le réseau. Cette topologie apparemment étrange s'expliquer par le fait que la machine est en fait physiquement composée de deux cartes mères superposées. En pratique cette partition en deux n'a pas d'impact sur les latences d'accès mémoire.

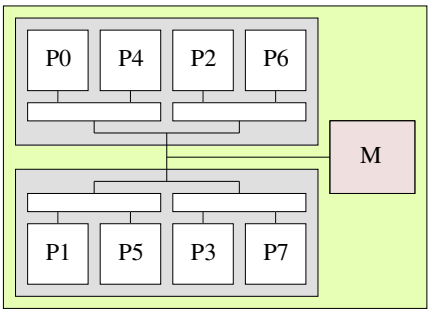


FIGURE 1.8 – Aragog, bi-quadricœur E5345 Xeon.
Cette machine est composée de deux puces cadencées à $2,33$ GHz, chacune embarquant quatre cœurs. L'organisation hiérarchique des caches peut s'observer en mesurant la vitesse d'une boucle intensive de calcul sur des variables *volatiles* situées dans une même ligne de cache (en situation de faux partage, donc), par exemple une simple décrémentation. Lorsqu'un seul thread est lancé, il peut effectuer environ 330 millions d'itérations à la seconde. Lorsqu'un autre thread est lancé de telle sorte que le cache L2 est partagé, tous deux peuvent effectuer environ 280 millions d'itérations à la seconde, soit environ $1,2$ fois moins. Lorsqu'ils ne partagent pas le cache L2 mais sont placés sur la même puce, ils ne peuvent effectuer qu'environ 220 millions d'itérations à la seconde, soit $1,5$ fois moins. Lorsqu'ils sont placés sur des puces différentes, ils ne peuvent effectuer que seulement 63 millions d'itérations à la seconde, soit environ $5,2$ fois moins !

sur plusieurs pages, le temps d'accès paraît assez homogène, égal à la moyenne entre le temps d'accès local et le temps d'accès distant. C'est une solution simple pour éviter les cas pathologiques qui surviennent très souvent avec les programmes qui ne sont pas pensés pour machines NUMA ; c'est cependant du gâchis de performances, puisque l'on n'obtient qu'un temps d'accès moyen. De plus, un agent commercial peut alors faire croire à ses clients que « ce n'est pas une machine NUMA »... À titre anecdotique, signalons qu'à l'arrivée d'une commande de deux machines, nous avons constaté que l'entrelacement mémoire était activé sur une machine mais pas sur l'autre !

Un autre artefact courant repose sur la numérotation des processeurs. La figure 1.8 montre une machine dont la numérotation matérielle des processeurs est à première vue étrange. Lorsque l'on exécute une simulation occupant tous les processeurs et que les communications sont localisées entre tâches de numéros consécutifs, la stratégie usuelle (et d'habitude plutôt efficace), qui est d'attribuer les tâches dans l'ordre des processeurs, devient une des pires idées que l'on puisse imaginer ! Cette numérotation particulière est en fait prévue pour le cas où l'on dispose de moins de tâches à effectuer qu'il n'y a de processeurs, et que l'on attribue bêtement ces tâches dans l'ordre des processeurs. Les tâches disposent alors ici de la plus grande taille de cache pour elles seules, et du maximum de bande passante possible. Ce scénario se reproduit de la même façon dans le cas des processeurs hyperthreadés, et de même que pour l'entrelacement mémoire, si cette numérotation est souvent configurable, la configuration effectivement activée est parfois aléatoire...

1.3 Du programmeur à l'environnement d'exécution

Face à de telles machines, on peut comprendre que les scientifiques non informaticiens soient quelque peu perplexes. Comment peut-on programmer ces machines de manière portable ? Il apparaît donc nécessaire de disposer d'un *environnement d'exécution* qui masque tous ces détails techniques et permette aux scientifiques de se concentrer sur leurs propres problématiques. Nous présentons ici pourquoi il reste tout de même nécessaire d'exprimer d'une manière ou d'une autre le parallélisme, et dans quelle mesure il est possible de le faire sans être spécialiste en la matière.

1.3.1 Une nécessité d'expliciter le parallélisme

La parallélisation automatique permet dans les cas simples d'exécuter un programme sur une machine parallèle sans modifications (ou très peu). Ainsi, HPF (*High-Performance Fortran* [KLS+94, Sch96]) est un ensemble de pragmas pour le langage Fortran 90 pour la parallélisation automatique. Fortran 90 fournit au programmeur des opérations qui agissent directement sur des matrices et vecteurs. HPF parallélise automatiquement ces opérations en découpant ces opérations en bandes de matrices et éléments de vecteurs, et en les distribuant aux différents processeurs. Cependant, comme nous l'avons vu à la section 1.1 (page 6), les applications de calcul scientifique deviennent de plus en plus complexes, loin d'un schéma aussi basique, et les parallélisations complètement automatiques sont alors incapables d'extraire du parallélisme. De plus, les parallélisations mises en jeu, nécessairement simplistes, ne peuvent pas passer à l'échelle des machines décrites dans la section précédente.

Ainsi, il n'y a pas de solution miracle, et il est nécessaire de modifier les programmes, même légèrement, pour exprimer du parallélisme de manière appropriée. La question qui se pose est la finesse de parallélisme qu'il faut exprimer. Des approches comme Cilk [FLR98] ont montré que pour obtenir les meilleures performances, il faut exprimer le plus de parallélisme possible. Il faut bien sûr savoir rester raisonnable pour éviter que l'expression même du parallélisme prenne plus de temps que les tâches ainsi isolées, mais plus l'environnement d'exécution dispose d'un parallélisme raffiné, plus il sera à même d'effectuer un équilibrage dynamique de charge adapté à la machine cible. C'est d'autant plus vrai dans le cas d'une application très irrégulière pour laquelle on ne sait pas *a priori* comment pourra se répartir la charge.

1.3.2 Des programmeurs non spécialistes en parallélisme

Il est certes nécessaire d'exprimer du parallélisme, mais les programmeurs de codes de calcul scientifique ne sont en général pas des informaticiens, et donc encore moins des spécialistes du parallélisme. De ce point de vue, l'interface de parallélisme la plus standard, les threads POSIX (PTHREAD), est en pratique d'un niveau de programmation bien trop technique pour être réellement utilisable telle quelle. Par ailleurs, en ce qui concerne le placement des données sur machine NUMA, aucun outil n'est fourni.

Pour ces raisons, des interfaces de plus haut niveau ont été conçues. OPENMP [ope, ME99] par exemple, va plus loin qu'HPF en proposant un ensemble d'annotations (sous forme de

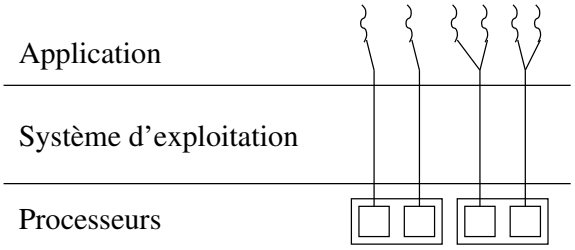


FIGURE 1.10 – Ordonnancement pré-calculé, court-circuitant le système d'exploitation.

suite déterminer un ordonnancement des threads et un placement des données adaptés à la complexité de la machine cible (voire optimaux). Enfin, on exécute simplement l'application en exigeant du système d'exploitation l'ordonnancement et placement voulus[3], tel qu'illustré figure 1.10, en supposant que la machine est dédiée à l'application. Les performances obtenues sont alors excellentes.

C'est ainsi qu'en connaissant à l'avance le graphe des tâches à effectuer (y compris leur temps de calcul), Lai et Chen [LC96a] et Narlikar [Nar02] sont à même de les exécuter d'une manière la plus appropriée possible, en privilégiant notamment un parcours local du graphe pour privilégier les affinités et ainsi bénéficier des effets de cache. Acar *et al.* [ABB02] y ajoutent une stratégie de vol local pour compenser les déséquilibres éventuels lorsque l'estimation du temps de calcul n'est pas assez fiable.

Le solveur PASTIX [HRR00] illustre également particulièrement bien cette approche : il résout de (très) grands systèmes linéaires creux par une méthode directe en calculant d'abord un ordonnancement statique des calculs par blocs et des communications, par l'intermédiaire d'une simulation utilisant une modélisation des opérateurs BLAS et de la communication sur l'architecture cible.

Enfin, pour distribuer de manière appropriée les données d'une application sur une machine NUMA sans autre connaissance approfondie de l'application que sa boucle principale, Marathe et Mueller [MM06] effectuent une première exécution de la première itération seulement de l'application, pendant laquelle ils utilisent les compteurs de performances de l'ITANIUM pour obtenir une trace approximative des accès effectués par chaque processeur pendant cette itération. Cette trace permet alors de décider du placement effectif des pages sur les différents nœuds NUMA. L'application est alors relancée avec ce placement mémoire, ce qui améliore grandement les performances par rapport aux approches opportunistes. Par rapport au mécanisme de migration automatique expliqué dans la section précédente, le point clé est que la mesure est effectuée pendant exactement *une itération*. Elle n'est perturbée ni par la phase d'initialisation, ni par le biais qui survient en faveur du début ou de la fin de la boucle principale lorsqu'une mesure dure pendant par exemple une itération et demie. L'estimation qui en résulte est ainsi très caractéristique de l'exécution de l'application.

Les performances obtenues par des méthodes de pré-calcul sont ainsi excellentes. Leur problème est qu'elles ne peuvent être utilisées que pour des applications dont le comportement

[3] Tous les systèmes actuels proposent une interface pour fixer les threads noyau sur les processeurs et les pages mémoire sur les nœuds.

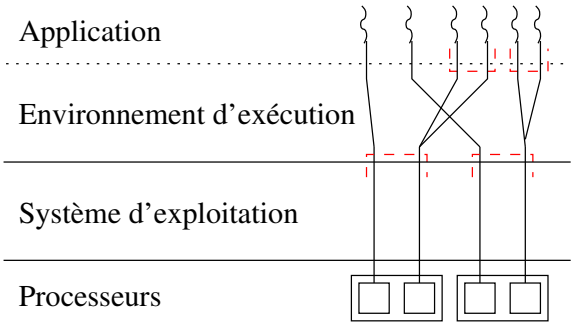


FIGURE 1.11 – Ordonnancement négocié, utilisant des informations de l'application et s'adaptant à la machine sous-jacente.

est régulier. Lorsque le comportement de l'application dépend des résultats intermédiaires obtenus, il est impossible de pré-calculer un ordonnancement ou un placement.

1.4.3 Approches négociées

Entre ces deux approches extrêmes, il existe des approches qui, tout en restant plutôt génériques, font intervenir des formes de négociation entre l'environnement d'exécution et la machine sous-jacente.

C'est ainsi que les compilateurs pour langages parallèles (OPENMP, HPF, UPC, etc. décrits à la section 1.3.2, page 16) compilent les programmes de manière suffisamment générique pour pouvoir s'adapter aux différentes architectures parallèles, et ajoutent au code généré une portion qui détermine *à l'exécution* l'architecture de la machine (le nombre de processeurs par exemple) et adapte le démarrage de threads et le placement des données à cette architecture (autant de threads que de processeurs par exemple).

Certains de ces compilateurs (Omni/ST pour OPENMP [TTSY00] par exemple) vont plus loin en embarquant un environnement d'exécution complet au sein de l'application. Celui-ci court-circuite complètement l'ordonnancement effectué par le système d'exploitation en supposant que la machine est dédiée à l'application et en effectuant explicitement en espace utilisateur les changements de contexte entre threads. Certains systèmes d'exploitation (tels que les anciennes versions de SOLARIS ou les versions récentes de FREEBSD) fournissent même le mécanisme de *Scheduler Activation* pour traiter le problème des appels systèmes bloquants que ce genre d'approche rencontre. L'environnement d'exécution embarqué peut alors contrôler complètement l'ordonnancement des threads, ce qui permet de prendre en compte les informations que le compilateur a pu collecter tout autant que l'architecture de la machine cible, tel qu'illustré figure 1.11. Un tel environnement est cependant plutôt difficile à mettre au point, et se limite donc assez souvent à un ordonnancement simple de tâches.

1.4.4 Paramètres et indications utiles à l'ordonnancement

Les approches négociées sont intéressantes, car elles sont à même de prendre en compte toute information utile tout en restant assez génériques. Or des informations peuvent venir

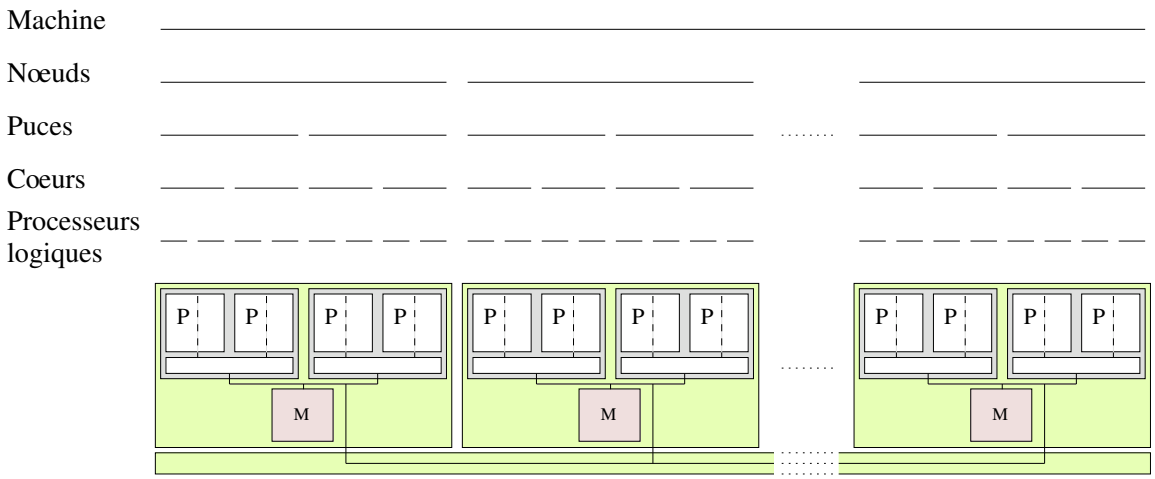


FIGURE 2.3 – Modélisation d'une machine très hiérarchisée par une hiérarchie de listes.

2.2 Coller à la structure de la puissance de calcul

Une des difficultés lorsque l'on écrit un ordonnanceur est due au modèle de VON NEUMANN : ce sont les processeurs eux-mêmes qui exécutent les instructions d'un thread et basculent éventuellement sur un autre lorsque le précédent se termine ou se bloque en attente d'une ressource. Ainsi, *a priori*, on ne *confie* pas un thread à un processeur, c'est plutôt le processeur qui doit de lui-même déterminer le prochain thread à exécuter. Toute la difficulté est alors que les différents processeurs doivent parvenir à faire ce choix *de concert* et le plus rapidement possible. Nous avons vu à la section 1.4.1 (page 18) que la solution généralement adoptée par les algorithmes opportunistes était de disposer de listes de threads dans lesquelles les processeurs viennent piocher. Par ailleurs, Dandamudi et Cheng [DC97], Oguma et Nakayama [ON01] et Nikolopoulos *et al.* [NPP98] s'accordent à dire qu'il vaut mieux utiliser au moins deux niveaux de listes de threads pour à la fois éviter des contentions et pouvoir facilement répartir la charge.

Nous poussons donc la logique jusqu'au bout en modélisant l'architecture de la machine cible par une hiérarchie de listes de threads. À chaque élément de chaque étage de la hiérarchie de la machine est associé (*bijectivement*) une liste de threads. La figure 2.3 montre une machine très hiérarchique et sa modélisation. La machine en entier, chaque nœud NUMA, chaque puce physique, chaque cœur, et chaque processeur logique d'un même cœur (SMT) possède ainsi une liste de threads prêts. On utilise alors, comme pour les stratégies opportunistes, un ordonnancement de base de type *Self Scheduling* : chaque processeur, lorsqu'il doit choisir le prochain thread à exécuter, examine les listes qui le « couvrent » à la recherche du thread le plus prioritaire. L'identification entre élément physique et liste de threads permet ainsi de déterminer le domaine d'exécution d'un thread donné : placé sur une liste associée à une puce physique, ce thread pourra être exécuté par tout processeur de cette puce ; placé sur la liste globale il pourra être exécuté par tout processeur de la machine. En outre, une notion de priorité permet de s'assurer de la réactivité d'éventuels threads de communication par exemple.

Il serait même aisé de modéliser un ensemble de machines NUMA reliées par un réseau

à capacité d'adressage (tel que SCI[1] [Sol96]), ou des machines exécutant un système SSI (*Single System Image*) tel que Kerrighed [MLV+03]. Il suffit simplement d'ajouter un niveau « réseau ».

Bien entendu, suivre l'organisation physique de la machine n'est pas obligatoire. Par exemple, lorsque l'*arité* entre les éléments d'un niveau et du niveau inférieur est grande et risque d'entraîner des problèmes de contention, des groupes artificiels de processeurs peuvent eux aussi être modélisés, de la même façon que Wang *et al.* le font pour leur algorithme CAFS (*Clustered AFinity Scheduling* [WWC97]).

La construction de cette modélisation s'effectue bien sûr de manière automatique lors du démarrage de l'application, en interrogeant le système d'exploitation sur le nombre de nœuds NUMA et la topologie du réseau d'interconnexion, le détail du partage des caches, etc. Tous les systèmes contemporains fournissent une interface donnant ce genre de renseignements. Il est bien sûr nécessaire pour chaque système de réécrire les fonctions d'interrogation, mais les différentes interfaces se ressemblant beaucoup, cela est plutôt aisé. Il est à noter que WINDOWS VISTA, dernier système à ce jour à avoir ajouté une telle interface, fournit une très grande quantité de détails tels que l'associativité des caches !

Certaines machines n'ont pas une hiérarchie régulière : sur la figure 1.7 (page 14), nous avons vu un exemple de machine dont le réseau d'interconnexion était asymétrique. D'autres réseaux d'interconnexion sont organisés en *tore 2D*, dans les machines X1 de CRAY ou les ALPHASERVER d'HP par exemple. La modélisation sous la forme d'une hiérarchie de listes n'est alors pas le reflet exact de la réalité, et il faudra à l'avenir utiliser des modèles plus complexes. En pratique, notre modèle hiérarchique reste cependant une assez bonne approximation de telles machines.

Dans le cas où la machine cible permet de débrancher et rebrancher des processeurs à chaud, ou si elle est partitionnée pour plusieurs utilisateurs, la structure de la machine reste tout de même la même, et l'on désactive et réactive simplement dynamiquement les niveaux devenus inutiles.

2.3 Combiner les modélisations

Une fois que l'on dispose de la structure de l'application sous forme d'une hiérarchie de bulles et de threads d'une part, et de la structure de la machine sous la forme d'une hiérarchie de listes d'autre part, le principe est de distribuer la hiérarchie de bulles et de threads sur les listes. En reprenant l'exemple de l'application qui avait été modélisée avec des bulles à la figure 2.1, exécutée par exemple sur une machine double-bicœur, deux distributions extrêmes sont représentées figure 2.4. Sur la figure 2.4(a), toute la hiérarchie de bulles (et donc l'ensemble des threads) a été placée tout en haut de la hiérarchie de listes. Une telle distribution permet certes d'utiliser tous les processeurs de la machine, puisque tous ont l'opportunité d'exécuter n'importe quel thread. Cela ne prend cependant pas en compte les affinités entre threads et processeurs, puisqu'aucune relation entre eux n'est réellement utilisée. À l'inverse sur la figure 2.4(b), toute la hiérarchie de bulles (et donc tous les threads)

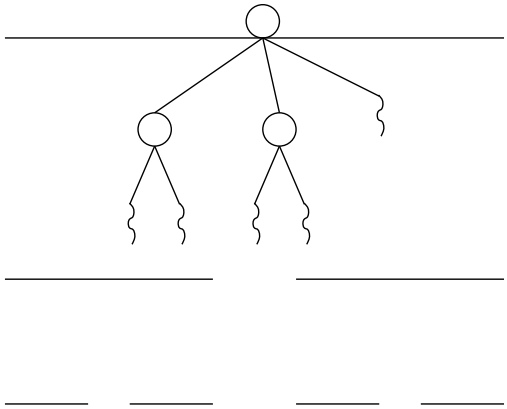
[1] Un réseau SCI permet de définir des segments de mémoire partagés par plusieurs machines de façon transparente, la machine SEQUENT NUMA-Q [LCS96] utilise cette technique.

a été placée sur la liste d'un seul des processeurs. La prise en compte des affinités est ici maximale : tous les threads seront exécutés par le même processeur ! Cependant, puisque l'application n'a pas d'autres threads à exécuter, tous les autres processeurs restent inactifs. La figure 2.4(c) montre comment les threads peuvent être distribués de manière spatiale en prenant fortement les affinités en compte, puisque les threads sont en fait ici correctement répartis sur les processeurs selon leurs affinités. Cependant, si certains threads s'endorment, les processeurs correspondants deviennent inactifs, et l'utilisation des processeurs est donc potentiellement partielle. La figure 2.4(d) montre enfin comment une distribution de threads sur les processeurs peut être incorrecte du point de vue des affinités : on voit des relations entre bulles et threads se croiser.

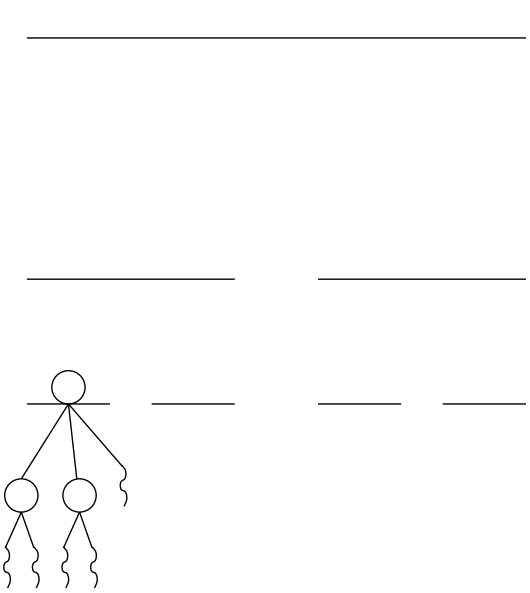
Toute la problématique se ramène alors à évoluer entre les différentes distributions possibles, en évitant les écueils tels que les placements extrêmes 2.4(a) et 2.4(b), au profit de placements intermédiaires tels que 2.4(c) qui établissent un compromis entre distribution de charge et prise en compte des affinités. Il faudra bien sûr autant que possible éviter des situations telles que 2.4(d). Les informations attachées aux bulles détaillées à la section 2.1.3 permettent de raffiner les choix de placement. Dans l'exemple de la figure 2.2, un ordonnancement simpliste pourrait aboutir au placement représenté figure 2.4(e). L'ordonnanceur *Spread*, que l'on présente plus loin à la section 3.2.3 (page 48), utilise l'information synthétique de charge pour obtenir une répartition plus équilibrée représentée figure 2.4(f) : il a commencé par distribuer la bulle de poids le plus fort (4) sur la première moitié de machine, et les deux autres (2+2=4) sur l'autre moitié.

Une fois une certaine distribution choisie, les processeurs peuvent, grâce à leur algorithme d'ordonnancement de base, déterminer rapidement quels threads ils devraient exécuter. On peut alors laisser l'application s'exécuter un certain temps : si la distribution est appropriée, l'exécution devrait être efficace et il n'y a pas de raison de redistribuer. Cependant, lorsque des évènements surviennent, la création, l'endormissement ou la terminaison de threads par exemple, il sera sans doute nécessaire de redistribuer. Ainsi, les décisions prises pour la distribution de threads et de bulles sont plutôt à *moyen terme*, en suivant les grandes étapes du déroulement de l'application.

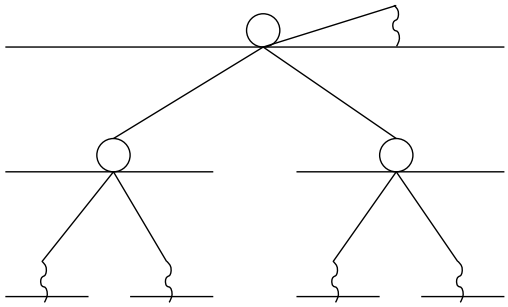
Pour réaliser de telles distributions, nous proposons une véritable plate-forme permettant de développer des *ordonnanceurs à bulles* adaptés aux différents types d'application. Cette plate-forme est décrite au chapitre suivant.



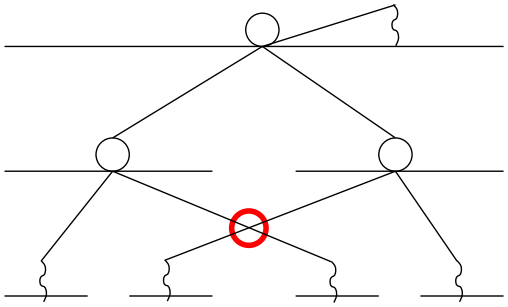
(a) Bonne utilisation processeur, affinités non prises en compte.



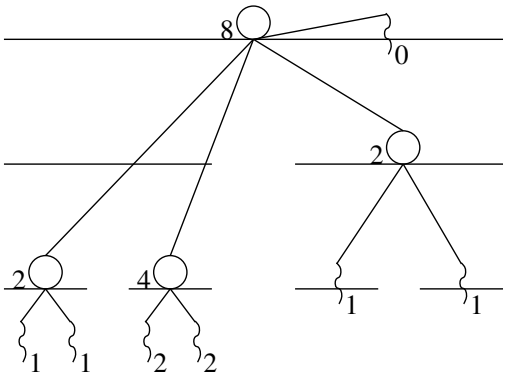
(b) Mauvaise utilisation processeur, affinités extrêmement prises en compte.



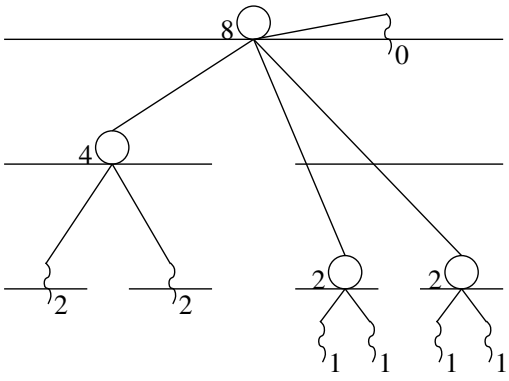
(c) Utilisation distribuée des processeurs, affinités correctement prises en compte.



(d) Utilisation distribuée des processeurs, affinités mal prises en compte.



(e) Répartition simpliste, non équilibrée.



(f) Répartition selon la charge estimée.

FIGURE 2.4 – Distributions possibles des threads et bulles sur la machine.

considérés. La figure 3.11 montre le déroulement de l'ordonnanceur *Affinity* sur un exemple de bulle déséquilibrée.

3.2.6 *MemAware*, un ordonnancement dirigé par la mémoire

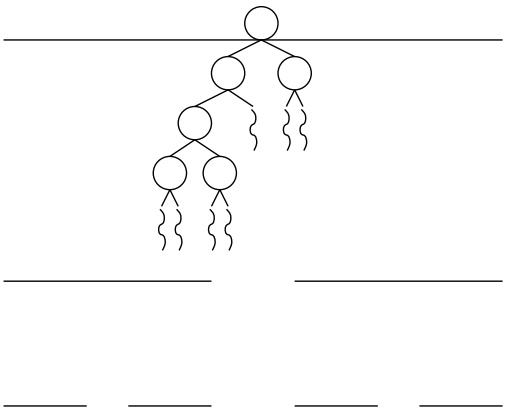
Nous avons vu à la section 1.2.1 (page 8) que sur les machines NUMA il était essentiel de prendre garde au placement des données en mémoire, or les ordonnanceurs décrits ci-dessus ne le font pas réellement. Pendant son stage de Master 2 [Jeu07] au sein de l'équipe dans le cadre du projet ANR NUMASIS, Sylvain JEULAND a donc affiné les ordonnanceurs *Spread* et *Steal* pour qu'ils prennent ce critère en compte lorsqu'ils choisissent le placement des threads sur les différents nœuds NUMA.

Il a pour cela travaillé avec l'équipe-projet Mescal de Grenoble sur l'interface de leur bibliothèque de gestion fine de tas, pour qu'elle lui permette notamment de savoir où les données sont effectivement allouées et d'enregistrer des informations fournies par l'application. Celles-ci peuvent être extraites de compteurs matériels ou bien déterminées par profilage logiciel, telles que la fréquence estimée d'accès, les taux de défauts de cache, etc. (voir section 1.4.4 page 22). Cette bibliothèque permet également de fusionner plusieurs tas, migrer tout ou partie des données entre nœuds NUMA, attacher artificiellement à la gestion des tas des zones déjà allouées par ailleurs (définitions statiques par exemple), etc.

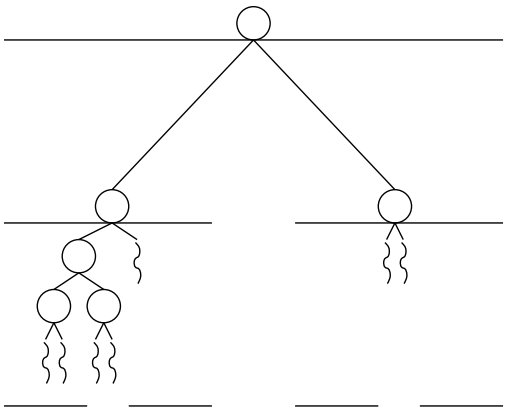
Le principe est alors de créer un tas pour chaque thread et chaque bulle. Lorsque l'application effectue une allocation de données, elle indique (ou bien le compilateur détermine automatiquement) quels threads accéderont le plus à ces données : le thread qui effectue l'allocation, un autre thread, ou bien l'ensemble des threads contenus dans une bulle donnée ; elle peut également indiquer une fréquence d'accès aux données. L'allocation est alors effectuée dans le tas correspondant au thread ou à la bulle indiquée. Ceci permet ainsi d'avoir à tout moment une connaissance hiérarchisée des affinités entre threads, données et nœuds NUMA.

Lors d'une distribution d'une hiérarchie de bulles, la version *MemAware* de l'ordonnanceur *Spread* prend garde à l'encombrement mémoire d'une bulle. Pour choisir quelle bulle éclater, elle considère la quantité de données qui a été allouée pour cette bulle et la fréquence d'accès (donc la quantité de données partagées correspondant à cette bulle). Elle préfère alors éclater les bulles pour lesquelles il y a le moins de données partagées (les moins « épaisses »), évitant ainsi de séparer les threads qui travailleront souvent sur les mêmes données.

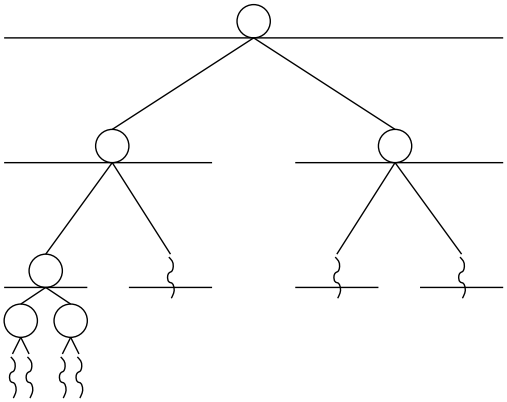
D'autre part, lors d'une redistribution d'une hiérarchie de bulles (pour s'adapter à une nouvelle situation de charge par exemple), le choix du placement des threads et bulles est guidé par les données effectivement allouées sur les nœuds NUMA. Pour chacun d'entre eux, on détermine en effet le *bassin d'attraction* : le nœud NUMA où se trouvent la plupart de ses données. Lors du déroulement de la distribution gloutonne, on privilégie alors les placements sur les bassins d'attraction. Lorsque c'est contre-indiqué pour des raisons de répartition de charge, on fait à l'inverse migrer les données sur le nouveau nœud NUMA choisi, adaptant ainsi le placement des données en fonction de la nouvelle répartition de charge. Le choix entre placement dans le bassin d'attraction existant ou déplacement (coûteux) des données pour pouvoir changer le bassin d'attraction fait bien sûr l'objet d'heuristiques qui prennent en compte les caractéristiques de la machine et dépendent de l'application. D'autre part, une



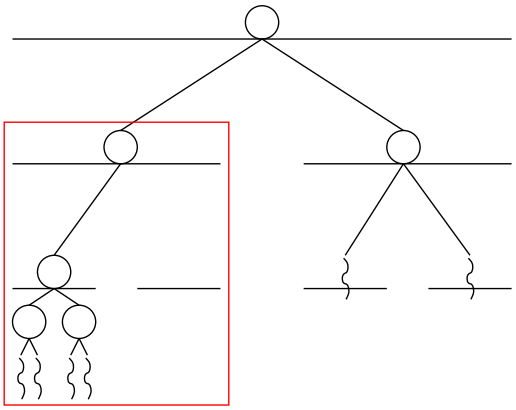
(a) Situation de départ.



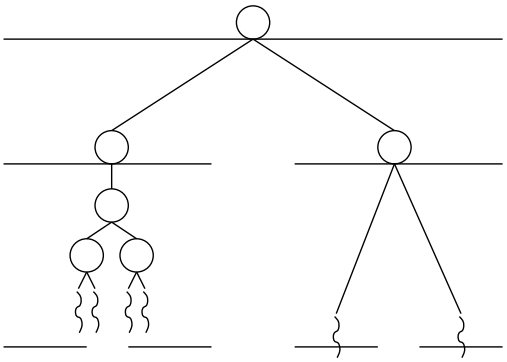
(b) Il faut éclater la bulle principale pour pouvoir alimenter les deux sous-listes.



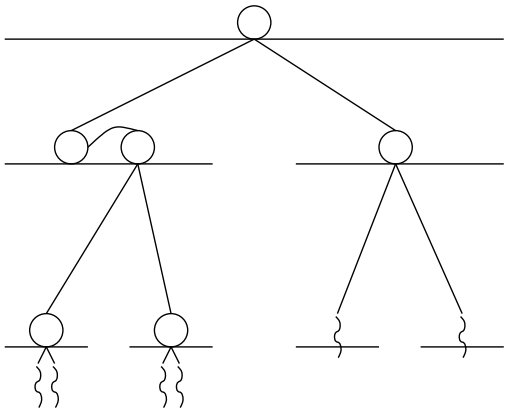
(c) Il faut éclater les deux sous-bulles pour pouvoir alimenter tous les processeurs. L'algorithme est terminé, malgré un déséquilibre en nombre de threads.



(d) Un thread s'est terminé, un processeur devient inactif et il n'y a pas de thread directement accessible chez le processeur voisin, l'algorithme de vol considère donc la liste juste supérieure et ses sous-listes.

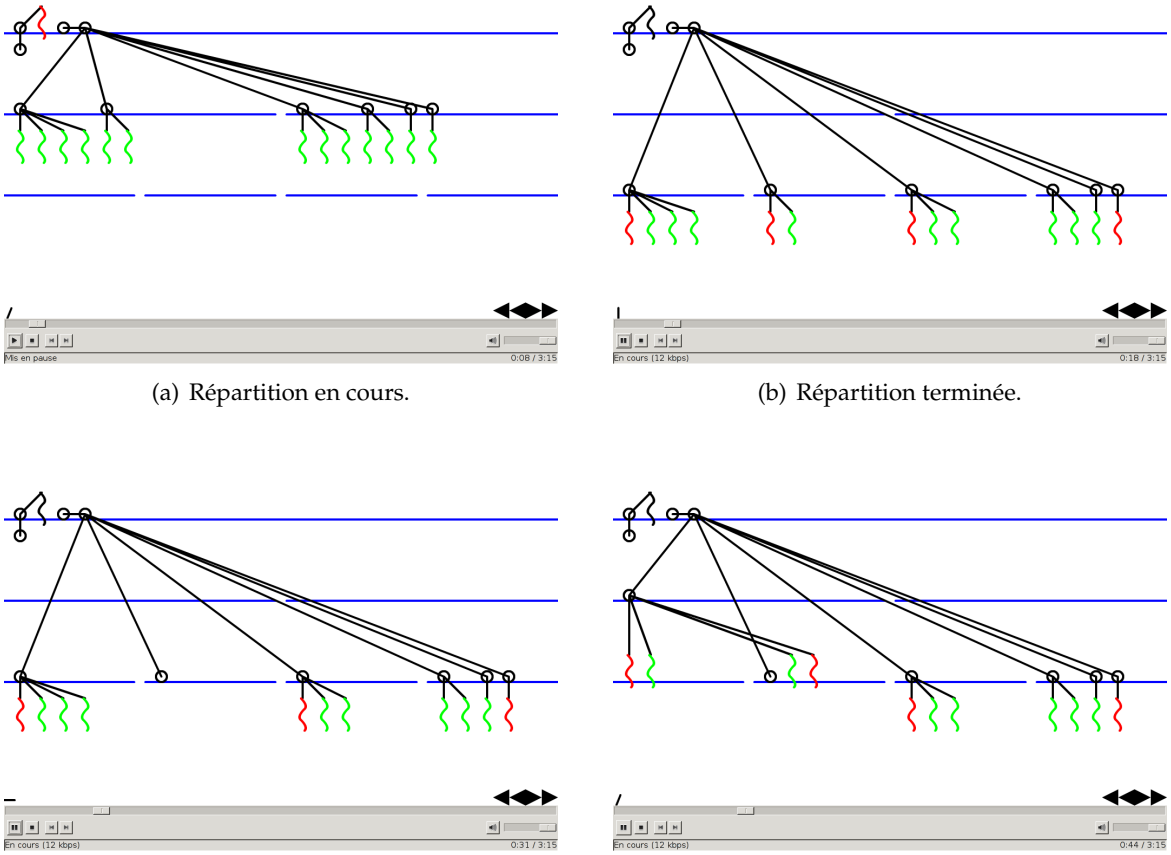


(e) Les entités de cette partie de machine sont rassemblées.



(f) L'algorithme de distribution est relancé sur cette partie de machine, il est alors obligé de percer deux bulles. On obtient une nouvelle distribution adaptée à la nouvelle charge.

FIGURE 3.11 – Déroulement de l'ordonnancement *Affinity*.



(a) Répartition en cours.

(b) Répartition terminée.

(c) Deux threads terminent, un processeur devient inactif.

(d) L'algorithme *Spread* a été relancé localement pour combler le déséquilibre.

FIGURE 3.12 – *Action Replay* de l'ordonnancement *Spread*.

peut être enregistrée de manière légère [DW05] : création, endormissement, destruction des threads, structuration en arbre des bulles, répartition le long de la machine, etc. Après l'exécution, il est alors possible d'analyser cette trace pour observer de manière fine ce qui s'est passé. Pour cela, elle est convertie en animation *flash* en s'appuyant sur la bibliothèque MING [min]. La figure 3.12 montre par exemple l'évolution de l'ordonnanceur *Spread* en réaction à l'inactivité d'un processeur.

Il est ainsi possible de vérifier rapidement et précisément ce qui s'est passé au cours de l'exécution, ce qui est précieux pour *déboguer* les ordonnanceurs. Bien sûr, la barre de défilement des lecteurs *flash* permet de se déplacer rapidement au sein de l'animation pour atteindre les périodes intéressantes. Il est par ailleurs possible d'ajouter toute information utile, telle que les attributs de threads (noms, priorité, charge, mémoire allouée, etc.) ou des bulles (élasticité, mémoire allouée, etc.) pour vérifier facilement le comportement de l'algorithme face à ces informations.

Cet outil est également très utile à des fins de démonstration : pour expliquer de manière visuelle un algorithme à un collègue ou au sein de présentations lors de colloques par exemple.

3.3.3 Multiplier les tests

Une fois un ordonnanceur développé, il est utile de vérifier qu'il se comporte bien quelle que soit la hiérarchie de bulles considérée. Pour pouvoir expérimenter rapidement de nombreux cas typiques de hiérarchies de bulles, notre plate-forme fournit un outil de génération de bulles appelé *BubbleGum*, dont le développement a été confié à un groupe d'étudiants. La figure 3.13 montre l'interface graphique de cet outil. La partie de gauche permet de construire récursivement des hiérarchies de bulles de manière intuitive par sélection et raccourcis claviers ou clics sur les boutons. Quelques attributs tels que la charge, la priorité ou le nom d'un thread peuvent être spécifiés en même temps que la construction. De plus, la charge est rendue visuellement en changeant les couleurs des threads. Il est possible de corriger les relations en effectuant une sélection puis un simple glisser-lâcher entre bulles. Un clic sur un bouton permet enfin de lancer la génération d'un programme C synthétique, son exécution, et la récupération de la trace, qui est ici encore convertie en une animation, intégrée cette fois à l'interface sur la partie droite. Cela permet notamment de sélectionner threads et bulles à partir de leur placement effectif sur la machine.

Bien sûr, il est possible de *sauvegarder* une hiérarchie de bulles pour pouvoir la reprendre plus tard, ou bien l'intégrer dans une autre hiérarchie. On peut ainsi se constituer un panel de hiérarchies de bulles synthétiques de toutes sortes : régulières, déséquilibrées, irrégulières, voire pathologiques. Cela permet également de s'échanger entre développeurs certains cas pathologiques par exemple.

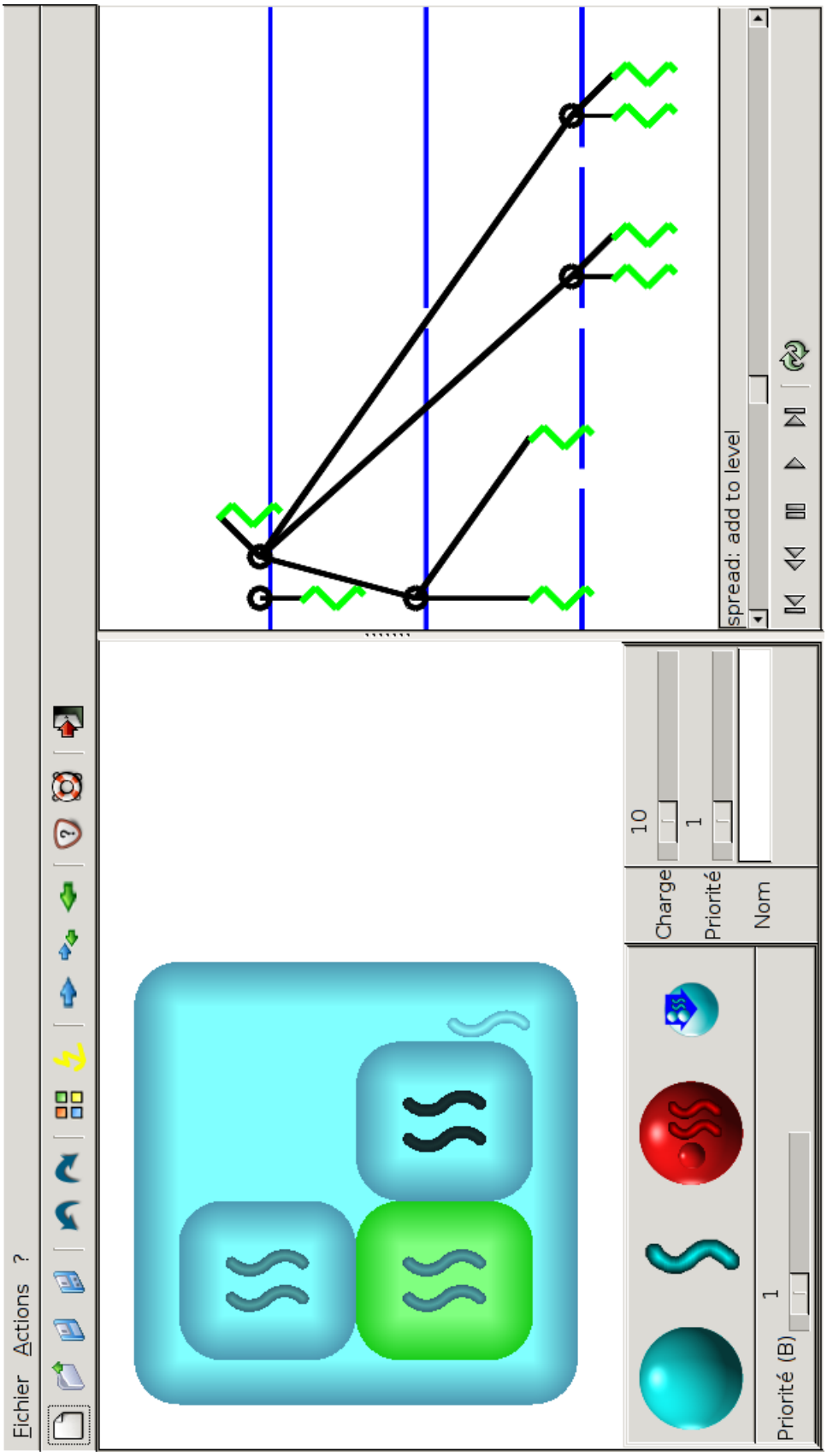


FIGURE 3.13 – Interface graphique de génération de bulles : *BubbleGum*.

Chapitre 4

Éléments d'implémentation

Sommaire

Dans ce chapitre, nous présentons l'implémentation en elle-même de la plate-forme BubbleSched. Nous expliquons d'abord pourquoi et comment elle a été réalisée au sein de MARCEL, la bibliothèque de threads de l'environnement d'exécution PM^2. Nous détaillons ensuite quelques points d'implémentation intéressants d'un point de vue méthodologie : de l'ordonnancement de haut niveau d'abord, puis des optimisations nécessaires à une exécution efficace.

4.1 Sur les épaules de MARCEL

La bibliothèque de threads utilisateur MARCEL a été originellement développée par Jean-François MÉHAUT et Raymond NAMYST pour l'environnement distribué de programmation multithreadée PM^2 [Nam97]. Cet environnement combine en effet MARCEL avec une bibliothèque de communication, MADELEINE, l'objectif étant de coupler de manière efficace calcul parallèle et communications, en utilisant notamment des threads pour faire progresser les communications de manière asynchrone et réactive. La librairie MARCEL fonctionne

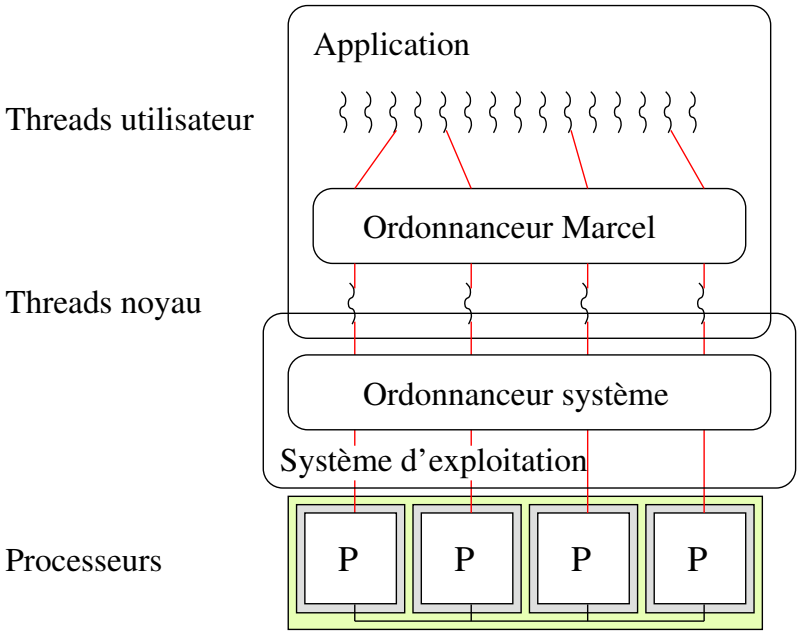


FIGURE 4.1 – Mode de fonctionnement de base de Marcel : l'ordonnanceur du système est en fait court-circuité.

entièrement en espace utilisateur à l'aide d'appels à `setjmp` et `longjmp` pour changer de contexte entre les threads MARCEL, ce qui lui permet d'être très légère en comparaison de toute bibliothèque de threads basée sur des threads de niveau noyau, ainsi que d'être utilisée sans avoir à changer le noyau du système. Son portage sur de nombreux systèmes (notamment GNU/Linux, BSD, AIX, Irix et OSF) et architectures (notamment x86, x86_64, Itanium, PowerPC, Alpha, Mips et Sparc) n'a par ailleurs nécessité que l'adaptation de quelques fonctions de bas niveau. À l'origine, MARCEL ne supportait que les machines monoprocesseurs. Vincent DANJEAN a par la suite étendu MARCEL aux machines multiprocesseurs en introduisant un ordonnanceur mixte [Dan98] qui exécute tour à tour les threads utilisateur sur autant de threads noyau qu'il y a de processeurs. De plus, lorsque le système d'exploitation le permet (ce qui est le plus souvent le cas), MARCEL lui demande de fixer les threads de niveau noyau sur les processeurs, ce qui lui permet alors de réellement contrôler l'exécution précise des threads sur les processeurs sans aucune interaction ultérieure avec le système (en supposant qu'aucune autre application ne s'exécute sur la machine). Ce mode de fonctionnement est représenté sur la figure 4.1. Vincent DANJEAN a par ailleurs implémenté le mécanisme de *Scheduler Activations* [DNR00a] pour que les threads Marcel puissent effectuer des appels système bloquants sans pour autant limiter l'exploitation de tous les processeurs de la machine. Enfin, il a intégré à MARCEL une des premières versions stables de l'ordonnanceur en $O(1)$ de LINUX 2.6 avec les outils associés (listes, *spinlocks*, *runqueues*, *tasklets*, etc.) [Dan04].

Il était ainsi naturel de poursuivre le développement de MARCEL pour lui permettre d'exploiter au mieux les machines NUMA. Il a d'abord fallu consolider le support SMP existant pour qu'il puisse exploiter de nombreux processeurs, en distribuant par exemple certains mécanismes qui étaient encore centralisés ; quelques points sont donnés en exemple à la

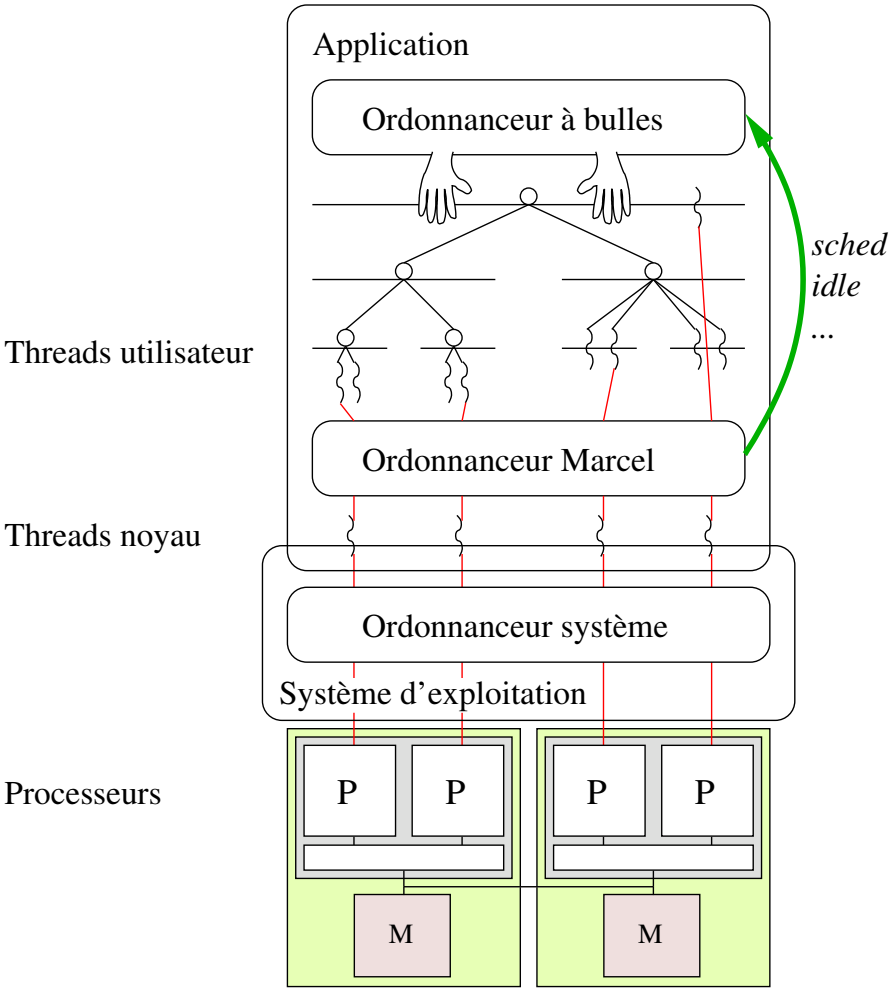


FIGURE 4.2 – Mode de fonctionnement de Marcel avec bulles : l'ordonnanceur de base de Marcel est conditionné par le placement des threads et bulles.

section 4.4. Il a ensuite été possible d'implémenter l'ordonnancement à bulles en modifiant l'ordonnanceur de base. Celui-ci n'était au départ prévu que pour consulter une liste locale au processeur, avec une répartition de charge ne prenant en compte ni affinités entre threads ni topologie de la machine sous-jacente. Le nouveau mode de fonctionnement est représenté figure 4.2. L'ordonnanceur de base consulte désormais une hiérarchie de listes de threads et de bulles, automatiquement construite à partir des informations de topologie fournies par le système d'exploitation. Par ailleurs, il appelle aux moments appropriés les méthodes de l'ordonnanceur à bulles courant, qui peut alors prendre l'initiative de déplacer threads et bulles sur la hiérarchie de listes. Ainsi MARCEL délègue la responsabilité de la répartition des threads à l'ordonnanceur à bulles, et se contente de suivre les contraintes imposées par les placements effectués sur la hiérarchie de listes.

Il serait bien sûr possible de réaliser une implémentation à l'aide d'approches micro-noyau telles qu'ExoKernel [EKO95] ou Nemesis [RF97] ou d'une approche *Scheduler Activations* telle que K42 [AAD+02], car elles permettent de réaliser des ordonnanceurs en espace uti-

lisateur. Cependant, de telles approches nécessitent d'amorcer la machine utilisée avec ces noyaux, or c'est en général impossible, par manque de support du matériel par ces noyaux, ou plus simplement pour des raisons administratives ou de sécurité. De fait, il est rare que l'administrateur de machines de calcul accepte de démarrer un autre système que celui qu'il administre[1]. L'approche que suit MARCEL permet par un petit effort de portage d'essayer différentes machines, indépendamment du système d'exploitation.

À plus long terme, il serait par contre intéressant d'essayer d'intégrer un ordonnanceur à bulles générique (ou plusieurs interchangeables) au noyau LINUX par exemple, en utilisant au moins les informations disponibles implicitement telles que l'imbrication de threads en processus, eux-mêmes regroupés en groupes et en sessions, etc. Il pourrait alors être utile d'étendre l'interface de threads POSIX pour permettre aux applications de spécifier au moins les affinités entre threads d'une manière analogue à nos bulles, pour permettre un ordonnancement plus fin. Une telle approche ne pourra pas atteindre les performances d'une approche complètement en espace utilisateur, car il n'est pas aussi aisé d'opérer des échanges d'informations entre l'ordonnanceur et l'application lorsque ceux-ci sont séparés par la barrière entre espace utilisateur et espace noyau. Cette approche devrait cependant permettre d'obtenir des gains appréciables. Nous reviendrons sur ce point dans la section 7.2 qui développe les principales perspectives que nous dégageons de nos travaux.

4.2 Ordonnancement

L'implémentation des concepts théoriques décrits dans les chapitres précédents a parfois nécessité des choix intéressants que cette section développe.

4.2.1 Ordonnanceur de base

À la section 2.2 (page 33), nous avons vu que l'ordonnanceur de base doit parcourir une hiérarchie de listes de threads et bulles pour trouver le prochain thread à exécuter. Il doit par ailleurs respecter les priorités indiquées par l'application. La figure 4.3 montre un exemple où un thread de priorité 2 a été laissé sur la liste principale, un thread de communication par exemple. Dans un tel cas, dès que ce thread est prêt à être exécuté, un des processeurs doit l'exécuter à la place des threads de moindre priorité, même si ceux-ci sont sur des listes plus proches des processeurs.

La structure de liste utilisée (les *runqueues* de l'ordonnanceur $O(1)$ de LINUX 2.6) permet de réaliser cela de manière assez efficace. En effet, ces listes sont *hachées* par priorité, ce qui permet d'extraire en $O(1)$ une entité (thread ou bulle) d'une priorité donnée. De plus, chacune mémorise en permanence dans un champ de bits les priorités qu'ont les entités qu'elle contient, ce qui permet de trouver rapidement[2] la priorité maximale parmi celles des

[1] Une exception notable est la plate-forme Grid5000, qui cependant ne comporte que des machines avec au plus 4 processeurs.

[2] L'ordonnanceur $O(1)$ de LINUX 2.6 n'est en réalité pas tout à fait en $O(1)$, mais en O(nombre de priorités), où le nombre de priorités est actuellement fixé à 140 pour pouvoir supporter une priorité absolue pour les threads noyau, 99 priorités temps réel, et des indices de politesse (`nice`) entre -20 et 19. En pratique, la recherche dans le champ de bits est écrite en assembleur pour qu'elle soit la plus optimisée possible. Dans le cas de MARCEL, le

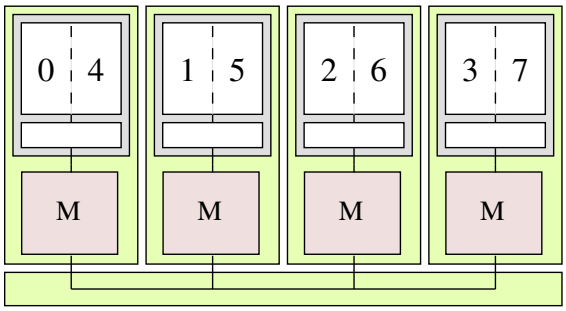


FIGURE 4.4 – Exemple de machine contenant deux nœuds NUMA comportant chacun un cœur hyperthreadé, avec une numérotation non consécutive (voir section 1.2.4 page15).

un paramètre non POSIX `_SC_NPROCESSORS_ONLN` qui lui fait retourner le nombre de processeurs disponibles. D'autres systèmes fournissent plutôt un paramètre `_SC_NPROC_CONF`, d'autres encore tels que DARWIN fournissent une fonction complètement non standard. On encapsule donc cette fonctionnalité dans une simple fonction `int ma_nbprocessors()` que l'on doit éventuellement réécrire pour un nouveau système d'exploitation. La fixation d'un thread noyau sur un processeur particulier nécessite par ailleurs l'emploi d'une fonction propre à chaque système, mais là aussi il est facile d'encapsuler cette fonctionnalité dans une fonction `void ma_bind_on_processor(int target)`.

La découverte des relations entre processeurs n'est absolument pas standardisée, chaque système possède sa propre interface complètement différente des autres, fournissant plus ou moins de détails de manières très variées. Il a donc été nécessaire, pour faciliter les portages, de trouver un dénominateur commun qui soit facile à implémenter sur chaque système, mais qui suffise cependant aux besoins de notre plate-forme. Ce dénominateur commun s'est révélé relativement simple, mais au prix d'une analyse générique complexe, si bien que le module de gestion de topologie est le troisième plus gros module de MARCEL, après l'ordonnanceur de base et la gestion des signaux Unix !

Le principe est que pour un système d'exploitation donné, une fonction `look_topology` décrit les niveaux de hiérarchie de la machine, des plus globaux aux plus locaux[3]. Cette description comporte simplement le type de hiérarchie (nœud NUMA, puce, partage de cache, cœur ou processeur logique) et l'ensemble des processeurs concernés par ce niveau (par un masque de bits). Sur le cas de la figure 4.4, `look_topology` pourra par exemple d'abord indiquer qu'il existe quatre nœuds NUMA ayant respectivement pour masque de processeurs `0x11`, `0x22`, `0x44` et `0x88`, puis qu'il existe quatre cœurs ayant respectivement pour masque de processeurs `0x11`, `0x22`, `0x44` et `0x88`, et enfin qu'il existe huit processeurs logiques ayant respectivement pour masque de processeurs `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40` et `0x80`. L'implémentation effective varie selon les systèmes : certains tels qu'OSF ne peuvent fournir d'information que pour les nœuds NUMA, c'est alors trivial. D'autres tels qu'AIX permettent d'utiliser une boucle `for` pour itérer la découverte sur les différents types de niveaux de topologie. Enfin, certains systèmes tels que LINUX nécessitent l'analyse d'un fichier texte.

[3] Il serait même possible de lever cette contrainte en effectuant automatiquement un tri topologique, mais en pratique elle s'est toujours révélée facile à respecter.

Machine :	0xff							
Nœuds NUMA :	0x11	0x22	0x44	0x88				
Cœurs :	0x11	0x22	0x44	0x88				
Processeurs logiques :	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80

(a) Informations brutes fournies par le système d'exploitation.

Machine :	0xff							
Nœuds NUMA :	0x11		0x22		0x44		0x88	
Cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(b) Sous-niveaux rassemblés selon le niveau supérieur.

Machine :	0xff							
Nœuds NUMA + cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(c) Niveaux redondants fusionnés.

Machine :	0xff							
Artificiel :	0x33				0xcc			
Nœuds NUMA + cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(d) Niveau intermédiaire artificiel ajouté.

FIGURE 4.5 – Déroulement de l'analyse de topologie ; à chaque étape est indiqué pour chaque niveau le masque de processeurs de chaque élément.

Une fois ce résultat brut obtenu, il s'agit de trier et filtrer ces informations. La figure 4.5 illustre le déroulement de cette analyse. Une première étape éventuelle, non nécessaire ici, s'occupe de trier les niveaux par masque de processeur pour obtenir une numérotation finale cohérente. Une deuxième étape rassemble pour chaque niveau les sous-niveaux qui le composent. Ce n'est ici pas nécessaire pour les cœurs, mais ça l'est pour les processeurs logiques. Une troisième étape fusionne les niveaux qui sont redondants : ici chaque nœud NUMA contient exactement un cœur. Enfin, une dernière étape éventuelle ajoute un niveau intermédiaire pour éviter une arité entre niveaux trop grande. Il est par ailleurs possible de choisir de n'utiliser qu'une partie de la machine en indiquant le numéro du premier processeur à utiliser, le nombre de processeurs à utiliser, et le « pas » (ici on peut indiquer par exemple 2 pour ignorer un processeur logique sur deux[4]). Une étape intermédiaire est alors insérée pour tronquer les niveaux non utiles et laisser la troisième étape simplifier éventuellement la topologie.

Enfin, pour choisir sur quel nœud NUMA une zone mémoire doit être allouée, ou

[4]On pourrait imaginer des manières plus portables d'effectuer cette sélection, en indiquant par exemple qu'on ne veut utiliser qu'un processeur logique par cœur, ou bien quatre processeurs partageant le moins de bande passante possible, etc.

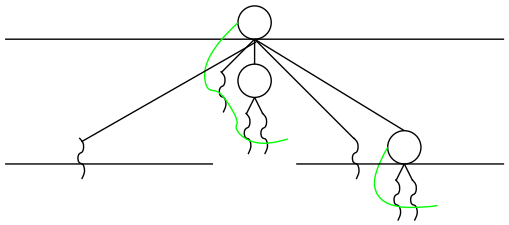


FIGURE 4.7 – Illustration du cache de threads.

4.4 Optimisations

Comme souligné par Edler [Edl95], la qualité en elle-même d'une implémentation est essentielle pour obtenir de bonnes performances. Cette section détaille donc certaines optimisations qu'il a été utile de faire, et dont les applications peuvent ainsi profiter de manière transparente.

4.4.1 Parcours des bulles à la recherche du prochain thread à exécuter

Lorsque l'ordonnanceur de base rencontre une bulle sur une liste, la méthode *sched* de l'ordonnanceur est appelée. L'implémentation par défaut parcourt cette bulle et ses sous-bulles à la recherche d'un thread à exécuter. La complexité de cette recherche n'est *a priori* pas bornée, puisque la hiérarchie de bulles peut être d'une taille arbitraire. Pour conserver un ordonnancement de base efficace, chaque bulle contient donc un *cache* de thread, illustré à la figure 4.7. C'est la liste des threads contenus dans cette bulle et ses sous-bulles qui ne sont pas placés sur d'autres listes. La recherche de thread peut alors s'effectuer en temps constant. La contrepartie est un surcoût en $O(1)$ de la mise à jour du cache lors des opérations sur les bulles et threads, mais ces opérations sont bien moins courantes que l'ordonnancement de base.

4.4.2 Plus léger que les processus légers

Lors de la parallélisation d'une application, une question qui revient souvent est « combien de threads créer ? ». En effet, si l'on crée peu de threads qui ont chacun une grande quantité de travail à faire, cela empêche une bonne répartition de charge puisque la granularité avec laquelle l'ordonnanceur pourra jouer est grande. À l'inverse, si l'on crée de nombreux threads qui ont chacun très peu de travail (dans certains cas, quelques microsecondes seulement), cela permet certes une bonne répartition de charge *a priori*, mais le coût de création d'un thread peut devenir prohibitif (typiquement quelques microsecondes, même pour les implémentations en espace utilisateur). Une approche typiquement utilisée est alors de gérer des « tâches » qui sont attribuées par l'application (ou par l'environnement d'exécution, dans le cas d'OPENMP par exemple) à un certain nombre de threads, nombre qui est choisi en fonction du rapport entre le coût de création d'un thread et le temps d'exécution d'une tâche. Cela ne permet cependant pas d'exprimer *tout* le parallélisme de l'application, ce qui est dommage. Engler *et al.* décrivent également ce problème [EAL93] et proposent d'alléger la notion de thread en une notion de *filament*, qui est une version très limitée de thread,

	Création	Exécution
Thread	$0,85\,\mu$s	$0,60\,\mu$s
Graine	$0,22\,\mu$s	$0,22\,\mu$s
NPTL	$6,4\,\mu$s	$11,3\,\mu$s

TABLE 4.2 – Coût de création et d'exécution d'un thread et d'une graine de thread. À titre de comparaison, valeurs pour la bibliothèque de threads native de Linux, la NPTL

spécialisée dans trois types de tâches particulières : s'exécuter sans interruption jusqu'à terminaison, itération avec barrière et *fork/join*. Ils sont exécutés par de véritables threads verrouillés sur chaque processeur. Cela permet d'obtenir une implémentation beaucoup plus simple et efficace, réduisant ainsi le coût de création presque à l'ordre de grandeur d'un appel de fonction. D'autres implémentations de bibliothèques de threads (MPC [Pér06] par exemple) utilisent aussi ce genre de simplification. Par ailleurs, la plupart des implémentations d'OPENMP utilisent ce genre de mécanisme pour effectuer les entrées et sorties de sections parallèles sans pour autant recréer à chaque fois tous les threads.

Pour répondre à ce besoin, la bibliothèque MARCEL utilise une approche qui obtient au final le même résultat, mais de manière légèrement différente. Elle dispose désormais de la notion de « graine de thread » : lors de la création d'un thread, on peut indiquer à l'aide d'un attribut que l'on ne veut pas nécessairement qu'un véritable thread soit créé tout de suite, et MARCEL peut ainsi se contenter de ne mémoriser que la fonction à exécuter et son paramètre, ce qui est très léger en comparaison de l'initialisation complète d'un thread. Ensuite, lorsque l'ordonnanceur rencontre une graine de thread, si le thread précédent vient de se terminer (cas le plus courant dans le contexte considéré), il est réutilisé tel quel pour exécuter la graine de thread. Sinon, un véritable thread est créé. En pratique, chaque processeur est alors capable de réutiliser en permanence le même thread pour exécuter rapidement les graines de thread qui ont été placées sur les listes qui le couvrent. La table 4.2 montre les coûts de création et d'exécution des threads et des graines de threads MARCEL sur une machine double-bicœur Opteron cadencée à $1,8$ GHz. On s'aperçoit que le coût (correspondant essentiellement à la gestion des listes de threads, relations de bulles et ordonnancement de base) devient réellement très faible, il est de l'ordre de 400 cycles à la fois pour la création et pour l'exécution. Il devient ainsi envisageable pour une application d'exprimer absolument tout son parallélisme intrinsèque en créant une graine de thread pour chacune de ses tâches. L'ordonnanceur ne pourra alors que mieux les répartir sur la machine. La contrepartie est que certaines opérations, telles que l'annulation d'une graine de thread en cours d'exécution, ne peuvent pas être effectuées. En pratique, cela n'est pas contraignant. Le résultat final est ainsi similaire à l'approche *filaments*, mais quasiment sans changer l'interface de programmation de threads de MARCEL, l'optimisation de création de thread se faisant de manière opportuniste seulement : ce n'est par exemple que si la fonction exécutée par une graine de thread se bloque réellement que l'on est obligé de créer un autre thread pour exécuter une autre graine de thread.

Il serait possible d'aller un peu plus loin si l'application peut promettre que la tâche qu'elle veut faire exécuter n'utilisera jamais d'opérations bloquantes (`mutex`, entrées/sorties, etc.), les opérations non-bloquantes (création de thread, verrous rotatifs, etc.) étant par contre toutes autorisées. Dans ce cas il n'est même pas nécessaire de créer de thread exécutant

une graine de thread, la tâche peut être exécutée par l'ordonnanceur directement depuis le thread `idle`.

4.4.3 Distribution des allocations

Il est essentiel d'éviter toute centralisation d'opération pour limiter les contentions. L'allocation d'objets est un des cas typiques pour lesquels une implémentation simpliste peut être catastrophique du point de vue des performances. Par exemple, la bibliothèque MARCEL utilise un « cache » de piles pour éviter d'avoir à solliciter le système à chaque fois qu'elle crée un thread. Dans la version originelle, ce cache est centralisé, cela pose donc des problèmes de contention lorsque MARCEL est utilisé avec de nombreux processeurs : la courbe du haut de la figure 4.8 montre que le temps de création concurrente de nombreux threads explose avec le nombre de processeurs. Pour éviter cela, il est possible de créer un cache par processeur, mais un cas pathologique apparaît alors lorsque par exemple un thread s'exécutant sur le processeur 0 passe son temps à créer des threads qui sont répartis sur la machine et se terminent ainsi loin du processeur 0. Dans ce cas, le cache du processeur 0 est toujours vide pendant que les caches des autres processeurs se remplissent... Ce genre de cas est très difficile à traiter. Lorsqu'elles peuvent être utilisées, les graines de threads permettent de le résoudre en retardant l'allocation effective éventuelle de la pile au moment où l'on exécute les graines. Sinon, une solution est d'avoir quelques niveaux supplémentaires de caches partagés entre processeurs, pour distribuer la réutilisation de piles.

La bibliothèque MARCEL intègre en fait une infrastructure, dont le développement a été confié à un groupe d'étudiants, de cache d'allocation distribuée qui, de plus, prend en compte la structure de la machine. Cette infrastructure est générique : on peut créer un cache d'allocation d'un type d'objet quelconque en fournissant simplement une fonction d'allocation (*e.g.* `malloc`) et une fonction de destruction (*e.g.* `free`). Elle est alors non seulement utilisée pour les allocations de piles de threads, mais aussi pour d'autres structures de la bibliothèque MARCEL telles que les graines de threads, et même les données internes à cette infrastructure elle-même.

Pour un cache donné, à chaque élément hiérarchique de la machine (processeurs, cœurs, nœuds NUMA, machine) est associé un *conteneur*. Lorsqu'un objet est libéré, la fonction de libération du cache essaie d'ajouter l'objet libéré au conteneur du processeur courant. S'il est plein (à cause d'un seuil réglable), elle essaie de l'ajouter au conteneur de la puce courante, puis du nœud NUMA courant, et si l'objet est indépendant des nœuds NUMA, au conteneur de la machine. Si tous ces conteneurs sont pleins, l'objet est réellement libéré à l'aide de la fonction fournie au moment de la création du cache. À l'inverse, lorsque la fonction d'allocation du cache est appelée, elle essaie de piocher un objet depuis le conteneur du processeur courant, puis de la puce courante, puis du nœud NUMA courant, puis éventuellement de la machine, et si tous ces conteneurs sont vides un objet est réellement alloué à l'aide de la fonction fournie.

La courbe du bas de la figure 4.8 montre que le coût de création de nombreux threads reste alors assez stable, même sur 16 processeurs. Cette infrastructure, relativement simple malgré sa généricité, pourrait être améliorée en introduisant par exemple des rééquilibrages entre les conteneurs pour compenser des comportements pathologiques. L'intérêt est que ce genre d'améliorations profiterait alors automatiquement à tous les types d'objets alloués ainsi dans

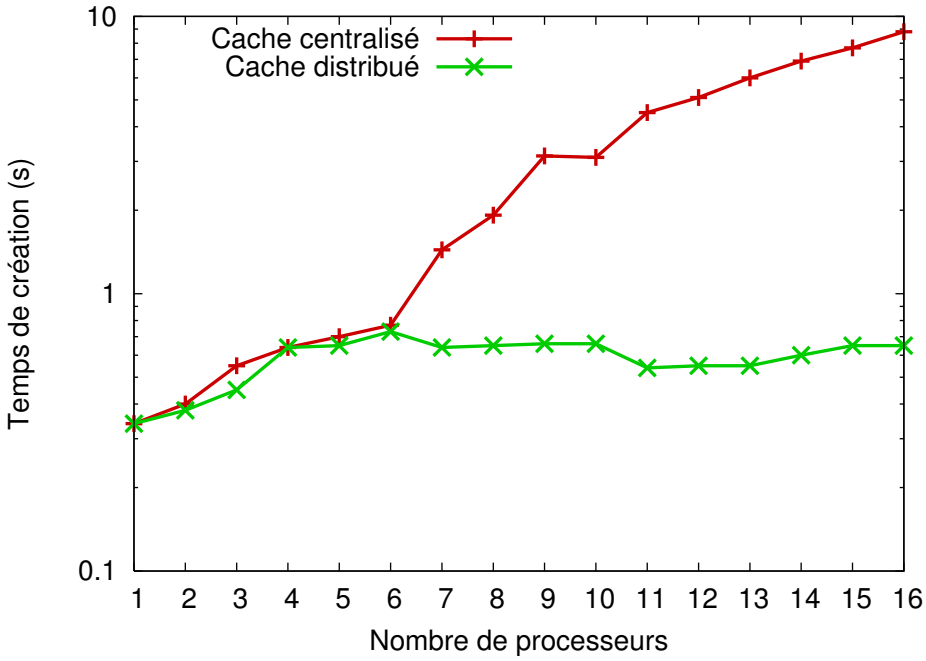


FIGURE 4.8 – Temps de création et de destruction parallèle de 100 000 threads.

MARCEL.

Chapitre 5

Diffusion

Sommaire

Dans ce chapitre, nous présentons les travaux que nous avons réalisés pour augmenter l'impact de nos résultats de recherche et la diffusion de notre plate-forme BubbleSched. En effet, pour qu'elle puisse être utilisable avec de véritables applications, il faut qu'elle puisse être utilisée avec les environnements de programmation parallèle existants. Nous avons ainsi d'abord adapté notre plate-forme aux threads POSIX, ce qui permet d'exécuter une très grande part des applications, puisqu'ils sont les briques de base classiques des environnements de programmation parallèle. Pour obtenir plus d'informations sur l'application, nous avons également adapté un support à l'environnement OPENMP, dont l'expressivité est plus riche, et pour lequel les compilateurs sont plus à même de déterminer des informations sur l'application.

5.1 Compatibilité POSIX et construction automatisée de hiérarchies de bulles

L'interface la plus naturelle que la bibliothèque MARCEL se doit de supporter est bien sûr l'interface des threads POSIX. L'interface de MARCEL, bien que plus ancienne, est déjà assez proche de celle de POSIX, elle ne diverge que par quelques détails de sémantique. Pendant son post-doctorat, Vincent DANJEAN a ajouté à MARCEL deux couches de compatibilité POSIX. La première est une couche de compatibilité assez légère au niveau interface de programmation (API), qui permet de recompiler pour MARCEL une application existante écrite pour utiliser l'interface de threads POSIX. La deuxième est une couche de compatibilité au niveau interface binaire (ABI), qui permet d'exécuter avec MARCEL une application déjà

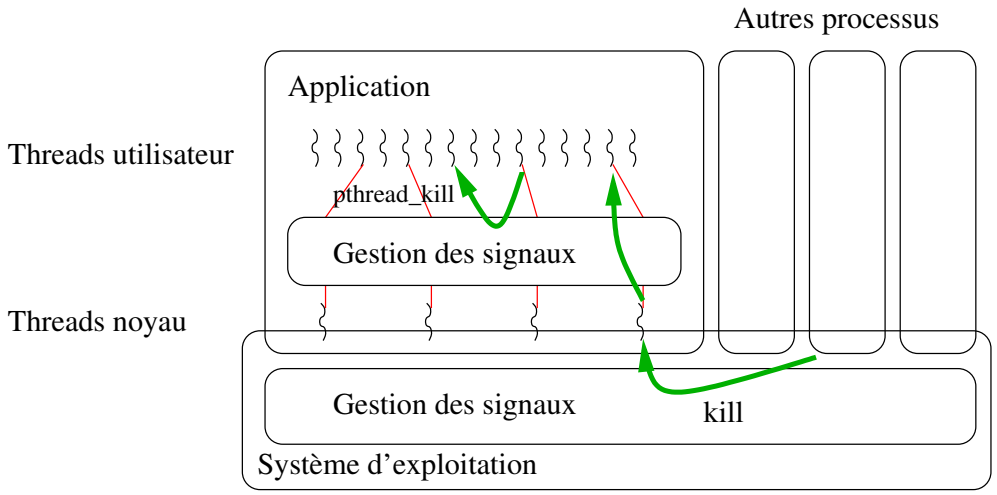


FIGURE 5.1 – Modèle hybride de gestion des signaux.
Lorsqu'un autre processus envoie un signal à l'application (représenté à droite), le noyau choisit un thread noyau auquel envoyer le signal. MARCEL choisit alors un thread MARCEL auquel envoyer le signal, si possible celui qui s'exécute déjà actuellement sur le thread noyau ayant reçu le signal par exemple. Les signaux envoyés entre threads MARCEL (tels que représenté au milieu) sont traités directement en interne sans passer par le noyau, sauf s'ils doivent terminer le processus et générer éventuellement un fichier *core*.

compilée pour utiliser la bibliothèque de threads POSIX de GNU/Linux, la NPTL. Elle permet également d'exécuter avec MARCEL une application utilisant un environnement d'exécution basés sur cette même bibliothèque. Cette deuxième couche est plus lourde à implémenter car elle impose de respecter les conventions binaires existantes (tailles des structures de données, valeurs des constantes, etc.). Elle a été portée sur les architectures x86, x86_64 et Itanium.

Ces couches de compatibilité n'étaient cependant pas complètes, il leur manquait notamment la gestion des signaux et la gestion des variables par thread (*Thread Local Storage*, TLS). J'ai donc encadré Sylvain JEULAND durant son stage de Master 1 [Jeu06], qui a consisté à finaliser le travail de Vincent DANJEAN, et notamment implémenter la gestion des signaux au sein de MARCEL. Pour conserver un fonctionnement léger, toute cette gestion est effectuée en espace utilisateur. MARCEL met en place auprès du noyau ses propres traitants de signaux pour réceptionner ceux qui proviennent des autres processus. Par contre, les signaux envoyés entre threads MARCEL ou par la fonction `raise` sont gérés complètement en espace utilisateur. Par ailleurs, j'ai implémenté le mécanisme de TLS [Dre03] au sein de MARCEL. La difficulté est que sur les architectures x86 (resp. x86_64), cela implique le registre de segment `gs` (resp. `fs`) dont la mise à jour est contrainte pour des raisons de sécurité. Il est alors nécessaire, pour chaque thread MARCEL, d'enregistrer auprès du noyau une entrée dans la table LDT (*Local Descriptor Table*). Heureusement, le mécanisme de cache générique présenté à la section 4.4.3 (page 75) permet d'éviter d'effectuer cet enregistrement pour chaque création de thread. Avec ces dernières finitions, les interfaces de compatibilité POSIX sont presque complètes (il manque essentiellement le support des verrous inter-processus), et il est alors possible de lancer sans les modifier ni même les recompiler des applications conséquentes : Mozilla FireFox, OpenOffice.org, mais aussi la JVM (*Java Virtual Machine*) de SUN !

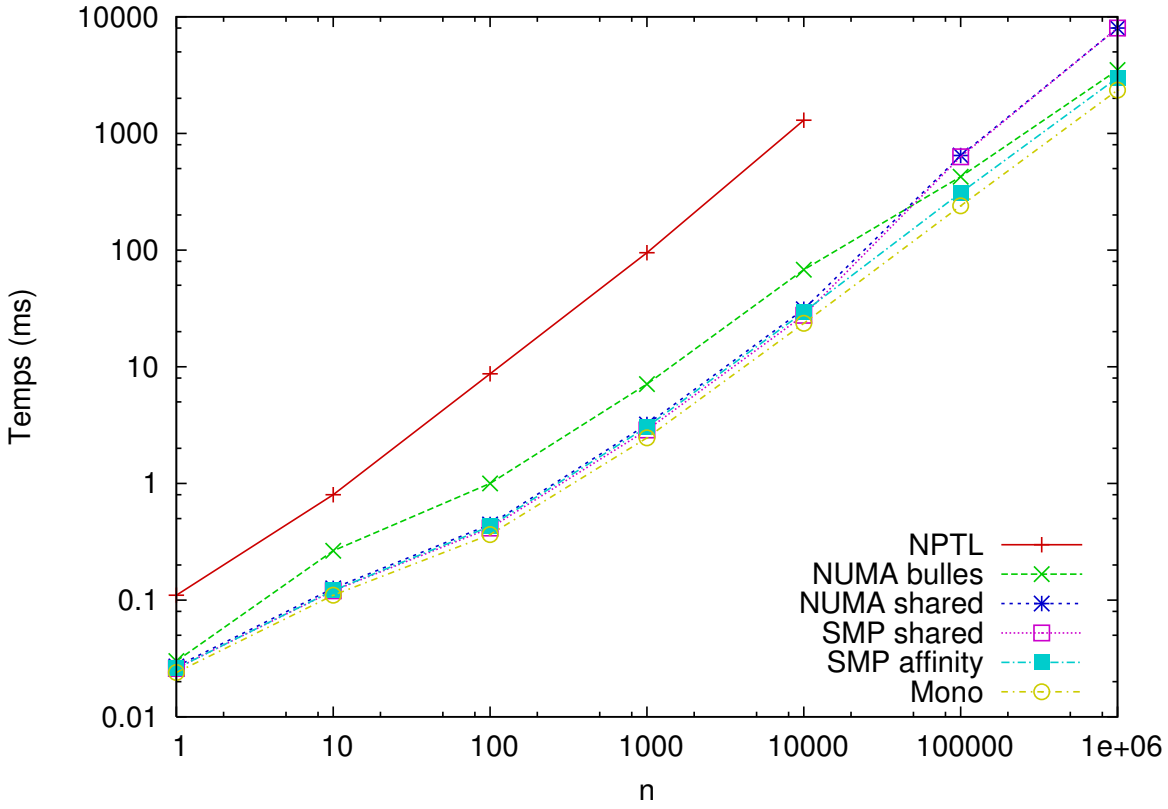


FIGURE 6.1 – Performances du programme *sumtime* en fonction des profils de compilation et des stratégies d'ordonnancement.

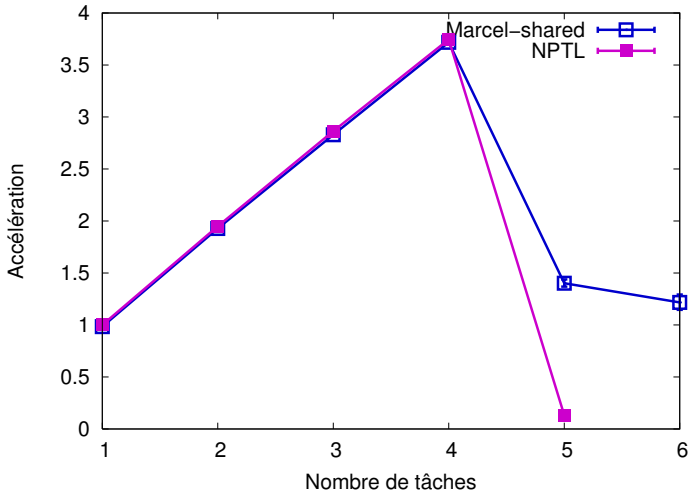
gang pour exécuter des jobs de manière concurrente. La troisième, BT-MZ, utilise l'interface OPENMP pour exprimer deux niveaux de parallélisme imbriqués, ainsi que l'ordonnanceur *Spread* pour répartir les bulles d'une manière équilibrée en terme de charge. La dernière, MPU, utilise l'interface OPENMP pour exprimer un parallélisme très déséquilibré et profite de l'implémentation des graines de threads pour pouvoir exprimer ce parallélisme à un grain extrêmement fin. Puisque la charge de calcul n'est *a priori* pas connue, MPU utilise l'ordonnanceur *Affinity*.

6.2.1 ADVECTION / CONDUCTION, un parallélisme régulier

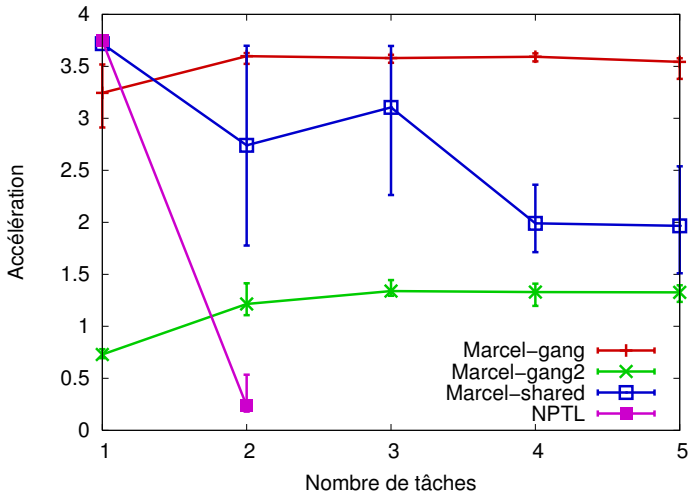
Les applications ADVECTION et CONDUCTION [Pér05], dont le support commun d'exécution MPC implémenté par Marc PÉRACHE [Pér06] a été porté pour MARCEL, effectuent des simulations d'advection et de conduction de chaleur au sein d'un grand maillage 2D régulier découpé en bandes. L'exécution est composée de cycles de calcul purement parallèle séparés par des étapes de communication globale hiérarchique. Dans les deux applications le coût des communications entre les mailles est plutôt important par rapport au temps de calcul (à cause de la latence pour l'advection, à cause du débit pour la conduction). Il est donc important lors de la parallélisation de prendre garde aux positions relatives des threads.

La machine cible est une machine BULL NOVASCALE comportant 4 nœuds NUMA composés de 4 processeurs ITANIUM II et 16 Go de mémoire, soit un total de 16 processeurs et 64 Go de mémoire. Le facteur NUMA est théoriquement de 3, en pratique on peut mesurer un facteur $1,5$.

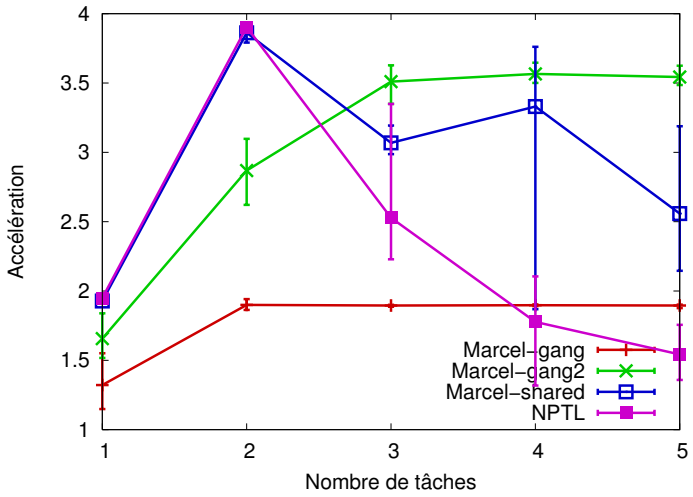
La table 6.4 montre les résultats obtenus selon les approches utilisées pour ordonnancer les threads. Dans la version Simple, on utilise la stratégie de placement *shared* de MARCEL, qui place les threads sur la liste globale, effectuant ainsi un simple ordonnancement opportuniste ne se préoccupant pas des affinités entre threads, l'accélération obtenue est alors assez limitée. Dans la version *fixée*, les applications ont été modifiées pour placer explicitement les threads sur les processeurs de la machine cible d'une manière adaptée à la hiérarchie de cette machine, ce qui permet ainsi d'obtenir un gain d'accélération appréciable. Le calcul de ce placement n'est cependant pas portable puisque la numérotation des processeurs ne peut être liée de manière standard à l'organisation hiérarchique de la machine. La version avec *bulles* a été obtenue en construisant simplement une hiérarchie de bulles selon la structure de l'application. En effet, pour éviter les contentions, le support d'exécution MPC effectue les communications entre threads de manière hiérarchique, en associant à 4 threads un thread de communication, qui lui-même est associé à 4 autres threads de communication et un thread de communication globale. Marc PÉRACHE a donc pu ajouter la construction de bulles d'une manière plutôt naturelle directement au sein du support d'exécution MPC, en rassemblant dans des bulles les threads par groupes de 4 avec le thread de communication associé, puis en rassemblant ces bulles et le thread de communication globale dans une bulle. Puisque l'application est ici très régulière, il n'y a besoin ni de répartition selon la charge ni de vol de travail, et l'on se contente d'utiliser l'ordonnanceur *Burst* (décrit à la section 3.2.3 page 48), qui fait en sorte que les processeurs « tirent » chacun les bulles vers eux de manière opportuniste. Le résultat est une distribution de threads équivalente (sans forcément être égale) au placement manuel effectué dans la version *fixée*. Les résultats obtenus sont ainsi très proches de cette version. La différence est que la version avec bulles a été plus naturelle



(a) Accélération de la parallélisation brute.



(b) Exécution par tâches de 4 threads.



(c) Exécution par tâches de 2 threads.

FIGURE 6.2 – Performances de la bibliothèque SUPERLU selon les stratégies d'ordonnancement.

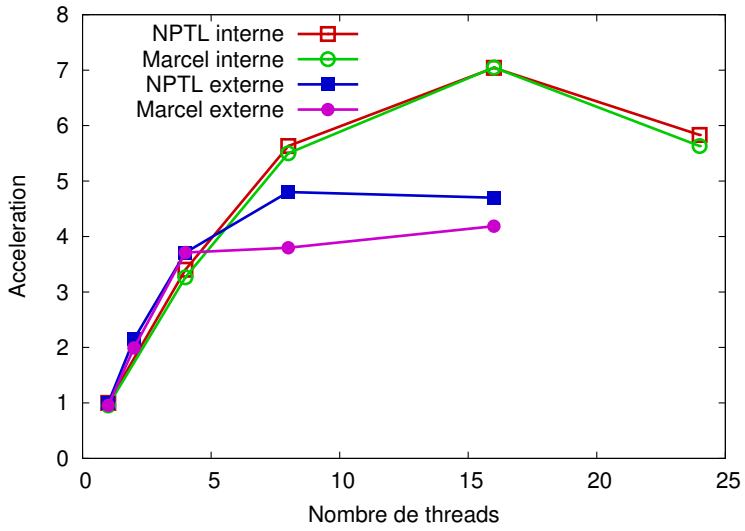


FIGURE 6.3 – Parallélisme externe et parallélisme interne de l'application BT-MZ.

la plus grande zone est typiquement 25 fois plus grande que la plus petite. De plus, le cas testé ne comporte que 16 zones. Il est donc essentiel d'effectuer un bon équilibrage de charge pour ne pas risquer d'obtenir une accélération limitée.

Cependant, comme indiqué à la section 5.2.1, la version OPENMP fournie sur le site de la NASA n'utilise pas d'imbrication de sections parallèles, mais lance plutôt plusieurs processus utilisant chacun une section parallèle, ce qui leur permet notamment d'utiliser une combinaison d'OPENMP et de MPI pour effectuer la parallélisation. Nous avons modifié cette version pour utiliser des sections parallèles imbriquées, nous permettant ainsi de contrôler le placement des threads au sein d'un même processus. Nous nous retrouvons ainsi avec une application comportant deux niveaux de parallélisme : un parallélisme *externe*, irrégulier, correspondant au découpage (x, y) du domaine en zones, et un parallélisme *interne*, régulier, correspondant à la parallélisation selon z de l'algorithme. Les résultats présentés ici ont été obtenus sur la machine Hagrid, octo-bicœur Opteron (donc composée de 8 nœuds NUMA de 2 cœurs, soit 16 cœurs en tout), détaillée à la page 14.

La figure 6.3 montre les résultats que l'on obtient si l'on active seulement un des deux niveaux de parallélisme. Si l'on n'exploite que le parallélisme *externe*, les zones elles-mêmes sont distribuées sur les différents processeurs. Puisque les tailles des zones varient fortement, l'accélération est limitée par le déséquilibre de charge de calcul dû aux zones les plus grandes. Lorsque l'on exploite le parallélisme *interne*, la répartition de charge est bonne, mais la nature même du calcul introduit de nombreux échanges de données entre processeurs. En particulier parce que la machine de test est une machine NUMA, l'accélération reste donc limitée à 7.

En combinant les deux niveaux de parallélisme, on peut espérer obtenir de meilleures performances en profitant des bénéfices des deux niveaux de parallélisme (localité et équilibrage de charge). Comme l'indiquent DURAN *et al.* [DGC05], l'accélération obtenue dépend du nombre relatif de threads créés au niveau du parallélisme externe et du nombre de sous-threads créés au niveau du parallélisme interne. Nous avons donc essayé toutes les com-

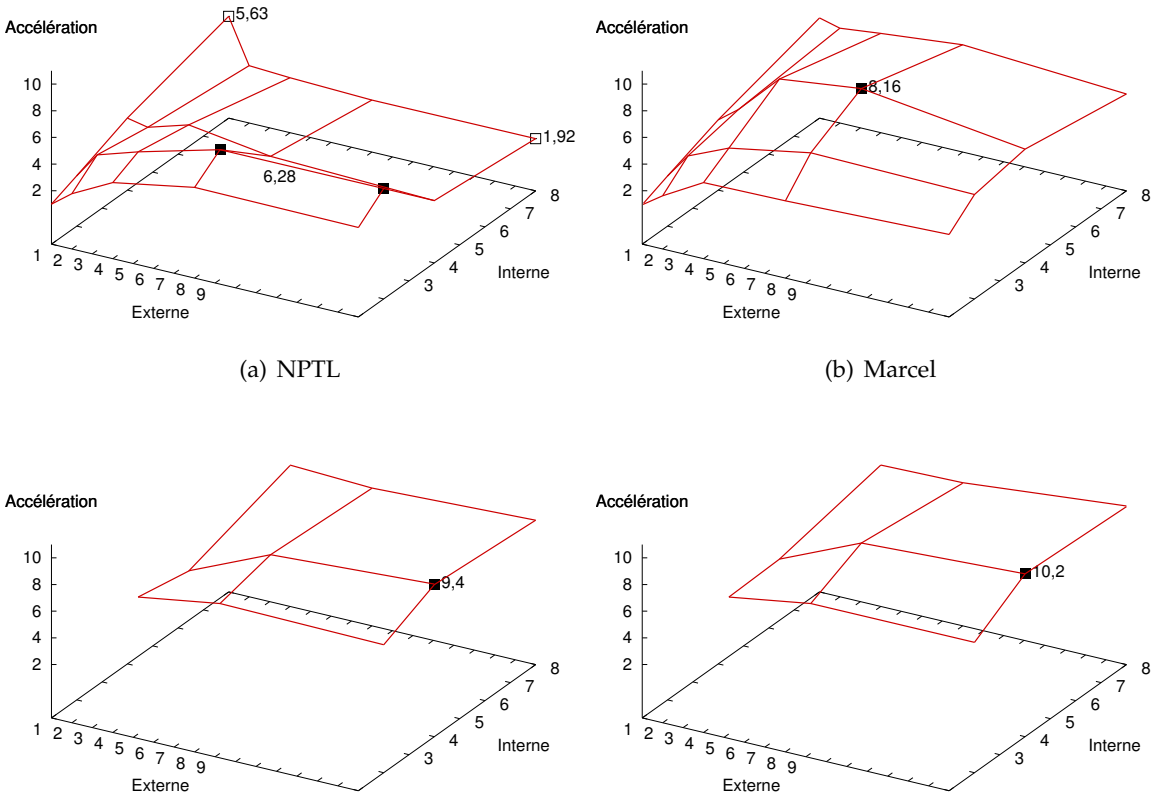
binaisons entre 1 et 16 threads pour le parallélisme externe (puisque ce parallélisme est de toutes façons limité aux 16 zones), et entre 1 et 8 sous-threads pour le parallélisme interne. La figure 6.4(a) montre les résultats obtenus lorsque l'on utilise simplement la bibliothèque de threads de Linux, la NPTL. Lorsque l'on choisit 1 pour le parallélisme externe ou que l'on choisit 1 pour le parallélisme interne (ce qui revient à désactiver l'un ou l'autre), on retrouve évidemment les courbes de la figure 6.3. Lorsque l'on combine réellement les deux niveaux de parallélisme, l'accélération n'est en fait pas meilleure ($6,28$, contre $7,02$ obtenu avec le parallélisme interne seul), et elle retombe même à $1,92$! La bibliothèque MARCEL, dont les résultats sont montrés à la figure 6.4(b), s'en sort mieux parce que la création de threads est bien plus légère et que l'ordonnancement de base est moins agressif, mais elle n'obtient, au mieux, qu'une accélération de $8,16$. Ces accélérations limitées sont dues au fait que ni la NPTL ni MARCEL ne prennent en compte les affinités entre threads, si bien que les sous-threads sont répartis assez aléatoirement sur la machine, conduisant à de nombreux accès distants, des défauts de cache, etc. Nous avons donc introduit l'utilisation de bulles.

Comme décrit à la section 5.2.3 (page 81), une hiérarchie de bulles peut être construite automatiquement à partir de l'imbrication des sections parallèles. Cela conduit alors ici à créer pour chaque thread du parallélisme externe (traitant donc un certain nombre de zones) une bulle qui contient les sous-threads traitant le parallélisme interne associé à ce thread. Puisque la charge de calcul des différentes zones est déséquilibrée mais connue, nous avons ajouté une ligne de code dans l'application pour qu'elle indique cette charge. Nous avons alors utilisé l'ordonnanceur à bulles *Spread* pour répartir cette hiérarchie de bulles sur la machine tout en prenant en compte leur charge de calcul. Le résultat est visible sur la figure 6.4(c) : grâce au placement judicieux des threads selon leurs affinités, l'accélération passe à $9,4$. Cependant, en observant le comportement de l'ordonnanceur *Spread* à l'aide de notre outil d'animation de traces décrit à la section 3.3.2, nous avons réalisé que, parce que cet ordonnanceur fait descendre tous les threads jusqu'aux cœurs, cela induit un certain déséquilibre de charge. Nous avons donc réglé cet ordonnanceur pour qu'il ne fasse descendre les threads que jusqu'aux huit nœuds NUMA, et le résultat est visible sur la figure 6.4(d) : l'accélération devient encore meilleure, $10,2$.

Avec cet exemple, on s'aperçoit donc bien qu'exploiter plusieurs niveaux de parallélisme peut réellement apporter un gain de performances substantiel, à condition que l'environnement d'exécution utilise un ordonnanceur de threads adéquat.

6.2.4 MPU, un parallélisme imbriqué irrégulier dans le temps

Les dernières technologies d'acquisition 3D permettent de numériser des objets entiers sous forme de nuages de points. Le projet *Digital Michelangelo* a, par exemple, pour but de numériser les sculptures de Michel-Ange avec une précision suffisante pour pouvoir distinguer les coups de ciseau du sculpteur. Ces numérisations comportent typiquement des milliards de points. Pour pouvoir effectuer efficacement un bon rendu de l'objet, il est courant de reconstruire une surface à partir d'un tel nuage de points. L'algorithme utilisé au LABRI par Tamy BOUBEKEUR [BRS04] au sein de l'équipe IPARLA est MPU (*Multi-Level Partition of Unity*) [YAM+03], dont le principe dans le cas 2D est expliqué à la figure 6.5. La figure 6.6 montre le code C++ de la boucle principale de MPU : c'est simplement une fonction qui effectue une approximation sur un cube, et si celle-ci n'est pas suffisamment bonne, subdivise



(a) NPTL (b) Marcel

(c) Bulles (d) Bulles avec ordonnancement optimisé

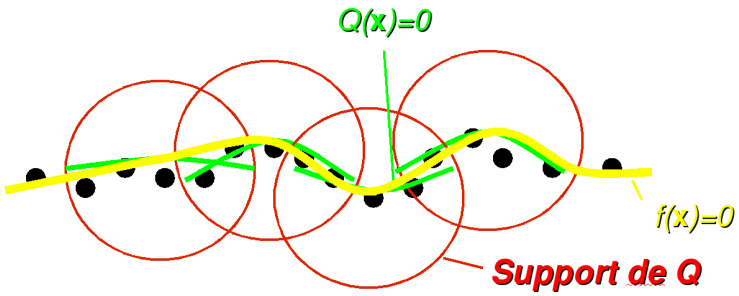
FIGURE 6.4 – Parallélisme imbriqué.

le cube en 8 sous-cubes pour lesquels elle s'appelle récursivement.

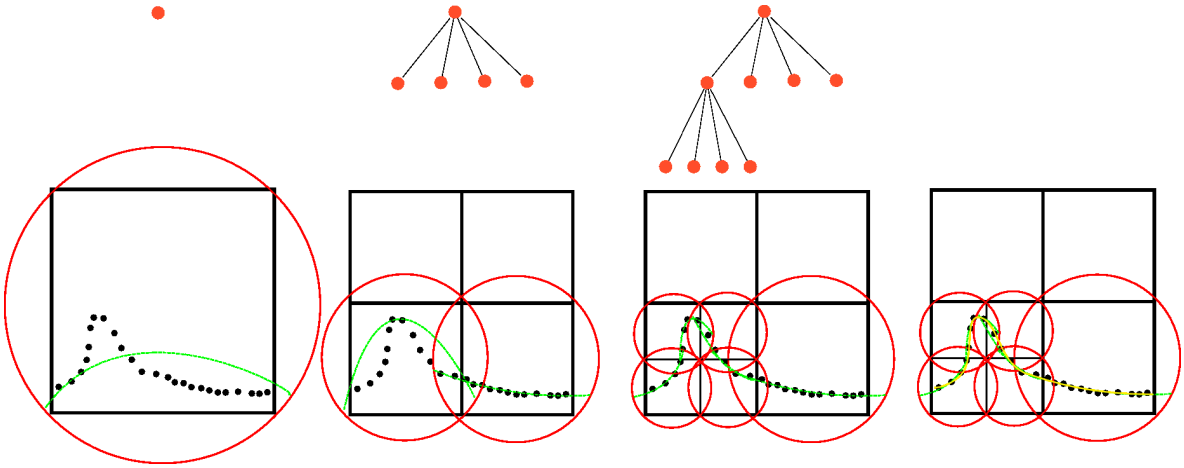
La parallélisation évidente de cet application à l'aide d'OPENMP revient alors à ajouter le `pragma` indiqué en gras. Les résultats obtenus avec la bibliothèque de threads de Linux, la NPTL, sont cependant loin d'être convaincants. Si l'on n'active pas le support des sections parallèles imbriquées, *i.e.* que seule la section parallèle la plus externe crée des threads, le parallélisme est limité à 8 voies, et la répartition de charge est en général plutôt déséquilibrée, si bien que même sur la machine Hagrid et ses 16 processeurs, l'accélération reste limitée à environ 5, comme on peut le voir sur la figure 6.8. Si l'on active le support des sections parallèles imbriquées, la répartition de charge est meilleure, mais Linux ne sait guère comment répartir la pléthore de threads ainsi créés (l'arbre de récurrence a typiquement une profondeur de 15 et une largeur de plusieurs centaines de threads), et l'accélération est, de fait, limitée à environ 4.

Durant son stage de Master 2 [Dia07] au sein de l'équipe, François DIAKHATÉ a développé sa propre version parallèle de MPU. Le principe consiste à lancer un thread par processeur, et de maintenir pour chacun d'entre eux une liste de cubes à traiter, implémentée à l'aide d'un *deque*, décrit à la figure 6.7. Lorsque le processeur local raffine un cube, il empile les sous-cubes de son côté de son propre *deque* et en pioche un, pendant que d'autres processeurs inactifs volent éventuellement depuis l'autre côté. Ainsi, les cubes des *deques* restent triés par ordre de taille, le processeur local piochant toujours un des plus petits, et les autres processeurs volant toujours un des plus gros. Cela permet d'assurer une bonne localité d'exécution du côté du processeur local (puisqu'au final il parcourt la hiérarchie de cubes en profondeur), et de limiter les vols des autres processeurs (puisqu'ils volent les cubes les plus gros). Les résultats obtenus sont bien meilleurs : comme on peut le voir avec la courbe *Manuel* de la figure 6.8, à l'aide des 16 processeurs de la machine Hagrid, l'accélération atteint 15,5 !

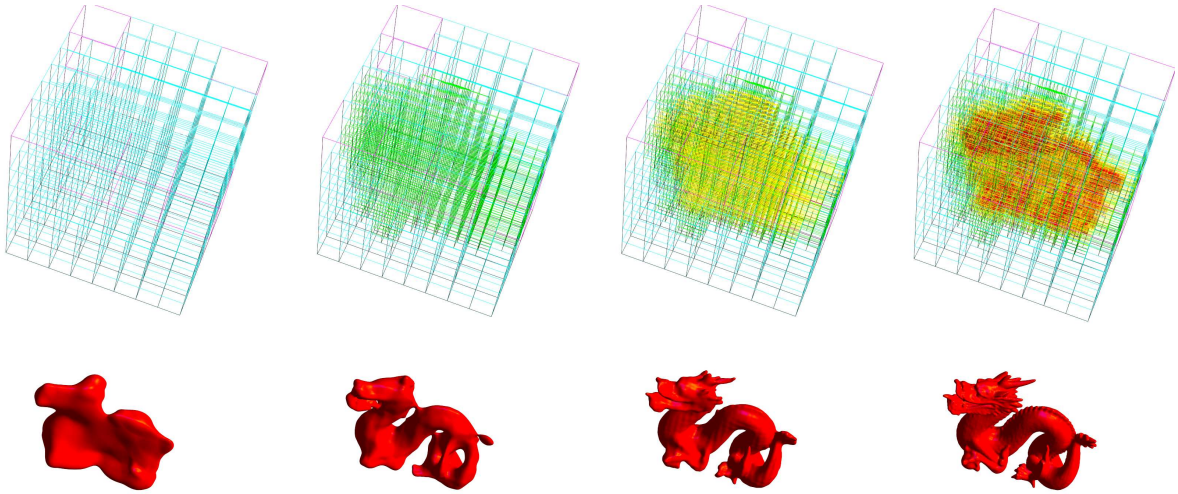
Ce très bon résultat a cependant nécessité des développements assez importants qui ne peuvent pas être facilement réutilisés pour d'autres applications. Il serait plus intéressant d'obtenir un résultat similaire sans modification de l'application (ou très peu). De nouveau, nous utilisons la construction automatique (décrite en section 5.2.3 page 81) d'une hiérarchie de bulles à partir de l'imbrication des sections parallèles. Le résultat est qu'à chaque cube correspond une bulle qui contient les threads travaillant sur les sous-cubes de ce cube. Puisque l'on ne connaît pas *a priori* la charge des threads et bulles, il vaut mieux utiliser l'ordonnanceur *Affinity*, pour au moins privilégier les affinités entre threads et utiliser un vol de travail pour équilibrer la charge. De plus, puisque la quantité de calcul effectuée par les threads traitant de très petits cubes est potentiellement très faible, nous activons l'utilisation des graines de threads. En pratique, l'ordonnancement obtenu est alors très proche de celui effectué à la main par François DIAKHATÉ. En effet, la partie haute de la hiérarchie de bulles se retrouve rapidement distribuée sur la machine, et sur chaque processeur la partie basse de la hiérarchie de threads qui s'y retrouve est exécutée par un parcours en profondeur, car les graines de threads générées par l'entrée dans une section parallèle sont mises en tête de la liste locale. De plus, lorsqu'un processeur est inactif, il vole du travail à un autre processeur en prenant en compte la structure de bulle, *i.e.* il vole si possible une bulle de niveau le plus externe, et donc une quantité de travail la plus potentiellement conséquente possible, ce qui revient bien au vol de travail de l'ordonnancement manuel. Le résultat est visible sur la figure 6.8 : sans être équivalente (la gestion des graines de threads et des bulles est bien plus lourde que celle des tâches de l'ordonnancement manuel), l'accélération obtenue avec un



(a) L'objectif est d'approcher implicitement un nuage de points (en noir) à l'aide d'une équation $f(x) = 0$ (en jaune). Le principe consiste à approximer localement une partie du nuage de points à l'aide d'une quadrique : pour chaque ensemble de points cerclé en rouge, une quadrique $Q(x) = 0$ est calculée (en vert). En sommant ces quadriques, on obtient une équation globale $f(x) = 0$.



(b) En pratique, des quadriques sont d'abord calculées pour les points compris dans des sphères englobant les cubes d'un maillage grossier. Les quadriques qui approximent suffisamment bien le nuage de point (c'est le cas en bas à droite) sont conservées telles quelles. Sinon, le maillage est raffiné localement, et de nouvelles quadriques sont calculées, comme illustré en bas à gauche. L'algorithme itère ainsi localement tant que l'approximation n'est pas jugée suffisante. La fonction globale est alors la somme de cette hiérarchie de quadriques.



(c) Selon la finesse d'approximation voulue, on effectue ainsi au final un raffinement hiérarchique du maillage adapté à la géométrie de l'objet, ici en 3 dimensions.

FIGURE 6.5 – Principe de la partition de l'unité multi-niveau (MPU).

```
void ImplicitOctCell::buildFunction() {
  /* Approximation sur le cube. */
  computeLA();
  if (error < maxError)
    return;


  /* Approximation trop grossière, raffiner. */
  /* Supprimer l'approximation. */   deleteLA();
  /* Découper en sous-cubes. */   Split();
#pragma omp parallel for
  for (int i=0; i<8; i++)
    subCell[i]->buildFunction();
}
```

FIGURE 6.6 – Boucle principale de l'application MPU.

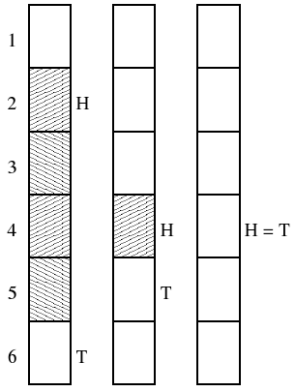


FIGURE 6.7 – Principe de fonctionnement d'un *deque* du protocole THE simplifié. Le processeur local empile et dépile les cubes à traiter de manière LIFO du côté T pendant que les autres processeurs peuvent voler du côté H. Ces opérations peuvent être effectuées sans verrouillage tant que H≠T.

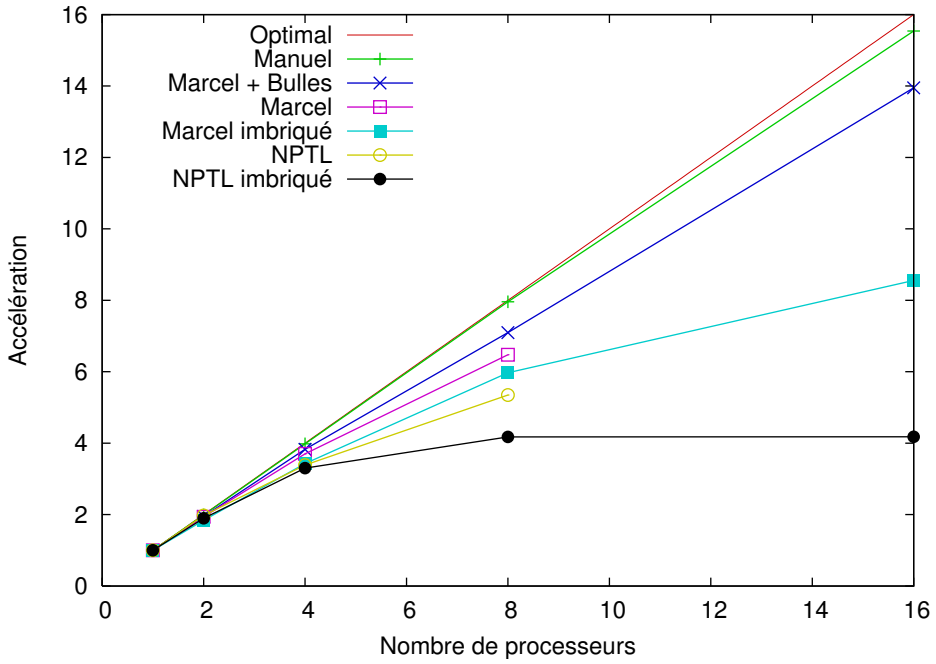
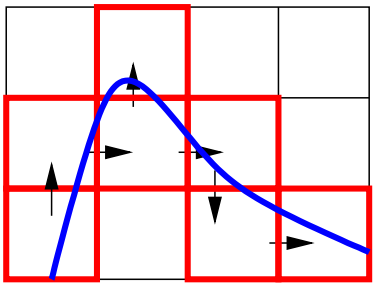


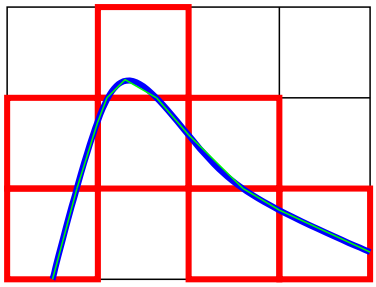
FIGURE 6.8 – Accélération de l'application MPU selon l'environnement d'exécution utilisé.

ordonnancement à bulles se rapproche de celle obtenue avec un ordonnancement manuel, pour un temps de développement très réduit pour le programmeur d'application, puisqu'il lui a suffit d'ajouter une directive OPENMP, de choisir l'ordonnanceur *Affinity* et d'activer l'utilisation de graines de threads.

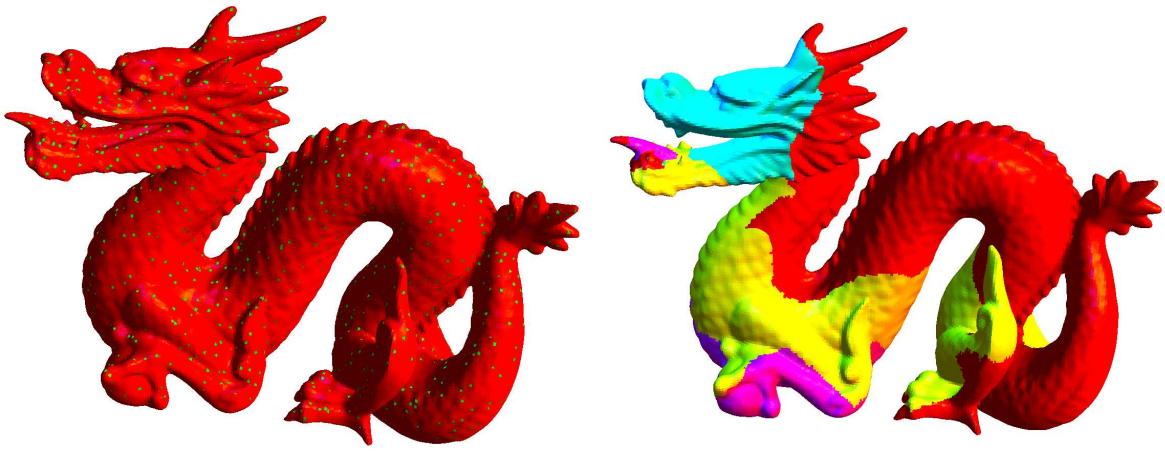
Une fois l'approximation du nuage de point effectuée, l'application MPU utilise une deuxième étape de construction d'un ensemble de polygones, détaillée à la figure 6.9. Il n'y a pas ici de structure hiérarchique qu'il serait véritablement naturel d'exprimer à l'aide de bulles, nous n'avons donc pas envisagé pour cette étape d'algorithme à bulles particulier, et répartissons donc simplement les threads sur chaque processeur. La parallélisation de cette étape a de toutes façons nécessité un certain travail qui se solde déjà par une accélération de presque 15. De telles performances n'ont pas réellement besoin d'être améliorées. Il est cependant important de noter ici l'utilité de pouvoir indiquer depuis l'application un changement d'étape : l'environnement d'exécution peut alors changer d'ordonnanceur, pour passer ici de l'ordonnanceur complexe *Affinity* à un ordonnancement bien plus simple, plus adapté à cette deuxième étape.



(a) Parcours de la surface à travers la grille.



(b) Résultat.



(c) Ensemble des cubes graines et distribution sur les processeurs.

FIGURE 6.9 – Construction d'un ensemble de polygones à l'aide de l'algorithme *Marching-Cube*. Le principe est que chaque processeur part d'un point de l'objet (une graine), et suit la surface implicite au travers d'une grille uniforme. Au sein de chaque cube de la grille, on approxime la surface à l'aide de polygones. Un protocole léger s'assure que deux processeurs ne se gênent pas lorsqu'ils traitent de cubes voisins, produisant ainsi une *frontière* entre les domaines traités par chaque processeur. Lorsqu'un processeur ne peut plus progresser, il pioche une autre graine.