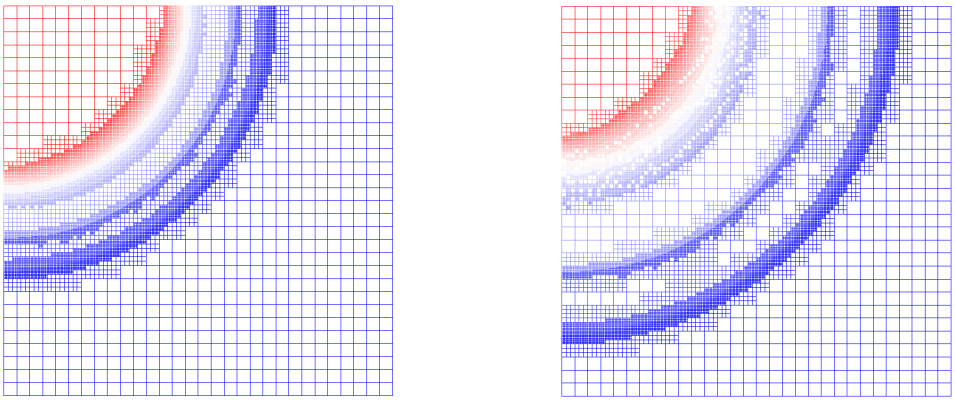


(a) Cycle 171. (b) Cycle 337.

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

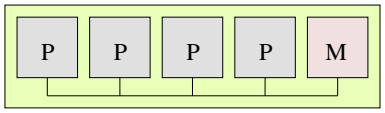


FIGURE 1.2 – Architecture SMP.

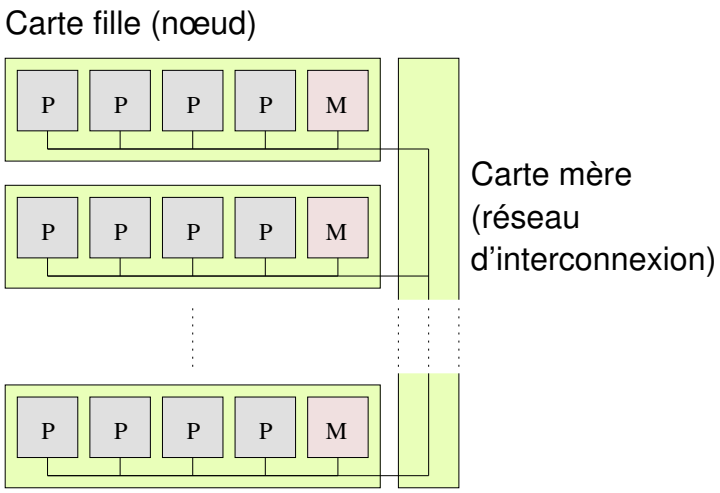


FIGURE 1.3 – Architecture NUMA.

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.

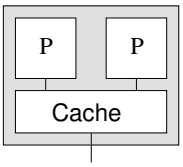


FIGURE 1.4 – Puce multicœur.

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

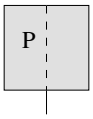


FIGURE 1.5 – Processeur multithreadé 2 voies.

	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

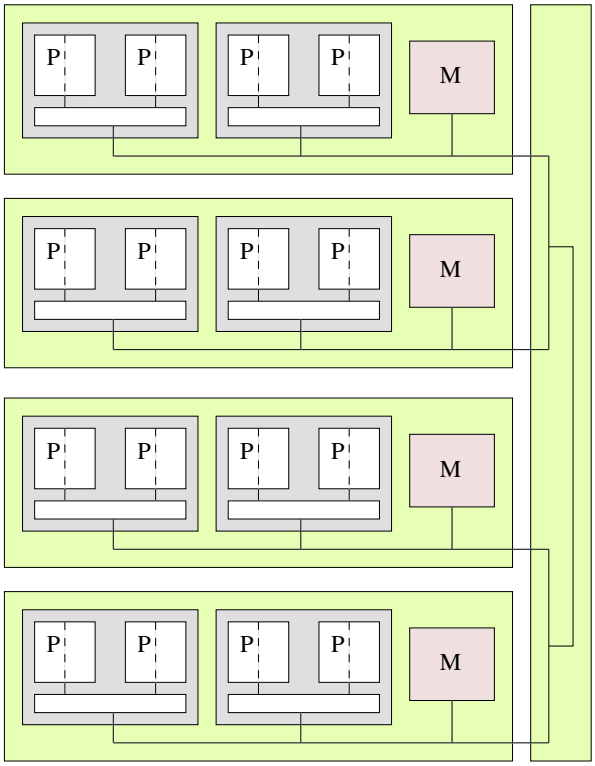


FIGURE 1.6 – Architecture très hiérarchisée.

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

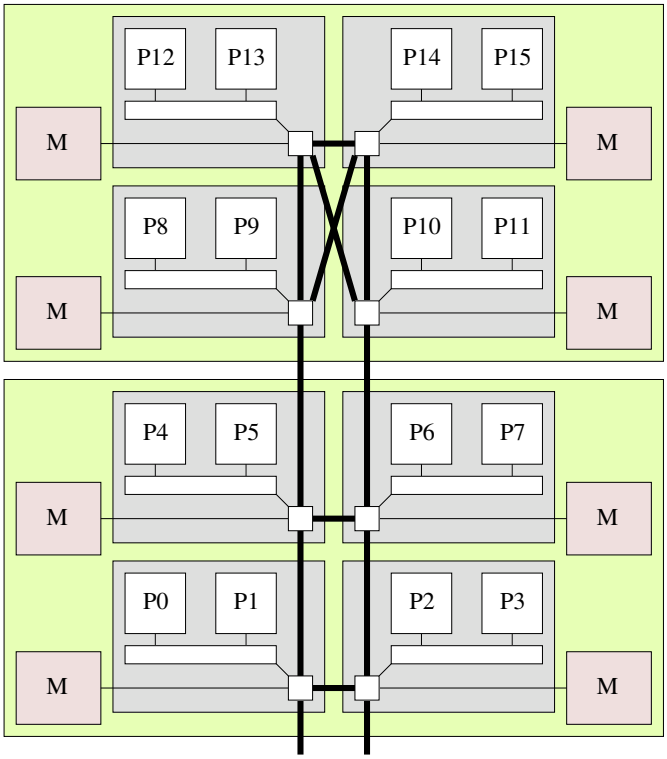


FIGURE 1.7 – Hagrid, octo-bicœur Opteron.
Chaque nœud NUMA est composé d'une puce bicœur Opteron cadencée à $1,8$ GHz et de 8 Go de mémoire. Ces nœuds sont reliés par un réseau de liens HyperTransport. Le facteur NUMA varie selon la position relative de l'emplacement mémoire accédée et du processeur 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.

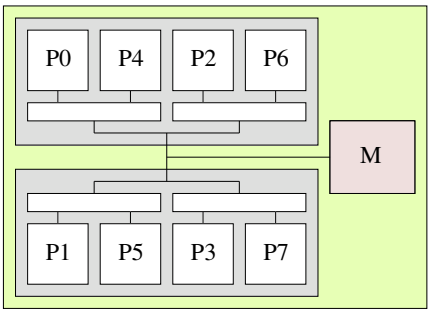


FIGURE 1.8 – Aragog, bi-quadricœur E5345 Xeon.
Cette machine est composée de deux puces cadencées à $2,33$ GHz, chacune embarquant quatre cœurs. L'organisation hiérarchique des caches peut s'observer en mesurant la vitesse d'une boucle intensive de calcul sur des variables *volatiles* situées dans une 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...

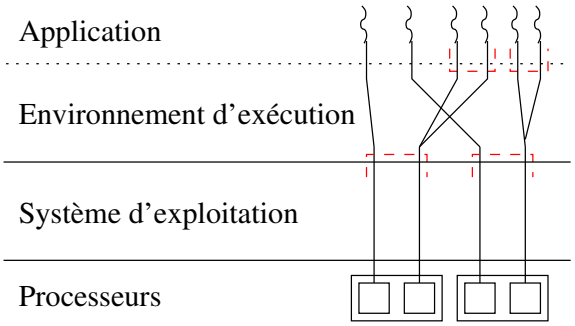


FIGURE 1.11 – Ordonnancement négocié, utilisant des informations de l'application et s'adaptant à la machine sous-jacente.

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

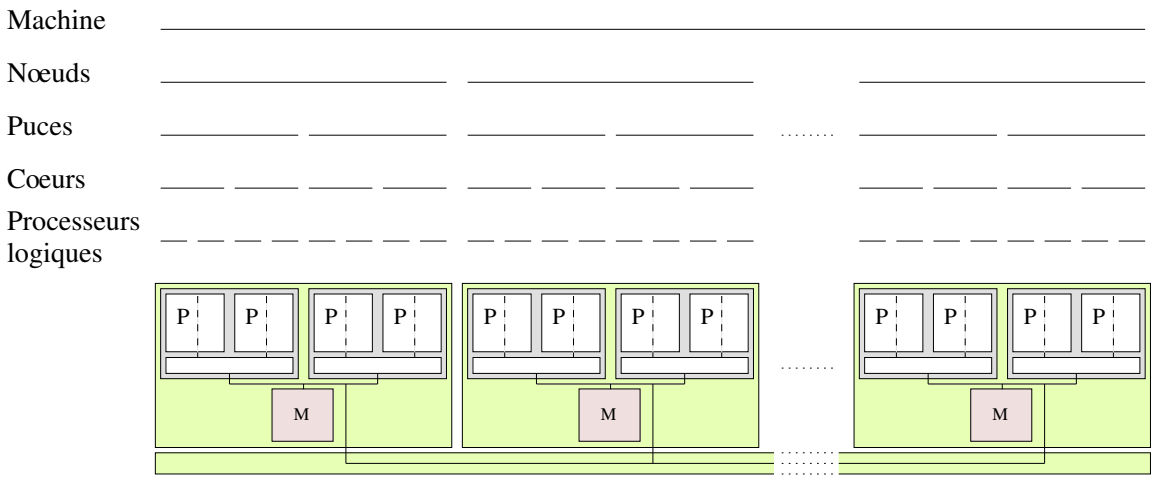


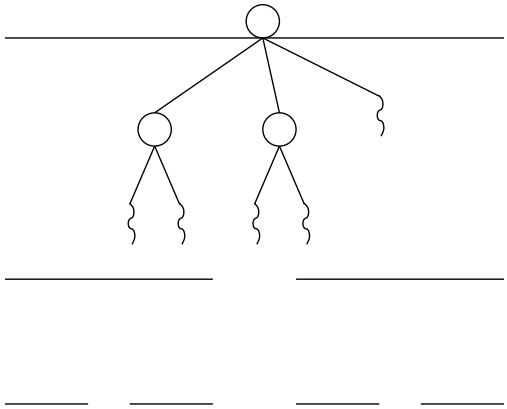
FIGURE 2.3 – Modélisation d'une machine très hiérarchisée par une hiérarchie de listes.

2.2 Coller à la structure de la puissance de calcul

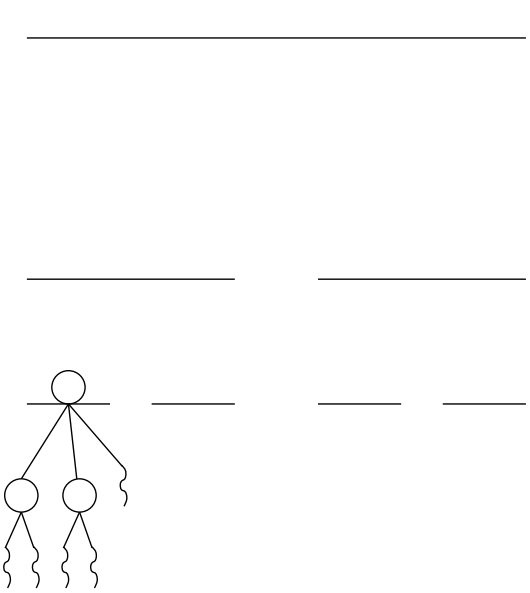
Une des difficultés lorsque l'on écrit un ordonnanceur est due au modèle de VON NEUMANN : ce sont les processeurs eux-mêmes qui exécutent les instructions d'un thread et basculent éventuellement sur un autre lorsque le précédent se termine ou se bloque en attente d'une ressource. Ainsi, *a priori*, on 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.

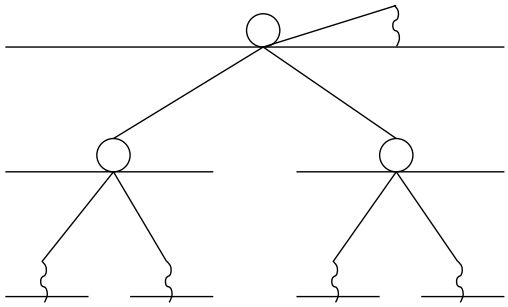
Il serait même aisé de modéliser un ensemble de machines NUMA reliées par un réseau



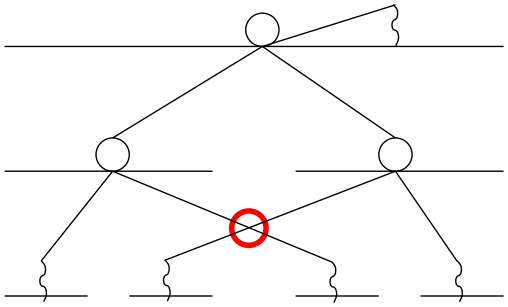
(a) Bonne utilisation processeur, affinités non prises en compte.



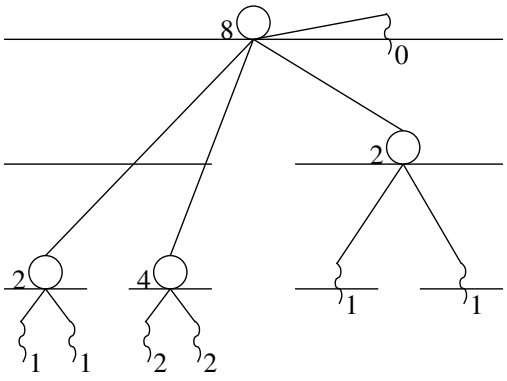
(b) Mauvaise utilisation processeur, affinités extrêmement prises en compte.



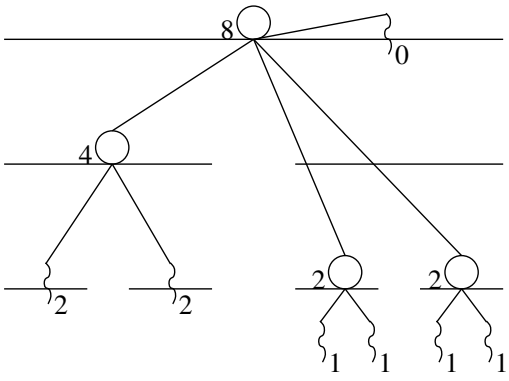
(c) Utilisation distribuée des processeurs, affinités correctement prises en compte.



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

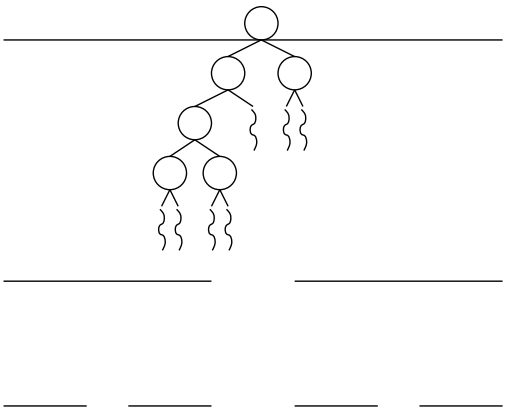


(e) Répartition simpliste, non équilibrée.

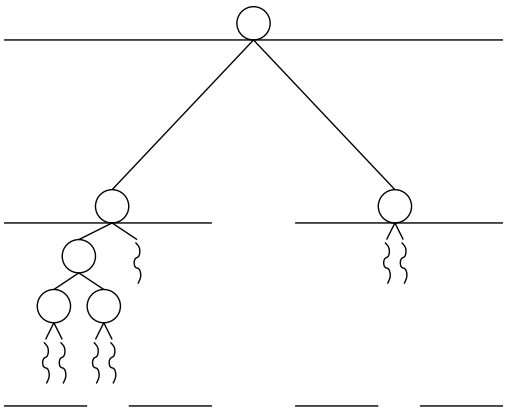


(f) Répartition selon la charge estimée.

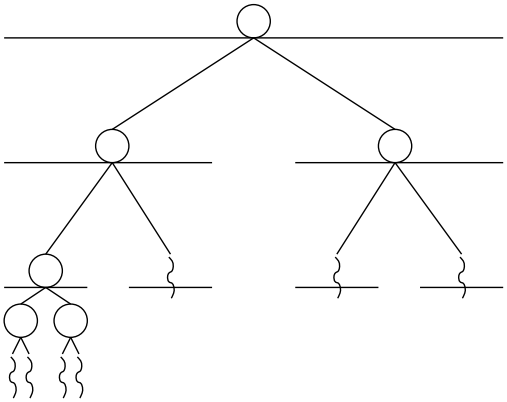
FIGURE 2.4 – Distributions possibles des threads et bulles sur la machine.



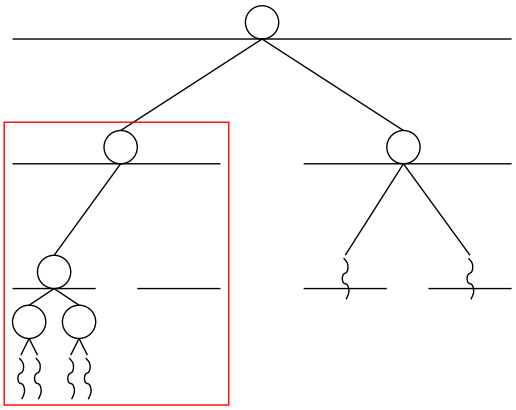
(a) Situation de départ.



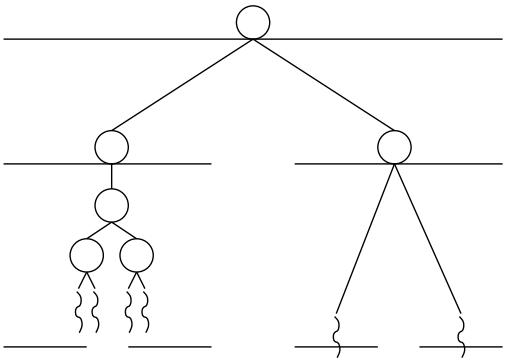
(b) Il faut éclater la bulle principale pour pouvoir alimenter les deux sous-listes.



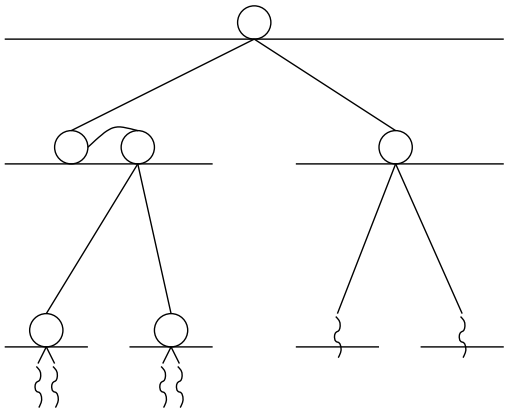
(c) Il faut éclater les deux sous-bulles pour pouvoir alimenter tous les processeurs. L'algorithme est terminé, malgré un déséquilibre en nombre de threads.



(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.

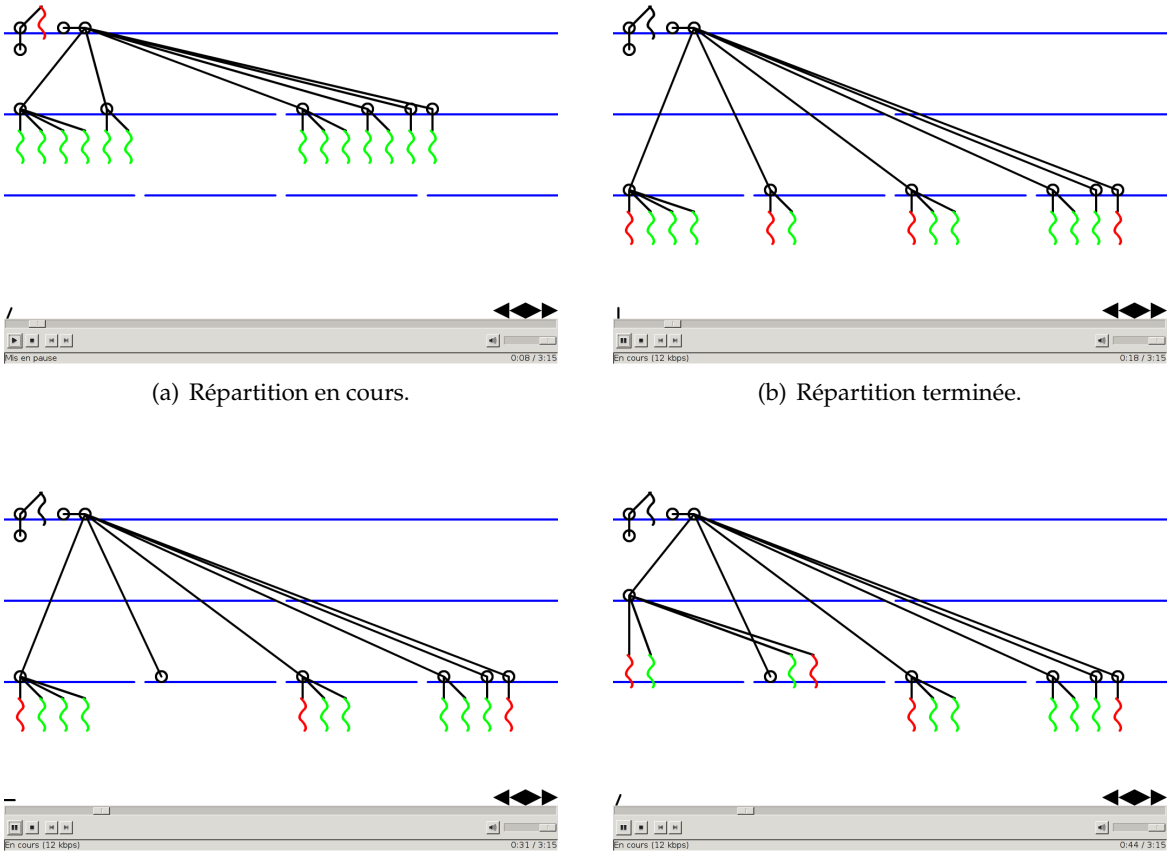


(e) Les entités de cette partie de machine sont rassemblées.



(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.

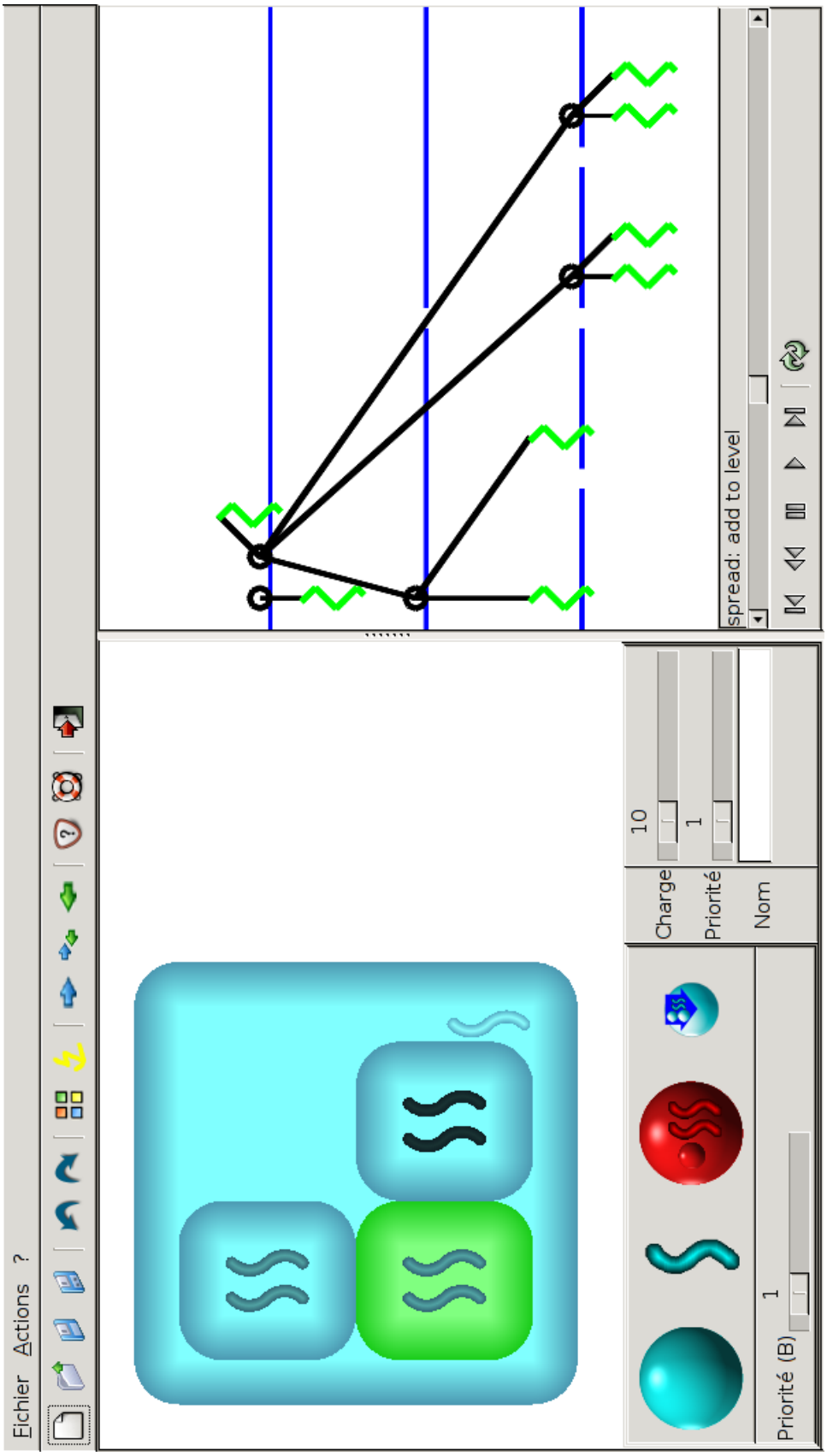


FIGURE 3.13 – Interface graphique de génération de bulles : *BubbleGum*.

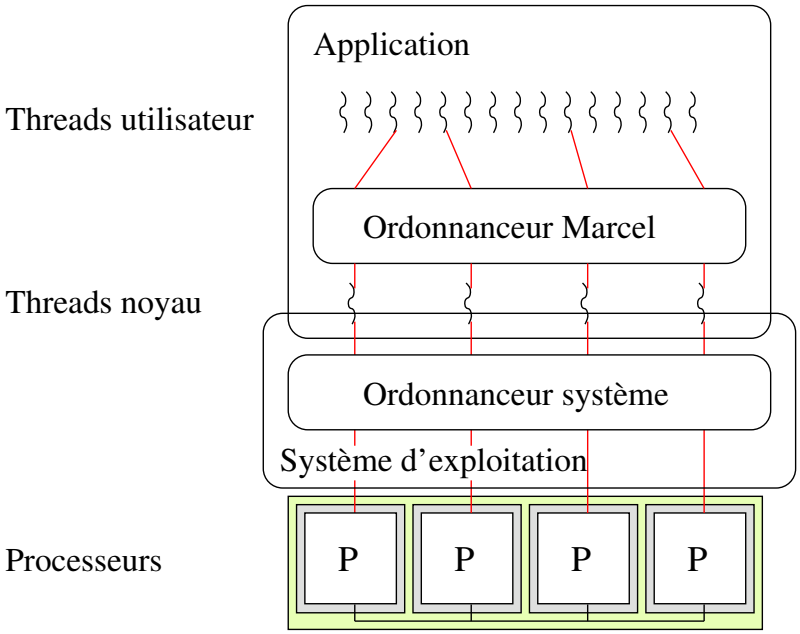


FIGURE 4.1 – Mode de fonctionnement de base de Marcel : l'ordonnanceur du système est en fait court-circuité.

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

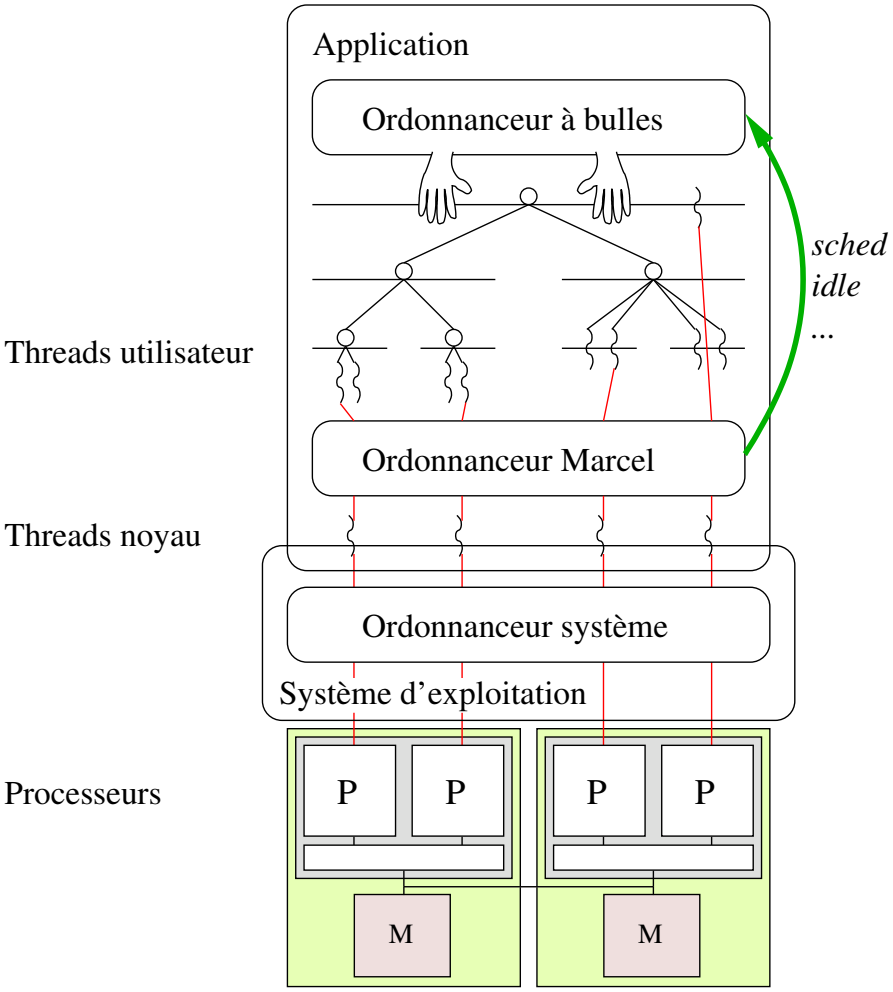


FIGURE 4.2 – Mode de fonctionnement de Marcel avec bulles : l'ordonnanceur de base de Marcel est conditionné par le placement des threads et bulles.

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-

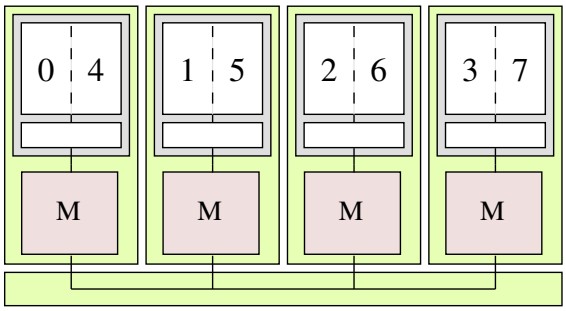


FIGURE 4.4 – Exemple de machine contenant deux nœuds NUMA comportant chacun un cœur hyperthreadé, avec une numérotation non consécutive (voir section 1.2.4 page15).

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.

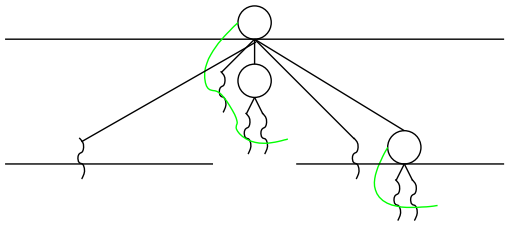


FIGURE 4.7 – Illustration du cache de threads.

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,

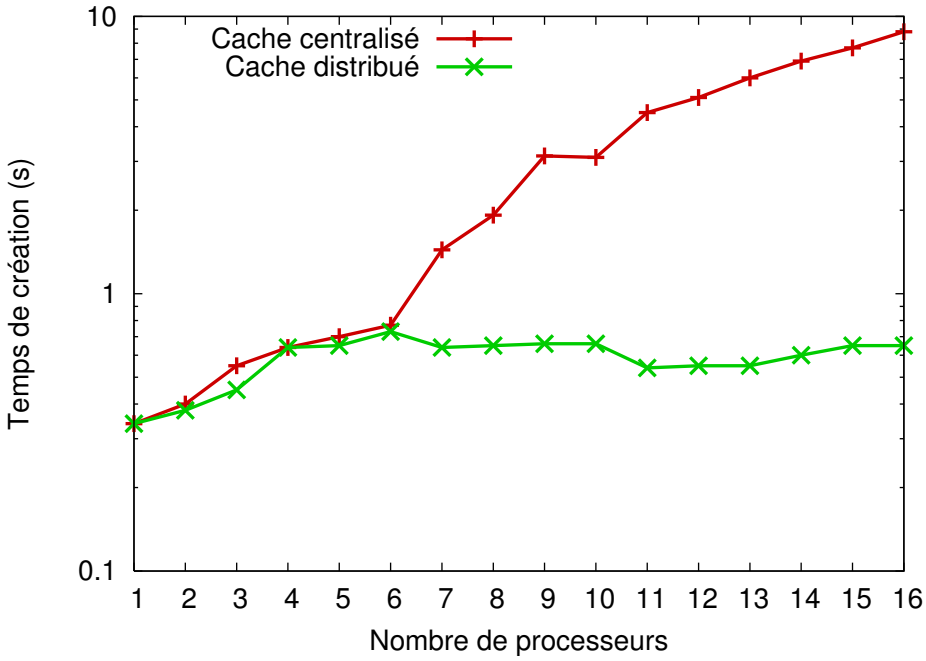


FIGURE 4.8 – Temps de création et de destruction parallèle de 100 000 threads.

MARCEL.

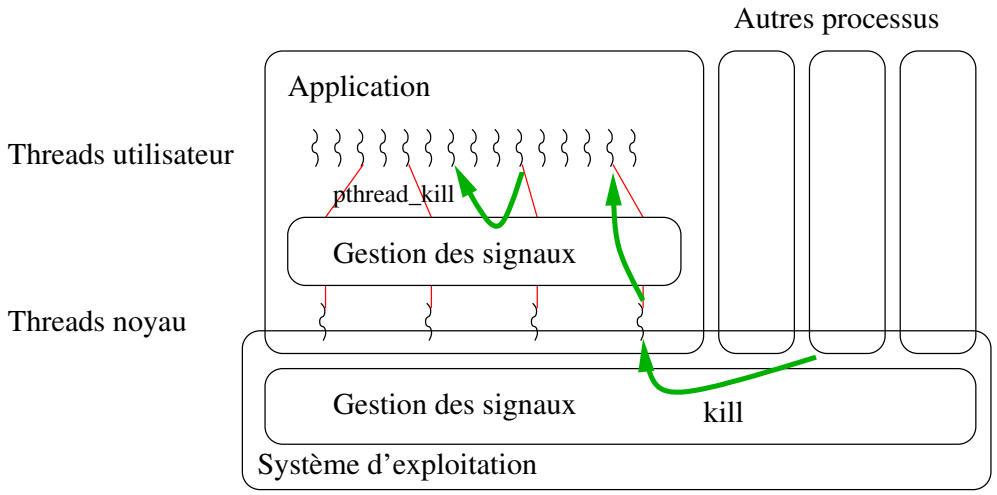


FIGURE 5.1 – Modèle hybride de gestion des signaux.
Lorsqu'un autre processus envoie un signal à l'application (représenté à droite), le noyau choisit un thread noyau auquel envoyer le signal. MARCEL choisit alors un thread MARCEL auquel envoyer le signal, si possible celui qui s'exécute déjà actuellement 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 !

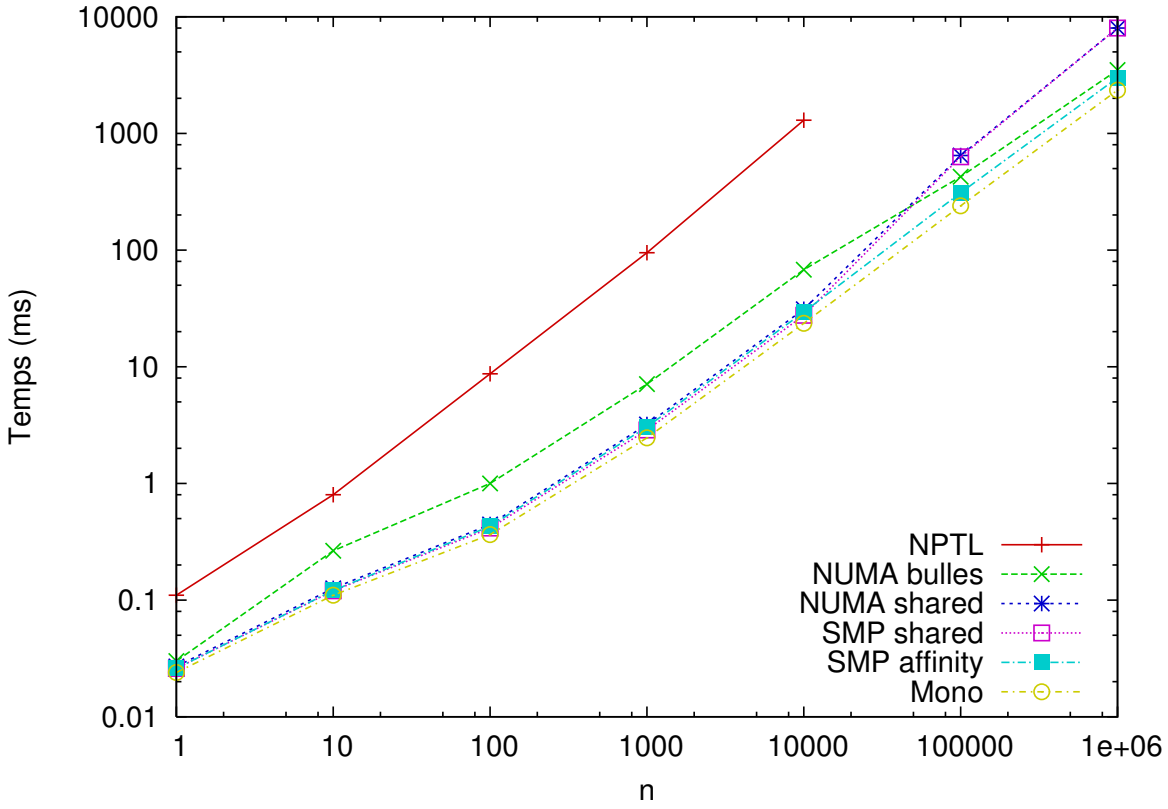
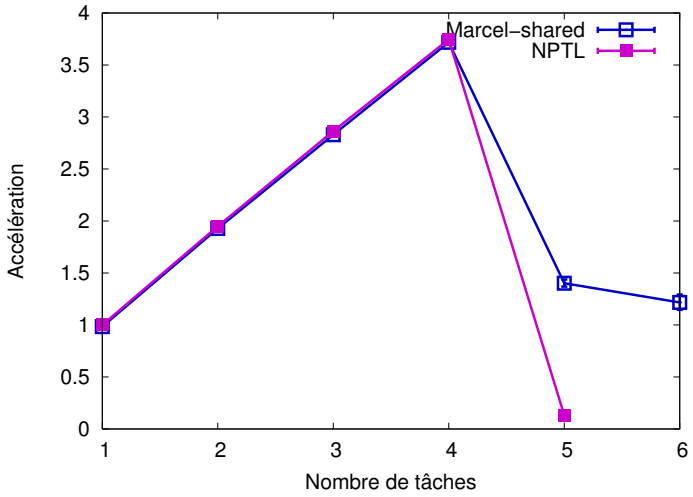
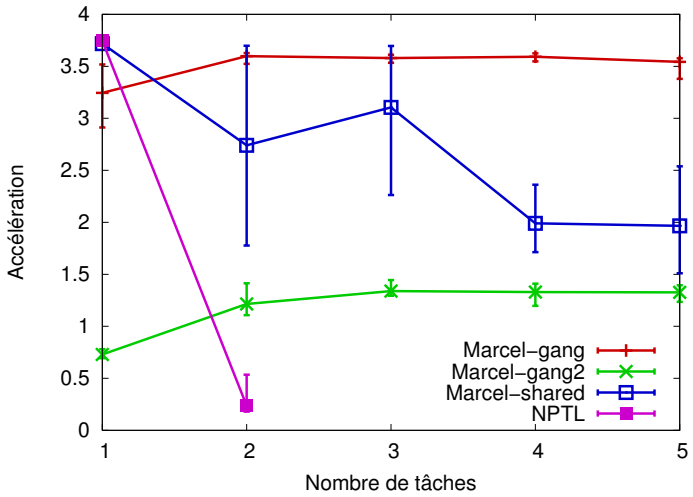


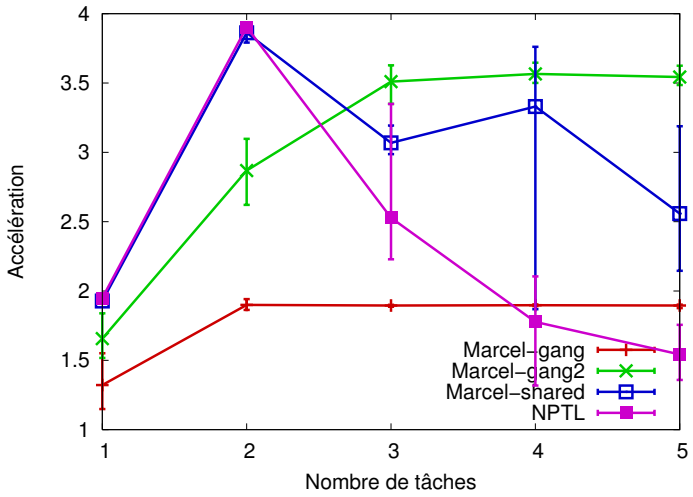
FIGURE 6.1 – Performances du programme *sumtime* en fonction des profils de compilation et des stratégies d'ordonnancement.



(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.

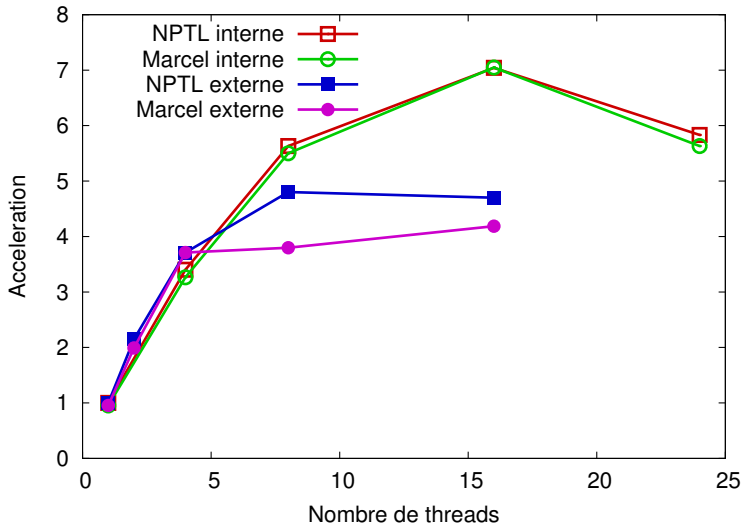


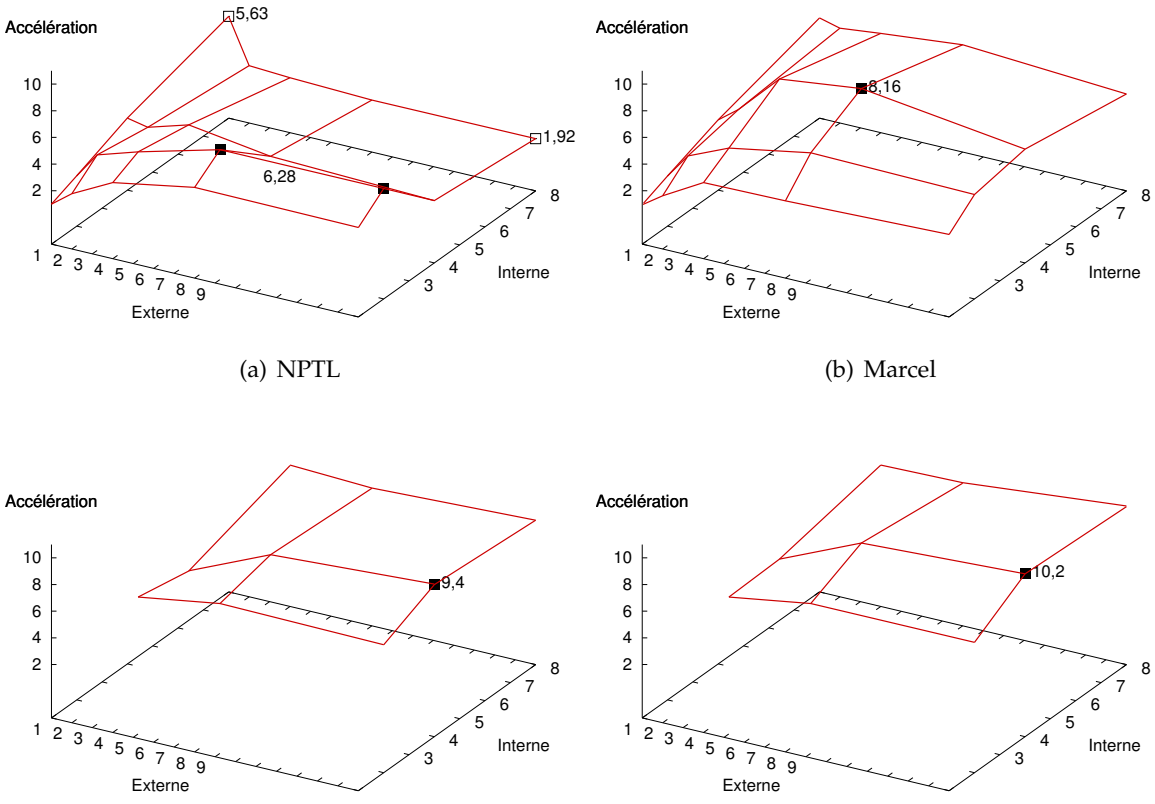
FIGURE 6.3 – Parallélisme externe et parallélisme interne de l'application BT-MZ.

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-



(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

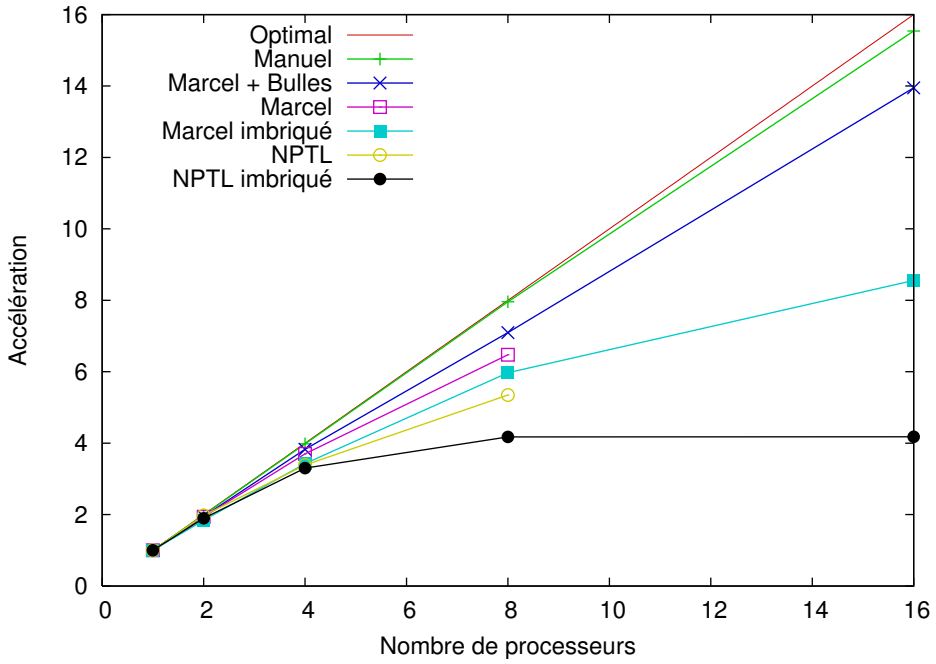
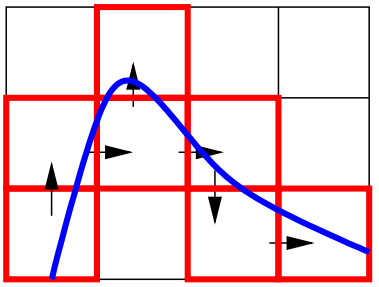


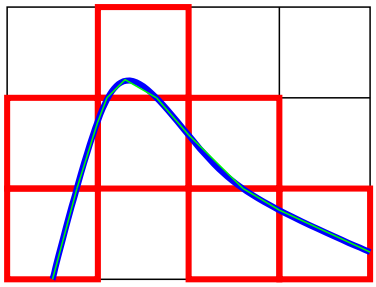
FIGURE 6.8 – Accélération de l'application MPU selon l'environnement d'exécution utilisé.

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.

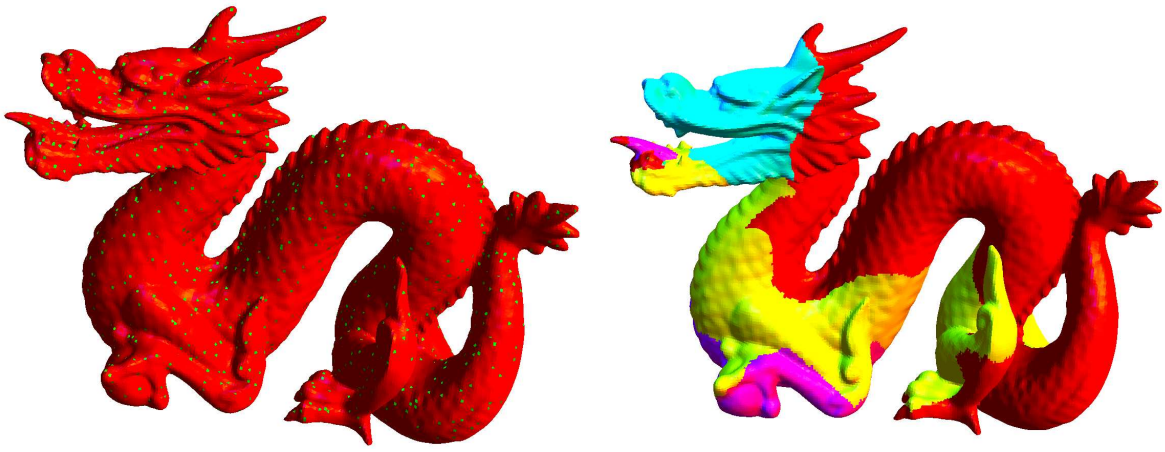
Une fois l'approximation du nuage de point effectuée, l'application MPU utilise une deuxième étape de construction d'un ensemble de polygones, détaillée à la figure 6.9. Il n'y a pas ici de structure hiérarchique qu'il serait véritablement naturel d'exprimer à l'aide de bulles, nous n'avons donc pas 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.