



Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance

Sébastien Jean Valat

► To cite this version:

Sébastien Jean Valat. Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance. Performance et fiabilité [cs.PF]. University de Versailles Saint-Quentin en Yvelines, 2014. Français. NNT : . tel-01253537

HAL Id: tel-01253537

<https://hal.science/tel-01253537>

Submitted on 11 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE DE DOCTORAT DE
L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN EN YVELINES

Spécialité

Informatique

à l'École doctorale des Science et Technologie de Versailles (STV)

présentée pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ DE VERSAILLES

et intitulée

**Contribution à l'amélioration des méthodes d'optimisation de la
gestion de la mémoire dans le cadre du Calcul Haute Performance.**

par **Sébastien Valat**

Organisme d'accueil :

CEA, DAM, DIF

F-91297 Arpajon France

Département des Sciences de la Simulation et de l'Information (DSSI)

Soutenue publiquement le 17 juillet 2014
devant le jury composé de :

M. William JALBY
M. Marc PÉRACHE

Directeur de thèse
Encadrant

Professeur à l'université de Versailles
Ingénieur chercheur au CEA,DAM,DIF

M. Allen D. MALONY
M. Jean-François MEHAUT
M. Alfredo GOLDMAN
M. Emmanuel JEANNOT

Président de jury
Rapporteur
Rapporteur
Examineur

Professeur à l'université de l'Oregon
Professeur à l'université Joseph Fourier
Professeur à l'université de São Paulo
Directeur de recherche INRIA

Résumé

L'évolution des architectures des calculateurs actuels est telle que la mémoire devient un problème majeur pour les performances. L'étude décrite dans ce document montre qu'il est déjà possible d'observer des pertes importantes imputables aux mécanismes de gestion de cette dernière. Dans ce contexte, nous nous sommes intéressés aux problèmes de gestion des gros segments mémoire sur les supercalculateurs multicoeurs NUMA de type Tera 100 et Curie. Notre travail est détaillé ici en suivant trois axes principaux.

Nous analysons dans un premier temps les politiques de pagination de différents systèmes d'exploitation (coloration de pages, grosses pages...). Nous mettons ainsi en évidence l'existence d'interférences néfastes entre ces politiques et les décisions de placement de l'allocateur en espace utilisateur. Nous complétons donc les études cache/allocateur et cache/pagination par une analyse de l'interaction cumulée de ces composants.

Nous abordons ensuite la problématique des performances d'allocation des grands segments mémoire en considérant les échanges entre le système et l'allocateur. Nous montrons ici qu'il est possible d'obtenir des gains significatifs (de l'ordre de 50% sur une grosse application) en limitant ces échanges et en structurant l'allocateur pour un support explicite des architectures NUMA.

La description de nos travaux s'achève sur une étude des problèmes d'extensibilité observés au niveau des fautes de pages du noyau Linux. Nous avons ainsi proposé une extension de la sémantique d'allocation afin d'éliminer la nécessité d'effectuer les coûteux effacements mémoire des pages au niveau système.

Abstract

Current supercomputer architectures are subject to memory related issues. For instance we can observe slowdowns induced by memory management mechanisms and their implementation. In this context, we focus on the management of large memory segments for multi-core and NUMA supercomputers similar to Tera 100 and Curie. We discuss our work in three parts.

We first study several paging policies (page coloring, huge pages...) from multiple operating systems. We demonstrate an interference between those policies and layout decisions taken by userspace allocators. Such interactions can significantly reduce cache efficiency depending on the application, particularly on multi-core architectures. This study extends existing works by studying interactions between the operating system, the allocator and caches.

Then, we discuss performance issues when large memory segments are allocated. To do so, we consider the interaction between the OS and userspace allocators. We show that we can significantly improve some application performances (up to 50%) by controlling the memory exchange rate with the OS and by taking care of memory topologies.

We finally study page fault extensibility in current Linux kernel implementation. We observe a large impact due to page zeroing which is a security requirement. We propose an improvement on memory allocation semantic aimed at avoiding page zeroing. It shows a new interest for huge pages to improve paging scalability without changing too much kernel algorithms.

Remerciements

Je tiens avant tout à remercier ceux qui ont acceptés d'être rapporteurs de ce travail, Jean-François Méhaut et Alfredo Goldman pour le temps qu'ils ont consacré à lire en détail ce manuscrit. Je remercie tout autant ceux qui ont acceptés d'être membre de mon jury Allen Mallony, Alfredo Goldman, Jean-François Méhaut, Emmanuel Jeannot, Marc Pérache et William Jalby. Un remerciement spécial pour les deux premiers membres de cette liste qui ont dû franchir l'atlantique pour venir assister à la soutenance.

Je donne toute ma gratitude à ceux qui ont permis la réalisation de ce travail et qui m'ont guidé tout au long de ces années de thèse. Je remercie tout particulièrement mon encadrant au CEA, Marc Pérache, pour avoir supervisé mon travail pendant ces années depuis mes débuts en stage. Merci à lui pour sa disponibilité et pour avoir laissé une part de liberté dans le travail réalisé. Merci d'avoir supporté la trop grande fougue dont j'ai parfois pu faire preuve sur l'avancée du projet MPC. Je tiens également à remercier mon directeur de thèse, William Jalby, pour avoir encadré ces travaux à l'UVSQ et de m'avoir fait découvrir les finesses des architectures des processeurs. Merci à lui de m'avoir accueilli à mes débuts au laboratoire Exascale.

Ma reconnaissance va aux membres du CEA pour m'avoir accueilli et d'avoir permis la réalisation de cette thèse, notamment Hervé Jourden, Bruno Scheurer et Pierre Leca. Également merci aux secrétaires du centre, Stéphanie, Isabelle et Éliane pour leur aide administrative et leur bonne humeur. Je n'oublie pas non plus Patrick Carribault également membre permanent de l'équipe MPC et qui a apporté son aide et sa bonne humeur tout au long de cette thèse.

Un grand merci aux doctorants Jean-Baptiste et Jean-Yves pour les discussions à n'en plus finir et pour leur enthousiasme pour les nouvelles idées geek. Merci à Bertrand et Alexandre pour les échanges sur la physique qui m'ont fait le plus grand bien. Merci à Julien, mon collègue de bureau, pour sa motivation, son enthousiasme et pour n'avoir jamais oublié de donner son propre avis lors de nos discussions techniques. Tout autant merci aux doctorants et post-doctorants du CEA avec qui j'ai partagé de très agréables moments dans nos bâtiments et autour de tables de repas animées : Marc, François, Emmanuel, Nicolas, Jordan, Jérôme, Antoine, Camille, Emmanuelle, Xavier, Thomas. Sans oublier les personnes avec qui j'ai pu travailler à l'UVSQ : Sylvain, Emmanuel, Cédric, Asma, Augustin, Aurèle et tous ceux dont il serait trop long de donner la liste complète. Merci à tout ceux précédemment cités qui ont fait parti du projet MPC et avec qui nous avons longuement échangé autour de ce projet.

Un grand merci à ma famille, notamment mes parents et ma sœur pour leur présence sur le chemin qui m'a amené jusqu'à cette thèse et pour leur soutien sans faille durant cette période. Merci à ceux qui ont pris du temps pour relire ce document, dont mon cousin, Roland.

Je tiens enfin à remercier l'ensemble des enseignants qui m'ont permis de concrétiser ce chemin, notamment à l'université de Savoie et à mes anciens collègues de l'association C-Net sans qui je ne serais pas où je suis aujourd'hui. Tout autant merci aux amis avec qui j'ai pu partager mes idées et mes rêves pendant ces années d'études et ceux qui m'ont soutenu pendant ces années de thèse.

Table des matières

Table des matières	9
Préambule	15
I Contexte	17
1 Introduction au calcul haute performance	19
1.1 Introduction	19
1.2 Historique du calcul haute performance	19
1.3 Évolution actuelle des architectures	21
1.3.1 Un équilibre entre débit et latence	21
1.3.2 L'arrivée du multicœur	22
1.3.3 Processeur versus GPGPU	24
1.4 Les défis de l'Exaflops	24
1.4.1 Gestion de l'énergie	25
1.4.2 La mémoire	25
1.4.3 Nombre de cœurs	26
1.4.4 Pannes	26
1.4.5 Entrées/sorties	27
1.4.6 Résumé	27
1.5 Environnement de calcul de la thèse	27
1.6 Modèles d'exécution	28
1.7 Modèles de programmation	29
1.7.1 Modèle à mémoire partagée ou distribuée	30
1.7.2 MPI : Message Passing Interface	31
1.7.3 Threads : pthread, OpenMP	32
1.7.4 CUDA / OpenCL	33
1.7.5 Tâches : Cilk, OpenMP-3	34
1.7.6 Les PGAS	34
1.7.7 Résumé	34
1.7.8 Problème du mélange	35
1.7.9 Le projet MPC	35
1.8 Question ouverte sur les DSL	36
1.9 Le système d'exploitation	37
1.10 Applications tests	38
1.11 Conclusion	38

2	Gestion de la mémoire	41
2.1	Introduction	41
2.2	Le système d'exploitation	41
2.2.1	Problématiques du multiprogramme/multiutilisateur	41
2.2.2	Adresses virtuelles, adresses physiques	42
2.2.3	Segmentation et pagination	43
2.2.4	Notion de processus et threads	45
2.2.5	Espace noyau et utilisateur	45
2.3	Interface avec les applications	46
2.3.1	Les appels systèmes	46
2.3.2	Fautes de pages	47
2.4	L'allocateur mémoire (malloc)	48
2.4.1	Interface	48
2.4.2	Fragmentation	48
2.4.3	Ramasse-miettes	50
2.4.4	Problématique d'implémentation	50
2.5	Accès à la mémoire	50
2.5.1	Les caches	51
2.5.2	MMU et TLB	53
2.5.3	Grosses pages	54
2.5.4	Accès mémoire non uniformes : NUMA	54
2.5.5	OS et support NUMA	55
2.6	Conclusion	56
II	Contribution	57
3	Interférences des mécanismes d'allocations	61
3.1	Pagination et associativité	61
3.2	Politiques de pagination	63
3.2.1	Pagination aléatoire : Linux	63
3.2.2	Coloration de pages	65
3.2.3	Grosses pages	66
3.2.4	Conclusion	67
3.3	Résultats expérimentaux	67
3.3.1	Protocole expérimental	68
3.3.2	Résultats des NAS séquentiels	68
3.3.3	Résultats des NAS parallèles	70
3.3.4	Résultats sur EulerMHD	71
3.4	Pagination et stratégie de malloc	71
3.4.1	Impact de l'implémentation de malloc	71
3.4.2	Problématique des paginations régulières	73
3.4.3	Associativité des caches partagés	74
3.4.4	Parallélisme des accès en lecture et écriture	75
3.4.5	Effet de la table des pages et des TLB	77
3.4.6	Impacte de la table des pages	79
3.5	Analyse générale et recommandations	80
3.5.1	Conséquence sur les politiques de pagination	81
3.5.2	Extension matérielle pour les grosses pages ?	83
3.5.3	Conséquence sur malloc	83
3.6	Outil d'analyse	84
3.6.1	Objectifs	84

3.6.2	Points techniques sur la méthode	84
3.6.3	Informations collectées	85
3.7	Application de règles de décalage	86
3.8	Conclusion	87
4	Mécanismes d'allocations parallèles et contraintes mémoires	89
4.1	Approche générale	89
4.2	Description du besoin	90
4.3	Aspects génériques des allocateurs	91
4.3.1	Gestion des blocs libres	91
4.3.2	Fusion et scission de blocs	92
4.3.3	Contraintes d'alignements	92
4.3.4	Placement des métadonnées	93
4.4	Allocateurs disponibles	93
4.4.1	Linux : dlmalloc et ptmalloc	94
4.4.2	Hoard	94
4.4.3	Jemalloc	95
4.4.4	TCmalloc	96
4.4.5	MAMA	97
4.5	Impact des allocateurs	97
4.6	Structure de l'allocateur	99
4.6.1	Organisation générale	99
4.6.2	Gestion des blocs	100
4.6.3	Discussion à propos des petits blocs	101
4.6.4	Suivi des macro-blocs alloués	101
4.6.5	Surcharge possible de la fonction de libération	102
4.6.6	Libérations distantes	103
4.6.7	Realloc	103
4.7	Réutilisation des gros segments	104
4.7.1	Méthode de réutilisation de TCMalloc	105
4.7.2	Méthode proposée	105
4.7.3	Recomposition de gros segments	106
4.7.4	Problème de surallocation	107
4.8	Remise à zéro pour calloc	107
4.9	Destruction du tas	108
4.10	Adaptation consommation versus performances	108
4.11	Aspect NUMA	109
4.12	NUMA et initialisation	110
4.13	Gestion de segments utilisateurs	110
4.14	Méthode d'implémentation	111
4.15	Évaluation	112
4.15.1	Micro-benchmarks	112
4.15.2	Résultats sur sysbench	114
4.15.3	Résultats sur la simulation numérique Hera	114
4.16	Bilan général	116
4.17	Discussion d'améliorations possibles	116
4.17.1	Niveaux topologiques	116
4.17.2	Politique de consommation dynamique	117
4.17.3	Surcharge de mmap/munmap?	118
4.17.4	Modification de Jemalloc?	118
4.17.5	API, sémantique NUMA?	119

TABLE DES MATIÈRES

5	Problématique de remise à zéro de la mémoire	121
5.1	Évaluation du problème de performance	121
5.2	Utilisation de grosses pages	123
5.3	Le problème de la remise à zéro	124
5.4	Solutions existantes	124
5.5	Proposition : réutilisation des pages	125
5.6	Extension de la sémantique mmap/munmap	126
5.7	Détails d'implémentation	127
5.7.1	Modification de Linux	127
5.7.2	Capture des zones sans réutilisation ?	128
5.7.3	Limite de consommation et réclamation	128
5.7.4	Intégration dans les allocateurs	129
5.8	Résultats expérimentaux	129
5.8.1	Micro-benchmark	129
5.8.2	Application HydroBench	130
5.8.3	Application Hera	131
5.9	Conclusion	132
6	Étude complémentaire sur le problème de consommation : KSM	135
6.1	Introduction	135
6.2	Mémoire partagée	136
6.3	Principe de KSM	137
6.3.1	L'idée maîtresse	137
6.3.2	Fonctionnement interne	137
6.3.3	Marquage des pages	138
6.3.4	Activation et configuration	138
6.4	Test sur Hera	138
6.4.1	Méthode de test	139
6.4.2	Résultats	139
6.5	Limitations de KSM	140
6.6	Bénéfices potentiels de KSM	142
6.7	Piste non évaluée : extension de la sémantique de mmap.	143
6.8	Conclusion	143
III	Conclusion et perspectives	145
7	Conclusion	147
8	Perspectives	151
	Bibliographie	155
	Annexes	159
A	Détail structurel des machines tests	163
B	Complémenté sur l'interférence des mécanismes d'allocations	165
B.1	SpecCPU 2006	165
B.2	Linpack	166
B.3	Alignements des tableaux de l'application MHD	168
B.4	Résumé des effets d'alignements	168

Préambule

Les sciences informatiques sont aujourd'hui devenues un outil essentiel dans le domaine de la recherche et de l'industrie. Ce domaine apporte en effet un moyen pratique de traiter de grandes quantités d'informations, que ce soit au travers de la simulation numérique comme outil prédictif ou pour l'analyse de données expérimentales. Au fil des années, les calculateurs mis en place pour résoudre ces problèmes sont devenus des objets extrêmement complexes offrant une puissance de calcul toujours croissante. C'est ainsi que l'on a aujourd'hui atteint l'échelle du *pétaflops* permettant idéalement d'effectuer 10^{15} opérations flottantes par secondes. Les nouvelles ambitions se tournent donc vers le pas suivant : *l'exaflop* avec 10^{18} opérations flottantes par secondes. L'évolution technologique a toutefois effectué un tournant au courant des années 2000. Ces années ont induit des changements en profondeur des architectures faisant apparaître de nouveaux problèmes qu'il nous faut aujourd'hui prendre en compte.

Ces problèmes viennent en partie de la consommation énergétique et des contraintes d'accès à l'information (mémoire, stockage) qui tendent à orienter les évolutions matérielles et logiciels. L'arrivée conjointe des architectures multicœurs permet de résoudre certains problèmes, mais en pose de nombreux autres en demandant notamment une prise en compte explicite de ce changement de paradigme par les programmeurs. Les échelles actuelles conduisent déjà à des calculateurs très hiérarchiques composés de quelques millions de cœurs. Il en résulte l'apparition de nouvelles problématiques notamment de passage à l'échelle nécessitant de revisiter certains points historiquement considérés comme "maîtrisés". Nous verrons que c'est le cas pour les problèmes de gestion de la mémoire au cœur de notre sujet de thèse.

Dans ce contexte en forte évolution, le choix d'un modèle de programmation exprimant efficacement le parallélisme devient une question sensible en pleine mutation. À l'heure actuelle, si la recherche fournit différentes approches, aucune n'a pour l'instant su se présenter comme LA solution retenue par tous. L'heure est donc au mélange de modèles du fait des choix des différentes bibliothèques que l'on peut être amené à utiliser. Ces changements doivent également cohabiter avec la présence de certains codes historiques (*legacy*) utilisant d'anciennes techniques. Le CEA et le laboratoire Exascale Computing Research ont donc investi dans le développement d'un support exécutif (MPC : *Multi-Processor Computing*) prenant en compte ces problématiques. Ce support a pour but de faire fonctionner les différents modèles de programmation sur une base unifiée permettant leur coopération. Dans ce contexte, nous avons observé des pertes de performances notables imputables aux mécanismes de gestion de la mémoire. Nous montrerons ainsi qu'il est possible dans certaines situations d'observer des écarts de performances pouvant dépasser les 50% du temps d'exécution total.

Les problématiques étudiées dans ce document seront donc abordées au travers d'un regard orienté *calcul haute performance* (HPC : *High Performance Computing*). Les points clés concerneront donc les aspects massivement multicœurs et de consommation mémoire. Le plan de la thèse sera donc le suivant.

La première partie de ce document dressera un état des lieux et introduira les concepts fon-

damentaux attachés au HPC, à son historique et aux méthodes de programmation ayant cours à l'heure actuelle. Y seront également introduits les points clés attachés aux problématiques de gestion mémoire sur les architectures modernes notamment au niveau du *système d'exploitation* (OS : *Operating System*).

La deuxième partie décrira les travaux propres à cette thèse. Nous y aborderons dans un premier temps le problème d'efficacité d'accès aux données. De ce point de vue, les architectures actuelles introduisent en effet des *mémoires locales* (*caches*) masquant les *latences* d'accès à la *mémoire centrale*. L'organisation actuelle du matériel est telle que l'OS et l'*allocateur mémoire* peuvent conduire à une perte d'efficacité de ces caches, pénalisant l'accès aux données. Dans ce sens, nous étudierons les effets d'interférences pouvant survenir entre les politiques des différents éléments en interaction (caches, OS, allocateur, application). Nous montrerons notamment qu'une analyse isolée de ces derniers n'est pas suffisante et qu'elle peut induire des effets néfastes. Le problème sera essentiellement étudié vis-à-vis des mécanismes de *pagination* en comparant les politiques de différents OS. Nous mettrons ainsi en évidence le couplage qui existe entre ces politiques et l'allocateur mémoire en espace utilisateur. Nous discuterons alors l'intérêt souvent négligé de la politique de pagination de Linux. L'utilisation de grosses pages sur les architectures modernes peut conduire à des effets similaires, impliquant une nécessité de prendre en compte ces dernières au sein même de l'allocateur en espace utilisateur.

On abordera dans un deuxième temps l'étude d'un *allocateur mémoire parallèle* spécifiquement conçu pour traiter les problèmes de performances d'allocation rencontrés au niveau des OS sur les nouvelles architectures. Ce travail prendra comme point focal la problématique des grosses allocations qui sont habituellement négligées dans la conception des allocateurs et redirigées directement vers l'OS. Or, ces derniers rencontrent des problèmes d'extensibilité sur les nouvelles architectures. Nous discuterons donc une méthodologie de réutilisation mémoire afin de limiter les échanges avec le système. À ce titre, nous verrons qu'il est pour l'instant nécessaire de rechercher un compromis entre consommation mémoire et performance, compromis que nous nous sommes efforcés de rendre configurable de manière dynamique. Ce travail sera complété par une prise en compte des architectures à mémoire non uniformes (NUMA : *Non Uniform Memory Access*) par l'organisation structurelle de l'allocateur. L'utilisation conjointe de ces techniques permet à l'heure actuelle d'obtenir des gains non négligeables (jusqu'à 50%) sur un code de simulation numérique.

La méthode de recyclage des gros segments discutée précédemment est introduite pour pallier le manque d'extensibilité de certains mécanismes du noyau *Linux*. Elle a toutefois l'inconvénient d'augmenter la consommation mémoire de l'application. Nous aborderons donc dans un troisième temps une proposition d'extension de la sémantique d'échange avec l'OS permettant des gains substantiels de performance démontrés par notre prototype. Pour ce faire, nous éliminerons l'obligation d'effectuer de coûteux effacements mémoires lors des *fautes de pages* engendrées par les allocations auprès de l'OS. Il est ainsi possible de favoriser les approches libérant plus régulièrement la mémoire vers l'OS. Comme la consommation mémoire devient un problème majeur, nous donnerons dans un quatrième temps les résultats obtenus lors d'une évaluation de la technique KSM (*Kernel Samepage Merging*) du noyau Linux. Cette dernière permet de fusionner les zones mémoire identiques et ainsi d'économiser la fraction de mémoire associée. Cette extension sera testée avec le maillage d'une simulation numérique afin d'évaluer sa capacité à réduire son empreinte mémoire.

La dernière partie clôturera ce document avec un bilan des travaux réalisés ainsi que des perspectives ouvertes par ces derniers.

Première partie

Contexte

Chapitre 1

Introduction au calcul haute performance

1.1 Introduction

Ce chapitre a pour but d'introduire les concepts fondamentaux ayant trait au domaine du *calcul haute performance (HPC)*, afin de disposer des éléments contextuels des travaux de cette thèse. La première section rappellera l'histoire de mise en place du HPC expliquant l'état actuel des connaissances dans le domaine. Elle sera suivie de détails sur les méthodes de programmation des calculateurs actuels, et d'un état de la recherche sur les méthodes envisagées pour les machines à venir. La mise en place de ces modèles sur les machines complexes actuelles requiert une construction au-dessus d'un système d'exploitation chargé de gérer et d'abstraire la ressource matérielle. La dernière partie donnera donc les points nécessaires à la compréhension de la construction de cette couche logicielle particulière, à laquelle sont associés nos travaux sur la gestion de la mémoire.

1.2 Historique du calcul haute performance

Historiquement, le *calcul haute performance* prend ses racines dans les centres de recherche avec des calculateurs visant à résoudre des systèmes d'équations complexes, notamment dans le domaine de la physique nucléaire. Les plus gros systèmes ouvrent alors la voie aux *supercalculateurs* modernes. On trouve dans un premier temps, de gros serveurs massivement vectoriels¹, conçus spécialement pour le calcul scientifique. On peut citer par exemple le CDC6600[Tho80] livré en 1964 ou le Cray-1[Rus78] de 1976 offrant 166 MFlops à 83 MHz doté de 8 Mo de mémoire. Ces calculateurs exploitent des unités vectorielles de $8 * 64$ bits et voient l'introduction de registres spécifiques dédiés aux instructions vectorielles pour compenser la lenteur d'accès à la mémoire centrale, qui, très tôt, s'avère être un facteur limitant les performances.

Très vite l'approche s'oriente vers une multiplication des composants en utilisant plusieurs processeurs partageant une même mémoire dite *partagée* telle que le Cray-X/MP[ABHS89] de 1982 avec 4 processeurs. Un an plus tard (1985), le Cray-2 fonctionne à 283 Mhz soit 1.7 GFlops pour 4 Go de mémoire, proche des ordres de grandeur de ce que l'on trouve dans nos téléphones portables actuels. L'évolution des machines uniques à mémoire partagée se poursuit ainsi jusqu'au milieu des années 90 en utilisant jusqu'à 2048 processeurs (Hitachi SR2201)[FYA⁺97].

1. Contrairement aux processeurs dits scalaires, les processeurs vectoriels permettent d'appliquer simultanément une opération sur un ensemble de valeurs contiguës en mémoire, dit vecteur.

Toutefois, l'accroissement de ces systèmes centralisés pose d'énormes problèmes de conception, notamment pour les accès à la mémoire qui ne parviennent plus à nourrir efficacement les unités de traitement. Les années 1990 orientent l'évolution vers des supercalculateurs de type grappes[BB99, EKTB99] (*clusters*) composés de multiples unités autonomes (*nœuds*) interconnectées par des réseaux spécialisés. Ce type d'approche est déjà exploité pour d'autres usages à moindre échelle tels que les systèmes VAX/VMS[KLS86] de 1986. Chaque unité est alors plus simple à concevoir et à exploiter. La complexité est donc repoussée sur l'utilisation de l'ensemble qui doit notamment prendre en charge les communications nécessaires entre les nœuds. L'évolution mène en 1992 au Paragon XP/S[EK93], doté de 2048 processeurs pour 32 nœuds. Ce contexte génère un besoin de nouveaux environnements de programmation multi-nœuds tels que PVM[FM90, Sun90]. Le travail conjoint de divers organismes et entreprises a également mené à l'interface MPI(*Message Passing Interface*)[MPI94] dont on reparlera plus en détail dans la suite et qui reste aujourd'hui la manière commune de programmer les calculateurs de type grappe.

Avec un coût croissant de développement, les processeurs spécialisés pour le calcul scientifique commencent à devenir un facteur limitant dans les années 2000. Certains se tournent alors vers les marchés grands publics pour y trouver des processeurs moins adaptés, mais moins chers, tels que ceux fabriqués pour les stations de travail plus classiques. Cette évolution pousse donc à un retour du calcul scalaire, l'utilisation de vecteurs intéressant moins l'industrie du PC. En terme de calcul scientifique, on observe dans ces années une divergence avec l'apparition des grilles de calcul[FK99]. Cette approche est axée sur des calculs intrinsèquement parallèles et indépendants, utilisés principalement pour les analyses de données expérimentales, notamment en physique des particules ou astrophysique. Pour ce type d'usage, chaque machine traite les données d'une mini-expérience sans dépendance avec les autres. On évite ainsi la contrainte des communications inter-nœuds qui, devenue inexistante, ne nécessite plus de placer l'ensemble des nœuds dans le même lieu ni de recourir à l'utilisation de réseaux rapides onéreux. La communauté scientifique dispose aujourd'hui de projets à l'échelle nationale, tels que Grid5000[CCD⁺05] ou mondiale avec la grille du CERN (WLCG)[Rob12]. De manière complémentaire, le domaine des supercalculateurs se concentre autour des simulations à grandes échelles et vise à répartir un calcul unique sur une part importante du supercalculateur. Ce besoin maintient la nécessité de faire communiquer efficacement un nombre croissant de composants.

Les années 2000 ont ainsi été marquées par la course au pétaflops avec une approche multi-nœuds, multi-processeurs et multi-cœurs entraînant une envolée du nombre de cœurs. La barre symbolique est franchie en 2008 par les Américains (DOE/IBM) avec Roadrunner [BDH⁺08] : 1.046 pétaflops pour 129 600 cœurs, dont une part d'accélérateurs de type Cell. Du côté français (CEA/Bull), c'est Tera 100 qui franchit le pas en 2009 : 1.05 pétaflops pour 138 368 cœurs. La barre symbolique du million de cœurs est atteinte fin 2012 par la machine Sequoia (DOE/IBM). Cette évolution continue aujourd'hui avec 3.1 millions de cœurs pour le supercalculateur chinois Thiane-2 (NUDT), offrant 33.8 pétaflops. La tendance se poursuit ; il nous faut donc appréhender le développement de logiciels devant à l'avenir fonctionner sur des millions de cœurs.

L'histoire du HPC est ponctuée de tests de performances qui permettent de comparer les capacités de calcul des machines dans le temps. À ce titre, le Linpack [Don88] sert de base au classement mondial des 500 calculateurs les plus puissants[Top10]. L'évolution de ce classement est donnée dans la figure 1.1 depuis 1993. On y observe bien une croissance régulière de la performance des plus gros systèmes, les projections menant à estimer l'arrivée de l'exaflops aux alentours de 2020.

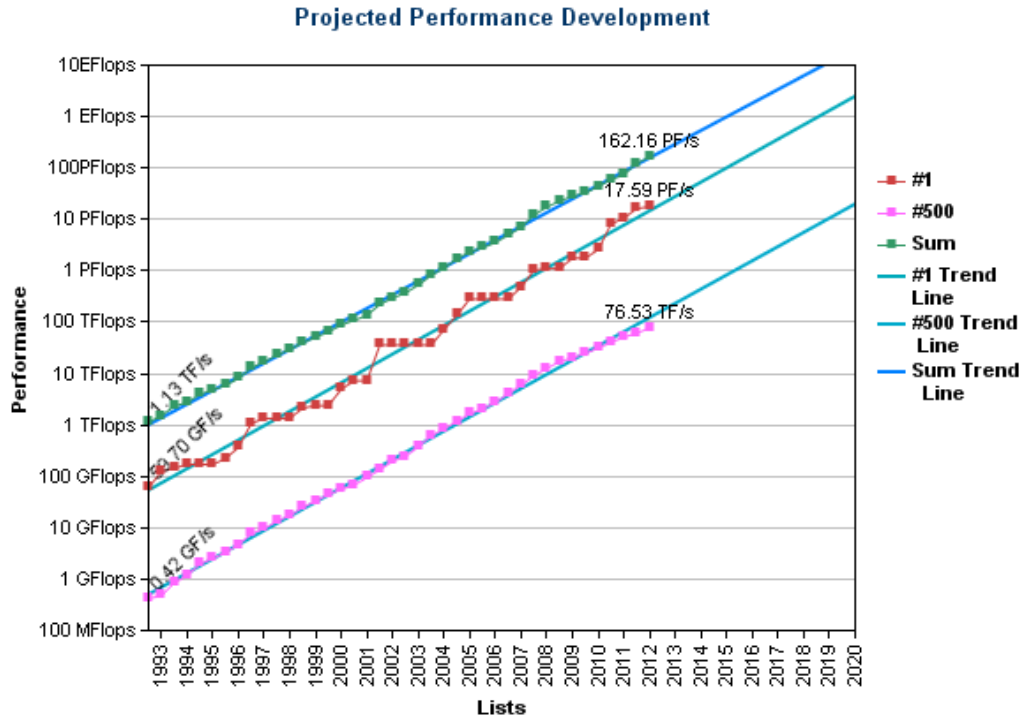


FIGURE 1.1 – Évolution des performances des supercalculateurs du Top500[Top10] en terme de calcul flottant (Flops) depuis les 1993.

1.3 Évolution actuelle des architectures

Dans cette section, nous allons nous centrer sur l'évolution survenue dans les processeurs au cours des années 2000. Cette évolution notamment marquée par l'arrêt de l'augmentation des fréquences s'est vue associée à l'arrivée du multicœur. Ceci change la manière d'exploiter le matériel et impacte les mécanismes de gestion de la mémoire en faisant apparaître les problématiques abordées dans cette thèse.

1.3.1 Un équilibre entre débit et latence

Avant d'entrer dans les détails de l'évolution des architectures, il semble important d'introduire les notions capitales de *débit* et *latence* pour les architectures informatiques. On définira tout d'abord le *débit* comme le nombre d'*actions* effectuées en un laps de temps déterminé. Remarquons que l'on ne s'intéresse pas ici au temps propre de ces actions, mais au temps de l'ensemble. À l'opposé, la *latence* s'intéresse au temps d'exécution d'une instruction, donc le temps que l'appelant doit attendre pour voir cette action se terminer. Les recherches d'optimisation des architectures, peuvent, d'une certaine manière, se décrire comme une recherche d'équilibre entre ces deux concepts.

Ces concepts s'appliquent à l'exécution des instructions par le processeur, aux transferts mémoires ou aux échanges réseau. Idéalement, le débit devrait être favorisé pour maximiser les performances en effectuant un maximum d'actions en un temps limité. Un programme est toutefois conçu comme une suite d'actions dont certaines dépendent du résultat de la précédente. Dans un tel contexte, la réduction de la latence est critique car elle contraint la capacité à exé-

cuter rapidement l'action suivante, donc le débit d'exécution. L'augmentation des fréquences de fonctionnement est par exemple un moyen mécanique de gagner sur les deux postes. Ceci en réduisant par définition la latence et en augmentant le débit. Nous verrons toutefois qu'elle a aujourd'hui atteint ses limites au niveau de l'exécution des instructions.

Les échanges entre les composants physiques peuvent toutefois introduire des contraintes sur ces débits et latences en fonction des capacités des liens utilisés pour les faire communiquer. Les améliorations basées sur la sémantique d'échange entre composants sont malheureusement souvent contraintes à favoriser l'un des deux points. Nous évoquerons indirectement ce problème dans les sections qui suivent au vu des architectures exploitées dans les calculateurs actuels.

1.3.2 L'arrivée du multicœur

À ses débuts, l'informatique a connu une croissance exponentielle des performances permise par la vérification des lois de Moore[Moo65, Moo75] et Dennard[DGR⁺74]. La première loi prédit un doublement du nombre de transistors imprimables sur une surface donnée tous les dix-huit mois. La seconde remarque que la réduction de taille des transistors permet de réduire leur consommation électrique. Ces deux lois ont une conséquence sur les performances. Tout d'abord, la réduction de taille des transistors permet d'augmenter leur fréquence de fonctionnement. À partir des quelques MHz des premiers calculateurs, on atteint le GHz peu avant l'an 2000, donc une croissance de 2 ordres de grandeur en 25 ans. D'autre part, la loi de Moore implique une augmentation du nombre de transistors utilisés dans les puces. Cela se traduit, dès les années 80[RF92], par une multiplication des unités de traitement à l'intérieur du processeur, offrant la possibilité d'extraire du parallélisme d'applications séquentielles (parallélisme d'instruction, ILP ou *Instruction Level Parallelism*) implémenté par deux approches complémentaires : utilisation de pipeline et d'architectures superscalaires.

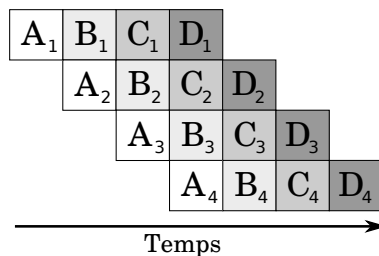


FIGURE 1.2 – Exemple de pipeline d'une instruction pouvant être découpée en 4 sous-étapes (A,B,C,D). Grâce à cette méthode, il est possible d'exécuter 4 de ces instructions simultanément sur le même pipeline sans avoir à dupliquer les unités de traitement associées. Le débit est ainsi augmenté, mais la latence reste la même voir augmente si le découpage entraîne une exécution plus lente.

L'introduction de la notion de *pipeline*[HP06] (Figure 1.2) permet de découper l'exécution d'une instruction en plusieurs étages, à la manière d'une chaîne d'assemblage d'usine. Plusieurs instructions peuvent donc être en cours de décodage et traitement. Cette technique ne réduit pas le temps d'exécution de l'instruction, mais permet d'augmenter le *débit* par l'exécution simultanée de plusieurs d'entre elles. Les processeurs actuels utilisent essentiellement des pipelines d'une profondeur allant de 10 à 20. D'autre part, certaines unités de traitement lentes peuvent être la source de ralentissement (accès mémoire, calcul flottant...). Disposer de plus de transistors permet donc de dupliquer certaines d'entre elles (*architectures superscalaires*). L'exploitation efficace de ces unités passe donc par un ordonnancement adapté des instructions pour maximiser l'utilisation des unités disponibles. Cet ordonnancement peut être pris en charge de

manière statique par le compilateur (architecture dite *in-order*) ou de manière dynamique par le processeur (architecture dites *out-of-order*). Le réordonnement dynamique des instructions offre également un moyen efficace de réduire l'impact des *latences* d'accès à la mémoire. Ceci en permettant au processeur d'exécuter dans l'intervalle les instructions disposant déjà de leurs données.

Or, au début des années 2000, ces démarches atteignent leurs limites. Les fréquences se stabilisent aux alentours de 3 GHz comme le montre la figure 1.3. Les raisons tiennent en partie à la fin d'application de la loi de Dennart[[DCK07](#)]. La fin d'application de cette loi entraîne des problèmes de consommation électrique et de dissipation thermique non compensés par l'évolution des finesses de gravure. De plus, l'évolution plus lente des technologies de stockage mémoire creuse le fossé entre la vitesse de traitement d'une donnée et son temps d'accès en mémoire. Les augmentations de fréquences sont donc rendues moins intéressantes, voir inefficaces. Ce problème est connu sous le nom de *mur de la mémoire* depuis sa formalisation en 1995 par Wulf[[WM95](#)]. D'un autre côté, l'exploitation du parallélisme d'instruction se heurte à une problématique de complexité croissante. Concernant l'allongement des pipelines, on observe ainsi chez Intel, un pic à 30 niveaux pour les Pentium 4, contre 14 pour les Core 2 Duo qui suivent. Avec des pipelines très longs, les opérations impliquant une purge de ces derniers deviennent très pénalisantes avec un temps important de re-remplissage de ce dernier. Le réordonnement des instructions, lui, se trouve limité par les dépendances entre ces dernières, limitant les gains au-delà d'un certain seuil.

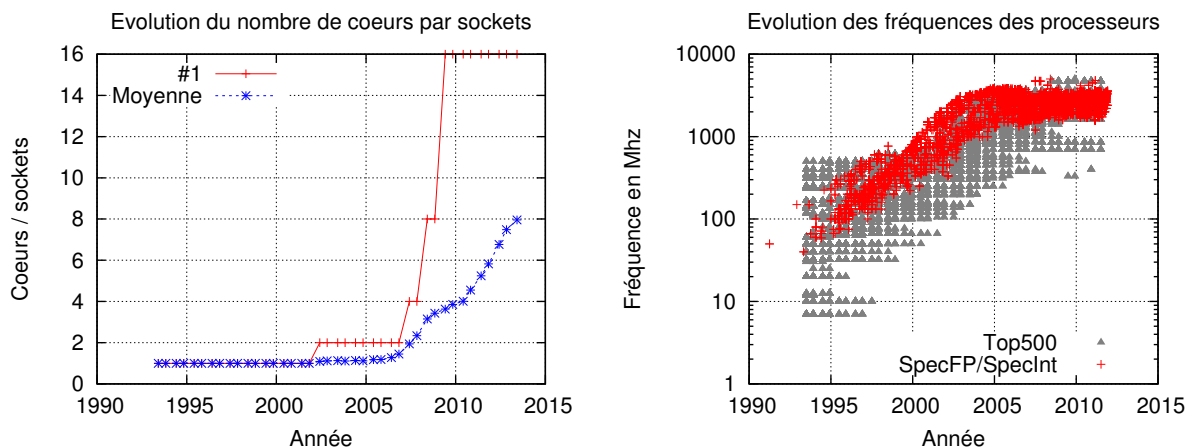


FIGURE 1.3 – Évolution du nombre de cœurs dans les processeurs et des fréquences à partir des données fournies par Top500[[Top10](#)] et Spec[[Hen06](#)].

La loi de Moore étant toujours vérifiée, le nombre croissant de transistors est exploité, depuis le milieu des années 2000, pour offrir un nombre croissant de cœurs, chacun étant capable d'exécuter son propre flot d'exécution. On retrouve donc à l'échelle du processeur, l'évolution qui a eu lieu à l'échelle des calculateurs dans les années 1980. Dès lors, l'augmentation de performance des nouveaux processeurs passe par une augmentation du nombre de cœurs, expliquant l'explosion de leur nombre dans les supercalculateurs modernes. Cette tendance est bien visible sur le graphique 1.3, mais risque d'être limitée par les phénomènes physiques survenant lorsque les pistes deviennent trop fines. À long terme, cette approche a donc, elle aussi, de fortes chances de devenir problématique.

1.3.3 Processeur versus GPGPU

L'informatique a été marquée par l'évolution des processeurs au travers de grandes sociétés telles que IBM, Intel, AMD, Cray et Fujitsu. Toutefois, l'essor du jeu vidéo a permis l'évolution d'un marché parallèle avec le développement des coprocesseurs graphiques, composants spécialisés dans le traitement d'images et vidéos. Avec le temps, ces composants massivement parallèles, principalement développés par les sociétés Nvidia et ATI/AMD, ont atteint des paramètres (coûts, performance et consommation énergétique) les rendant attractifs pour le domaine du HPC. Les constructeurs sont passés des GPU (*Graphical Processing Unit*) à des conceptions visant une utilisation plus générique dite GPGPU (*General Purpose Graphical Processing Unit*). Ces architectures offrent des possibilités pour le calcul scientifique de par leur conception plus vectorielle, dédiée à l'application de traitements sur de grands volumes de données. Les processeurs conventionnels sont habituellement conçus pour minimiser les latences d'accès aux données et donc, principalement optimisés pour des codes séquentiels. En comparaison, ces coprocesseurs sont spécialisés pour maximiser les débits et traitements parallèles au prix de la latence. Cela suppose des codes massivement parallèles et peu dépendants de la latence pour les exploiter.

Ces approches technologiques ont notamment l'intérêt d'apporter une démarche intéressante de modèles de programmation (CUDA[Buc07] OpenCL[SGS10] ...) qui forcent le développeur à exprimer ses traitements sous une forme parallèle. On trouve plusieurs des calculateurs du TOP500 équipés de ces coprocesseurs (notamment le premier du classement en novembre 2012 : Titan). Ces approches présentent toutefois une rupture technologique qui nécessite la réécriture des codes existants. Ceci complique le choix de migration vers cette technologie. Le besoin d'une présence de processeurs classiques pour les contrôler rend également la prise en main de ces environnements hybrides d'autant plus délicate.

Les langages tels que CUDA et OpenCL ont toutefois de fortes contraintes qui ne permettent pas d'exprimer aisément tous les algorithmes actuellement exploités sur processeurs conventionnels. La migration vers ces architectures doit donc pour l'instant être réfléchie en fonction de la fraction d'étapes adaptées à ces architectures. Remarquons que les fabricants de GPU tendent aujourd'hui à complexifier leurs composants pour supporter une sémantique plus vaste et donc, élargir la gamme de problèmes accessibles au travers de ces outils. À l'opposé, certains processeurs généralistes tendent vers une certaine simplification pour augmenter leurs performances parallèles et tendent, d'une certaine manière, à se rapprocher des GPGPU. C'est notamment le cas du nouveau coprocesseur Xeon-Phi, actuellement mis en œuvre par Intel et présent dans les supercalculateurs Tianhe-2 et Stampede lancés en 2012 et 2013. D'une manière générale, les algorithmes doivent donc être pensés ou repensés pour prendre en compte des architectures, qui de toute façon, deviennent massivement parallèles.

1.4 Les défis de l'Exaflops

Le *pétaflops* ayant été atteint en 2009, les avancées s'orientent désormais vers l'objectif de l'*exaflops*, estimé aux alentours de 2020 si la tendance actuelle se poursuit. Il convient toutefois de noter que cette évolution présente un certain nombre de défis importants qui nécessitent des efforts soutenus. Les gains nécessaires ne pourront en effet être apportés par une simple juxtaposition à l'infinie de composants bien maîtrisés. En effet, si l'on s'en réfère à certaines études [DBa11], l'exaflops pose de sérieuses questions en ce qui concerne les points listés ici.

1.4.1 Gestion de l'énergie

Jusqu'à récemment, l'augmentation de puissance des supercalculateurs s'est vue associée plus ou moins directement à une augmentation de la consommation électrique de ces derniers. En 2001, Earth Simulator, premier au TOP500 consommait 3.2 mégawatts pour fournir 35 téraflops. Avec les technologies actuelles, le calculateur Titan, premier au Top500 fin 2012, consomme 9 mégawatts pour produire 17.5 pétaflops. En 2013 Tianhe consomme 17 mégawatts pour ses 33 pétaflops. Dans ces conditions, la facture et les problèmes techniques, électriques deviennent un problème crucial.

L'exaflops ne pourra pas être atteint en poursuivant une augmentation de ces consommations, tant pour des raisons financières que pratiques. Il faut pour cela compter sur une multiplication par 1000 des performances pour une augmentation de consommation de quelques unités. Des limites de l'ordre de 20-30MW[TC12] sont considérées par certains comme un problème pratique et financier. Il importe donc d'utiliser des composants plus efficaces énergétiquement, voir des programmes eux-mêmes optimisés pour tenir compte des aspects énergétiques. Cela se traduit par l'apparition récente d'un nouveau classement, le Green500[SHcF06] prenant pour critère l'efficacité énergétique des calculateurs.

Sur le plan technique, la dissipation thermique des processeurs est fortement liée à leur fréquence de fonctionnement. Ce paramètre devient donc un levier important pour la maîtrise de la consommation. Pour le programmeur, cela signifie une réduction des fréquences de fonctionnement, donc une nécessité d'exploiter le parallélisme pour maintenir les performances des programmes existants. Les nouveaux Xeon-Phi d'Intel, utilisent par exemple une fréquence de 1.6 GHz, bien inférieure aux 3 GHz des Xeon utilisés sur les calculateurs actuels du CEA : Tera100 et Curie. En compensation, ce coprocesseur offre 60 cœurs capables d'exécuter jusqu'à 240 threads contre 8 cœurs et 16 threads (si *hyperthreading* activé) pour les processeurs de Tera100 et Curie. Certains voient plus loin et avancent qu'il sera nécessaire dans un futur proche de désactiver les transistors inutilisés parfois appelés *dark silicon*[EBSA⁺12].

En ce qui concerne la gestion de la mémoire, on trouve actuellement des travaux visant à permettre d'éteindre les portions non utilisées de cette dernière[Kje10, Bha13, LPMZ11]. Suivant l'évolution du matériel grand public, ces technologies sont actuellement très étudiées dans le cadre des applications mobiles. Ces rapprochements offrent des opportunités d'échanges constructifs avec ce domaine, confronté à des problèmes en partie similaires à ceux du HPC, notamment sur le plan de la consommation électrique. Remarquons que ce type de travaux a un impact sur les stratégies de gestion avec la mise en place de technique de compactage mémoire pour déplacer tous les blocs présents dans les zones mémoires candidate à l'extinction.

Les marchés actuels tendent également à favoriser la production d'appareils mobiles aux dépens du PC classique. A moyen/long terme, il est donc probable que les technologies exploitées en HPC devront s'adapter à cette évolution du marché en réutilisant ses technologies fortement rentabilisées.

1.4.2 La mémoire

La mémoire est par construction un composant *extensif* puisque l'augmentation de capacité est directement reliée au nombre de transistors. Les technologies dites DRAM (*Dynamic Random Access Memory*) actuellement employées nécessitent en effet un rafraîchissement[Dre07] régulier de leur contenu et donc, un coût énergétique proportionnel à la capacité de stockage. Ce facteur devient donc un paramètre d'optimisation de la consommation électrique des calculateurs tendant à limiter l'augmentation de l'espace mémoire des supercalculateurs. De plus, ces

mêmes technologies n'évoluent pas à la même cadence que les processeurs. En l'état, l'augmentation du nombre de cœurs va donc avoir tendance à augmenter plus rapidement que la quantité de mémoire. Ceci, pour des raisons tant énergétiques, que de contrainte de volumes physiques.

Pour le programmeur, cela signifie moins de mémoire par cœur, donc un besoin d'une attention plus grande sur ces aspects qui s'étaient estompés avec le temps. D'autre part, cela exclut la possibilité de reposer sur une extensibilité faible² stricte pour tirer parti du parallélisme. Le sous-problème traité par cœur doit en effet se réduire (ou être réorganisé) en proportion de la mémoire disponible. Les bibliothèques et l'allocateur mémoire lui-même doivent donc être construits de manière à pouvoir limiter leur empreinte mémoire. Il en résulte un besoin de réévaluer les compromis réalisés dans leur conception. Nous verrons que cela impacte également les modèles de programmation envisageables pour ces architectures. Remarquons que la mémoire totale continuera quant à elle à augmenter, même si ce n'est pas en proportion du nombre de cœurs. Or, le coût de gestion d'un octet mémoire par l'OS est lui, plus ou moins constant (à fréquence fixe). Il importe donc de remarquer que la problématique liée aux méthodes de gestion de cette ressource est donc vouée à prendre une importance croissante. Ce point peut donc devenir un problème majeur pour exploiter le gain de capacité de traitement des processeurs s'il n'exploite pas les gains de parallélisme.

Le problème sera discuté plus en détail dans la suite, mais le nombre croissant de cœurs à alimenter en données oblige les architectures à devenir très hiérarchisées pour distribuer les contentions d'accès à la mémoire. Ce facteur structurel complique le développement des programmes en introduisant des dépendances sur des paramètres architecturaux en pleine évolution.

1.4.3 Nombre de cœurs

L'évolution actuelle nous mène déjà à disposer de calculateurs composés de près d'un million de cœurs. La programmation de tels calculateurs représente un défi pour partie liée à la loi d'Amdahl[Amd67]. Cette dernière rappelle en effet qu'un programme contenant une fraction s de code séquentiel ne pourra pas obtenir une accélération supérieure à $S_{max} = \frac{1}{s}$. Avec une fraction de code séquentiel de 1%, il est donc impossible d'obtenir une accélération supérieure à 100, quel que soit le nombre de processeurs utilisé. De plus, cette loi est optimiste, car elle ne considère pas le coût croissant des communications dépendantes de l'augmentation du nombre d'unités de traitement. On comprend donc qu'avec 1 million de cœurs, les parties séquentielles deviennent très rapidement un problème.

Ce point impacte l'ensemble des codes utilisés, du système d'exploitation aux codes applicatifs eux-mêmes. Dans ce contexte, il est nécessaire de revisiter le fonctionnement de certains aspects systèmes. Nous nous intéresserons donc à la pile de gestion de la mémoire afin de prendre en compte ces nouvelles contraintes.

1.4.4 Pannes

Une telle augmentation du nombre de cœurs rend difficile le débogage des applications, du fait de l'interaction d'un nombre trop important de tâches ne pouvant plus être naïvement énumérées de manière lisible par le programmeur. D'autre part, l'utilisateur fait face à un problème statistique. Considérant une chance de panne fixe pour un composant, il est clair qu'une multiplication de ces derniers augmente les chances de pannes de l'un d'entre eux. Ceci vaut pour le

2. L'extensibilité faible est une méthode qui consiste à profiter du supplément de parallélisme pour traiter un problème plus grand (en proportion du nombre d'unités de traitement supplémentaire).

matériel comme pour les logiciels. Les défauts ont donc tendance à produire de plus en plus rapidement un plantage des applications. Il en résulte un besoin de fiabiliser ces composants (tant logiciels que matériels). Le cas extrême consiste à rendre les programmes tolérant aux pannes. L'état actuel des connaissances implique toutefois que la panne d'un seul composant (logiciel ou matériel) a de grandes chances d'entraîner un plantage de l'ensemble du programme.

1.4.5 Entrées/sorties

La problématique liée à l'évolution lente des performances des mémoires vaut également pour les supports de stockage à long terme (disques durs, bandes...) qui restent très en deçà des capacités de transfert nécessaires pour alimenter les processeurs actuels. L'utilisation de ces supports doit donc limiter l'impact sur les performances. On notera que la tolérance aux pannes est actuellement gérée dans les programmes par le biais des méthodes de *reprise sur écriture*. Ces dernières procèdent par écriture régulière de l'ensemble de l'état d'un programme sur un système de fichier partagé, en général l'ensemble ou une partie de sa mémoire. La reprise peut alors être réalisée en rechargeant cet état en mémoire. Cette approche a toutefois le défaut de générer un flux important de données, tant en débit qu'en volume.

1.4.6 Résumé

D'une manière générale, la complexité croissante des architectures dont on dispose est liée en partie à l'augmentation du nombre de composants, mais aussi à l'application cumulée des diverses approches historiques. Les architectures récentes exploitent en effets beaucoup de technologies maîtrisées par le passé (programmation vectorielle, mémoires partagées, mémoires distribuées...). Les nouvelles difficultés proviennent donc de leur utilisation simultanée à grande échelle. On remarquera également les contraintes importantes liées à la gestion des données (stockage et transfert) qui tendent à orienter les développements actuels. Dans ce cadre, cette thèse s'intéresse aux méthodes de gestion de la ressource mémoire tant que niveau du système d'exploitation que des bibliothèques système en espace utilisateur. Nous reviendrons plus en détail sur les points structurels associés à cette problématique dans le chapitre suivant (2).

1.5 Environnement de calcul de la thèse



FIGURE 1.4 – Photo du calculateur Tera 100 du CEA.

Cette thèse a été réalisée au centre CEA de Bruyères-le-châtel, site regroupant l'essentiel des moyens de calcul du CEA, notamment par le biais du centre lui-même (pour les calculs clas-

sifiés), du CCRT (Centre de Calcul Recherche et Technologie) et la création récente du TGCC (Très Grand Centre de Calcul) associée à l'initiative européenne PRACE (Partnership for Advanced Computing in Europe). L'environnement de travail de cette thèse comprend donc deux supercalculateurs classés au TOP500 développés par le constructeur Bull. Tera 100, 6e au rang mondial en 2010 composé de 4370 nœuds soit 138368 cœurs et offrant une puissance sur Linpack de 1.05 pétaflops. Curie, 9e en 2012 composé de 5040 nœuds pour un total de 77184 cœurs offre quant à lui, une puissance de 1.36 pétaflops pour les projets européens de l'initiative PRACE. Ces deux supercalculateurs exploitent une architecture similaire basée sur un assemblage de processeurs Intel x86_64 groupés en nœuds NUMA et interconnectés en infiniband³. Au-delà de ces supercalculateurs, le travail a également été en partie réalisé sur un petit cluster (Cassard) composé de 3 nœuds et une station autonome biprocesseur Nehalem. Un résumé des caractéristiques de ces nœuds est donné dans la table 1.1 et une description détaillée peut être trouvée en annexe A.

D'autres travaux plus annexes ont également pu avoir lieu sur certains calculateurs prototypes du CEA offrant de l'ordre de 900 cœurs avec des structures proches des architectures listées précédemment. Sont également à ajouter les nœuds prototypes mis à disposition par le projet PerfCloud[Per12]. Ces derniers fournissent des coprocesseurs Intel Xeon-Phi sur lesquels nous avons pu réaliser quelques évaluations de performance du noyau Linux.

Machine	Processeurs	Famille	Cœurs	NUMA	Mémoire
Tera100 - noeuds fins	4	Intel Nehalem EX	32	4	64 Go
Tera100 - noeuds larges	16	Intel Nehalem EX	128	4*4	128 Go
Curie - noeuds fins	2	Intel Sandy-Bridge EX	16	2	64 Go
Curie - noeuds larges	16	Intel Sandy-Bridge EX	128	4*4	128 Go
Cassard	2	Intel Westmere	12	2	48 Go
Station autonome	2	Intel Nehalem	8	2	32 Go

TABLE 1.1 – Structure synthétique des nœuds de calcul utilisés lors des développements de cette thèse. Sont donnés, par nœud : le nombre de processus, de cœurs total, la structure NUMA et la mémoire disponible. Tera100 et Curie sont respectivement des calculateurs pétaflopiques classés au Top500 avec respectivement 1.05 pétaflops (138368 cœurs) et 1.36 pétaflops (77184 cœurs).

Tous ces systèmes sont opérés sous système d'exploitation Linux (Redhat 6, noyau 2.6.32). Notre étude prendra donc Linux comme OS principal bien que considérant les approches Unix d'une manière plus générale. En ce qui concerne les parties matérielles, nous nous intéresserons principalement aux architectures de type x86_64 fournies par Intel sous la forme de nœuds NUMA agrégeant jusqu'à 16 processeurs (nœuds de 128 cœurs) grâce à la technologie d'interconnexion BCS (Bull Coherence Switch) développée par Bull.

1.6 Modèles d'exécution

Au vu de l'histoire de l'informatique, plusieurs auteurs ont défini une taxonomie des architectures utilisées. Celle de Flynn[Fly66] décrit quatre catégories majeures en considérant l'unicité ou multiplicité des instructions et données :

SISD (Single Instruction Single Data) : C'est le modèle de base correspondant à l'architecture initialement définie par Von Neumann, une instruction est appliquée sur une donnée unique.

3. Technologie de réseau rapide utilisé dans les centres de calcul évitant les surcoûts liés au protocole Ethernet, notamment en terme de latence.

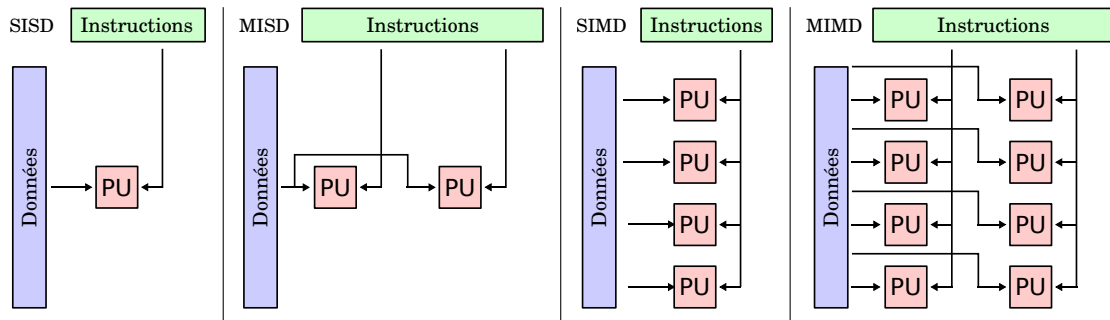


FIGURE 1.5 – Illustration des modèles d'exécution établis par la classification de Flynn (schémas extraits de Wikipedia[[wik](#)]).

SIMD (Single Instruction Multiple Data) : L'une des extensions du modèle très présente en HPC, elle correspond à la notion de calcul vectoriel où une instruction unique est appliquée à un ensemble contigu de données. Ce modèle permet de réduire le coût de décodage des instructions à répéter sur des ensembles de données et d'optimiser leurs transferts mémoires. Il est aujourd'hui concrétisé à grande échelle dans l'architecture des GPU et sous forme d'extension avec les normes SSE, AVX⁴ et équivalents de la famille x86.

MISD (Multiple Instruction Single Data) : Ce modèle n'a pas d'implémentation pratique reconnue même si certains auteurs[DL95] considèrent que les pipelines peuvent être considérés comme tel.

MIMD (Multiple Instruction Multiple Data) : Plusieurs instructions sont traitées simultanément, chacune, sur des données différentes. C'est le modèle principal des architectures modernes. On notera qu'il peut s'appliquer en utilisant des mémoires partagées (SPMD : Single Program Multiple Data) ou distribuées (MPMD : Multiple Program Multiple Data). Comme discuté, les calculateurs actuels exploitent un mode hybride en mixant ces deux approches sous la forme de grappes de nœuds multicœurs.

1.7 Modèles de programmation

En informatique, il est très rare que le développeur interagisse directement avec les composants matériels devenus complexes. L'exploitation des ressources matérielles passe donc par la mise en place de couches d'abstractions permettant une expression du besoin compréhensible par la machine. Cette approche se concrétise en informatique sous la forme du trio *langage*, *compilateur/interpréteur* et *support exécutif* représentés sur la figure 1.6. Ces trois aspects couvrant respectivement : une manière formalisée d'exprimer la résolution d'un problème opérationnel, la traduction de ce langage et la gestion des aspects dynamiques éventuels en cours d'exécution.

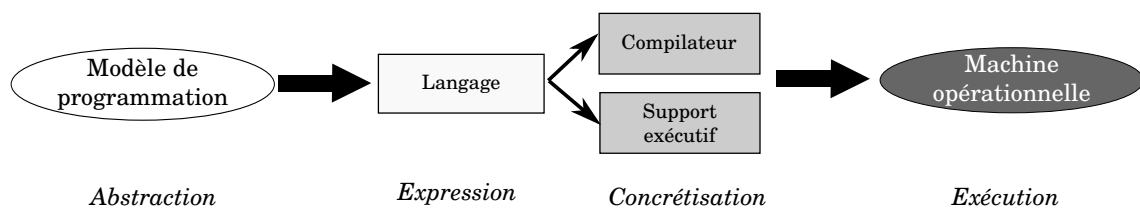


FIGURE 1.6 – Illustration de passage du modèle abstrait de programmation à une exécution sur machine réelle en passant par une expression formalisée du besoin.

4. SSE et AVX sont des normes d'extensions permettant le calcul vectoriel sur les processeurs Intel et AMD pourtant centrés sur un fonctionnement scalaire.

Concrètement, la mise au point de nouveaux langages pose de réels problèmes pratiques. Dans un domaine s'intéressant à la performance, il est en effet important de noter que la qualité et robustesse des compilateurs est un facteur clé. De ce fait, leur mise au point peut prendre des années. De plus, un nouveau langage nécessite un ensemble de fonctionnalités pré-fournies afin de pouvoir exprimer rapidement des problèmes complexes. De la même manière, un ensemble d'outils est nécessaire pour aider le développeur dans son travail, outils qui sont pour partie dépendants des langages utilisés (débugueur, profileur...). Il y a donc une nécessité de réécrire l'ensemble de l'existant, ou d'offrir un mécanisme permettant d'y faire appel.

Il en résulte une forme de compromis impliquant une tendance à limiter l'évolution brutale des langages. Actuellement, les modèles de programmation des architectures parallèles tendent donc à se concrétiser sous la forme de *bibliothèques de fonctions* ou d'*extensions* de langages existants. Dans le domaine du HPC, on trouve principalement C, C++ et Fortran. Les modèles de programmation parallèle tendent donc à se concrétiser autour de ces derniers. Une autre approche consiste à introduire des *directives de compilation*, des mots clés ajoutés au langage permettant d'activer, si supporté, des extensions du compilateur pour que ce dernier modifie le code source. En C, C++ et Fortran, il s'agit de la notation portée par le mot clé *#pragma* qui est ignoré si le compilateur ne supporte pas l'extension associée. Ces formalismes doivent abstraire le fonctionnement détaillé du matériel, ils reposent donc sur un modèle plus abstrait de la machine. Nous allons donc voir les deux modèles mémoires dominants exploités en HPC.

1.7.1 Modèle à mémoire partagée ou distribuée

Si l'on reprend l'historique de construction des supercalculateurs en considérant les architectures parallèles, on remarque la présence de deux modèles génériques dominants, fonction du mode d'accès à la mémoire (Figure 1.7). On trouvera d'un côté, les architectures dites à *mémoire partagée*, dès lors que les différents *flux d'exécutions* ont accès à l'ensemble des données en mémoire de manière unifiée et transparente. La limite de cette conception est principalement liée aux effets de saturation survenant sur le *bus* reliant les différentes unités de traitement à la mémoire. De plus, les architectures, dites *cohérentes*, doivent assurer de manière automatique, qu'une modification d'une donnée par un processeur soit immédiatement visible par les autres. Or, ces mécanismes se complexifient avec l'augmentation du nombre de processeurs à interconnecter.

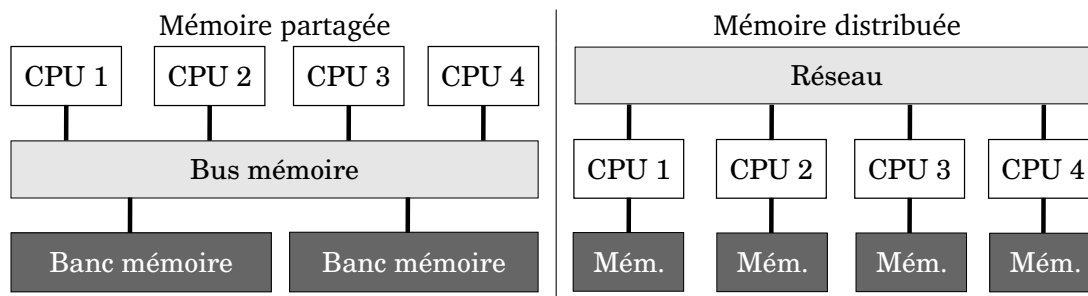


FIGURE 1.7 – Illustration de passage du modèle abstrait de programmation à une exécution sur machine réelle en passant par une expression formalisée du besoin.

Le modèle complémentaire dit à *mémoire distribuée* tente donc de régler ces problèmes en distribuant la mémoire entre les unités de traitement. La contention sur le *bus* est ainsi éliminée. Les échanges entre les unités sont toutefois transformés, d'un mode implicite vers un mode explicite, au travers de la gestion d'un réseau déléguée à l'utilisateur ou aux supports exécutifs. Ces architectures sont habituellement exploitées sur la base d'échanges de messages : modèle

dit MP (*Message Passing*) dont nous verrons la concrétisation dans la prochaine section.

Ce point a déjà été abordé précédemment, mais rappelons que les calculateurs actuels mixent ces deux approches en utilisant des grappes de nœuds multicœurs. Les nœuds sont donc à programmer sur la base de modèles à mémoire distribuée par échange de messages. Les cœurs internes aux nœuds sont idéalement à programmer sur la base de modèle à mémoire partagée. Remarquons toutefois que les modèles à mémoire distribuée représentent une vue plus abstraite, pouvant tout à fait être exécutée sur des machines à mémoire partagée.

1.7.2 MPI : Message Passing Interface

MPI est une interface de programmation (API) internationalement reconnue, s'intéressant au fonctionnement cohérent d'un ensemble de programmes sur les architectures à *mémoires distribuées*. Elle fournit une concrétisation du modèle *Message Passing*. MPI définit donc une sémantique pour mettre en relation différents programmes et la manière dont ils vont s'échanger les informations. Cette norme offre essentiellement un ensemble de fonctions permettant : d'identifier les tâches distribuées par un numéro unique, d'établir des connexions entre deux tâches (*point à point*), d'effectuer des synchronisations, échanges ou réductions sur l'ensemble ou des sous-ensembles de tâches (*collectives*). MPI offre une méthode de programmation pour architectures distribuées, mais cette approche englobe également les architectures à mémoires distribuées, il est donc possible d'utiliser MPI sur ces architectures. D'un point de vue plus pratique, MPI se trouve actuellement concrétisée sous la forme de bibliothèques de fonctions utilisables en C, C++ et Fortran. Il existe différentes implémentations de cette convention, on pourra notamment citer OpenMPI, MPICH, IntelMPI, MVAPich ou encore MPC dont on parlera plus loin.

Dans les systèmes d'exploitation modernes, un *processus* est une instance d'une application composé : d'un code programme à exécuter et d'un ensemble de ressources en cours d'utilisation (mémoire, fichiers, connexions...). Un processus contient au minimum un flux d'exécution. Chaque processus est isolé et n'utilise que ses propres ressources, sauf demande explicite d'échanges au travers du système d'exploitation. Cette notion de processus correspond donc bien à une programmation en mémoire distribuée. En dehors du projet MPC que nous discuterons plus loin, les implémentations MPI traditionnelles se construisent en établissant une correspondance unique entre tâche et processus.

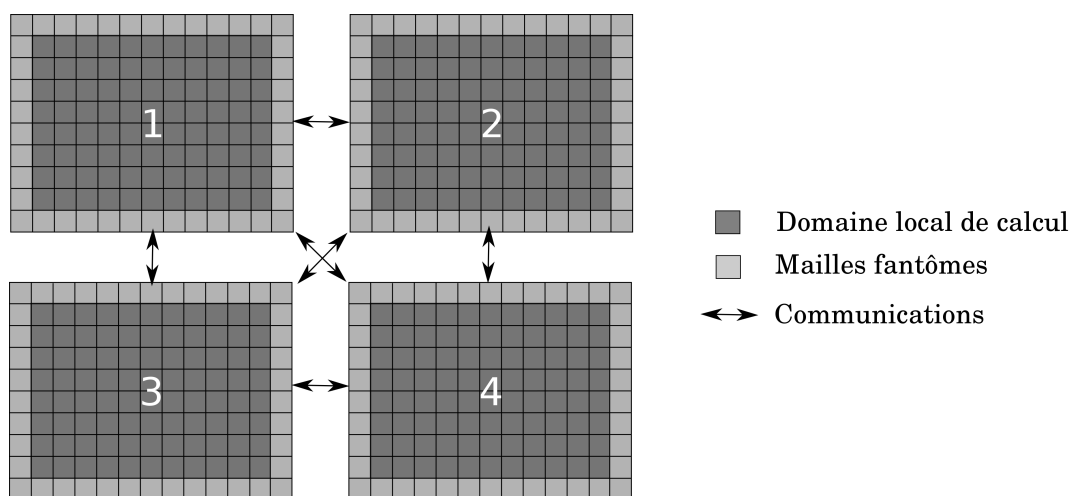


FIGURE 1.8 – Exemple de décomposition de domaine MPI en considérant un maillage 2D décomposé sur 4 tâches en considérant une épaisseur de maille fantôme de un élément.

Dans le cadre de la simulation numérique, le modèle MPI est souvent utilisé pour faire de la décomposition de domaine. Chaque tâche MPI a la charge d'une portion du maillage utilisé pour le calcul. Les mailles en bordure des domaines voisins doivent être répliquées (*mailles fantômes*), car nécessaires aux calculs locaux. MPI est donc utilisé pour synchroniser leurs valeurs par échanges de messages, comme cela est visible sur la figure 1.8. Cette approche a pour contrepartie une surconsommation mémoire induite par la présence des mailles fantômes qui, dans le cas de simulation 3D, peut devenir non négligeable. Avec le nombre croissant de cœurs, le modèle MPI, utilisé seul et implémenté à base de processus, pose donc trois problèmes qui deviennent des facteurs limitant à grande échelle :

Duplication de données : sur une architecture à mémoire partagée, les implémentations MPI à base de processus imposent la duplication de certaines données (mailles fantômes, tables de constantes physiques...), qui, sans cela, ne seraient pas nécessaires sur ces architectures. Cette approche, devient donc limitante dans un contexte où la quantité de mémoire par cœur tend à la stabilisation, ou réduction.

Communications inter-nœuds : MPI doit établir et maintenir des connexions entre les tâches qui échangent des données. Cela implique des connexions au niveau du système d'exploitation et la mise en place de tampons mémoires pour les communications. Avec un nombre croissant de cœurs, le nombre de tampons et de connexions établies par certains schémas de communications va de manière croissante sur chaque nœud, finissant par poser problème pour des raisons de consommation mémoire, mais aussi du fait de limites sur le nombre de connexions autorisées par l'OS et le support matériel des cartes.

Communications intra-nœud : Bien que le modèle MPI fonctionne parfaitement sur les architectures à mémoire partagée, il importe de noter que le fonctionnement basé sur l'échange de messages est sous-efficace par rapport à ce que peuvent offrir ces architectures. Ces échanges nécessitent en effet des recopies de données, qui, sinon, ne seraient pas nécessaires. Il existe bien sûr des techniques d'optimisations allant dans ce sens[PCJ09, BMG07], mais le problème de fond demeure.

1.7.3 Threads : pthread, OpenMP

En mémoire partagée, les demandes explicites d'échanges peuvent s'avérer sous-efficaces, on préfère donc exploiter la notion de *processus léger* (*threads*) plus efficace pour ces architectures. Dans ce cas, un processus est découpé en plusieurs flux d'exécutions (les *threads*) partageant les mêmes ressources (mémoire, fichiers...). De cette manière, chaque flux d'exécution a accès en permanence et, de manière immédiate, à l'ensemble des données manipulées par le processus. Le modèle se calque ainsi directement sur l'aspect mémoire partagé de l'architecture sous-jacente. Cette approche pose toutefois le problème de la *synchronisation*, car deux flux d'exécutions ne doivent pas modifier simultanément un même élément ou ensemble d'éléments sous peine de le rendre *incohérent*. Il en ressort un besoin de synchronisation explicite permettant d'assurer une exécution par un unique thread des sections dites *critiques*, qui accèdent à des données pouvant être manipulées par plusieurs threads. On trouve essentiellement quatre approches basiques pour assurer la propriété d'accès synchronisé aux données, classées ici par granularité et coût croissant.

Les instructions atomiques assurent (au niveau assembleur) l'exécution de leur tâche de manière dite *atomique*. Ceci sous-entend que vue depuis les autres unités de traitement, l'action ne laisse pas transparaître d'étapes intermédiaires en ce qui concerne les données manipulées. Autrement dit, tout se passe comme si l'instruction effectuait son traitement (chargement d'une donnée, manipulation, écriture de la donnée) en un cycle. Disponibles au niveau assembleur, ces instructions représentent un problème de portabilité et se limitent à des opérations très simples.

Les **spinlocks** sont construits à l'aide d'instructions atomiques et permettent de protéger des *sections critiques* de code constituées de plusieurs instructions. Ils fournissent un mécanisme d'attente active, une forte réactivité, mais consomment le temps de calcul au lieu de le partager. Ils sont disponibles à partir d'interfaces de programmation (API) standard telles que *pthread*.

Les **mutex** sont des verrous logiciels offerts par le système d'exploitation ou certaines bibliothèques parallèles. Ces derniers permettent de rendre la main à une autre tâche lorsqu'ils sont bloqués, en permettant un meilleur partage de la ressource de calcul. La contrepartie est un coût supérieur et une réactivité moins importante.

Sur les OS de la famille Unix, l'utilisation standardisée des threads repose sur l'interface *pthread* définie par la norme POSIX⁵. Cette interface est toutefois destinée à une manipulation très orientée système, mais, peu adaptée à la parallélisation de simulation numérique, rendant son utilisation fastidieuse. À l'heure actuelle, on préfère donc utiliser des directives de compilation permettant d'exprimer le parallélisme de manière plus abstraite et déléguer une partie du travail au compilateur. On trouvera notamment les normes de *directives* OpenMP et HMPP comme extension des langages existants C, C++ et Fortran. Ces directives offrent l'avantage d'introduire simplement la création de threads dans un programme existant et la possibilité de maintenir le fonctionnement originel du programme si ces dernières sont ignorées. Un exemple de parallélisation de boucle en OpenMP est donné dans le code 1.1. Ces approches ont l'intérêt de permettre une transition en s'appliquant sur des codes existants avec un nombre réduit de changements. La contrepartie est une sémantique restreinte par la nécessité de maintenir un code fonctionnel lorsque leur interprétation est désactivée.

Code 1.1– Exemple de parallélisation de boucle avec OpenMP

```
1 #pragma omp parallel for
2 for (int i = 0 ; i < SIZE ; i++)
3     //action
```

1.7.4 CUDA / OpenCL

Dans la section 1.3.3 nous avons discuté de l'arrivée récente des architectures dérivées des GPU. Ces architectures mises au point par Nvidia et ATI apportent leur propre modèle de programmation, avec notamment CUDA, une extension propriétaire du C et C++ offert par la société Nvidia pour exploiter ses cartes et le fruit d'un effort de normalisation : OpenCL. Dans les deux cas, ces langages cherchent à forcer une expression du parallélisme comme brique de base. Certains fondements du langage sont donc modifiés avec notamment l'introduction de *noyaux* (*kernels*) de calcul traitant une donnée "unique", mais devant pouvoir être exécutés en parallèle sur chacune des données d'un ensemble correspondant aux contraintes de la puce. Il en résulte des gains substantiels dans l'expressivité du parallélisme, le développeur étant contraint à écrire un code adéquat. Ceci entraîne toutefois une difficulté pour exprimer efficacement certains problèmes non adaptés à ce type d'expression restrictive.

En pratique, l'utilisation de ces technologies prend pour l'instant la forme d'un duo processeur généraliste associé à un coprocesseur spécialisé nécessitant une programmation hybride de façon à faire fonctionner chaque portion de code sur l'architecture la plus adaptée. Ce type d'approche ajoute toutefois la difficulté de gérer explicitement le transfert des données entre les différents modules de calcul. On trouve pour cela de nombreux travaux de recherche autour d'environnement d'exécution prenant en charge le besoin d'équilibrage hybride : StarPU[AN09],

5. **POSIX** : *Portable Operating System Interface* est une norme visant à standardiser l'interface avec les systèmes d'exploitation. Elle est reprise par la plupart des Unix.

StarSS[ABI⁺09], XKaapi[GFLMR13] ou les études similaires conduites dans le cadre du projet MPC[JYV12]. Des langages à base de directives sont en cours de mise au point pour générer des codes GPU à partir de codes existant sur une base de *directives* de compilations : HMPP et OpenACC.

1.7.5 Tâches : Cilk, OpenMP-3

Les langages de programmation précédents offrent essentiellement un parallélisme qui vise à exécuter en parallèle des flux d'instructions. L'ordonnancement est alors entièrement pris en charge par le développeur. Ce type d'ordonnancement prédéfini présente de réels problèmes d'adaptabilité sur des architectures en pleine évolution et pour traiter des problèmes non nécessairement équilibrés en terme de charge de calcul. Pour cela, certaines approches tendent à modifier l'expression du travail à effectuer en le déclarant sous la forme de tâches. Une tâche étant un ensemble cohérent et restreint d'instructions visant à résoudre un sous-problème. Contrairement à la notion de simples fonctions s'enchaînant, les tâches ont la propriété de pouvoir être réordonnées à l'exécution sous réserve de vérifier les contraintes de dépendances. On trouve ainsi des extensions du langage C, C++ et Fortran permettant de générer des tâches à partir de fonctions ou de sous-blocs existants. On dispose par exemple de Cilk et des extensions ajoutées à la norme OpenMP 3.0.

1.7.6 Les PGAS

Les trois modèles de programmation précédents permettent d'exploiter des architectures à mémoire partagée à partir d'extensions de langages existants et permettent en théorie au moins de paralléliser les codes existants avec un nombre limité de modifications. En ce qui concerne l'utilisation de machines à mémoires distribuées, on trouve des démarches similaires visant à éviter l'utilisation explicite de MPI pour simplifier l'écriture de programme parallèle. Pour cela, des mécanismes peuvent être mis en place afin de simuler un fonctionnement à mémoire partagée au-dessus d'un système à mémoire distribuée. Ceci prend habituellement la forme de DSM (*Distributed Shared Memory*) implémenté au niveau des bibliothèques[SSC96] ou globalisé au niveau de l'OS, au travers de solutions telles que MOSIX[BGW93]. Ces solutions automatiques prises en charge à l'exécution ont toutefois l'inconvénient de rencontrer des difficultés en terme de performances en traitant les accès de manière aveugle. Une approche complémentaire consiste donc à étendre des langages existants et fournir une notion d'adressage local et distant de manière native pour prendre en compte les échanges nécessaires lors d'accès distants. L'intégration au langage permet de disposer de plus d'informations sémantiques. C'est par exemple le cas des langages ou extensions de langages : UPC, CAF, X10 d'IBM, chapel, Fortress...

1.7.7 Résumé

Nous venons de voir qu'il existait différentes manières d'aborder l'expression du parallélisme sur les architectures actuelles. Il semble qu'il existe actuellement un certain goût pour une prise en charge au niveau langage, à base de directives ou d'extension plus profonde de langages existants. Ces dernières approches permettent une vision plus abstraite et plus simple pour le programmeur. Remarquons que les approches initiales MPI et pthread tendent ainsi à devenir les éléments sous-jacents d'approches plus abstraites visant à automatiser l'utilisation de ces outils. En terme de production, OpenMP devient un standard relativement adopté pour remplacer l'interface pthread sur les systèmes à mémoire partagée. Toutefois, en mémoire distribuée, MPI reste pour l'instant, l'outil principal utilisé dans les codes de calculs. L'important ici est de noter la tendance vers une expression à base de thread qui entraîne les problèmes traités dans cette thèse.

1.7.8 Problème du mélange

Jusqu'à présent, MPI s'est avéré être l'un des modèles dominants en HPC, mais il est désormais clair que ce dernier commence à montrer ses limites avec la croissance du nombre de cœurs à l'intérieur des nœuds de calcul. Les développements se tournent donc vers une démarche hybride utilisant MPI pour le parallélisme inter-nœuds et une base de thread pour le parallélisme intra-nœud. Cette démarche mène à deux problèmes techniques :

Utilisation parallèle de MPI : Depuis sa version 2.0, la norme MPI introduit une compatibilité explicite des applications multithreadées (*MPI_THREAD_MULTIPLE*). Dans les faits, bien que les implémentations disponibles progressent, il existe encore de nombreux problèmes, notamment de performance, qui surviennent lorsque les fonctions de MPI sont appelées en parallèle.

Problème du mélange de modèles : Dans ce qui précède, nous avons listé un certain nombre de normes permettant d'exprimer le parallélisme. En pratique, chacune est associée à une ou plusieurs implémentations différentes. Lorsque l'on mélange différentes bibliothèques n'utilisant pas le même modèle de parallélisme ou la même implémentation d'une norme, il apparaît la question de la collaboration de ces supports exécutifs pour le partage des ressources (cœurs) disponibles. Le problème est d'autant plus marqué que chaque support exécutif tend à monopoliser l'ensemble des cœurs sans tenir compte des autres.

1.7.9 Le projet MPC

MPC[PJN08, CPJ10] est un environnement de développement visant les grappes HPC multi-cœurs NUMA. Il vise en partie à aborder les problèmes cités précédemment. Tout d'abord, l'implémentation MPI qu'il fournit se veut parallèle dès sa base, offrant naturellement un support du mode *MPI_THREAD_MULTIPLE*. Afin de résoudre le problème du mélange, MPC implémente un support de *threads utilisateurs*. Il est ainsi possible de gérer les threads de chaque support exécutif au sein d'un même ordonnanceur, qui, étant en espace utilisateur, peut exploiter plus d'informations associées à la sémantique des différents modèles en interaction. Bien que pour l'instant, focalisé sur le mélange MPI et OpenMP, MPC propose une approche facilitant l'intégration de supports exécutifs existant par le biais de l'interface de la norme *pthread*. Il est ainsi possible de porter avec des efforts limités les supports exécutifs existant sur ce dernier.

MPC a également l'originalité de fournir une implémentation MPI basée sur les threads[PCJ09] au lieu des processus. Il est ainsi possible de faire fonctionner une application préexistante uniquement MPI dans un mode multithread par simple recompilation. Ceci permet de réduire certains problèmes liés aux implémentations MPI à grande échelle (consommation mémoire de la bibliothèque, nombre de processus à lancer et à connecter, efficacité des communications intra-nœud). Ceci permet également d'aider la transition de codes existants vers un mode de programmation hybride MPI+X avec X un modèle de programmation à base de thread. La figure 1.9 donne un exemple d'exécution dans ce mode couplé à une utilisation d'OpenMP. MPC fournit dans les grandes lignes :

- Une implémentation de threads utilisateurs qui respecte la topologie physique de la machine par contrôle de leur placement physique.
- Un support-exécutif MPI basé sur la notion de thread plutôt que processus.
- Un support-exécutif pour OpenMP exploitant les threads utilisateurs de sorte à obtenir un mélange équilibré avec le support exécutif MPI.
- Un compilateur modifié (GCC) pour le support d'OpenMP et l'exploration d'extensions telles que la privatisation automatique de variables ou les HLS (*Hierarchical Local Storage*)[TCP12].
- Un débogueur modifié (GDB) prenant en charge explicitement les threads utilisateurs.

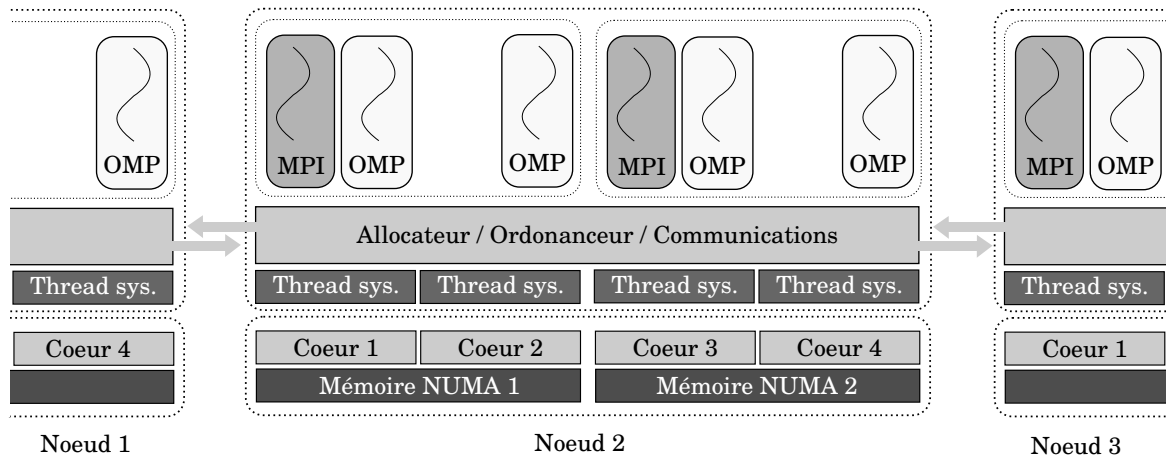


FIGURE 1.9 – MPC exécuté sur un cluster composé de deux nœuds NUMA bicœurs en considérant une tâche MPI par nœud NUMA et deux threads OpenMP pour chaque tâche MPI. Remarquons que contrairement à OpenMPI, les deux tâches partagent bien le même espace d'adressage en étant exécutées dans le même processus.

- Un allocateur mémoire parallèle supportant explicitement les hiérarchies NUMA, qui est l'objectif de cette thèse.

1.8 Question ouverte sur les DSL

La programmation hybride (*Message Passing + Threads*) devient essentielle pour exploiter les calculateurs dont nous disposons aujourd'hui. Or, comme nous l'avons vu, chaque modèle apporte ses propres difficultés, qui, combinées, rendent la programmation d'application hybride délicate. Une solution consisterait à obtenir un modèle de programmation générique, couvrant aussi bien l'intra que l'inter nœud. Or, si cet objectif est dans les esprits, aucun modèle n'a pour l'instant émergé comme remplaçant reconnu des modèles existants.

D'un autre côté, les simulations numériques profitent de l'augmentation de performance pour exploiter des modèles physiques plus complexes et les coupler. Cela conduit à une complexification des codes de simulation : code multi-physiques, multi-matériaux, raffinement de maillages, couplage de solveurs... Dans l'attente d'obtenir un hypothétique modèle généraliste, on peut se demander s'il n'est pas intéressant d'explorer une approche à base de langages spécialisés (DSL : *Domain Specific Language*), de façon à construire une expression adaptée aux différents types de problèmes.

La construction d'un DSL passe par la formalisation des concepts nécessaires à la résolution du problème ou de la classe de problèmes visés. D'une certaine manière, le langage va alors "absorber" l'expérience pratique de ses concepteurs. L'idée n'est pas ici de tomber dans le travers des systèmes experts, mais d'extraire la notion sémantique importante. On peut ainsi permettre une meilleure collaboration entre l'humain et la machine, voir également entre les humains. En suivant cette démarche, chaque équipe ou groupe d'équipes exprime son besoin particulier, avec les biais que cela implique. Dans un second temps, on peut alors espérer parvenir à extraire les points communs des différents DSL pour construire des langages plus généraux tenant compte des différentes problématiques et idées émergentes. Cette construction peut alors se faire de manière croissante, passant de sous-domaine particulier à des ensembles plus vastes, favorisant au passage, l'échange de points de vue entre les domaines respectifs. À noter que rien n'interdit l'empilement de DSL, chacun traitant des problèmes rencontrés aux différents niveaux d'abs-

traction. Mener cette démarche jusqu'à l'interface d'expression de la physique des simulations permettrait, certainement de découpler au mieux le travail du physicien et de l'informaticien, en assurant une interface d'échange saine et clarifiée entre ces domaines. Trouver des langages se situant à mi-chemin entre ces deux spécialités serait également un moyen de faciliter les échanges entre les personnes de ces domaines respectifs. Il semble en effet désormais important de ne pas se limiter aux expertises extrêmes, toujours nécessaires, mais employant des vocabulaires trop différents parfois nuisibles aux échanges inter-disciplinaires.

1.9 Le système d'exploitation

Tout au long de ce chapitre, nous avons beaucoup discuté des aspects matériels et modèles structurant les calculateurs. À l'heure actuelle, un ordinateur complet fait intervenir une dizaine d'éléments principaux (processeurs, cartes mère, disques durs...). Ceux-ci sont eux-mêmes composés d'une grande quantité de composants élémentaires, avec par exemple, plus d'un milliard de transistors pour un processeur. L'utilisation d'assemblages complexes de composants ne peut se faire en programmant directement ces derniers, ceci pour des raisons de difficulté de développement et de portabilité. Les composants des ordinateurs sont donc habituellement *orchestrés* par un logiciel spécifique dit *système d'exploitation* (OS). Ce dernier est chargé du double rôle de faire fonctionner le matériel de manière cohérente et d'offrir une abstraction permettant aux utilisateurs, programmeurs et programmes de ne pas prendre en compte les détails précis du fonctionnement de la machine. Au-delà des aspects purement fonctionnels, l'OS se voit attribuer deux rôles supplémentaires. L'un, lié à la sécurité des données, dès lors que les machines sont accessibles à plusieurs utilisateurs, notamment en réseaux. L'autre tenant à l'affectation des ressources, qu'il s'agisse du temps de calcul (utilisation du processeur), d'accès au moyen de transferts (réseaux) ou du stockage (mémoire, disque, fichiers...).

Le grand public a souvent entendu parler des OS historiques : DOS, Windows et Mac OS. Il est toutefois important de remarquer que le domaine du HPC a lui, une longue tradition d'utilisation de la famille Unix, plus industrielle. On y observe également les OS ayant historiquement mis en place les concepts fondamentaux au cœur de la constitution de ces outils logiciels. Vis-à-vis des mécanismes de gestion mémoire, on notera la forte influence des OS développés pour le calculateur Atlas[KELS62] (1962) et le système Multics[BCD69] (1969) dédié aux calculateurs de Burroughs Corporation.

On remarquera toutefois le tournant observé une fois de plus en 2000, avec l'arrivée de Linux dans les calculateurs du Top500. Cet OS de type Unix initié par Linus Torvald a été développé dans le mouvement *open source* de manière indépendante et est aujourd'hui répandu chez le public confirmé, dans les serveurs Web, nos téléphones et télévisions. Dans le domaine du HPC, il équipe plus de 50% des configurations du Top500 depuis 2004 avec un seuil actuel voisin de 80%. L'utilisation de cet OS offre l'avantage d'un accès pratique à ses sources pour modification, ainsi qu'une communauté active et une vaste gamme de logiciels et supports matériels. Il faut toutefois remarquer qu'il n'est pas développé spécifiquement pour le HPC. Avec l'augmentation massive du nombre de cœurs, certains problèmes sont donc rencontrés de manière pionnière par ce domaine et nécessitent certains efforts d'amélioration, notamment vis-à-vis des problèmes dits d'extensibilité sur un nombre de cœurs toujours croissant.

Dans le cadre de cette thèse, nous nous intéressons essentiellement aux problématiques liées à la gestion de la mémoire, tant au niveau des décisions prises par l'OS que celles prises par les bibliothèques systèmes servant d'interface entre l'OS et les langages de type C, C++ et Fortran. Le dernier point est celui qui nous intéresse plus particulièrement dans ce document, essentiellement vis-à-vis de la fonction *malloc* au cœur des méthodes d'allocation dynamique de ces

langages. Avec l'évolution vers le multicœur il devient en effet nécessaire de revisiter les compromis des mécanismes exploités par cette fonction. Nous étudierons donc ces points vis-à-vis des architectures parallèles multicœurs d'aujourd'hui et de l'évolution des modèles de programmation utilisés pour les exploiter. Les méthodes de gestion mémoire des OS seront traitées en détail dans le prochain chapitre.

1.10 Applications tests

Dans ce document, nous fournirons à plusieurs reprises des résultats expérimentaux de mesure de performances sur la base de simulations numériques. La première, EulerMHD est un code de magnéto-hydrodynamique sur maillage cartésien implémenté en MPI. Ce code a été développé par un ancien doctorant du CEA, Marc Wolf[Wol, DEJ⁺10]. Ce code effectue relativement peu d'allocations mémoires, mais fournira une base intéressante d'analyse pour notre premier cas d'étude des problématiques d'accès mémoires.

En complément, nous validerons nos méthodes d'allocation avec un code plus conséquent en terme de taille et de complexité. Hera[Jou05] est une plateforme de simulation multi-physiques multi-matériaux opérants sur maillage de type AMR (*Adaptive Mesh Refinement*). Ce type de maillage exploite des techniques de raffinement dans les zones du maillage contenant des géométries plus complexes ou ayant des termes aux dérivées élevées. Cette approche permet de traiter des problèmes plus fins sans payer le prix parfois trop élevé d'un raffinement du maillage complet. La contrepartie de ces approches est une complexification des méthodes numériques et une génération d'un nombre plus important d'allocations mémoires. L'application Hera dispose d'un parallélisme de type MPI. L'utilisation d'un maillage AMR dans ce type de contexte tend également à complexifier les échanges et générer des problèmes d'équilibrage de charge entre processus. Ceci fait de cette application un bon cas d'étude en fournissant un cas concret de résolution de problème représentatif des codes de production en complément des micro-benchmarks.

Dans les deux cas, les études en mode multithreads seront réalisées en utilisant le support exécutif MPC décrit en section 1.7.9. L'exploitation de sa capacité à exécuter chaque tâche MPI en tant que thread plutôt que processus permet en effet de convertir rapidement des applications MPI en applications multithreads.

1.11 Conclusion

Nous venons de voir dans ce chapitre que le domaine du HPC a connu une forte évolution en terme architecturale depuis ses commencements. Au cours des années 2000, les limites technologiques ont poussé les constructeurs à s'orienter vers une stratégie menant aux calculateurs massivement parallèles d'aujourd'hui. Les contraintes de conception des calculateurs devant nous conduire à l'exascale vont poursuivre cette évolution en amplifiant les échelles d'exploitation. Dans ce contexte, les modèles de programmation utilisés et les outils logiciels (OS, bibliothèques...) sont mis à rude épreuve et doivent prendre en compte ce nouveau paradigme. Pour ce faire, il importe d'exploiter des algorithmes passant à l'échelle en terme de performance et de consommation mémoire.

C'est dans ce cadre que cette thèse s'intéresse au sous-ensemble que constituent les mécanismes de gestion de la mémoire tant au niveau de l'OS que de la fonction utilisateur *malloc* défini par le langage C. L'étude est notamment conduite pour ré-évaluer le fonctionnement de ce composant vis-à-vis d'un parallélisme massif à base de threads et des caractéristiques propres aux simulations numériques. Nous verrons notamment que les simulations tendent à utiliser des

tailles d'allocation qui leur sont propres (grands tableaux). Ce type d'allocation peut nécessiter un travail autre que les nombreuses allocations de petite taille des applications habituellement étudiées dans le domaine de la gestion mémoire. Nous discuterons donc largement cette spécificité.

Le second chapitre de cette partie contextuelle permettra au lecteur de se familiariser avec les mécanismes de gestion de la mémoire. Nous y introduirons donc essentiellement les concepts de mémoire virtuelle, traduction d'adresse, caches processeurs, nœuds NUMA et définition de l'API POSIX de gestion mémoire au travers des fonctions associées à *malloc*.

Chapitre 2

Gestion de la mémoire

2.1 Introduction

Si un ordinateur est avant tout une machine à calculer, il est important de noter qu'il doit être alimenté par des données. Ces dernières doivent être identifiables et accessibles efficacement. D'autre part, un ordinateur ne peut stocker qu'une quantité finie de données dans sa mémoire. Il importe donc de gérer le cycle de vie de ces dernières et leur arrangement dans l'espace de stockage dédié. S'ajoute à cela un besoin de partager la ressource mémoire entre divers programmes. Dans ce chapitre, nous rappellerons les concepts fondamentaux ayant trait à la gestion des données par le *système d'exploitation*, notamment en ce qui concerne la construction des espaces d'adressage.

Ce chapitre introduira les concepts fondamentaux liés à la gestion de la mémoire dans les OS modernes. Nous traiterons dans un premier temps de la constitution de l'espace d'adressage virtuel, de l'isolement des différents composants logiciels et de la méthodologie de partage de la ressource mémoire. Nous pourrons alors étudier plus en détail l'interface d'échange entre l'OS et les applications pour permettre d'effectuer des requêtes mémoires. Nous terminerons enfin par une description des mécanismes mis en œuvre pour distribuer la mémoire au sein des applications et bibliothèques en espace utilisateur. Après l'introduction des principes de gestion des données, nous rappellerons dans une dernière section que l'accès aux données peut être affecté par certaines contraintes sur les processeurs modernes. Nous décrirons donc les méthodes d'accès à la mémoire pouvant impacter les décisions de l'allocateur. Pour ce faire, nous aborderons principalement la notion d'associativité des caches du processeur et les mécanismes de traduction d'adresses. Ces concepts seront essentiels pour comprendre la première partie de nos travaux.

2.2 Le système d'exploitation

Les systèmes d'exploitation modernes ont la charge de mettre en œuvre des méthodes permettant l'exécution simultanée de plusieurs programmes sur un même matériel. À ce titre, ils doivent garantir l'isolation de ces derniers et le partage des ressources matérielles, notamment mémoire. Cette section décrit la notion fondamentale de mémoire virtuelle exploitée par tous les OS modernes pour atteindre ces objectifs.

2.2.1 Problématiques du multiprogramme/multiutilisateur

Les premiers ordinateurs étaient utilisés par un unique utilisateur et ne faisaient fonctionner qu'un seul programme à la fois, par exemple sous DOS. Dans ce contexte, un programme

peut sans difficulté travailler dans un espace d'adressage unique partagé avec le système d'exploitation, c'est donc la méthode retenue dans les premiers temps. Or, certains systèmes permettent d'exécuter plusieurs programmes simultanément[Tan05, Han73]. Ce type d'exécution est rendu possible par la présence de plusieurs processeurs/cœurs ou par l'utilisation d'un mode dit *interruptible*, exécutant tour à tour chaque programme pendant un laps de temps déterminé[Lor72, GCO65]. Le système de la machine Atlas[KELS62] offrait par exemple un mode interruptible. Ces techniques rencontrent toutefois certaines limitations que l'on peut décrire comme suit.

Problème de relocation : lorsque les programmes sont compilés, leur code est conçu pour être chargé à une adresse fixe. Il en va de même pour certains éléments tels que les constantes, les variables globales, la pile¹, etc... Si l'on dispose d'un espace d'adressage unique, relancer plusieurs fois un tel programme est alors impossible sans obtenir une superposition des bandes d'adresses utilisées par ce dernier. On doit donc recompiler le programme pour le lancer plusieurs fois, ou passer par la complexité de génération d'un code *relocalisable*². Ce problème est déjà présent en 1965 chez IBM[McG65].

Problème de sécurité, robustesse : dans une telle organisation, les programmes ont accès à toute la mémoire et peuvent donc lire ou corrompre les données d'autres programmes voir du système d'exploitation lui-même. Ce problème se pose en cas d'attaque de la part d'un utilisateur malveillant ou d'un bogue entraînant l'utilisation non prévue de zones mémoires réservées à un autre usage.

Il est possible de traiter le premier problème en modifiant les programmes générés au prix d'une complexification des *compilateurs*, *éditeurs de lien* et *lanceurs* de programmes. Aujourd'hui, c'est toutefois une autre approche qui est retenue, réglant le second point du même coup. Elle est détaillée dans ce qui suit.

2.2.2 Adresses virtuelles, adresses physiques

Une solution plus efficace au problème du multiprogramme consiste à ajouter un niveau d'abstraction virtualisant les adresses. On crée ainsi une séparation entre les adresses manipulées par les programmes et les adresses physiques en mémoire centrale. Les premières idées (non implémentées) de cette approche remontent aux années 1956. Le problème initial concernait l'optimisation d'accès à des données sur de multiples tambours rotatifs[Rob04]. Elle a également été mise en place sous une forme proche des concepts d'aujourd'hui sur la machine Atlas[KELS62] puis reprise par les systèmes suivants, tels que Multics[BCD69].

Depuis, les architectures modernes disposent toutes d'au moins deux types d'adressage au niveau matériel. *L'adressage physique* fait correspondre les adresses à la position des données dans la mémoire centrale. *L'adressage virtuel*, plus abstrait, fournit des adresses qui n'ont pas de correspondances directes avec la mémoire centrale. L'utilisation d'un système d'adresses virtuelles offre plusieurs avantages du point de vue de l'OS et des applications.

Efficacité de gestion : Cette approche permet de disposer d'un espace d'adressage couvrant toute la plage accessible pour une taille d'adresse donnée (32 bits ou 64 bits par exemple). Ceci offre un espace beaucoup plus grand que l'espace physique qui est naturellement restreint à la taille de la mémoire disponible. Disposer de ce grand espace permet de gérer plus simplement et plus efficacement le placement des différents objets dans la mémoire. Ce problème est rencontré au quotidien, le rangement d'un grand placard étant habituellement plus aisé qu'un espace réduit.

1. La pile est un espace mémoire utilisé pour stocker les variables locales. Son adresse haute est incrémentée à chaque appel de fonction de sorte à *empiler* les variables locales et les *dépiler* lorsqu'une fonction se termine.

2. Dont les différents éléments peuvent être chargés en mémoire à des adresses non fixées par avance.

Isolement : Chaque programme peut disposer de son propre espace d'adressage. Il est donc possible d'assurer qu'aucun processus ne pourra accéder aux données d'un autre de manière non autorisée. Chaque programme est ainsi isolé du reste du système, ce qui offre un bon moyen de sécuriser les données de ces derniers et d'isoler le système d'exploitation.

Multi-instances : La question des codes non relocalisables est réglée en permettant l'utilisation des mêmes adresses virtuelles pour différents programmes, chaque instance travaillant dans son propre espace. Cela suppose toutefois que les adresses physiques sont associées à une unique adresse virtuelle, sauf demande explicite (*mémoire partagée*).

Projection passive : Il est possible de projeter un autre espace de stockage dans la mémoire (fichier, disque...) en générant des transferts au moment des premiers accès aux données. Les accès à ces données se font donc de manière transparente au travers de simple accès mémoire.

Extension de la mémoire : La virtualisation de la mémoire permet de mettre en place facilement un système de pagination disque (*swap* dans la terminologie Unix) permettant d'utiliser plus de mémoire physique que disponible. Dans ce cas, une fraction, si possible rarement utilisée, de la mémoire est copiée sur le disque pour laisser place à une nouvelle donnée plus utile. Les futurs accès à la donnée déplacée nécessitent alors un échange (*swap*) pour être chargés en mémoire avant utilisation. Cette extension se fait toutefois au prix d'un surcoût lié à la relative lenteur des systèmes de stockages permanents tels que les disques durs.

Compression mémoire : Dans la même idée que le *swap*, certains proposent de compresser les données non utilisées pour économiser de la mémoire [Riz97, YDLC10] sans être pénalisé par les accès aux disques durs. On notera une certaine activité récente autour de cette question notamment vis-à-vis des machines virtuelles pour Linux.

2.2.3 Segmentation et pagination

Le principe général étant posé, il convient de trouver une manière de concrétiser cette approche conceptuelle. Pour cela, il faut offrir un moyen efficace de décrire la traduction d'une adresse virtuelle en son équivalent physique. Il existe deux méthodes historiques : la *segmentation*, mise en place par exemple dans les machines de Burroughs Corporation avec le B5550 en 1961 [Den70, DD68] notamment sous Multics et la *pagination* mise en place dans le calculateur Atlas de 1962 [KELS62, KPH61]. Ces dernières sont décrites dans ce qui suit avec un accent plus prononcé sur la méthode retenue historiquement : la *pagination*.

Pour la *segmentation*, l'espace virtuel se définit par assemblage de "segments" décrits comme des espaces mémoire contigus de tailles variables. Ces segments sont associés à un décalage permettant la correspondance avec les adresses physiques par simple addition à l'adresse virtuelle. Ils sont donc décrits dans une ou deux tables dont les entrées sont identifiées par le *sélecteur de segment*, habituellement défini par un registre. Différents sélecteurs sont généralement disponibles pour distinguer les différentes parties du programme : code, constantes, pile... Sur les architectures à faible largeur d'adressage, cette approche a posé de nombreux problèmes du fait de la nécessité de changer (explicitement) de segment pour accéder à des tableaux de tailles importantes.

La *pagination* reprend la notion de segment, mais lui applique une taille fixe. L'espace virtuel est ainsi découpé en segments de taille prédéfinie, dits *pages* (généralement 4 Ko). L'espace mémoire physique est découpé de la même manière de sorte qu'il est possible d'associer une *page virtuelle* à une *page physique* (Figure 2.1). L'association entre ces deux espaces est alors décrite dans une table maintenue par l'OS. Ce découpage de la mémoire en segments de tailles fixes permet de faciliter la gestion des allocations mémoires au niveau du système d'exploitation :

1. L'utilisation d'une taille unique (ou ses multiples) réduit les problèmes de fragmentation de la mémoire, c'est à dire l'introduction de trous non alloués, mais trop petits pour être utilisés par la suite. Ce problème sera discuté plus en détail dans la suite (section 2.4.2).
2. L'utilisation d'un grand nombre de pages pour décrire l'espace d'adressage permet de décrire efficacement un espace creux. De cette façon, les pages virtuelles non utilisées peuvent être libérées afin de réduire la consommation mémoire des programmes, et ce, même pour celles situées au milieu d'un segment alloué.
3. Les différentes pages voisines n'ayant pas de contraintes de contiguïté, il est possible d'allouer efficacement la mémoire en utilisant plus aisément les pages physiques disponibles.

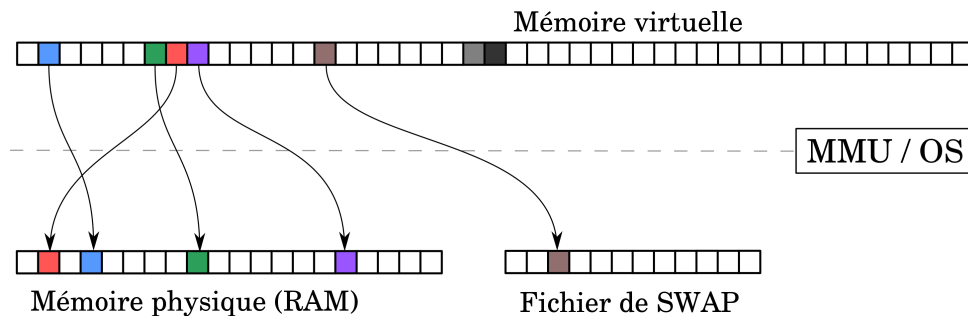


FIGURE 2.1 – *Pagination de la mémoire pour gérer la traduction des adresses virtuelles.*

L'invention du système de pagination est décrite par certains comme l'une des plus belles ingénieries de l'informatique. Toutefois, sa mise en place effective a longtemps été en but au problème pratique de construire une manière efficace de traduire les adresses au niveau matériel. Cette nécessité de modifier le matériel a dans un premier temps restreint cette technique à des études liées aux machines virtuelles pour lesquelles une émulation logicielle pouvait rapidement être mise en place.

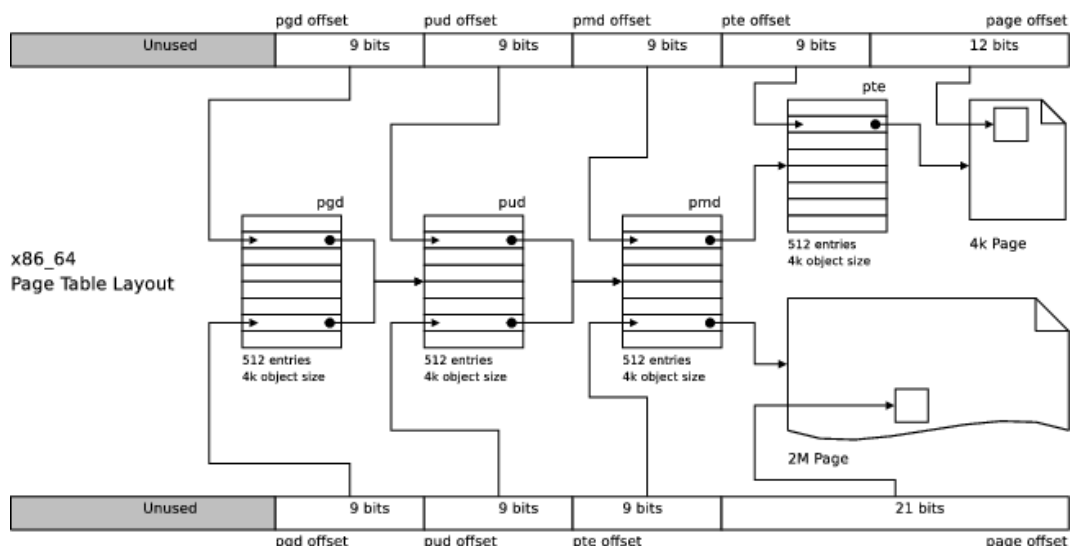


FIGURE 2.2 – *Structure de la page des tables utilisée par Linux pour les architectures x86_64 extraite de <http://linux-mm.org/>.*

Notons que les espaces virtuels actuels couvrent généralement un adressage sur 48 bits, soit 256 To. Les 64 bits d'adresse des processeurs ne sont en effet pas réellement exploitables pour l'instant, mais laissés pour un usage futur[[Int10a](#)]. Stocker la table des pages sur un seul niveau

nécessiterait $256To/4Ko = 64Go$ de mémoire. En pratique, cette dernière prend donc la forme d'un arbre dont chacun des nœuds occupe la taille d'une page (4Ko). Un exemple est donné par la figure 2.2 pour Linux sur architecture x86_64. La traduction d'adresse peut ensuite être effectuée de manière logicielle par l'OS ou matérielle par une unité dédiée, la MMU (*Memory Management Unit*).

2.2.4 Notion de processus et threads

La notion abstraite de mémoire virtuelle permet de recréer au niveau logiciel l'équivalent des notions *distribuées* et *partagées* des architectures matérielles. Dans les systèmes actuels, chaque *instance* (réalisation concrète) d'un programme est dite *processus*. Chaque processus se voit attribuer son propre espace d'adressage virtuel et au minimum un flux d'exécution matérialisé par une pile. Cette entité rassemble également d'autres notions dans sa définition (descripteurs de fichiers, droits utilisateurs...), mais qui sortent du cadre particulier de la gestion mémoire. Les *processus* forment l'équivalent des architectures à mémoire distribuée, car chacun dispose de son propre espace mémoire privé. Les échanges entre processus doivent donc être établis de manière explicite par les deux participants et passer par le système d'exploitation (IPC : *Inter-Process Communication*). Or, nous l'avons vu en section 1.7.1, ce type d'échange implique un surcoût sur des architectures à mémoire partagée.

Les systèmes d'exploitation modernes étendent donc la notion de processus en permettant à ces derniers de contenir plusieurs flux d'exécution, matérialisés par la notion de *thread*. Chaque *thread* dispose alors de sa propre pile d'exécution, mais partage le même espace d'adressage et les mêmes descripteurs de fichiers avec tous les *threads* d'un même *processus*. Certains ajoutent un niveau de *threads utilisateurs* gérés par des bibliothèques utilisateurs et non par l'OS. Ces derniers sont en générale plus rapide à lancer et endormir, car ne nécessitant pas un passage coûteux par l'OS comme nous allons le voir dans ce qui suit.

2.2.5 Espace noyau et utilisateur

La constitution d'un espace virtuel permet de construire une protection autour du système d'exploitation. Dans le mode dit "protégé", le processeur peut en effet être configuré pour fonctionner selon différents niveaux de privilèges limitant l'exploitation des instructions modifiant l'état général du matériel. Ces méthodes ont une fois de plus été initialement mises en place dans le cadre des développements de du système Multics. Bien que le matériel en offre plusieurs, on distingue habituellement deux niveaux de privilèges au niveau logiciel. Le *mode noyau* limité au système d'exploitation qui est seul à disposer de tous les droits et le *mode utilisateur* dans lequel fonctionnent les programmes. L'exploitation de machines virtuelles tend aujourd'hui à rendre intéressante l'utilisation d'un troisième niveau de contrôle. Le *mode hyperviseur* dédié à l'OS racine contrôle les machines virtuelles. L'implémentation d'une telle approche se fait efficacement si elle est couplée à la notion de mémoire virtuelle en permettant de garantir les points suivants :

1. La modification des registres de sélection des tables d'adresses (segmentation ou pagination) est limitée au mode noyau.
2. La modification du contenu de la table des pages est également restreinte au mode noyau. Cette protection est naturellement assurée par la présence de mémoire virtuelle si la table n'est pas projetée dans l'espace des programmes.

Cette séparation permet une politique de sécurité, mais complique les échanges entre les applications et le système d'exploitation. Les fonctions de ce dernier ne peuvent en effet plus être appelées directement par un saut d'adresse. Ce type d'appel dit *appel système* repose donc sur la levée d'*interruptions* logicielles qui permettent au processeur de basculer en mode noyau avec une montée de privilège. Le gestionnaire d'interruption de l'OS traite alors la requête avant

de rendre le contrôle à l'application. Cette mécanique impacte les performances, notamment parce qu'elle implique une sauvegarde en mémoire de l'état du processeur, donc de l'ensemble de ses registres (adresse de l'instruction en cours, adresse de retour de fonction, registres temporaires...). Elle sous-entend donc une utilisation raisonnée de ce type d'appel.

En plus de décrire l'association des pages virtuelles et physiques, la table des pages définit les droits d'accès à la mémoire. Le contenu d'une page peut en effet être lu, écrit ou exécuté. Sur de nombreuses architectures, telles que x86_64, les pages sont également associées à un niveau de privilège. Ceci permet de projeter l'ensemble de la mémoire utilisée par le noyau dans l'espace virtuel du processus (en général à la fin). Cet espace est inaccessible en temps normal par l'utilisateur. Cette astuce utilisée par Linux[BPO5] et d'autres OS permet d'éviter d'invalider certains caches (TLB que nous verrons en section 2.5.2) en changeant de table des pages lors des appels systèmes. Cela a toutefois conduit à la limitation des 3.5 Go de mémoire adressable sur architecture 32 bits au lieu des 4 Go attendus. Pour les architectures 64 bits, l'espace est suffisamment grand pour que cette approche ne pose plus de problème.

2.3 Interface avec les applications

Lorsqu'une application a besoin de manipuler sa structure mémoire, elle interagit avec le système d'exploitation par le biais d'*appels systèmes* ou de *levée d'exception* par le matériel lors d'accès à des zones non autorisées. Nous allons décrire ici la sémantique d'échange permettant la négociation d'allocation mémoire entre l'OS et les applications en espace utilisateur.

2.3.1 Les appels systèmes

On trouve cinq appels systèmes principaux pour la gestion de la mémoire, donnés ici dans leur syntaxe Unix. Des équivalents existent sur les différents OS :

brk/sbrk : Ces appels permettent de changer la taille d'un segment contigu traditionnellement utilisé pour implémenter le tas, un espace utilisé pour stocker les tableaux dynamiques. Les données qui y sont stockées peuvent au contraire de la pile être libérées dans n'importe quel ordre et sans être lié à la durée de vie de la fonction les ayant créé.

mmap : Historiquement cet appel système était dédié à la projection de fichiers directement dans la mémoire, de sorte que son contenu soit chargé automatiquement lors des accès aux adresses associées. Plus abstrait que *brk*, cet appel permet de créer un segment de taille voulue n'importe où dans l'espace virtuel. Il est aujourd'hui étendu pour l'allocation mémoire dite *anonyme* (non associée à un fichier) utilisé pour gérer le tas de manière plus souple qu'avec *brk*.

munmap : Complémentaire de l'appel *mmap*, il permet de libérer une zone préalablement réservée par *mmap*. À noter qu'il n'est pas limité à s'appliquer sur un segment complet, mais peut-être utilisé pour réduire ou couper des segments existants. La sémantique de l'équivalent Windows est toutefois différente sur ce dernier point.

mremap : Spécificité de Linux, cet appel système permet de redimensionner ou déplacer un segment existant dans l'espace virtuel.

mprotect : Moins utilisé, cet appel permet de changer les droits associés à un segment mémoire (lecture, écriture, exécution).

Tous ces appels travaillent à la granularité de la *page*, ils manipulent donc uniquement des tailles et adresses multiples de leur taille (habituellement 4 Ko). D'un point de vue plus abstrait, ces appels génèrent des zones mémoires virtuelles (VMA³ dans la terminologie Linux, nous

3. VMA : Virtual Memory Area

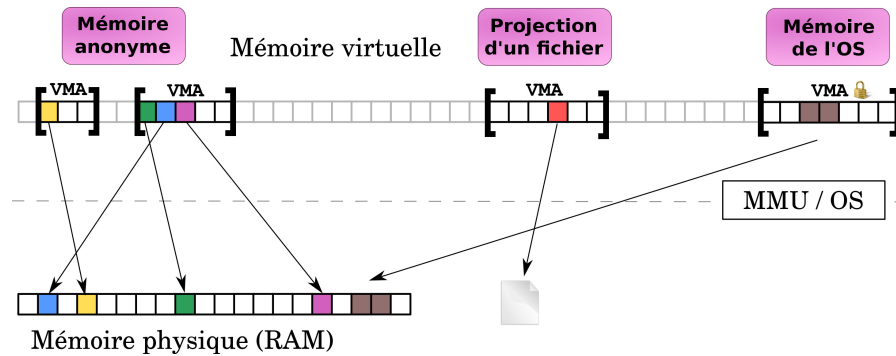


FIGURE 2.3 – Exemple d'organisation de l'espace virtuel Linux en VMA avec différentes propriétés. Les zones inter-VMA sont interdites et conduisent immédiatement à un arrêt du processus au travers du signal *faute de segmentation*.

utiliserons le terme de *zone mémoire* dans ce document) auxquels sont associées des informations (taille, adresse de base, droits, source de donnée...). Un exemple de découpage de l'espace virtuel en VMA est donné dans la figure 2.3.

2.3.2 Fautes de pages

Nous l'avons vu, l'espace d'adressage virtuel est beaucoup plus grand que l'espace physique. Par conséquent, la majeure partie de ce dernier n'est pas associée à une zone mémoire physique. Lorsqu'un processus accède à ces zones, on dira qu'il y a *faute de page*. Une interruption est donc levée pour donner la main au système d'exploitation qui va alors terminer le processus fautif. Cet arrêt est effectué en envoyant un signal de type *faute de segmentation* bien connu des développeurs C/C++. Ce type d'erreur peut survenir si le processus accède à une zone non associée à une page physique, ou bien outrepassé les droits d'accès aux données de cette page (lecture, écriture, exécution ou niveau de privilège).

Cette capacité de notifier l'OS d'un accès à une page est mise à profit pour mettre en place une politique d'*allocation paresseuse*. Selon cette approche, les appels de types *mmap* autorisent l'utilisation d'un *segment mémoire*, mais ne lui associent pas immédiatement de mémoire *physique*. Le segment est alors dit *virtuel*. Ces segments provoquent donc une *faute de page* lors du premier accès par le processus. À ce moment, l'OS peut alors dynamiquement associer une page physique à la zone concernée et rendre le contrôle au processus pour qu'il poursuive son exécution de manière transparente. L'utilisation de la mémoire physique est donc ajustée dynamiquement en fonction de l'utilisation réelle par le processus. Cela permet d'éviter un gaspillage mémoire pouvant survenir avec les applications qui demandent explicitement plus de mémoire qu'elle n'en utilise réellement.

Cette méthode est particulièrement utile pour les fichiers projetés dans la mémoire par *mmap*. De cette manière, ne sont lues et chargées que les parties utiles de ces fichiers. Ceci est notamment utilisé pour les exécutables et bibliothèques. Ce mécanisme est d'ailleurs la base même du système de *pagination disque* survenant en cas de congestion mémoire⁴. Dans ce cas, une page considérée comme inutile est copiée sur le disque et supprimée de la table des pages. Lors d'un accès futur, une faute de page sera levée pour demander le rechargement de cette donnée en mémoire.

4. Lorsque toute la mémoire est utilisée par l'OS et les applications et que des requêtes sont en attente.

2.4 L'allocateur mémoire (malloc)

Le système d'exploitation gère les allocations mémoires des applications à la granularité d'une page (4 Ko). Toutefois, le programmeur peut vouloir utiliser des segments mémoires plus petits, allouer une page entière pour chacun d'entre eux conduirait à un gaspillage évident. Au niveau applicatif, une page doit donc être partagée en *blocs* plus petits. Ce rôle est dévolu à la fonction du standard C : *malloc*, chargée de gérer la mémoire à l'intérieur de l'espace virtuel. Cette dernière prend en charge la demande de pages auprès du système d'exploitation et leur découpage éventuel pour les requêtes de petite taille.

2.4.1 Interface

Pour son interface, l'allocateur définit dans le langage C suit une démarche proche de ce que l'on retrouve au niveau du système pour manipuler les pages, à savoir le nécessaire pour allouer, redimensionner et libérer des segments mémoires :

malloc : Fonction utilisée pour allouer un nouveau segment mémoire de taille demandée.

calloc : Fonction similaire à *malloc*, mais assure contrairement à cette dernière que la mémoire nouvellement allouée est initialisée à 0.

free : Fonction permettant de libérer la mémoire occupée par un segment alloué avec l'une des autres fonctions.

realloc : Fonction permettant de redimensionner un segment. Cette méthode provoque un éventuel changement d'adresse du segment si l'espace contiguë suivant est occupé et ne permet de satisfaire la requête. Le segment sera alloué s'il n'existe pas déjà (adresse nulle), ou bien libéré pour une taille nulle.

memalign : Fonction similaire à *malloc* mais permettant de contraindre l'alignement mémoire du segment.

Dans le langage C++, cette interface est étendue par l'adjonction des mots clés *new* et *delete* se traduisant par un pré- ou post-traitement éventuel aux appels respectifs de *malloc* et *free*. Cet opérateur prend automatiquement en charge la détection de taille du type et l'appel des constructeurs et destructeurs de l'objet considéré s'il en définit.

Ces fonctions d'allocation de C et C++ sont utilisées pour les allocations dites *dynamiques* par opposition aux allocations *statiques* placées par le compilateur sur la *pile*. On décidera de recourir à leur usage dans les cas suivants :

1. La taille de l'allocation n'est pas connue au moment de la compilation, la décision doit donc être prise au moment de l'exécution. Ce critère est une des raisons principales de l'approche *dynamique*.
2. La taille du segment à allouer est trop grande pour tenir dans la pile.
3. Le ou les éléments placés dans le segment alloué doivent survivre après la sortie de la fonction.

On notera que les langages de plus haut niveau (de type Java, C#...) n'exposent pas cette distinction au développeur en ne lui fournissant que la sémantique d'allocation dynamique.

2.4.2 Fragmentation

Dans un programme, l'allocateur mémoire est appelé par l'ensemble des fonctions du programme ; les requêtes sont donc associées à des tailles potentiellement très variables, dépendantes des objets manipulés par ces dernières. D'autre part, les segments alloués peuvent avoir des durées de vie très différentes. Ces deux paramètres créent le problème dit de *fragmentation*,

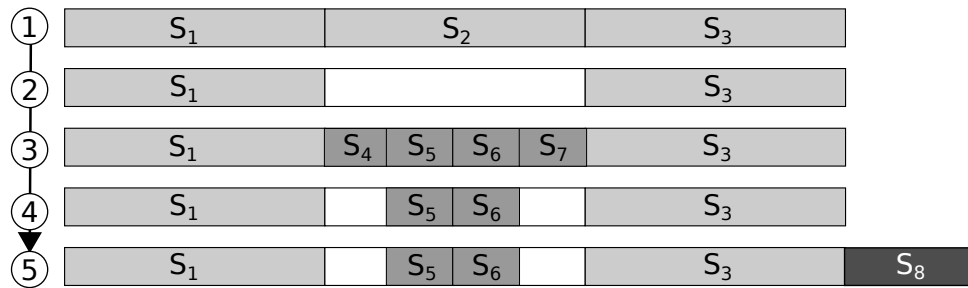


FIGURE 2.4 – Illustration du problème de fragmentation externe en considérant une séquence d'allocation d'éléments de 256o, 64o et 128o.

correspondant à l'apparition de trous non réutilisables (taille plus petite que les requêtes à venir) entre segments alloués. Ce problème est très discuté dans la littérature[JW98] et représente l'un des principaux problèmes de conception des allocateurs mémoires. Ce problème peut s'illustrer en supposant un schéma d'allocation du type :

1. Allocation de trois segments de 256 octets. Dans un état initial, ils peuvent être alloués de manière contiguë. Notons-les s_1, s_2 et s_3 par adresse croissante.
2. Libération du segment s_2 situé au milieu des deux autres.
3. Allocation de 4 segments de 64 octets qui peuvent être placés dans la zone libérée par s_2 .
4. Libération des segments de 64 octets 1 et 4.
5. Allocation d'un nouveau segment de 128 octets.

Dans cette configuration, le segment final ne peut être placé dans l'espace laissé libre entre s_1 et s_3 car cet espace est fragmenté. Le bloc doit donc être alloué après s_3 , conduisant à une augmentation de la consommation mémoire. Si toutes les allocations suivantes nécessitent plus que 64 octets, la zone fragmentée entre s_1 et s_3 ne pourra jamais être réutilisée. La fragmentation tend donc à dépendre de deux paramètres :

1. Le mélange de blocs de tailles différentes pouvant conduire à un mélange non compact de l'espace mémoire.
2. Le mélange de blocs ayant des durées de vie différentes peut entraîner la formation de trous non compatible avec les tailles mémoires requises dans le futur ou rend impossible la libération des pages associées non totalement utilisées.

La littérature distingue habituellement deux types de fragmentations :

Fragmentation interne : Cette dernière décrit la quantité de mémoire perdue à l'intérieur du segment alloué. En effet, l'allocateur ajoute généralement des en-têtes de description au segment et applique certains décalages pour maintenir des alignements compatibles avec les contraintes de l'architecture. Ces espaces supplémentaires s'ajoutent donc à celui demandé par l'utilisateur. Un bon exemple de fragmentation interne apparaît avec la pagination : si l'on ne dispose pas d'un allocateur intermédiaire, la demande de 1 Ko conduit à la réservation pratique de 4 Ko, c'est la fragmentation interne.

Fragmentation externe : On s'intéresse ici à l'espace perdu par le placement des blocs, donc relié à la présence de trous non réutilisables. En ces termes, la gestion des pages physique n'est pas impactée par la fragmentation externe, car une seule taille de bloc est utilisée, il est donc toujours possible de ré-utiliser une page physique libre.

Remarquons que les politiques oscillent souvent entre ces deux types de fragmentation, la réduction de l'une tendant généralement à favoriser l'autre comme c'est par exemple le cas pour la pagination.

2.4.3 Ramasse-miettes

La gestion de la mémoire est souvent une tâche délicate dans le développement d'un programme. La libération des segments alloués est souvent l'objet d'oubli. Ce problème de *fuite mémoire* est traité par des outils tels que Valgrind[NS07] et demande des efforts systématiques de la part des développeurs. Ce problème peut être pour parti réglé avec l'introduction d'un *ramasse-miettes* (*Garbage Collector*), introduit pour le langage récuratif LISP dans les années 1950[McC60]. Cette approche vise à libérer automatiquement toute zone mémoire non référencée. Remarquons toutefois que le problème subsiste si le développeur oublie de déréférencer un objet. Il n'est donc résolu que pour les segments orphelins pour lesquels aucun pointeur ne permet plus d'y accéder.

Ce type d'approche est au centre des langages de plus haut niveau tels que Java, Python, C#... en leur permettant d'abstraire les problèmes de gestion mémoire. Ces langages basés sur la notion de référence plutôt que pointeur offrent également la possibilité de déplacer un segment après son allocation. Ceci lève donc l'une des contraintes principales des allocateurs mémoire des langages C, C++ et Fortran. Avec cette liberté, il est en effet possible d'effectuer une défragmentation régulière de la mémoire. On trouve ainsi une très large littérature dans ce sens[DP00, BOP03], notamment dans le cadre des architectures parallèles actuelles. Ceci permet par exemple d'éliminer le problème de fragmentation survenant sur la figure 2.4 de la section précédente. Cette thèse s'intéresse toutefois aux principaux langages utilisés en HPC (C, C++ et Fortran), nous ne pourrions donc pas profiter de cette liberté.

2.4.4 Problématique d'implémentation

La problématique majeure de l'allocateur est de décider rapidement où placer un bloc mémoire, trouver rapidement un bloc qui pourrait être ré-utilisé et décider ou non de le subdiviser s'il offre un espace trop grand. Ces décisions à prendre rapidement sont malheureusement celles qui peuvent conduire aux problèmes de fragmentation si elles font apparaître un nombre trop important de trous inutilisés. Ceci d'autant plus qu'il n'est pas possible de déplacer un bloc précédemment alloué. Les allocations à venir ne peuvent être prédites, les algorithmes mis en place sont donc nécessairement construits sur la base d'heuristique. Le choix d'un bon algorithme d'allocation peut être décrit comme une lutte contre les cas pathologiques. Le meilleur algorithme est donc celui affecté par un nombre réduit de cas problématiques. Certains essaient à contrario d'obtenir des prédictions de durée de vie sur les blocs pour guider leurs décisions[SZ98, BZ93].

2.5 Accès à la mémoire

Au-delà de la simple notion de gestion mémoire, il est important de remarquer que l'évolution des processeurs doit satisfaire à de nombreuses contraintes autres que sa seule capacité à calculer. En terme de performance, la mémoire est actuellement un facteur limitant. Les processeurs se sont donc structurés de manière à compenser ces effets. On obtient ainsi la structure en cache hiérarchique que l'on observe aujourd'hui. Les systèmes d'exploitation ont également nécessité certaines adaptations au niveau matériel afin de permettre leur développement. C'est notamment ce que l'on observe avec l'introduction des mécanismes de pagination au centre de la gestion de la mémoire de tout système moderne. Cette section dresse un inventaire des points clés décrivant l'accès physique aux données en considérant les problématiques propres à la pagination, donc aux traductions d'adresses et aux méthodes de transfert de données entre la mémoire et les unités de calcul.

2.5.1 Les caches

Nous avons rappelé dans la section 1.2 que l'évolution de l'informatique s'est rapidement confrontée au problème d'accès à la mémoire. Structuellement, la première parade consiste à introduire la notion de *registre* permettant de travailler sur des données locales au processeur évitant le surcoût systématique d'accès à la mémoire centrale. Il est ainsi intéressant de travailler sur une copie locale, accessible rapidement et ne la synchroniser avec la mémoire centrale que lorsque l'espace occupée par cette copie est nécessaire pour l'utilisation d'une autre donnée. La notion de localité spatiale et temporelle fait référence au fait que la plupart des programmes ont tendance à réutiliser plusieurs fois les mêmes données. La notion de registre se poursuit donc avec l'introduction de mémoires locales (caches). Ces derniers peuvent contenir des copies d'une portion de la mémoire centrale. Ces mémoires de petite taille peuvent alors bénéficier pour un coût raisonnable de techniques plus onéreuses et plus efficaces. Les faibles tailles permettent également de limiter les effets liés à l'adressage de grands volumes de données. Lors d'un accès à une donnée, on dira qu'il y a *faute de cache* si cette dernière n'est pas présente dans le cache. Un transfert doit alors être déclenché de la mémoire vers la zone de copie. Sur les architectures actuelles, ces transferts sont effectués par unité dite *ligne de cache* correspondant habituellement à 64 octets contigus. Les caches peuvent éventuellement être spécialisés pour ne gérer que les données ou instructions du programme.

Hiérarchie

En 1994, Wulf et McKee[WM95] définissent le problème du *mur de la mémoire* en pointant un écart de vitesse dans l'évolution des technologies des processeurs et de la mémoire. Selon eux, si rien n'est fait, les architectures finiront par être limitées essentiellement par les capacités de la mémoire. Ils estiment l'arrivée de ces problèmes majeurs pour 2010. L'utilisation de cache est un moyen de repousser la limite, en augmentant la réutilisation locale de données et donc en réduisant la pression sur la mémoire globale. La récupération spéculative de données (*pré-fetcheur*) est également une manière de réduire l'impact de l'écart de performance. Toutefois, l'accroissement de cet écart doit être compensé par un accroissement de la taille des caches. Or, construire des caches de taille plus importante implique également de réduire leur performance. À un certain point, il devient donc nécessaire de mettre en place une hiérarchie de caches. Les plus petits caches, plus rapides, compensent ainsi le manque de performance du niveau supérieur, plus gros, mais plus lent. Les processeurs actuels contiennent 3 niveaux de caches allant de quelques kilo-octets à une dizaine de méga-octets.

La présence de plusieurs cœurs au sein d'un même processeur impacte cette hiérarchie. Les caches peuvent en effet être locaux ou partagés entre plusieurs cœurs. On trouve en général des caches *locaux* pour les premiers niveaux et *partagés* pour les derniers. La figure 2.5 (a) donne la structure détaillée d'un processeur Intel Sandy-bridge, typique de ce que l'on observe sur les processeurs actuellement utilisés par les super-calculateurs. L'impacte de cette hiérarchie peut être mis en évidence en mesurant le temps d'accès répétés à un segment de taille variable. Comme le montre la figure 2.5 (b), l'accès aux petits volumes de données se fait d'autant plus rapidement qu'ils peuvent être contenus dans les caches les plus internes.

Associativité des caches

Les caches sont des mémoires contenant une copie locale d'information. Leur utilisation se fait de manière transparente en étant entièrement prise en charge par les mécanismes matériels du processeur. Les échanges entre la mémoire et le cache se font par *lignes de cache*, soit des segments contigus de 64 octets, et ce, même si l'utilisateur n'a demandé qu'une fraction de cette zone mémoire. Toutefois, le cache est par définition une entité qui ne peut pas contenir

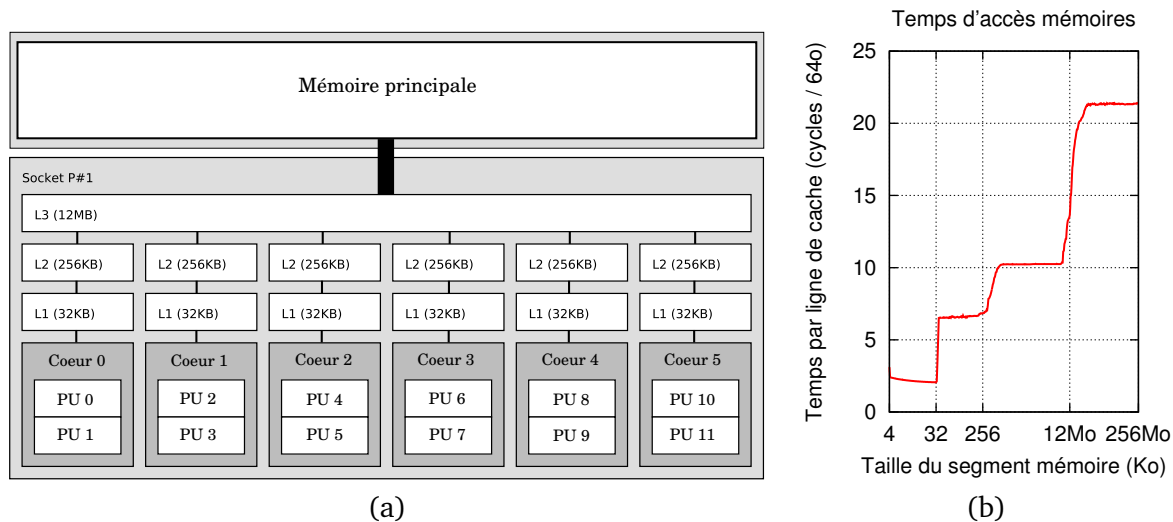


FIGURE 2.5 – Exemple de topologie de l'architecture Intel Westmere utilisée sur les nœuds de la grappe Cassard. Le schéma (a) donne le rattachement et les tailles des caches privés L1 et L2 des différents coeurs et le cache L3 partagé à l'ensemble du processeur. Le graphique de droite (b) donne les temps d'accès par ligne de cache en fonction de la taille du segment lue en boucle.

l'ensemble de la mémoire. Lors d'un accès à une nouvelle donnée, il faut donc *évincer* une donnée présente et supposée non utilisée. De plus, à chaque accès, les copies présentes doivent pouvoir être identifiées pour savoir si la donnée est présente et où. On dispose de trois techniques pour réaliser cette tâche :

Cache entièrement associatif (*fully associative caches*) : Dans cette approche, les lignes de cache peuvent être associées à n'importe quelle adresse, cela suppose que chaque ligne de cache est associée à un registre mémorisant l'adresse des données qu'elle contient. Lors d'une requête, il faut effectuer une recherche exhaustive sur tout le cache pour trouver la ligne correspondante. Cette méthode a donc une complexité proportionnelle à la taille du cache (complexité en $O(s)$ avec s la taille du cache). Cette technique n'est donc utilisée que sur les très petits caches (quelques kilo-octets). Elle se limite en général aux caches de premier niveau et aux TLB (*Translation lookaside buffer*) que nous décrirons dans la prochaine section.

Association directe (*direct caches*) : Chaque adresse est associée à une ligne de cache unique. La recherche est donc très rapide (complexité en $O(1)$). L'association est en général déterminée par les bits de poids faible de l'adresse. Ce mode a toutefois le défaut de créer beaucoup de collisions si les données utilisées sont associées à la même position du cache.

Associatif à N voies (*N-way caches*) : Il s'agit d'un compromis entre les deux approches précédentes. Les collisions sont réduites en associant chaque adresse à N lignes de cache. Pour trouver une donnée, on effectue une recherche exhaustive sur au plus N éléments (complexité en $O(N)$). Les caches processeurs utilisent généralement cette approche pour les caches de tailles importantes. La figure 2.6 représente l'association des segments mémoires avec les lignes de cache de cette approche. Remarquons que ce modèle plus général englobe les deux précédents en considérant respectivement le cas d'une taille de voie égale à une ligne de cache ($N = \frac{s}{64}$) ou une voie unique ($N = 1$).

On notera à titre de complément qu'il est possible de s'abstraire de ces problèmes si les caches sont manipulés explicitement. Ces caches sont dits en terminologie anglophone *scratch-pad*[BSL⁺02]. Le placement des données est alors défini par l'utilisateur ou plus généralement le compilateur.

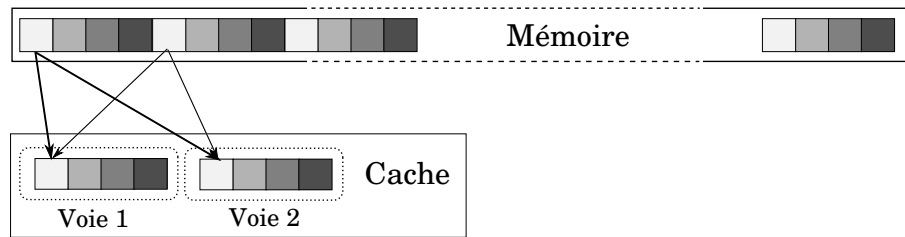


FIGURE 2.6 – Exemple d’associativité à 2 voies montrant les points de stockage possible pour les cellules de couleurs gris clair.

2.5.2 MMU et TLB

La section 2.2.3 a décrit le mécanisme de pagination central pour la gestion de la mémoire. Pour devenir effective, la description en page de l’espace virtuel a besoin d’un composant support au niveau matériel. Ce dernier prend place dans l’unité de gestion mémoire (MMU : Memory Management Unit). La MMU est dédiée à la traduction des adresses virtuelles en adresses physiques. Historiquement, on observe deux tendances. Les processeurs les plus légers relayent les traductions d’adresses au système d’exploitation. Un appel au code noyau est donc nécessaire pour chaque nouvelle traduction. Cette méthode a l’avantage de ne nécessiter qu’un support très simplifié au niveau matériel, mais présente une pénalité en terme de performance. On trouve donc son opposé, retenue sur les processeurs Intel actuels, avec une gestion entièrement matérielle des traductions à partir de la table des pages mise en place par l’OS. Notons l’existence d’architectures mixtes permettant l’utilisation de l’un ou l’autre des modes (par exemple Itanium).

Nous avons vu que la table des pages était constituée d’un arbre. Une traduction nécessite donc l’accès à plusieurs entrées mémoires ou à un appel logiciel à l’OS, rendant la traduction coûteuse. Afin de contrebalancer cet effet, les processeurs disposent d’un cache (TLB : *Translation lookaside buffer*). Il est ainsi possible de garder en mémoire un nombre réduit de traductions afin de s’y référer rapidement en cas de réutilisation. Une page contient un nombre important de données. On peut donc compter sur le fait qu’un accès linéaire ne nécessitera qu’une seule traduction pour l’ensemble des accès de la page avant de passer à la suivante.

L’existence des TLB a une contrepartie pour l’OS qui doit invalider leur contenu en cas de modification de la table des pages. Ces invalidations engendrent un surcoût associé à toute manipulation de la table des pages, à la fois pour exécuter l’invalidation elle-même, mais aussi parce qu’elle nécessite une nouvelle traduction d’adresse de la part de l’application lorsqu’elle reprend le contrôle. Certains processeurs disposent donc de techniques permettant de n’invalider qu’une partie des entrées et éviter les invalidations globales utilisées par les premières générations.

Dans le cas d’architectures multicœurs/multiprocesseurs, le noyau doit invalider les TLB de l’ensemble des cœurs utilisés [SKR⁺04, VDGR96] par le processus modifiant sa mémoire. On remarquera que les architectures de type SGI proposent des instructions spécialisées permettant d’invalider les TLB d’un processeur voisin [RB03]. Ce n’est toutefois pas le cas des architectures x86_64 proposées par Intel et AMD qui nécessitent l’envoi d’un signal d’exception sur chaque processeur pour donner le contrôle à l’OS. Ce dernier prend alors en charge l’invalidation des entrées TLB si elles concernent le processus en cours. Avec un nombre de cœurs croissant, cette méthode semble être risquée puisque chaque modification de la table des pages nécessite un changement de contexte sur l’ensemble des cœurs.

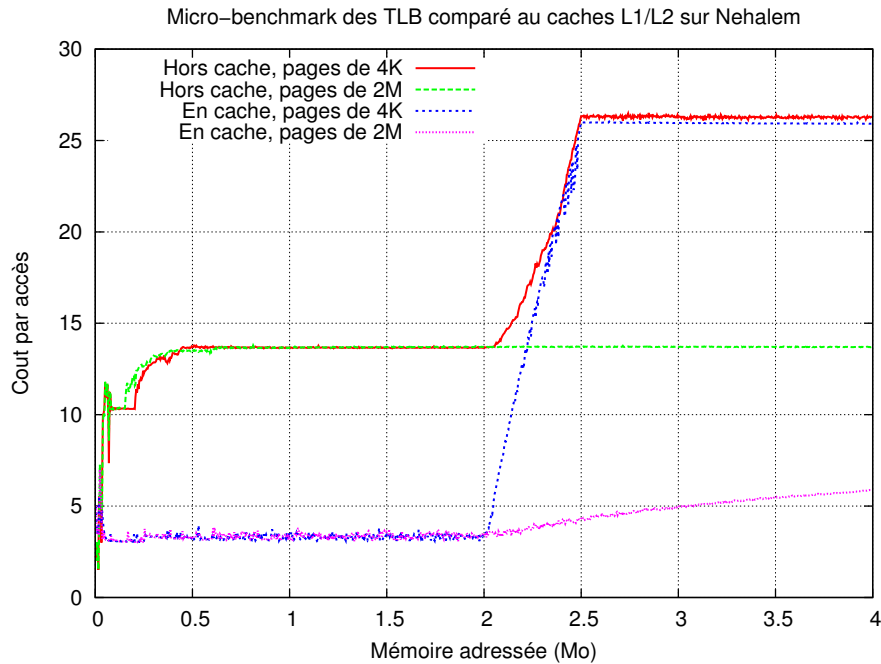


FIGURE 2.7 – Mise en évidence de l’effet des TLB avec leur limite d’adressage à 2 Mo pour les pages de 4Ko. Le coût relatif aux performances des caches est donné en générant des accès dans ou hors des caches L1 (32 Ko) et L2 (256 Ko) mais toujours dans le L3 (8 Mo) de l’architecture Nehalem.

2.5.3 Grosses pages

Les TLB actuels des processeurs Intel ne permettent d’adresser que 2 Mo de mémoire (512 entrées[[Int10a](#)]), c’est à dire bien moins que la mémoire disponible sur les nœuds actuels. Depuis quelques années, c’est également moins que la taille des caches (8 Mo sur Nehalem). Pour remédier à ce problème, il est possible de grossir la taille des pages. On adresse ainsi un espace plus grand avec un nombre d’entrées fixe[[KNTW93](#)]. Ce type de support est offert par les architectures actuelles avec une taille de page de 2 Mo pour x86_64. Il est possible de mettre en évidence l’impact des TLB sur un micro-benchmark en accédant à un élément tous les 4 Ko sur processeur Nehalem avec ses 3 niveaux de caches. Du fait de l’associativité, il est possible de profiter ou non des caches L1/L2 en accédant toujours à la même position dans la page (conflits d’associativité rendant inefficace les caches L1 et L2) ou en variant cette dernière pour profiter du premier niveau de cache. Il est ainsi possible de comparer l’impact relatif visible sur la figure 2.7. Attention, nous considérons ici un unique accès par page de 4Ko simulant plus des accès aléatoires qu’un accès linéaire, le coût relatif des TLB est donc en partie surestimé.

2.5.4 Accès mémoire non uniformes : NUMA

Dans un modèle à mémoire partagée, tous les processeurs ou cœurs d’un nœud doivent accéder à la même mémoire. Dans une approche dite UMA (*Uniform Memory Access*), les processeurs utilisent un même lien physique pour accéder à la mémoire. Cette approche crée une contention au niveau de ce lien d’accès qui ne peut fournir suffisamment de bande passante pour un nombre trop important de cœurs. Bien que compensé pour partie par les caches, ce problème finit par devenir un frein à l’augmentation des performances.

Ce problème peut être contourné en utilisant une approche dite NUMA (*Non Uniform Memory Access*) dans laquelle les différents processeurs d’une machine disposent d’une *mémoire locale* avec un accès rapide. Afin de maintenir un mode mémoire partagée, un mécanisme est

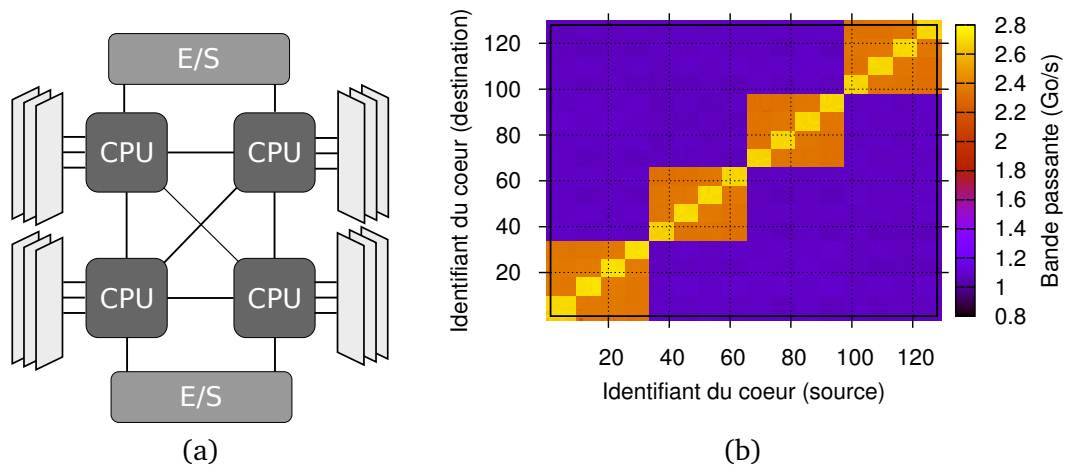


FIGURE 2.8 – (a) Exemple de noeud NUMA composé de 4 processeurs avec 4 liens chacun et deux modules d’entrées/sorties (E/S). (b) Coût effectif du passage par les liens NUMA sur les noeuds 128 coeurs Curie avec leurs 3 niveaux NUMA[VJYA13].

alors mis en place pour accéder automatiquement à une *mémoire distante*. Ces accès se font en passant par le processeur propriétaire au travers de liens d’interconnexions (QPI chez Intel, HyperTransport chez AMD). Les accès se séparent donc en deux catégories, *locaux* et *distants*. Les accès distants sont impactés par une latence plus importante. On augmente donc la bande passante mémoire locale au prix d’une augmentation de la latence des accès distants et du maintien d’une contention sur ces derniers s’ils sont réalisés en trop grand nombre. Sur ces architectures, les programmes doivent veiller à accéder autant que possible à des données locales et limiter les accès distants.

Les nœuds peuvent exploiter différentes topologies structurées en plusieurs *niveaux NUMA* (nombre de sauts à faire pour rejoindre une mémoire donnée) en fonction du nombre de liens offerts par le processeur. La figure 2.8 (a) donne l’exemple d’une topologie à 4 processeurs semblables à ce que l’on trouve sur les nœuds 32 coeurs de Tera 100. Les nœuds 128 coeurs de Curie et Tera 100 exploitent eux un niveau supplémentaire reliant 4 nœuds de 32 coeurs par la technologie BCS (Bull Coherence Switch Architecture) développée par Bull. Cette architecture qui est détaillée en annexe implique 3 niveaux NUMA visibles sur le graphique de bande passante des nœuds larges de Curie : 2.8 (b).

On remarquera que l’approche NUMA permet d’accéder à l’ensemble de la mémoire, mais ne précise pas que la vue de ses données est *cohérente* pour l’ensemble des processeurs s’ils disposent de cache. L’approche non cohérente est difficile à programmer. On utilise donc plutôt des architectures dites NUMA-cc (NUMA Cache Coherent) disposant de mécanismes de cohérence entre nœuds NUMA. Ces mécanismes de cohérences peuvent toutefois être un réel frein à la performance si l’application recourt à eux de manière trop intensive, notamment en actionnant fréquemment des verrous entre nœuds NUMA.

2.5.5 OS et support NUMA

Pour l’exécution de programmes séquentiels, le problème est entièrement traité par le système d’exploitation, et ce de manière transparente pour le développeur et l’utilisateur. Dans ce mode, un processus ne fonctionne que sur un seul processeur à la fois. L’OS doit donc essentiellement veiller à allouer la mémoire localement et à maintenir un ordonnancement du processus

à l'intérieur de son nœud NUMA d'origine. En cas de besoin de migration du processus sur un autre nœud NUMA, il est possible de recourir à une migration de la mémoire. Ce support n'est pas nécessairement optimal dans les OS actuels, mais les informations sont à disposition pour mettre en place une politique correcte.

Dans le cas d'une programmation multi-threads, un processus peut fonctionner simultanément sur plusieurs nœuds NUMA. Dans ce mode, tous les threads partagent la même mémoire, il est donc de la responsabilité du programmeur de limiter au maximum les accès distants. Avec les systèmes d'exploitation courants, le placement en mémoire des pages est malheureusement dépendant d'une interaction *implicite* entre l'OS et le programme. La décision d'utiliser une page d'un nœud NUMA déterminé est en effet prise lors du premier accès à la mémoire (*first touch*). En pratique cela signifie qu'il faut distribuer le travail d'initialisation d'une manière similaire au mode d'accès majoritaire des données. Cette approche n'est toutefois pas nécessairement possible si l'on considère une répartition complexe du travail. Rappelons également que l'allocateur mémoire (*malloc*) tend à réutiliser des segments précédemment libérés. Or, actuellement, aucun allocateur générique ne prend en compte ce paramètre. Ils peuvent donc générer une réutilisation de bloc mémoire précédemment alloué sur un nœud NUMA distant. Les choix de conception de l'allocateur mémoire décrits dans cette thèse seront donc orientés pour prendre en compte cette problématique.

2.6 Conclusion

Nous venons de voir que la conception des architectures mémoires actuelles agrégeait de nombreux concepts pour résoudre le problème du mur de la mémoire et permettre l'implémentation de systèmes d'exploitations fiables et robustes. Dans la suite de ce document, nous allons montrer que certains de ces points méritent d'être revisités en prenant en compte le nombre croissant de cœurs des architectures modernes et l'utilisation à grande échelle d'architectures NUMA. Comme nous venons de le voir, l'OS est l'outil chargé de gérer cette infrastructure, nous allons donc essentiellement nous intéresser à ce dernier et à sa couche d'interaction (en terme mémoire) avec le niveau applicatif (*malloc*).

Deuxième partie

Contribution

La partie précédente a présenté un rapide historique de l'évolution du matériel utilisé pour le calcul haute performance. Elle a également introduit les notions structurantes des architectures modernes et de leur programmation. Cette description s'est terminée par un rappel détaillé des mécanismes d'accès à la mémoire et de gestion de cette ressource par le système d'exploitation. Cette seconde partie va se baser sur ces connaissances pour décrire les travaux attachés aux quatre problématiques majeures abordées durant cette thèse.

Nous donnerons dans un premier temps les résultats obtenus lors de notre étude du comportement collaboratif du système de pagination de l'OS et de l'allocateur mémoire en espace utilisateur, au vu de leur impact sur les performances des caches. Le second chapitre se concentrera sur l'allocateur lui-même en tâchant de prendre en compte les problématiques de passage à l'échelle sur architecture NUMA. Ce chapitre donnera une description de l'allocateur conçu dans le cadre du projet MPC et pointera notamment un problème d'extensibilité de l'implémentation actuelle des fautes de pages du noyau Linux. Le troisième chapitre se focalisera sur ce problème et étudiera les mécanismes de fautes de pages de cet OS en proposant une modification de sa sémantique d'échange avec l'espace utilisateur. Nous montrerons qu'il est ainsi possible d'éliminer les effacements mémoires coûteux imposés par la sémantique standard. Comme la consommation mémoire constitue actuellement un enjeu, nous terminerons par une analyse du module noyau KSM (Kernel Samepage Memory) offert par les noyaux Linux récents. Nous décrirons un test de cette technique sur un maillage de l'application Hera afin de réduire son empreinte mémoire en fusionnant les pages au contenu identique.

Chapitre 3

Interférences des mécanismes d'allocations

Ce chapitre décrit une étude préliminaire réalisée sur l'interaction, et plus précisément les interférences, des politiques d'allocation en espace noyau et utilisateur. Ici, nous nous intéresserons principalement à la gestion des gros tableaux mémoires et à leur agencement dans les caches du processeur. Le problème traité peut se découper en deux sous-problèmes : d'un côté, l'impact de la politique de pagination vis-à-vis des caches, de l'autre, l'impact des choix de l'allocateur sur ces mêmes caches. Ces deux problèmes pris séparément sont très largement discutés dans la littérature. On ne trouve toutefois que peu de discussions sur l'interaction des politiques proposées. En ce qui concerne l'allocateur mémoire, les études liées aux caches sont habituellement centrées sur la problématique des petites allocations, négligeant le problème spécifique des gros segments. Ici, notre contribution sera donc centrée sur l'analyse des interférences potentielles entre les stratégies retenues au niveau de l'OS et celles de l'allocateur mémoire en espace utilisateur. Cette analyse sera également l'occasion de discuter la politique de pagination de LINUX sous un angle nouveau, donnant des arguments originaux en faveur de l'approche, certes améliorable, retenue par cet OS vis-à-vis de sa robustesse aux cas pathologiques.

Dans un premier temps, nous rappellerons les liens connus qui s'établissent entre pagination et associativité des caches. Nous pourrions alors lister les différentes approches actuelles exploitées par les OS disponibles. Suivra une partie expérimentale permettant de fournir les données nécessaires à notre étude. Pour cela, nous étudierons les performances de différentes applications sur plusieurs OS afin de comparer leurs politiques. S'en suivra une analyse détaillée des différents problèmes répertoriés. Il nous sera alors possible d'extraire de nouvelles recommandations concernant les aspects OS et espace utilisateurs de la gestion mémoire. Nous décrirons enfin un programme prototype permettant de repérer certains des problèmes de notre inventaire et une exploration de méthode de résolution pour la partie espace utilisateur.

3.1 Pagination et associativité

Nous avons rappelé dans le chapitre 2 que les caches processeurs actuels étaient généralement *associatifs*. Nous avons ainsi vu que cette structure implique des restrictions quant à la manière de placer les données. Ce chapitre a également introduit le double système d'adressage (*physique* et *virtuel*) mis en place par le système d'exploitation. Dans ce contexte, les caches matériels peuvent être *indexés* sur la base des adresses *physiques* ou *virtuelles*. Nous allons voir ici les conséquences que cela peut avoir pour le développeur.

Une *indexation virtuelle* est habituellement retenue pour le premier niveau de cache. Ce dernier ne peut en effet pas se satisfaire de la latence induite par le mécanisme de traduction

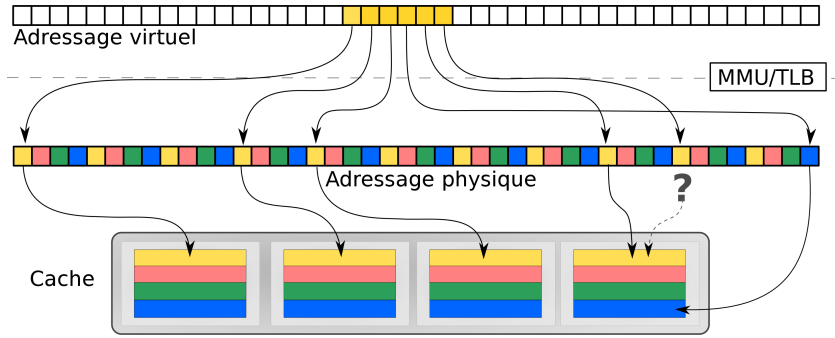


FIGURE 3.1 – Exemple de conflit de cache lié à un placement non optimal des pages en considérant un cache associatif à 4 voies indexées physiquement.

d'adresse [LBF92, CD97]. Dans ce cas de figure, la prise en compte de l'associativité du cache reste de la responsabilité du développeur ou de l'allocateur mémoire. On trouve ainsi nombre de travaux visant à optimiser le placement des données en espace virtuel vis-à-vis de ce type de cache : [CHL99, HBHR11]. Dans ce cas, la position p_{cache} d'une donnée dans le cache peut être modélisée par l'équation 3.1 en considérant S la taille du cache, A son associativité, $p_{virt.}$ l'adresse virtuelle de la donnée et $W = \frac{S}{A}$ la taille d'une voie du cache :

$$p_{cache} \equiv p_{virt.} \pmod{W} \quad (3.1)$$

À l'opposé, une *indexation physique* fait intervenir la table des pages dans le processus. En conséquence, le sous-système mémoire de l'OS devient un paramètre entre l'application et le cache. L'OS décide en effet de la position physique d'une page virtuelle. Il choisit donc par la même occasion le placement correspondant dans le cache associatif si ce dernier fournit des voies plus grandes qu'une page. Considérons B l'adresse physique de la base de la page, U la taille d'une page et $o \equiv p_{virt.} \pmod{U}$ le décalage dans la page. L'adresse physique $p_{phys.}$ d'une donnée se définit alors comme $p_{phys.} = B + o$. La position dans le cache devient donc dépendante de B :

$$p_{cache} \equiv p_{phys.} \pmod{W} \equiv (B + o) \pmod{W} \quad (3.2)$$

Les voies du cache ont habituellement une taille multiple de la taille de page. On peut donc définir la *couleur* c d'une page comme sa position dans un cache indexé physiquement. Cette couleur dépend uniquement de la table des pages au travers de B :

$$c = \lfloor \frac{p_{cache}}{U} \rfloor \equiv \lfloor \frac{B \pmod{W}}{U} \rfloor \quad (3.3)$$

On notera que dans un cache associatif à A voies, l'utilisation de $A + 1$ pages de la même couleur génère un *conflit* ; la dernière page devant occuper l'une des places prises par les A autres pages. Le problème peut-être illustré en considérant un segment continu en mémoire virtuelle et couvrant plusieurs pages. Du fait de la pagination, nous avons vu au chapitre 2 que la mémoire physique associée n'avait aucune raison d'être contiguë. En considérant un cache A fois associatif on peut donc se trouver dans la situation décrite par la figure 3.1. Dans cette situation, l'accès à la dernière page du segment conduira nécessairement à l'éviction d'une des pages précédemment chargées dans le cache. Les accès successifs à ce segment seront donc pénalisés par des transferts mémoires, même si ce dernier aurait pu être placé entièrement dans le cache.

Ce problème a été largement étudié et décrit, on pourra par exemple citer [AHH89, KH92, BKW98] ou certains travaux plus récents avec une reprise d'activité de ce sujet [HK, PTH11, ADM11]. Il peut-être décrit avec une approche statistique en considérant un choix aléatoire des pages physiques lors de l'allocation. Un cache associatif offre $C = \frac{W}{U}$ couleurs possibles. Si les

pages sont choisies de manières aléatoires et indépendantes, alors la probabilité d'obtenir une couleur c est :

$$p = \frac{1}{C} \quad (3.4)$$

Ce problème de tirage aléatoire de N éléments parmi C types d'éléments se résout mathématiquement sous la forme d'une loi binomiale donnant une probabilité d'obtenir k éléments d'une couleur donnée :

$$P(N, X = k) = \binom{N}{k} \left(\frac{1}{C_{max}} \right)^k \left(1 - \frac{1}{C_{max}} \right)^{(N-k)} \quad (3.5)$$

On peut donc évaluer le nombre de fautes de cache en considérant les couleurs présentes en quantité supérieure aux A voies du cache :

$$f(N) = C_{max} \sum_{k=A}^N (k - A) P(N, k) \quad (3.6)$$

Cette description prend en compte la part de faute liée au manque de *capacité* du cache qui peut s'exprimer plus simplement en considérant un remplissage idéal :

$$f_{capa.}(N) = \max(0, N - AC_{max}) \quad (3.7)$$

On peut donc extraire la part de faute induite par la politique de pagination aléatoire comme étant :

$$f_{poli.}(N) = C_{max} \sum_{k=A}^N (k - A) P(N, k) - f_{capa.}(N) \quad (3.8)$$

Pour être calculé en pratique, on peut utiliser la définition de l'*espérance*¹ de P et de sa *norme*² pour remanier l'équation sous une forme dépendante de la *fonction de répartition* connue par ailleurs. Cette forme est valide pour pour $N > A$, le nombre de conflits étant strictement nul pour $N \leq A$:

$$f_{poli.}(N) = C_{max} \left(Np - \sum_{k=0}^A kP(N, k) - A \left(1 - \sum_{k=0}^A P(N, k) \right) \right) - f_{capa.}(N) \quad (3.9)$$

Cette évaluation théorique est comparée à l'expérience dans la figure 3.2 de la section suivante en considérant un accès linéaire à un tableau mémoire et les paramètres du cache L3 des processeurs INTEL NEHALEM, à savoir une associativité de 16 pour une taille de 8 Mo.

3.2 Politiques de pagination

La section précédente vient de rappeler un des problèmes inhérents à la mise en place d'un système de pagination sur une architecture disposant de caches associatifs. Nous allons discuter ici les différentes approches actuellement disponibles dans les systèmes d'exploitation de production (Linux, FreeBSD, et OpenSolaris) vis-à-vis de ce problème.

3.2.1 Pagination aléatoire : Linux

Sous Linux, les demandes de pages sont satisfaites avec la première page disponible dans les listes de son gestionnaire de pages[BP05]. Il en résulte que l'OS ne prête aucune attention à la distribution de couleur des pages physiques qu'il projette dans l'espace virtuel. Nous utiliserons

1. $\sum_{k=0}^N kP(k) = Np$
2. $\sum_{k=0}^N P(k) = 1$

le terme de *pagination aléatoire* pour désigner cette approche.

Comme rappelé dans la section 3.1, ce type de politique génère des conflits de caches directement imputables à la politique de placement des pages et non à un manque de place dans le cache. Afin de confirmer ces observations sur les larges caches actuels, nous avons comparé la politique de Linux à une politique simulée strictement aléatoire et à la prédiction théorique de la section 3.1. Pour cela, nous avons alloué un segment de taille voulue et effectué une initialisation de son contenu en séquentiel ou parallèle. La table des pages de Linux est alors lue afin de calculer le nombre de conflits pour un cache de taille et associativité fixées en considérant un accès linéaire aux données. Cette mesure est finalement comparée à une table aléatoire générée par nos soins.

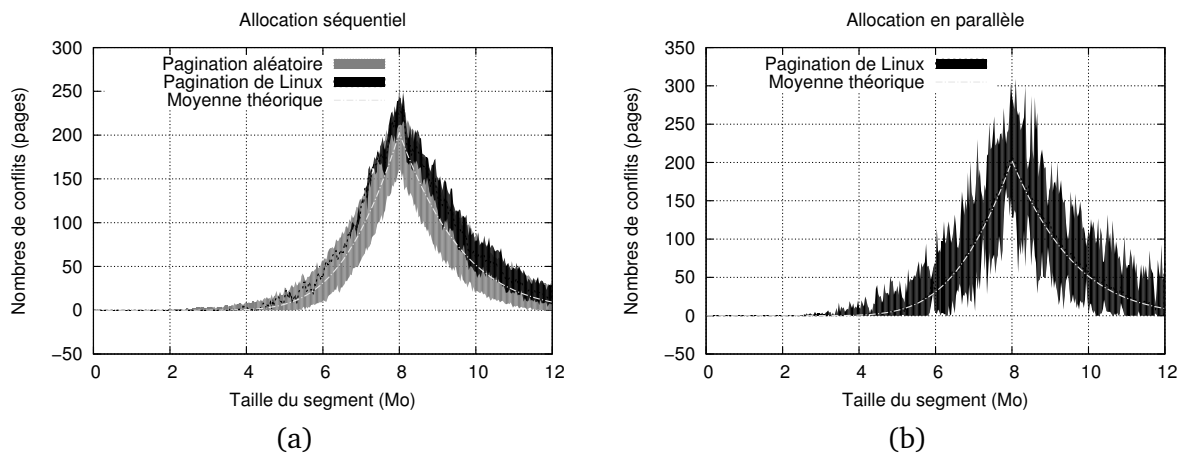


FIGURE 3.2 – Comparaison des conflits de caches provoqués par une table des pages générée par Linux et une table aléatoire. La simulation considère le cache L3 d’un processeur Nehalem, soit 8Mo partagés en 16 voies. Le graphique rassemble 500 exécutions pour chaque taille mémoire. Il représente les minimums et maximums de conflits obtenus en fonction de la taille du tableau considéré. À gauche (a), une mesure séquentielle, à droite (b) en parallèle.

La figure 3.2 montre clairement que Linux réagit d’une manière proche de la méthode aléatoire en terme de conflits de cache. Nous confirmons également l’observation de Bahadur[BKW98] en montrant que la table aléatoire fournit des résultats plus dispersés que la table générée par Linux. Ce phénomène s’explique par les relations pouvant intervenir entre deux allocations successives. Les pages libres sont en effet gérées par un allocateur dit “buddy allocator” groupant les pages contiguës par blocs allant jusqu’à 2 Mo. Les allocations des pages de ces blocs ne vérifient donc pas une distribution strictement aléatoire.

On peut également remarquer que les allocations réalisées en parallèle conduisent à une distribution identique à une table aléatoire. Ceci confirme les observations l’étude de Bahadur sur des charges de travail plus intensives en allocation mémoire avec notamment des tests utilisant plusieurs applications. Nos observations ne montrent aucun changement en fonction de la durée de fonctionnement de l’OS. Avec 24Go de mémoire, nous observons strictement les mêmes effets immédiatement après le démarrage alors que l’OS consomme de l’ordre de 100 Mo de mémoire, pouvant laisser espérer moins d’aléas dans la sélection des pages dans un grand espace. La politique de pagination de Linux peut donc raisonnablement être considérée comme *aléatoire* vis-à-vis des caches associatifs.

La figure 3.3 confirme l’impact de cette politique sur les performances en considérant un

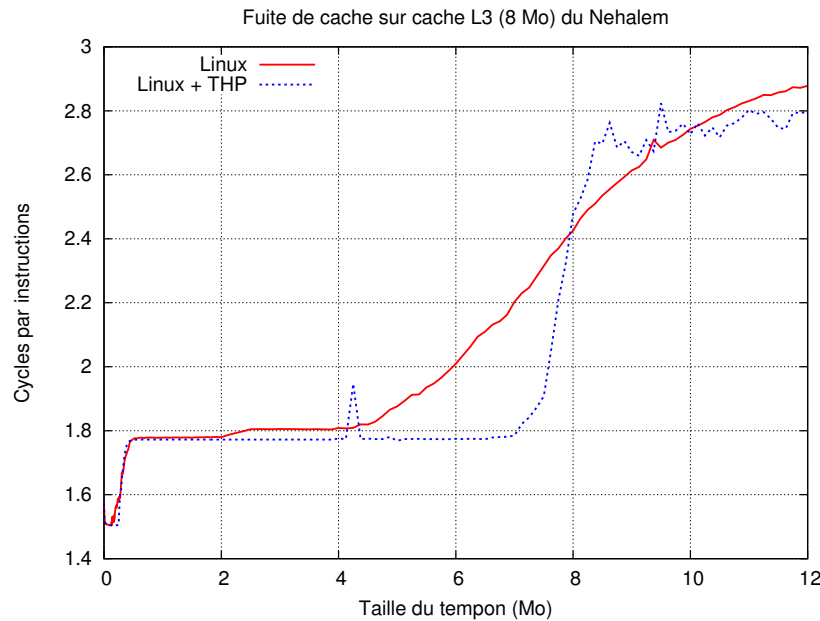


FIGURE 3.3 – Mesure expérimentale de l'effet de fuite de cache sur les temps d'accès la mémoire. On considère ici un accès répété et linéaire à un tableau sur un processeur INTEL NEHALEM avec ou sans support des grosses pages de Linux (THP : Transparent Huge Pages).

accès séquentiel répété à un tableau sur INTEL NEHALEM. Sur ce type de processeurs, on note un début de fuite du cache à partir de 5 Mo (62%) causé par la pagination aléatoire de LINUX. Lors de l'utilisation de techniques de type *cache blocking*, le développeur ne doit donc pas oublier que la politique de pagination de Linux ne permet pas une utilisation complète des caches indexés physiquement. On remarquera que le développeur ne peut pas facilement compenser cet effet au niveau espace-utilisateur et devra donc éventuellement considérer un cache plus petit pour obtenir de bonnes performances.

Ce problème bien que connu est rarement discuté et pris en compte dans les modélisations concernant les techniques de *cache blocking* [LRW91]. Il est également courant de présenter des figures similaires à la figure 3.3 en utilisant des échelles logarithmiques pour les tailles mémoires. Couplé à un échantillonnage des points de mesure en puissance de deux, ce type d'approche tend à masquer le problème en occultant la perte des derniers 40% du cache. Il convient donc d'être prudent lorsque l'on analyse ce type de graphique.

3.2.2 Coloration de pages

Les articles [KH92, BKW98] décrivant le problème de conflits de caches liés à la politique de pagination critiquent les approches de type aléatoire tels qu'utilisées dans LINUX. Ils proposent donc des solutions prenant en compte la présence de caches associatifs au niveau de l'implémentation du mécanisme de pagination de l'OS. Ces approches visent à sélectionner les pages physiques de sorte à assurer une distribution égale des pages sur l'ensemble du cache. Ces méthodes sont généralement dites *coloration de pages* et reposent entièrement sur les mécanismes internes à l'OS. Un placement idéal pourrait être établi si l'on connaissait à l'avance le schéma d'accès aux données. Ce type d'information ne peut toutefois pas être connu dans un cadre général, l'OS doit donc se baser sur des heuristiques telles que listées ici :

Hashage des adresses virtuelles : Cette méthode consiste généralement à appliquer un modulo de la taille d'une voie (W) sur l'adresse virtuelle nécessitant une page physique. Le

nombre obtenu constitue alors la couleur de la page physique à sélectionner. L’utilisation d’un unique modulo multiple du nombre de couleurs génère une association directe entre les adresses virtuelles et physiques en terme d’associativité du cache. Ainsi, le choix d’une adresse virtuelle implique directement le choix de la position dans le cache. Cela se démontre en reprenant l’équation 3.2 avec $B = p_{virtu.} \pmod{W}$. Dans ce cas, les modulus liés à la pagination et au placement dans le cache se compensent et font disparaître le terme de pagination :

$$p = (B + o) \pmod{W} = p_{virtu.} \pmod{W} \quad (3.10)$$

Les différentes instances (processus) d’un même programme ont tendance à exploiter les mêmes adresses pour certains éléments du programme (pile, tas, constantes...). Ces derniers tendent donc à générer des conflits inter-processus au niveau des caches partagés. Il est donc possible d’ajouter [KH92] un décalage basé sur l’identifiant du processus (PID) pour éliminer ces conflits.

Tourniquet : Le système se rappelle de la couleur utilisée lors de la précédente faute de page et incrémente cette dernière à chaque faute de page. Cette méthode permet d’obtenir une certaine adaptation en capturant le schéma du premier accès mémoire, supposant donc que les suivants seront proches de ce dernier.

Équilibré : Le système compte l’utilisation des différentes couleurs et tente d’allouer les moins utilisées. Dans cette approche, il est aussi possible de prendre en compte le nombre de pages libres pour chacune des couleurs comme cela est discuté par [KH92]. Cette méthode est surtout conçue pour mieux s’adapter aux contextes de pénurie mémoire.

Les OS OPENSOLARIS[MM06] et FREEBSD offrent un support de ces méthodes. Le premier permet de choisir parmi les deux premiers modes (avec ou sans ajout du PID). Ce choix n’est toutefois disponible que sur architecture SPARC, le support x86 se limitant au mode de hachage sans ajout du PID. Le deuxième génère implicitement un support de la méthode de hachage sans PID de par son implémentation des *superpages*[NIDC02] discutée dans ce qui suit. Remarquons l’existence de certains travaux visant à exploiter des informations fournies par le compilateur [BAM⁺96] ou par des compteurs matériels [SCE99, RLBC94, BLRC94] pour améliorer la politique de coloration. Ces méthodes ne sont toutefois supportées par aucun OS de production.

3.2.3 Grosses pages

Nous l’avons discuté dans le chapitre 2, à chaque accès, le processeur doit traduire les adresses virtuelles en adresses physiques. Comme décrites, ces traductions sont accélérées par la présence d’un cache dédié : le TLB. Sur architecture NEHALEM, on remarquera que ce cache peut contenir 512 entrées[Int10a] adressant un maximum de 2 Mo en considérant des pages de 4 Ko. Ceci est bien moins que la taille du cache de dernier niveau de cette architecture : 8 Mo. L’effet du TLB est donc observable sur la figure 3.3 avec un léger renflement pour les pages standards à 2 Mo. Cet effet est amplifié dans le cas d’accès aléatoires à la mémoire, ne permettant pas de réutiliser les entrées du TLB.

Pour résoudre ce problème, certains processeurs offrent la possibilité d’utiliser des pages de tailles supérieures dites *grosses pages*. De cette manière, il est possible d’adresser efficacement un espace plus grand sans augmenter la taille des TLB. Remarquons que, comme cela est rappelé dans [KNTW93], il n’est pas raisonnable d’exploiter ce type de pages de manière générale pour tout l’OS. Une trop grande taille peut en fait impacter les sous-systèmes entrées/sorties qui risquent de générer beaucoup plus de trafic de données. Ce type de pagination s’ajoute donc en complément des pages standard de 4 Ko.

Sur les architectures `x86_64`, il est possible d'utiliser des pages de 2 Mo ou 1 Go [Int10a]. L'implémentation de ce double support présente toutefois quelques difficultés notamment liées à l'utilisation de deux tailles différentes réintroduisant les problèmes de fragmentation au niveau de la pagination [GH12]. Les méthodes de résolution actuelles dépendent donc fortement des OS considérés. On trouve chez FreeBSD l'une des méthodes natives les plus abouties dites *superpages* [NIDC02]. OpenSolaris offre un support bien intégré, bien que moins avancé par rapport à la méthode de FreeBSD. Linux quant à lui, a longtemps maintenu son support sous une forme très externe au gestionnaire principal, rendant son utilisation difficile et très contraignante. On trouve toutefois de nombreux travaux ces dernières années visant à améliorer ce support, avec notamment le travail de Ian Wienand [Wie08, CWHW03] en 2008 ou de K. Yoshii nommée *big memory* pour Linux sur Blue Gene [YIN⁺11]. Pour cet OS, on citera entre autre le travail de Andreas Arcangeli [Arc10] (Transparent Huge Pages) désormais inclu officiellement dans le noyau de RedHat 6 (2.6.32) et dans les branches officielles depuis la version 2.6.38. On pourra trouver une étude complémentaire concernant les grosses pages dans domaine du HPC avec l'étude de Zhang [ZLHM09] ou des propositions intermédiaires facilitant l'intégration aux OS par Talluri [TH94].

En terme d'associativité, les grosses pages génèrent de par leur définition une forme de coloration de page similaire à ce qui est obtenu par une méthode de hachage avec un simple modulo. Les processeurs actuels disposent de caches de l'ordre de 8 Mo pour une associativité de 16, correspondant à une taille de voie de 512 Ko. Les grosses pages ont une taille supérieure (2 Mo) et doivent être alignées en mémoire sur leur propre taille. On obtient donc naturellement un placement en cache qui ne dépend que de la taille d'une voie et de l'adresse virtuelle. Cela peut se démontrer en reprenant la définition de placement des données en cache (eq. 3.2) en considérant U_{huge} la taille d'une grosse page tel que $U_{huge} \geq W$ et $U_{huge} \pmod{W} = B_{huge} \pmod{W} = 0$.

$$p_{cache} = p_{phys.} \pmod{W} = (B_{huge} + o_{huge}) \pmod{W} = p_{virt.} \pmod{W} \quad (3.11)$$

Plus simplement, cela correspond à remarquer qu'une grosse page couvre un nombre entier de voie. Cette propriété implique une relation directe entre adresse virtuelle et physique en terme d'associativité. On remarquera de plus que l'aspect attentif aux caches des grosses pages repose sur leur définition matérielle, non logicielle. Il n'y a donc aucun moyen d'appliquer des règles de pagination pour modifier leur impact sur l'associativité des caches. Des techniques telles que l'ajout du PID comme décalage sont donc inapplicables.

3.2.4 Conclusion

Dans cette section, nous avons vu différentes politiques de pagination et leurs interactions avec les caches associatifs. Sur le plan théorique, Linux semble désavantagé par rapport à FreeBSD et OpenSolaris qui recourent explicitement à des politiques qui prennent en compte les caches. Dans ce qui suit, nous allons comparer expérimentalement ces techniques et montrer que l'approche aléatoire présente tout de même certains intérêts.

3.3 Résultats expérimentaux

Comme discuté précédemment, chaque système d'exploitation propose des politiques de pagination différentes. On peut donc s'interroger sur l'impact de ces dernières sur des applications intensives en terme de calcul. Cette section s'intéresse donc à la comparaison des effets de ces politiques sur diverses applications MPI et OpenMP.

3.3.1 Protocole expérimental

Pour ces expériences, nous visons la comparaison des politiques de trois OS de classe production : OPENSOLARIS, FREEBSD et LINUX. Ces derniers exploitent des techniques différentes permettant l’exploration des différentes possibilités avec tout de même certains recouvrements possibles. Il nous sera ainsi possible d’analyser l’impact de ces choix. Afin d’éliminer les effets annexes, nous avons tenté de limiter au maximum les autres paramètres intervenant dans le protocole. Toutes les expériences ont été réalisées sur une unique station décrite par la table 3.1. Comme OPENSOLARIS ne supportait pas un démarrage en mode NUMA nous avons désactivé ce mode au niveau du BIOS pour fonctionner en mode *entrelacé*³ pour tous les OS. Cela permet également d’éliminer la question de qualité de support NUMA entre ces derniers, point ne nous intéressant pas spécialement pour ce chapitre.

Dell T5500	
Processeur	Bi Intel Xeon-E5502 (Nehalem)
Fréquence	2.27 GHz
Cache L3	8 Mo, 16 voies
Cache L2	256 Ko, 8 voies
Cache L1	32 K, 8 voies
Mémoire	24 Go

TABLE 3.1 – *Plaforme de test utilisée pour l’évaluation des politiques de pagination.*

Au niveau logiciel, nous avons essayé de fixer au maximum les composants utilisés de manière à garder autant que possible le noyau comme unique variable. Nous avons donc utilisé le compilateur GCC-4.4.1 recompile manuellement sur chaque système. De la même manière, nous avons fixé les bibliothèques nécessaires aux benchmarks utilisés tels que BINUTILS-2.20, GMP-5.0.1, MPFR-2.4.2, LIBATLAS-3.8.3 et MPICH2-1.2.1p1. Chacune de ces dépendances est recompilée si possible avec les mêmes options pour l’ensemble des systèmes testés. Tous les benchmarks ont ainsi été compilés par les outils précédemment décrits en utilisant les options de compilation suivantes, nécessaires pour assurer un mode de compilation valide et similaire sur chacun des trois OS : `-m64 -fomit-frame-pointer -O3 -march=core2`.

En procédant ainsi, à part pour la LIBC, nous laissons le noyau comme principale variable de manière à comparer LINUX FEDORA 16 (noyau 2.6.38), OPENSOLARIS 2009.06 et FREEBSD 8.2, chacun de ces OS fonctionnant en mode 64 bits. Nous avons également cherché à limiter l’impact des ordonnanceurs de ces OS en limitant au maximum les processus en fonctionnement pendant les tests (serveur graphique...). En ce qui concerne les benchmarks, nous avons sélectionné les NAS[BBB⁺91][JJF⁺99] et EulerMHD, un solveur de magnétohydrodynamique 2D sur maillage cartésien développé en MPI[DEJ⁺10]. Les sections suivantes donnent donc les résultats obtenus sur ces applications.

3.3.2 Résultats des NAS séquentiels

Les benchmarks NAS (version 3.2)[BBB⁺91][JJF⁺99] fournis par la NASA sont constitués d’une suite de programme reprenant certains types de résolutions de problèmes classiques rencontrés en HPC. On en trouve des versions MPI et OpenMP. Le benchmark DC a été mis de côté car s’intéressant plus aux performances du système de fichier ne nous intéressant pas ici.

3. Dans ce mode, la mémoire est découpée de sorte qu’une ligne de cache sur deux soit sur un nœud NUMA différent. Ceci assure donc une distribution statistiquement égale sur chacun des nœuds afin de masquer leur présence.

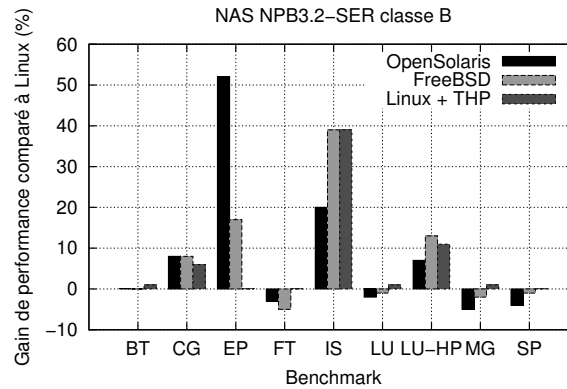


FIGURE 3.4 – Gains de performances relatifs à Linux obtenus sur la série de benchmarks NAS-SER. Sur ce graphique, les valeurs positives montrent des gains de performances.

Ces benchmarks ont dans un premier temps été exécutés en mode séquentiel. Les résultats obtenus et confirmés par plusieurs exécutions (de l'ordre de cinq) sont donnés dans la figure 3.4. Cette dernière représente les gains de performances relativement à Linux avec des pages standards. On observe une amélioration des performances en faveur de FreeBSD et OpenSolaris comparé à Linux. Les gains obtenus allant de 7% à 51%. Le benchmark EP est particulièrement sensible à ces effets avec une amélioration des performances de 51% sur OpenSolaris et 19% sous FreeBSD. Les benchmarks dont les performances sont réduites sur ces OS ne voient en comparaison leurs performances diminuer de 5% sous FreeBSD et OpenSolaris.

Le graphique donne également des résultats obtenus avec les *Transparent Huge Pages* (THP) disponibles sous Linux. Ces dernières ont été utilisées en activant leur support en mode permanent, impliquant l'usage de grosses pages de manière implicite pour toutes les allocations. La figure 3.4 montre clairement la corrélation des résultats de cette politique de pagination avec ceux obtenus sur FreeBSD. On notera toutefois que l'on n'obtient pas strictement la même politique puisqu'elle ne s'applique que pour les segments plus larges qu'une grosse page (2 Mo) et alignés sur ces dernières. Pour les parties de segment non entièrement couvertes par de telles pages, Linux maintient sa politique d'allocation aléatoire qui diffère des autres OS. Ceci explique les différences de résultats observées sur les benchmarks EP et IS.

En ce qui concerne le mode séquentiel, on note une amélioration moyenne des performances de 7% par rapport aux pages standards de Linux. On montre donc que la politique de Linux semble pouvoir être améliorée. Comme les NAS sont centrés sur l'utilisation du processeur, nous pouvons supposer que les écarts observés viennent de l'ordonnanceur ou du système de gestion mémoire, non du système d'entrées/sorties. Comme nous avons limité l'activité de l'ordonnanceur, nous pouvons raisonnablement supposer que le gestionnaire mémoire est l'une des sources principales des effets observés. Cette supposition est renforcée par les corrélations obtenues sur les résultats avec les grosses pages de Linux.

Par défaut, OpenSolaris n'utilise pas de grosses pages, mais uniquement une politique de coloration. Comme nous observons aussi des améliorations notables sur cet OS, nous pouvons admettre que la réduction des TLB n'est pas l'effet dominant dans nos observations. Nous supposons donc que les effets proviennent de l'interaction de la politique de pagination avec les caches associatifs. Nous verrons dans la suite que cet argument est largement confirmé par les résultats obtenus avec l'application EulerMHD.

3.3.3 Résultats des NAS parallèles

La figure 3.5 fournit les résultats obtenus avec les NAS en mode MPI sur 8 processus. Dans cette configuration, on peut observer une dégradation importante des performances avec des ralentissements de l'ordre de 30%. Ce comportement peut-être également observé avec les classes S, W et A en activant les grosses pages sous Linux, éliminant une fois de plus les politiques d'ordonnancement comme paramètre majeur. Les mêmes résultats sont obtenus en utilisant *hw-loc*[BCOM⁺10] pour verrouiller les threads et processus sur les coeurs de manière déterministe.

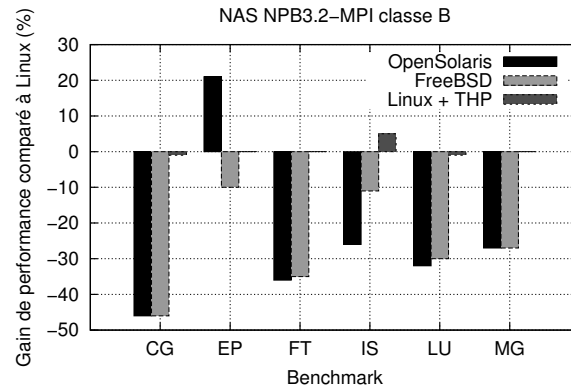


FIGURE 3.5 – Gains de performances des NAS-MPI pour 8 processus comparés aux pages standards de Linux. Les valeurs positives montrent des gains de performances.

Afin d'obtenir une vue plus synthétisée des résultats, on peut grouper les benchmarks et regarder les impacts extrêmes (amélioration maximum ou dégradation maximum) obtenus en fonction du nombre de coeurs exploités. Ces résultats sont mis en forme dans la figure 3.6 montrant une corrélation nette entre les différentes politiques de pagination. Avec un nombre croissant de flux d'exécution, on observe que les différents OS tendent à favoriser et à dégrader les cas pathologiques alors que les gains potentiels tendent à se réduire, principalement en ce qui concerne les grosses pages. Ce point particulier critiquant l'impact des grosses pages ou de certains types de coloration sera discuté plus en détail dans la suite à la lumière des résultats obtenus avec l'application EulerMHD.

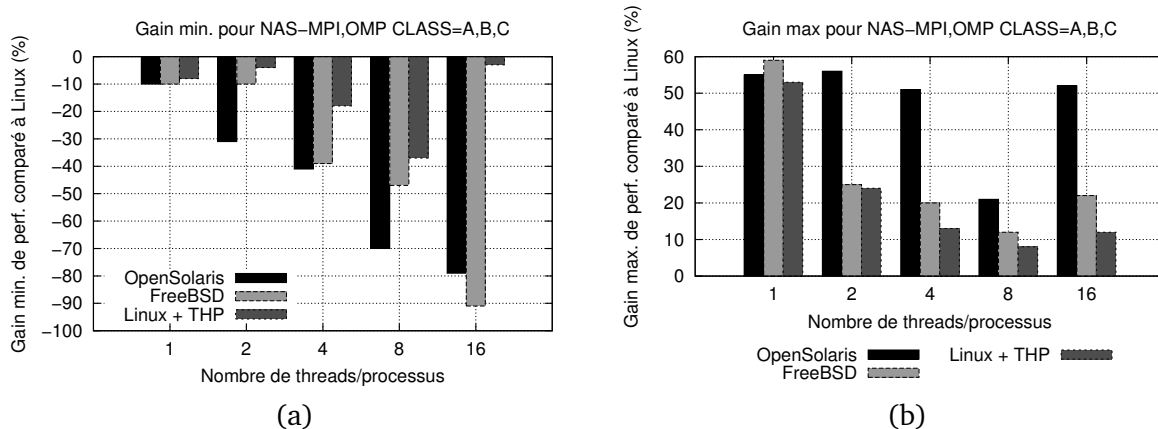


FIGURE 3.6 – Représentation des (a) gains et (b) pertes extrêmes de performances pour l'ensemble des benchmarks MPI en fonction du nombre de threads/processus utilisés. Ces résultats agrègent les classes A,B et C. Des valeurs positives impliquent des gains de performances.

On remarquera que les effets des grosses pages de Linux sont systématiquement en retrait par rapport aux résultats des autres OS. Cela s'explique par les effets de seuils liés à leur intégration partielle dans l'OS et le maintien d'une pagination aléatoire en dessous de ces seuils.

3.3.4 Résultats sur EulerMHD

Afin de tester une application plus complexe, nous avons retenu un solveur de magnétohydrodynamique 2D sur maillage cartésien avec des schémas d'ordres élevés : EulerMHD[DEJ⁺10, Wol]. Les premiers résultats obtenus avec cette application sont présentés dans le graphique 3.7. On peut y observer une amélioration de 4% avec les grosses pages, mais une forte dégradation des performances sous FreeBSD. De manière surprenante, la fonction affectée contient uniquement une boucle de calcul intensif sans appel système.

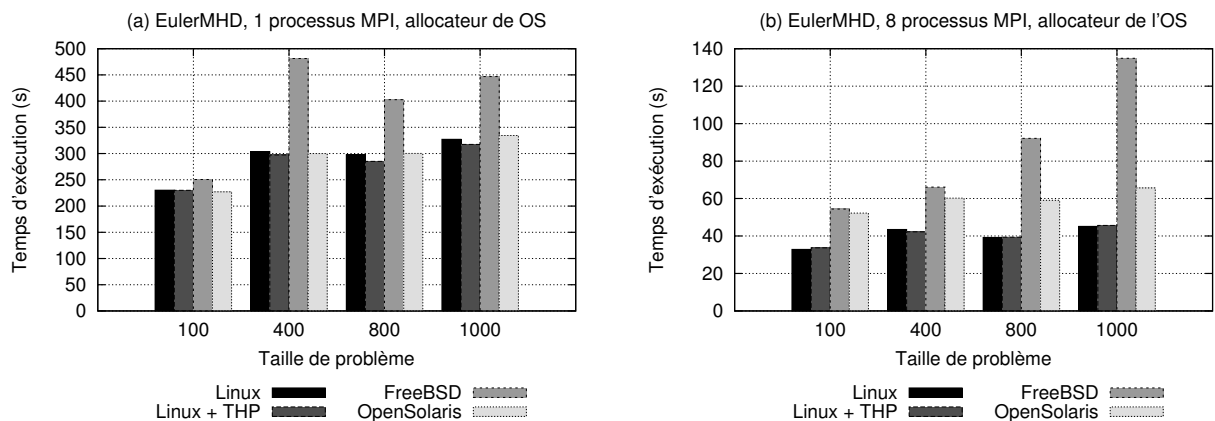


FIGURE 3.7 – Temps d'exécution de l'application EulerMHD sur les différents OS (a) en séquentiel (b) avec 8 processus MPI.

Ce phénomène de dégradation de performance va être analysé et expliqué en détail dans les sections suivantes. On remarquera toutefois qu'il a également été repéré sur le benchmark 410.bwaves de la suite SpecCPU2006 avec des variations pouvant atteindre 40% en fonction de l'OS sur Core 2 duo. Les résultats associés peuvent être trouvés en annexe B.

3.4 Pagination et stratégie de malloc

Cette section se propose d'analyser en détail le problème observé sur EulerMHD en étudiant l'interaction de la politique d'alignement des blocs de l'allocateur vis-à-vis de la politique de pagination de l'OS. Nous allons montrer ici que la conjonction de certaines décisions peut conduire aux effets observés. Nous montrerons que ces conditions sont notamment rassemblées dans l'implémentation de FreeBSD.

3.4.1 Impact de l'implémentation de malloc

Comme cela a été étudié dans [CLT], le choix relatif des adresses de base de différents tableaux peut impacter les performances des applications par des effets de caches. Dans cette étude, C. Lemuët étudiait l'impact du placement des tableaux relativement aux bancs d'accès à la mémoire de l'Itanium 2. Si deux tableaux sont utilisés simultanément avec un même alignement, alors le programme peut-être pénalisé par des accès concurrents au même banc. Dans leur description, l'introduction de décalages dans les adresses permet de régler le problème. On

peut trouver des remarques similaires dans les documentations de Nvidia quant à l’utilisation de certaines mémoires du GPU[RM09] pour l’implémentation de multiplication de matrices. Ces études travaillent plus généralement sur la notion de tableau en négligeant la présence de l’allocateur. Nous allons donc étendre leurs remarques en prenant en compte ce paramètre comme cela est discuté dans la publication [ADM11] réalisée en parallèle de nos travaux.

Rappelons que pour nos expériences nous avons gardé la LIBC de chacun des systèmes et donc maintenu leurs allocateurs mémoires (malloc) respectifs. Un simple test prouve que les alignements mémoires des grands tableaux d’EulerMHD diffèrent d’un OS à l’autre. On pourra trouver les données en annexe B. On remarque ainsi que sous Linux, les tableaux plus grands que 128 Ko sont alloués directement par un appel à *mmap*. Cette méthode implique que chacune de ces allocations s’aligne sur le début d’une page avec un décalage de 16 octets lié à l’en-tête ajouté par l’allocateur. Sur FreeBSD, les gros tableaux tendent à être alignés directement sur les limites des grosses pages (2 Mo)[Eva06]. À l’opposé, OpenSolaris tend à utiliser une distribution, qui dans le cas d’EulerMHD s’avère plus aléatoire selon des multiples de 16 octets.

De façon à éliminer ce facteur de différence, nous avons implémenté un allocateur simpliste permettant d’assurer une politique unique d’alignement des tableaux. Cette implémentation repose directement sur des appels à *mmap* pour chaque allocation. Cette approche génère un gâchis important de mémoire, mais fournit en retour un moyen simple d’obtenir une structure plus reproductible et contrôlable. Tous les tableaux sont alignés sur les limites de pages, ce qui tend à augmenter les effets de contentions sur le cache, mais permettra de comparer objectivement les résultats de chaque système. Sous Linux, nous avons forcé un alignement sur 2 Mo pour reproduire le problème qui n’est pas présent avec le comportement par défaut de l’allocateur de la GLIBC.

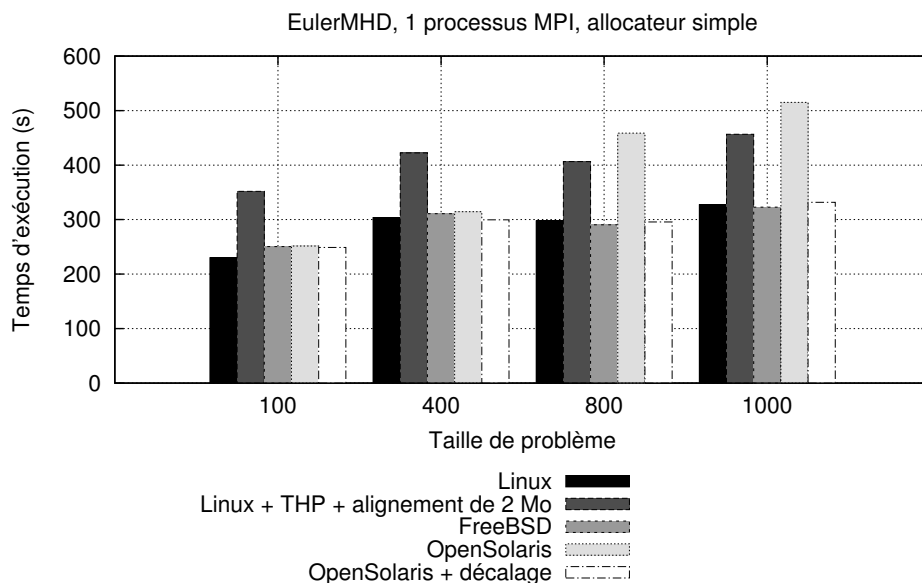


FIGURE 3.8 – Résultats obtenus avec EulerMHD en mode séquentiel en utilisant notre allocateur simplifié en remplacement de celui de l’OS. Sur les grosses pages Linux, nous avons délibérément forcé un alignement sur 2 Mo. Sur OpenSolaris, nous avons également testé un mode introduisant un décalage aléatoire multiple de 4 Ko

La figure 3.8 montre les résultats obtenus avec cet allocateur simplifié. Avec cette méthode, FreeBSD donne des performances équivalentes à Linux. De manière similaire, le problème peut-

être reproduit sous Linux en forçant un alignement sur les grosses pages, problème qui ne se pose pas sans l'application forcée de cette contrainte. Notre allocateur simplifié génère également le problème sous OpenSolaris. Toutefois, le problème disparaît si l'on ajoute manuellement un décalage aléatoire multiple de 4 Ko aux adresses générées par ce dernier. Une analyse approfondie du fonctionnement d'OpenSolaris montre que les requêtes *mmap* au delà de 1 Mo sont automatiquement alignées sur 2 Mo au lieu des 4 Ko habituellement utilisés sous Linux et FreeBSD. L'ajout de décalages aléatoires permet donc de réduire l'alignement de 2 Mo à un alignement de 4 Ko et de corriger le problème. On démontre ainsi l'importance du choix de placement des segments dans l'espace virtuel. Les NAS utilisent essentiellement des allocations statiques, cette méthode ne permet donc pas de mettre en évidence des changements de performance sur ces derniers.

3.4.2 Problématique des paginations régulières

Nous venons de voir que la modification de *malloc* permettait de reproduire le cas pathologique d'EulerMHD sur Linux et OpenSolaris ou de l'éliminer sur les trois plateformes. Pour ce faire, nous avons forcé ou évité le maintien d'alignements mémoires sur les frontières de grosses pages lors des appels à *mmap*. Le placement en mémoire virtuelle est donc un paramètre clé. Pour expliquer ce phénomène, on remarquera que l'allocation d'un tableau dynamique aligné sur les grosses pages implique que le premier élément du tableau est nécessairement de couleur "0". On remarquera également que la fonction problématique d'EulerMHD exploite simultanément plus de huit tableaux (donc plus que l'associativité des caches L1 et L2 de l'architecture Nehalem). Or, dans ces conditions d'alignements, les débuts de tous les tableaux vont nécessairement se reporter sur la même partie du cache. Tous les accès de la fonction génèrent donc des conflits permanents qui annulent l'efficacité des deux niveaux de caches (L1 et L2) pour un accès séquentiel linéaire.

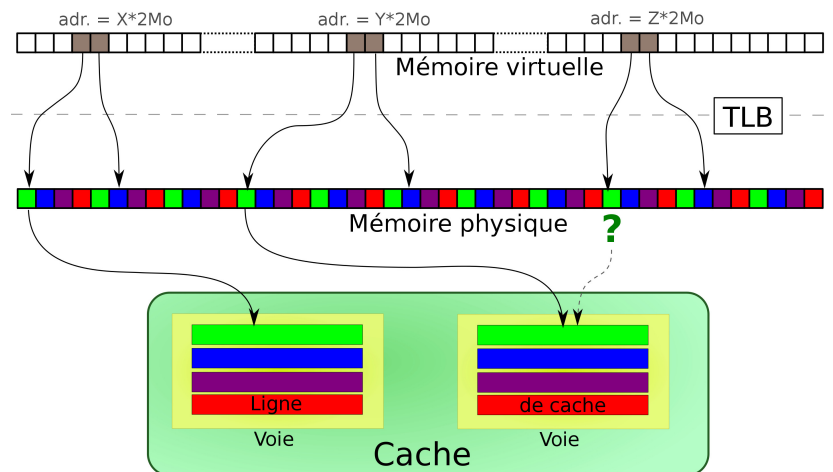


FIGURE 3.9 – Exemple de conflit lié à certains alignements en mémoire virtuel en conjonction d'une politique de coloration régulière sur un cache associatif à 2 voies et un accès à 3 tableaux alignés sur 2 Mo.

Le problème se pose de la même manière sur une coloration de pages régulière, c'est à dire produisant un motif répété sur l'ensemble de l'espace virtuel et aboutissant à une relation *linéaire* entre les adresses virtuelles et les positions en caches. Ceci explique l'observation du phénomène sous OpenSolaris. Ce cas de figure est illustré par la figure 3.9 en considérant un cache associatif à 2 voies. Sous Linux, avec une pagination aléatoire, le déplacement d'un élément de 4 Ko dans l'espace virtuel ne change statistiquement pas sa position dans le cache. Une page aléatoire est

en effet remplacée 4 Ko plus loin par une autre page aléatoire. Dans ce contexte, on comprend que Linux ne soit pas affecté par le problème observé sur les deux autres OS.

Nous pointons ainsi une limite des techniques de coloration de pages exploitées par FreeBSD et OpenSolaris. Avec ces colorations que nous dirons *régulières*, la fonction malloc devient sensible à l’associativité des caches au-delà de la taille standard des pages. Ces décisions peuvent alors conduire à des interférences avec la politique de l’OS. Nous verrons en section 3.5.3 les implications sur le design des allocateurs.

3.4.3 Associativité des caches partagés

Dans la section précédente, nous avons vu que dans EulerMHD, l’utilisation d’un nombre de flux de données supérieur à l’associativité du cache pouvait conduire à l’apparition de cas pathologiques très pénalisants. Ce type d’accès est une erreur en terme d’optimisation, la meilleure solution serait donc de corriger le code. Comme les caches actuels offrent des associativités de 8 à 16, voire 24 on pourrait considérer ce problème comme réglé. Toutefois, les applications modernes doivent exploiter des machines multicœurs, donc disposant de caches partagés. Considérant cet aspect, on peut par exemple observer les 16 voies d’un cache partagé (tel que le cache L3 de l’architecture Nehalem). Ces caches sont typiquement partagées par 4 flux d’exécution. Ceci permet donc d’exploiter 4 tableaux en interférence par flux d’exécution. En tenant compte de l’hyperthreading, ce nombre tombe à deux tableaux par flux. Dans ces conditions, les efforts d’optimisation d’un développeur peuvent très rapidement se voir annulés par une politique d’allocation inadaptée. On notera par ailleurs que les équations de physique des simulations numériques peuvent rapidement faire intervenir un nombre de tableaux supérieur à 4 ou 8, on se trouve donc dans une situation où le développeur non expert aura tendance à se trouver dans cette situation défavorable.

Ce phénomène explique les pertes de performances observées sur les NAS parallèles en section 3.3.3 l’effet parallèle plus marqué sur EulerMHD. Comme ces derniers utilisent des adresses statiques, en mode MPI, toutes les instances vont avoir tendance à utiliser les mêmes adresses. Sur une pagination régulière, on observera l’effet discuté précédemment, augmentant la pression sur les caches à mesure que l’on augmente le nombre de flux d’exécution. Pour mettre en évidence les limites du problème, nous étudions ici les performances du code 3.1 en considérant un nombre variable de tableaux et en répétant ces opérations dans différents threads.

Code 3.1– Noyau de calcul utilisé pour l’évaluation des effets d’alignements.

```
1 #pragma omp parallel
2 {
3     float **arrays = alloc_arrays();
4     float * tb0 = arrays[0];
5     float * tbj;
6     for ( k = 0 ; k < NB_REPEAT ; ++k ) {
7         for ( j = 1 ; j < NB_ARRAYS ; ++j ) {
8             tbj = arrays[j];
9             for ( i = 0 ; i < size ; ++ i )
10                 tb0[i] += tbj[i];
11         }
12     }
13 }
```

La figure 3.10 donne les résultats de ce benchmark. En fonction de l’alignement des tableaux, il est possible d’observer des pertes de performance d’un facteur 2 si l’on utilise plus de tableaux que l’associativité ne le permet. Éliminer le facteur d’alignement permet d’assurer des perfor-

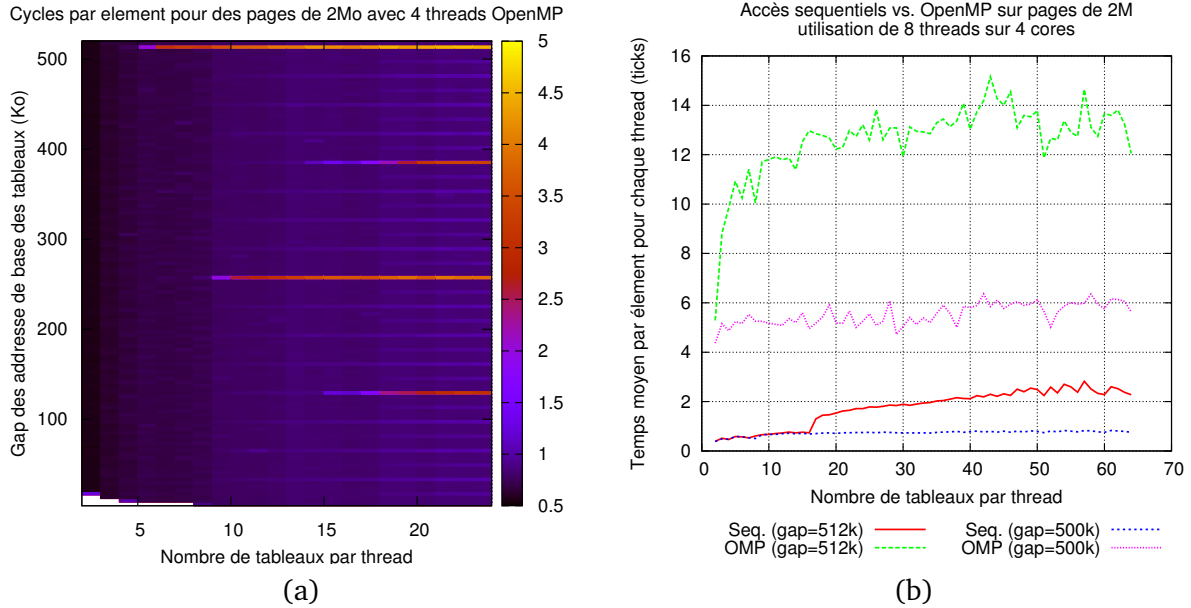


FIGURE 3.10 – Exécution du code 3.1 avec 4 threads sur INTEL NEHALEM en utilisant des grosses pages (a). Les tableaux sont dimensionnés pour que les données manipulées par chaque thread tiennent dans les 32Ko de leur cache L1 respectif. Rappelons que le cache L3 de 8Mo dispose de 16 voies de 512Ko. Le graphique (b) donne une coupe horizontale ($gap = 500$ Ko et $gap = 512$ Ko) pour mieux observer les effets des threads.

mances décentes même en exploitant 64 tableaux, c’est à dire bien plus que l’associativité de 16 du cache L3 du processeur. On montre donc qu’il est possible d’intervenir au niveau de la politique d’allocation pour prévenir ou limiter l’apparition de ces cas pathologiques.

3.4.4 Parallélisme des accès en lecture et écriture

Les processeurs modernes permettent généralement d’effectuer plusieurs transferts mémoires simultanément, mais aussi d’effectuer des écritures et lectures simultanées. Toutefois, sur certains processeurs il est possible d’observer des cas pathologiques empêchant ce parallélisme. Par exemple, l’architecture Core 2 n’autorise pas les lectures et écritures simultanées sur des éléments distants d’un multiple de 4Ko. Cette limitation est liée à l’évaluation des dépendances qui se limitent à l’analyse d’une partie réduite de l’adresse (les 12 bits de poids faibles) [Int10a]. Ce problème peut se mettre en évidence avec un accès similaire au code 3.2. Remarquons que ce type d’accès peut être commun en HPC dès lors que les codes effectuent par exemple des interpolations.

Code 3.2– Mise en évidence du problème de lecture/écriture.

```
1 for (i = 1 ; i < SIZE ; ++i)
2   X[i] = Y[i-1]
```

Du fait du modèle superscalaire⁴ utilisée par ces processeurs, l’itération $i + 1$ génère un chargement de $Y[i]$ alors que l’opération $X[i]$ de l’itération i est encore dans le pipeline d’exécution. Si les tableaux X et Y ont les mêmes adresses de base modulo 4Ko, le détecteur de dépendance considérera qu’il y a conflit et mettra l’instruction en attente. Cela peut être observé avec le

4. Rappelons que les architectures superscalaires peuvent exécuter plusieurs instructions simultanément si elles utilisent des unités de traitement différentes et qu’il n’y a pas de dépendances entre ces instructions.

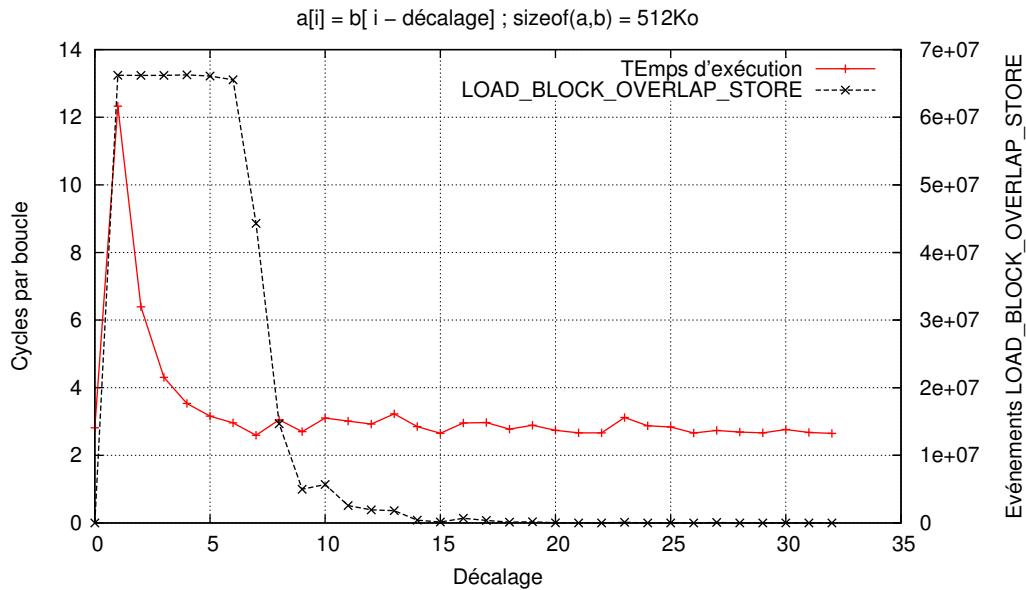


FIGURE 3.11 – Observation du problème de conflit lecture/écriture en fonction de l’alignement relatif (modulo 4Ko) des tableaux manipulés. Ce graphique montre aussi les limites de la pertinence du compteur matériel associé à ce problème : `LOAD_BLOCK_OVERLAP_STORE`. Ce compteur donne en effet de faux positifs puisque le phénomène n’a pas nécessairement un impact sur le temps d’exécution.

compteur matériel⁵ `LOAD_BLOCK_OVERLAP_STORE` sur Core 2 Duo. Comme cela est montré sur la figure 3.11, ce type de problème peut conduire à un ralentissement du code d’un facteur supérieur à 2.

Sur architecture Nehalem et Sandy Bridge, le problème est résolu pour l’exemple précédemment, mais nous observons qu’il reste présent pour des cas plus compliqués avec plusieurs flux mémoires à problèmes tels que donnés dans le code 3.3.

Code 3.3– Accès plus complexe ayant des problèmes de concurrence lecture/écriture.

```
1 for (i = 1 ; i < SIZE ; ++i) {
2     X1[i] = Y1[i-1]
3     X2[i] = Y2[i-1]
4 }
```

Sur un cas réel, nous avons observé qu’il était possible d’obtenir des gains de performance aussi bons en désalignant les tableaux qu’en faisant des optimisations complexes du code. La table 3.2 donne les mesures effectuées sur Core 2 Duo avec ce code. La version optimisée manuellement offre une meilleure performance, plus stable en fonction des architectures. Elle a toutefois nécessité beaucoup plus d’effort et abouti à un code peu lisible non nécessairement compatible avec une maintenance dans la durée de vie d’un code massif.

Ce problème connu est déjà discuté sur les problèmes de piles [MDHS09], mais nous démontrons ici que les politiques de l’allocateur peuvent conduire au même problème. Ceci est particulièrement vrai dans le cas où ce dernier tend à aligner les grands tableaux sur les débuts de pages comme c’est le cas par défaut sur la quasi-totalité des allocateurs utilisant *mmap* pour ces segments. La glibc de Linux génère ce type de placement à risque pour tous les tableaux dé-

5. Ces compteurs permettent de suivre certains phénomènes survenant directement à l’intérieur du processeur, on y accédera par exemple avec l’outil Likwid, VTune ou perf.

	Originale	Optimisée
Allocateur standard	51.0	21.7
Tableaux décalés	21.8	21.3

TABLE 3.2 – Évaluation de performance d’une fonction optimisée et sa version d’origine. Les performances sont données avec l’allocateur standard ou en modifiant l’alignement des tableaux utilisés. Les temps sont donnés en cycles processeurs par itération, les valeurs faibles indiquent donc un gain.

passant 128 Ko. Éviter ce type d’alignement pourrait permettre de maintenir des performances décentes en limitant l’apparition de ce type de cas pathologiques dépendants de la micro architecture. Les fonctions d’un programme ne méritent en effet pas nécessairement des corrections lourdes et risquées au niveau du code des programmes.

3.4.5 Effet de la table des pages et des TLB

Au cours de cette étude, nous avons également observé certains effets de résonances liés à des alignements bien au-delà de 4 Ko sur les pages standards de Linux. Ces observations ne peuvent s’expliquer avec les points précédents. Dans cette section, nous allons donc étudier ces phénomènes et montrer qu’ils prennent leur source dans la structure de la table des pages et des TLB.

Alignements au-delà de 4 Ko

De manière similaire aux tests précédents, nous allouons des tableaux avec des alignements induisant des effets de résonances et y appliquons des opérations simples pour évaluer les performances d’accès. Pour ce test, nous étudions les alignements au-delà de la taille d’une page, nous allouons donc des tableaux directement avec *mmap* pour forcer le placement en mémoire virtuelle en respectant la distribution suivante :

$$address = base_address + array_id * alignement \quad (3.12)$$

Avec *base_address* un point fixe pour l’ensemble des tableaux, *array_id* un identifiant de tableau débutant à 0, *alignement* l’alignement visé (multiple de 4 Ko). Le noyau de calcul est donné par le code 3.4.

Code 3.4– Noyau utilisé pour tester les TLB avec un nombre variable de tableaux.

```
1 for (int i=0;i<SIZE;++i)
2     a[i] = b[i] + c[i] + d[i]....;
```

La figure 3.12 est obtenue en échantillonnant les alignements (en mémoire virtuelle) par puissances de deux au-delà de 4Ko et ce jusqu’à plusieurs Go. Cette figure montre clairement l’apparition de différents niveaux de performances. On y distingue clairement sur Core 2 Duo les niveaux à 256 kB et 1 GB ; sur Core i7 : 64 kB, 512 kB et 1 GB. Ici nous ignorons le cas à 32Mo pour lequel nous n’avons pour l’instant pas d’explication confirmée. On remarquera que le problème n’apparaît que dans le cas d’utilisation de plus de 4 tableaux et uniquement pour les alignements multiples cités précédemment, la structure lissée de la courbe n’apparaît sous cette forme que parce que notre échantillonnage suit ces multiples par puissance de deux.

Limitation des TLB

Comme ces plateaux ne sont observés que pour des alignements spécifiques, nous pouvons supposer qu’il s’agit, comme dans les sections précédentes, d’effets liés aux caches. Pour confir-

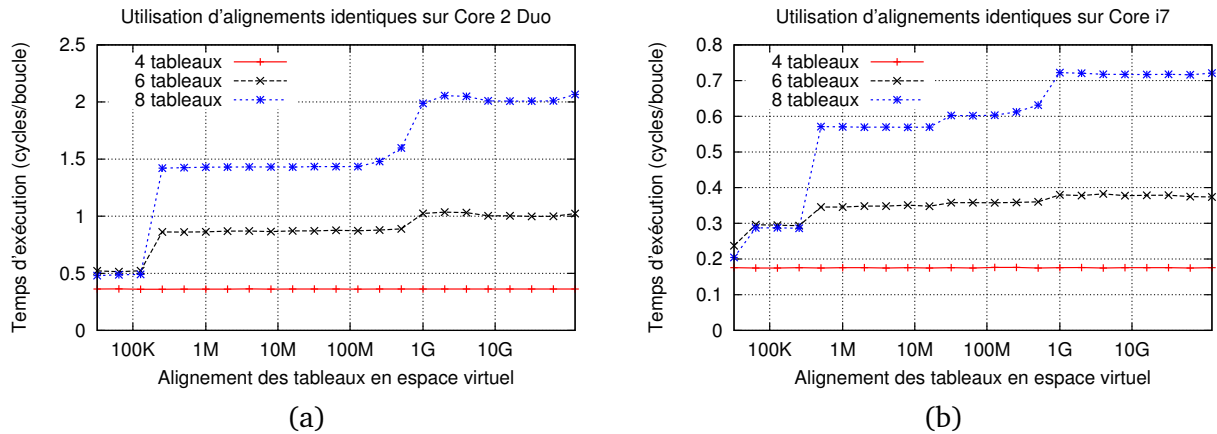


FIGURE 3.12 – Impact de l'alignement relatif des tableaux au-delà d'une page en utilisant le code 3.2 sur CORE 2 DUO (a) et CORE i7 (b).

mer cette hypothèse, nous pouvons explorer les compteurs matériels pour ces différents paliers. La figure 3.13 donne les résultats de ces expériences sur Core 2 Duo en donnant les écarts entre une mesure alignée et non alignée. Une valeur non nulle indique donc que nous avons identifié une source du problème.

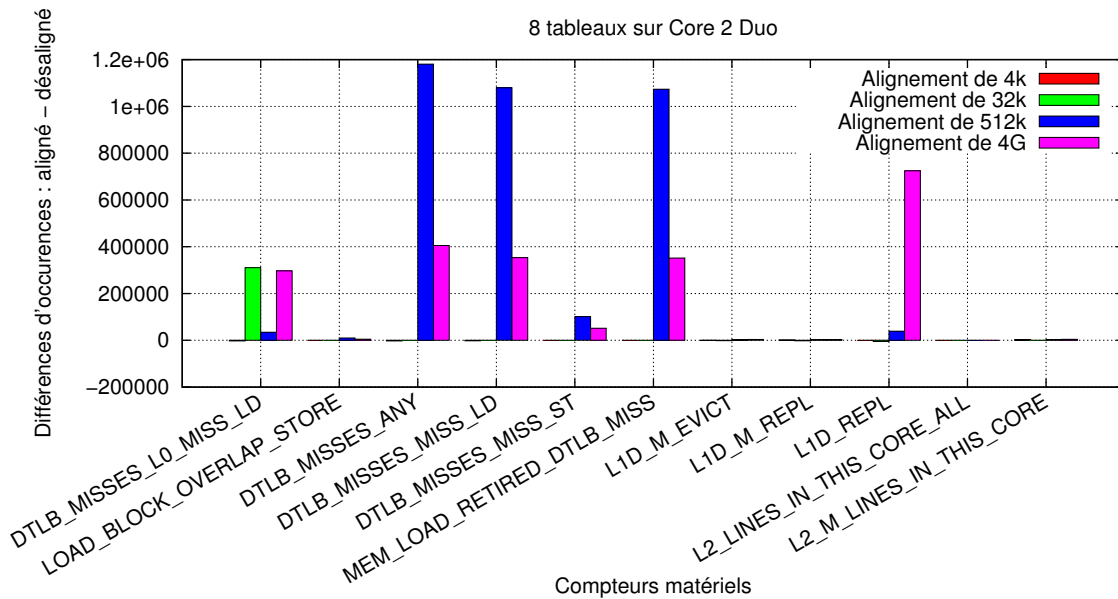


FIGURE 3.13 – Exécution du benchmark précédent en utilisant les compteurs matériels pour les alignements relatifs 4kB, 32kB, 512kB et 4GB. Le graphique représente les valeurs des compteurs soustraites de la part présente pour le cas non aligné.

Pour analyser ces données, rappelons que les architectures Core d'Intel disposent de plusieurs niveaux de TLB décrits dans la table 3.3. Pour nos analyses, nous étudions uniquement les effets relatifs aux TLB de données (DTLB) et non ceux dédiés aux instructions (ITLB), ces dernières n'ayant pas de liens étroits avec les allocations dynamiques.

Les données de la figure 3.13 montrent que le niveau à 512Ko sur Core 2 Duo est lié à une augmentation des fautes de TLB au niveau du DTLB1. En effet, si les tableaux sont distants de 256 Ko (ou multiples), avec une associativité de 4, nos huit tableaux essaient d'utiliser deux fois

TLB level	Nombre d'entrées	Mémoire adressée
Core 2		
DTLB0	16	64k
DTLB1	256, 4 voies	1M, 256k par voie
Core i7		
DTLB0	64, 4 voies	256k, 64k par voie
DTLB1	-	-
STLB (DTLB+ITLB unifié)	512, 4 voies	2M, 512k par voie

TABLE 3.3 – Configuration des TLB des architectures CORE 2 et CORE i7. Extrait de la documentation Intel[[Int10a](#)]. Chaque cœur exploite ses propres TLB.

chaque entrée du TLB et génèrent donc des conflits. Même remarque sur Core i7, les niveaux correspondants à la structure différente des TLB, notamment l'ajout d'un niveau unifié (STLB). La raison spécifique au problème de DTLB0 à 32k sur Core 2 Duo n'est pas confirmée. Ce dernier pourrait s'expliquer si le DTLB0 est directement associatif, mais cette information n'est pas précisée dans les documents Intel[[Int10a](#), [Int10b](#)].

3.4.6 Impacte de la table des pages

Nous venons précédemment d'expliquer les problèmes liés aux paliers de 256 Ko et 512 Ko. Il reste toutefois à expliquer le dernier palier observé sur les deux architectures au-delà de 1Go. Pour cela, rappelons que la table des pages est structurée en arbre[[Int10a](#), [BP05](#), [Gor04](#)] afin de limiter son empreinte mémoire pour la description d'un espace virtuel essentiellement vide. Cette table est rappelée par la figure 2.2 de la section 2.2.3. Notons que cette structure est spécifiée par la norme x86_64 et doit donc être la même sur chaque OS. Le problème que nous décrivons est donc indépendant des choix de l'OS sur ces architectures. Chacun des niveaux de la table permet d'adresser un espace croissant dont les échelles sont listées ici :

Page : adresse un total de 4 Ko.

Table : contient 512 entrées donc adresse jusqu'à $512 * 4 \text{ Ko} = 2 \text{ Mo}$.

Directory : contient 512 entrées donc adresse jusqu'à $512 * 2 \text{ Mo} = 1 \text{ Go}$.

Le problème que nous observons à 1 Go est donc causé par la présence du dernier niveau de la table (PDE : *Page Directory Entry*). Chacun de ces niveaux doit en effet être lu pour obtenir la traduction de la table. Pour plus que 4 tableaux, nous avons vu que les alignements au-delà de 512Ko provoquaient des fautes en série au niveau du TLB. Les compteurs matériels nous montrent une augmentation des fautes du cache L1. On peut donc supposer que les entrées des niveaux intermédiaires de la table sont simplement stockées dans ce cache, mais cela n'est pas clair dans la documentation Intel.

Pour confirmer notre hypothèse, nous pouvons remarquer que si les PDE expliquent les pertes de performance, alors la position absolue dans l'espace virtuel devient un paramètre important. Cette position guide en effet la position des entrées en conflits vis-à-vis des limites d'adressage des PDE. Nous pouvons donc revenir au micro benchmark 3.4, y utiliser un alignement de 512Mo et déplacer l'adresse de base du tableau pour balayer une bande d'adresse. Le croisement des frontières des PDE devrait donc être visible sur les performances. En utilisant 8 tableaux de 4 Ko alignés sur 512 Mo nous couvrons donc un espace 3.5 Go. En fonction de la position relative aux PDE l'adressage de ces tableaux, l'adressage nécessite un total de 4 ou 5 PDE. Ceci donne les adresses décrites dans le code 3.5 pour *base_address* = 4 Go et *alignment* = 512 Mo.

Code 3.5– Positionnement des tableaux pour une mise en évidence de l'impact des PDE.

```

1 addr_array[0] = base_address = 4 Go
2 addr_array[1] = base_address + 1 * alignment = 4.5 Go
3 addr_array[2] = base_address + 2 * alignment = 5 Go
4 addr_array[3] = base_address + 3 * alignment = 5.5 Go
5 ...

```

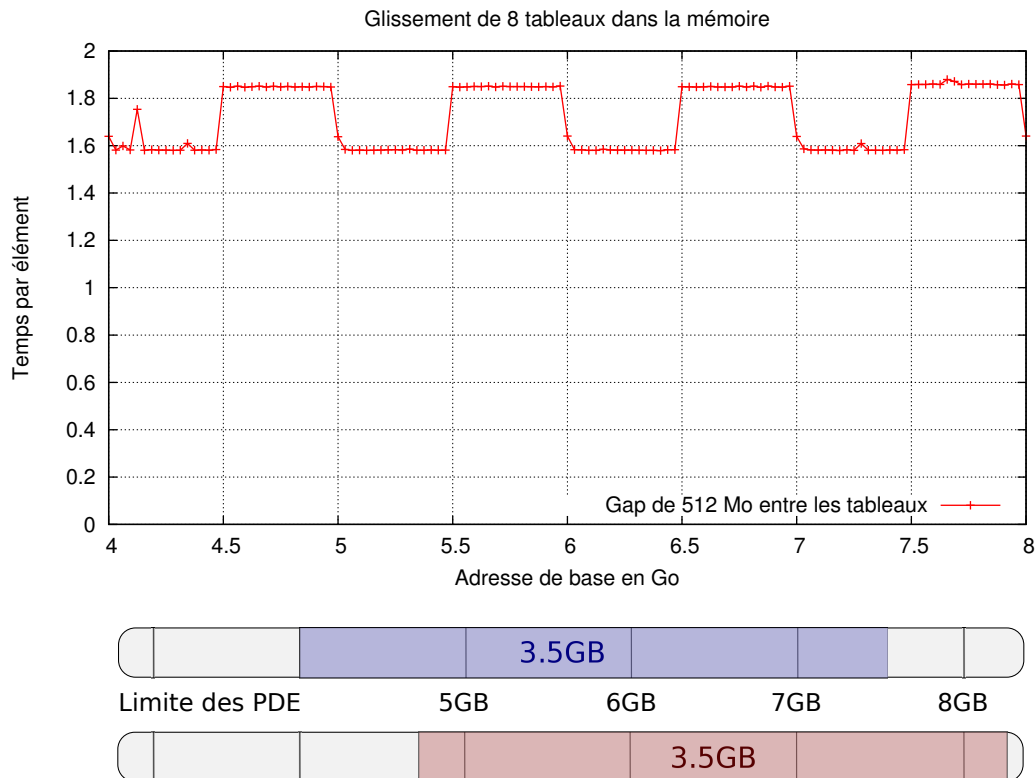


FIGURE 3.14 – Déplacement des données dans l'espace virtuel afin de croiser un nombre variable de frontières des PDE. Le benchmark utilise 8 tableaux de 4 Ko distribués de manière à couvrir une zone de 3.5 Go de mémoire virtuelle en utilisant un gap de 512 Mo. En fonction du placement, ces segments nécessitent 4 ou 5 PDE pour être adressés.

La figure 3.14 montre le résultat de ce test en déplaçant les données dans l'espace virtuel pour croiser les limites des PDE. Pour une adresse de base de 4 Go à 4.5 Go l'espace mémoire utilisé est couvert par 4 PDE. Par contre entre 4.5 et 5 Go le recouvrement nécessite une PDE de plus, donc un coût d'accès supérieur. Ajouter 512 Mo fait revenir à la situation initiale. Les résultats expérimentaux confirment donc bien que les effets d'alignement au-delà du Go sont liés au dernier niveau de la table des pages. Rappelons toutefois que ce problème ne se pose qu'en conjonction de lecture répétée de la table, donc cumulé avec les problèmes précédemment évoqués. Ce problème particulier lié à des alignements de l'ordre du Go a donc très peu de chance de survenir en pratique et peut être ignoré. Il ajoute toutefois un argument visant à considérer les alignements trop réguliers comme source de problèmes.

3.5 Analyse générale et recommandations

Nous venons de lister un certain nombre de problèmes générés par des effets de résonance entre la politique de pagination et les choix de placement en espace virtuel de la part de l'al-

locateur. Résumons ici ces derniers avant de poursuivre sur une série de recommandations applicables à chacun des deux niveaux de gestion de la mémoire (OS et malloc). Ces problèmes sont essentiellement étudiés en considérant l'utilisation de gros segments mémoires (plusieurs centaines de Ko). Un tableau récapitulatif pourra être trouvé en annexe [B.4](#).

Fuite de caches : Nous avons dans un premier temps (section [3.1](#)) étudié le problème de la coloration de pages sur son plan historique. Pour ce faire, nous avons initialement observé des effets connus de perte d'efficacité du cache liée aux politiques de paginations aléatoires de la part de l'OS. Nous avons également vu que Linux entrainait dans cette catégorie.

Grosses pages : Nous avons ensuite étudié une amélioration potentielle apportée par les grosses pages (section [3.2.3](#)) et avons montré que ces dernières tendaient à amplifier les effets pénalisants de cas pathologiques. Nous avons notamment montré que dans ce contexte, l'allocateur devenait responsable de la prise en compte des caches pour le placement des gros tableaux. Ce point particulier n'est explicitement traité par aucun des allocateurs actuels. Avec l'application EulerMHD, nous avons également montré (section [3.3.4](#)) que l'allocateur de FreeBSD appliquait une règle tendant à générer ces cas pathologiques alors même que cet OS offre un support natif des grosses pages.

Coloration de pages régulière : Au cours de nos analyses, nous avons également montré que les politiques de coloration de pages proposées par OpenSolaris tendaient à produire les mêmes effets néfastes que les grosses pages en rendant les décisions de la fonction malloc sensible aux caches.

Effet des TLB : Sur la fin de notre étude, nous avons (section [3.4.5](#)) également mis en évidence quelques effets liés à la forme associative des TLB dans le cas où l'allocateur générerait des alignements particuliers entre les tableaux qu'il fournit à l'utilisateur.

Effet de la table des pages : En poussant le raffinement, nous montrons également qu'il est possible d'observer quelques rares effets liés à la structure de la table des pages. Nous considérons toutefois qu'il est peu probable d'observer ces derniers dans une application réelle du fait des échelles en jeux (alignement de l'ordre du Go).

De manière générale, nous avons rappelé que les paginations aléatoires de type Linux réduisaient l'efficacité des caches. Toutefois, nous avons montré que les techniques de coloration de pages actuellement mise en place pour corriger ces effets peuvent s'avérer inefficaces. Cette inefficacité apparaît parce que ces techniques tendent à générer au niveau de l'OS des motifs réguliers déléguant la responsabilité d'utilisation des caches à l'espace utilisateur. Ce dernier point est une bonne chose pour une application optimisée finement par des spécialistes. Il est toutefois clair qu'elle peut devenir problématique pour le développement d'applications mises au point par des développeurs moins experts ou dont les priorités ne sont pas orientées sur l'optimisation fine dépendante du matériel.

Dans ce contexte, nous pensons qu'il est possible de faire certains efforts au niveau du système d'exploitation et de l'allocateur pour limiter les problèmes observés en trouvant un compromis entre les méthodes actuellement utilisées. Dans tous les cas, la conception de l'allocateur doit prendre en compte la politique de pagination sous-jacente et ne peut donc la considérer comme étant une simple boîte noire.

3.5.1 Conséquence sur les politiques de pagination

Les politiques de *paginations aléatoires* telles qu'exploitées par Linux sont habituellement critiquées sur le plan théorique. Les techniques de coloration de pages semblant être une solution

intellectuellement satisfaisante au problème. Au vu de nos résultats, il semble clair que les techniques de coloration actuellement employées génèrent autant de nouveaux problèmes qu'elles n'en règlent. Cela peut expliquer les avis mitigés observés dans de nombreuses publications traitant de l'implémentation de ces techniques[HK, BKW98] en dehors de certains cas particuliers.

Pour des accès strictement aléatoires, nous pouvons considérer les différentes politiques de pagination comme équivalentes. En revanche pour des accès plus linéaires, nous pouvons proposer de trouver un meilleur compromis. Une certaine maîtrise du processus d'allocation des pages est nécessaire pour ne pas aboutir aux défauts d'une pagination aléatoire, néfaste pour des accès linéaires. Ces techniques de contrôle des pages physiques ne doivent toutefois pas conduire à l'apparition trop aisée de phénomènes de résonances tels que nous l'avons observé. Pour cela nous préconisons d'utiliser des méthodes de coloration évitant l'apparition de schémas réguliers et répétitifs sur l'ensemble de l'espace d'adressage. Ceci est surtout vrai s'ils sont synchrones avec les frontières des voies du cache. Dans ce sens, afin d'éliminer les effets intra-processus, il est possible de complexifier les fonctions de hachages comme cela a été fait par l'introduction du PID pour éliminer les effets inter-processus.

Le problème peut donc se poser en considérant la construction d'heuristiques vérifiant les points suivants :

1. Considérons W_{max} la taille maximale d'une voie dans la hiérarchie de cache de l'architecture en considérant que les éventuels caches inférieurs exploitent des tailles sous-multiples de W_{max} .
2. Pour tout segment de taille $s = W_{max}$, le nombre de pages d'une couleur donnée doit être aussi proche que possible de A l'associativité du cache.
3. La distance séparant deux segments ayant le même motif de coloration doit être maximisée.

On peut pour cela étendre la méthode de coloration habituelle basée sur un modulo en introduisant un terme de brassage dépendant du PID et de la *position dans l'espace virtuel*. En brassant différemment chaque segment $[x, x + W]$ par une fonction H , on peut alors obtenir l'assurance d'une parfaite distribution sur les segments de taille W alignés sur cette même taille. On évite également les schémas répétés pour deux segments proches. Cela implique que H doit être constant sur les intervalles $[x * W, (x + 1) * W]$.

$$C(p_{virt.}) = (p_{virt.} \oplus H(p_{virt.}, PID, W)) \pmod{W} \quad (3.13)$$

Cette méthode n'a pas été expérimentée en pratique, nous proposons donc ici de choisir arbitrairement la fonction H comme étant définie à partir de l'index du segment $\lfloor p_{virt.}/W \rfloor$:

$$H(p_{virt.}, PID, W) = \lfloor p_{virt.}/W \rfloor * p_1 + PID * p_2 \quad (3.14)$$

Avec p_1 et p_2 des nombres premiers supérieurs à la taille des pages. Ce choix est ici arbitraire et l'on pourrait également choisir de hasher $\lfloor p_{virt.}/W \rfloor$ et le PID avec une fonction de type MD5. Remarquons que l'utilisation de l'opérateur XOR limite la méthode à un nombre de couleur puissance de 2 sous peine de perdre l'équilibre de distribution de couleur du fait du modulo qui suit.

Concernant les problématiques multicœurs/multithreads, on pourra noter les travaux de certaines équipes sur le partitionnement de cache via la politique de pagination [ASBC09, KPKZ11]. Si ces méthodes peuvent résoudre certains problèmes de conflits inter-threads, on notera qu'elles doivent également prendre en compte la remarque précédente pour chaque thread. Le micro benchmark de la section 3.4.1 montre en effet qu'il est possible d'obtenir une amélioration notable des performances du cache dans ce type de contexte en éliminant les cas pathologiques liés aux résonances.

3.5.2 Extension matérielle pour les grosses pages ?

Nous venons de voir qu'il était possible de prendre en charge le problème au niveau de l'OS pour les pages standards. Cette technique n'est toutefois pas applicable aux grosses pages du fait de leur définition matérielle. On peut donc se demander s'il n'est pas possible de résoudre le problème à ce niveau. On peut alors considérer l'idée précédente en brassant les adresses à l'intérieur des grosses pages au travers d'une fonction de hachage. La fonction de hachage peut alors être implémentée directement au niveau matériel ou bien fournie de manière logicielle.

Il nous semble plus judicieux d'implémenter un mécanisme logiciel permettant à l'OS de prendre une décision adaptée à l'application. Cette approche peut être mise en place en ajoutant un champ aux entrées de la table des pages de sorte à associer une clé de brassage à chaque grosse page (valeur effective de $H(p_{virt.}, PID, W)$). Cette clé doit alors être appliquée (XOR) sur les bits décrivant les pages internes aux grosses pages afin de ne pas impacter les caches adressés virtuellement. Remarquons qu'une valeur 0 pour cette clé permet de retrouver une projection linéaire équivalente à la définition actuelle. Sur architecture x86_64 on remarque qu'il nous faut 9 bits pour décrire ce hash ce qui peut tout à fait se placer dans certains bits inutilisés de la structure actuelle de la table de cette architecture d'après la documentation Intel [Int10b].

L'application de cette clé peut être effectuée de manière globale pour la traduction générale des adresses virtuelles en adresses physiques. Cela pose toutefois un problème de prise en charge par les périphériques externes (carte PCI...) qui peuvent être amenés à faire ce type de traduction. Nous n'avons pas expérimenté ou étudié en détail cette technique au cours de la thèse. L'idée semble toutefois intéressante si l'utilisation des grosses pages se généralise. En effet, nous verrons dans le chapitre 5 que ces dernières offrent certains avantages pour améliorer les performances de gestion de grands volumes.

3.5.3 Conséquence sur malloc

Dans le cadre de nos expériences, les pertes de performances se sont manifestées lors de la conjonction d'alignements particuliers au-dessus des politiques de pagination régulières. Nous venons de décrire une amélioration possible au niveau de ces politiques de pagination. Notons toutefois que cette approche n'est pas applicable aux grosses pages qui génèrent ce type de pagination par leur définition matérielle. Dans cette situation il est nécessaire de prendre en compte ces questions au niveau de la fonction *malloc* elle-même.

D'une manière générale nous pouvons conseiller d'éviter de produire artificiellement des alignements sur des ordres trop élevés afin d'éviter de faire apparaître par défaut des résonances potentielles sur les paginations régulières (grosses pages ou coloration régulière). Ce type de politique tend malheureusement à être la méthode employée par défaut par certains allocateurs comme *jemalloc* [Eva06] de FreeBSD qui tend à forcer l'alignement des grosses allocations sur 2Mo. Nous avons également observé qu'OpenSolaris effectue ce type d'opération au sein même de la fonction *mmap*, impliquant ce type de comportement par défaut pour tout allocateur appelant *mmap* sans adresses pour l'allocation de gros segments. Sur ce type d'OS, l'allocateur doit donc prendre en main explicitement ses alignements. Nous pensons que ce type de politique au niveau de *mmap* est à prohiber bien que permettant le placement en mémoire d'un plus grand nombre de grosses pages. Ce problème est à surveiller au niveau de Linux qui tend actuellement à voir un développement de sa politique de gestion des grosses pages.

D'autre part, si les problèmes étudiés précédemment concernaient principalement les caches indexés physiquement (L2 et L3), il importe de rappeler que les caches L1 des processeurs tendent à utiliser un découpage correspondant exactement à la taille d'une page (4Ko). Il convient donc de noter qu'aligner les grands tableaux en début de page suite à un appel à *mmap* peut conduire à une perte d'efficacité de ce cache dès lors que l'application utilise plus

de tableaux que l’associativité ne le permet. Ce point est d’autant plus fâcheux qu’un simple décalage permet comme nous l’avons vu en section 3.4.1 de réduire très largement les pertes de performances. Ce type d’alignement est toutefois généré par la majorité des allocateurs comme cela est critiqué dans [ADM11].

3.6 Outil d’analyse

Lors de cette étude préliminaire, nous avons développé un prototype permettant d’analyser l’exécution du programme et repérer les fonctions potentiellement impactées par les cas pathologiques étudiés. Ce dernier est rapidement discuté ici afin de montrer qu’il est possible de détecter ces problèmes de manière dynamique.

3.6.1 Objectifs

Dans ce chapitre, nous avons montré que les paramètres clés des problèmes étudiés étaient *l’adresse de base* et *le nombre* de tableaux utilisés simultanément dans les boucles. Notre outil se propose donc d’extraire ces paramètres pour chaque fonction d’une application.

Ici, nous ne nous intéressons qu’aux grands segments mémoires, au-delà de 4 Ko. De plus, ne seront analysés que les segments dynamiques alloués par le biais des fonctions *malloc*, *calloc* et *realloc*. Notre approche sera basée sur une instrumentation de l’exécutable et une collecte d’information pendant l’exécution du programme. Afin de ne pas tomber dans des problèmes de volumes de données, nous limiterons la prise d’information à une forme échantillonnée. Les entités fondamentales seront constituées des *fonctions* et des *blocs alloués*. Nous n’analyserons donc pas les accès détaillés à l’intérieur des blocs eux-mêmes ou bien la prise en compte de présence de plusieurs boucles dans une même fonction.

Cette approche simplifiée a permis d’obtenir rapidement un prototype fonctionnel permettant de pointer les fonctions potentiellement sensibles aux effets recherchés. Elle a toutefois le défaut de pouvoir déclencher de faux positifs.

3.6.2 Points techniques sur la méthode

Notre objectif est de collecter des informations sur l’utilisation des tableaux dynamiques. Nous avons donc utilisé la démarche suivante :

Instrumentation du binaire : La première étape consiste à instrumenter chaque appel de fonction de l’exécutable et les bibliothèques éventuelles avec l’option *-finstrument-functions*⁶ du compilateur GCC. De cette manière notre outil est notifié à chaque appel de fonction afin de pouvoir attacher les tableaux utilisés à une pile d’appels.

Suivi des gros tableaux : Les fonctions *malloc*, *calloc*, *realloc* et *free* sont ré-implémentées par la bibliothèque d’analyse afin de marquer les allocations mémoires supérieures à un seuil pour suivre leur utilisation et connaître leur taille.

Suivi des accès : Le suivi d’accès aux tableaux surveillés est réalisé à l’aide de captures des signaux de type *faute de segmentation*. Pour cela, lors de l’entrée dans une fonction, l’outil d’instrumentation rend les tableaux inaccessibles à l’aide de l’appel système *mprotect*. Lors d’un accès, la faute est capturée, l’accès noté pour la fonction en cours et le tableau entier est rendu accessible. Il est possible d’analyser les accès à la granularité de la page, mais par souci de simplicité nous avons limité notre prototype au grain des tableaux complets.

6. Cette option demande au compilateur d’appeler une fonction particulière à chaque entrée/sortie de fonction.

Chaque entrée/sortie de fonction nécessite une remise en place des verrous d'accès aux tableaux.

Système de liste noire : Avec les points précédents nous générons potentiellement un nombre trop important de données et d'appels système donc un surcoût d'exécution important proportionnel au nombre d'entrées/sorties de fonction. Pour simplifier le problème, nous avons appliqué une méthode d'échantillonnage limitant le nombre d'analyses réalisées sur une fonction donnée. Une fois ce seuil d'analyse dépassé, la fonction concernée est placée dans la liste noire et ne nécessitera plus de verrouillage d'accès aux tableaux. Le programme est donc fortement ralenti lors des premiers appels de fonction puis reprend une cadence normale d'exécution uniquement impactée par le surcoût d'instrumentation. Pour être efficace, la liste noire dispose de l'équivalent d'un cache maintenant les dernières entrées récemment utilisées.

Avec cette approche sélective et une implémentation naïve des fonctions de liste noire, nous obtenons un surcoût d'un facteur 2 pour de petites exécutions d'EulerMHD. Ce dernier se réduit toutefois à 30% pour des cas tests de plus grande taille toujours avec EulerMHD. L'approche retenue à base de signaux de type faute de segmentation implique toutefois que le prototype n'est pas utilisable en contexte multithread. Les protections des segments sont en effet globales au processus. Il n'est donc pas possible de maintenir un verrouillage des accès pour un thread donné. Cette approche nous permet néanmoins de trouver rapidement les fonctions à problème pour les programmes séquentiels et MPI.

Comme discuté, la méthode précédente n'est pas applicable sous cette forme en contexte parallèle. Pour supporter ce type d'application il faudrait recourir à une émulation (par exemple avec valgrind[NS07]) ou instrumentation des accès (MAQAO[BCRJ⁺10], Pintool[HLC09]). Une autre possibilité serait d'exploiter certaines propriétés de la segmentation mais elle n'est plus disponibles sur les architectures intel 64 bits. Dans le cadre de notre étude, nous nous sommes limité au prototype séquentiel.

3.6.3 Informations collectées

Code 3.6– Exemple d'informations obtenues avec l'outil de trace.

```

1  ===== ALLOC =====
2  main() [test.c:55]
3      - Bloc 0 of size 128Ko [ 16 , 16 ]
4      - Bloc 1 of size 128Ko [ 16 , 16 ]
5      - Bloc 2 of size 128Ko [ 16 , 16 ]
6  ===== FREE =====
7  main() [test.c:55]
8      - Bloc 0 of size 128Ko
9      - Bloc 1 of size 128Ko
10     - Bloc 2 of size 128Ko
11  ===== MEM USAGE =====
12  main() [test.c:55]{ 1 : 23.3759% }
13     - Using 0 of size 128Ko [ 16 , 16 ] {+0}
14     - Using 1 of size 128Ko [ 16 , 16 ] {+0}
15     - Using 2 of size 128Ko [ 16 , 16 ] {+0}
16  test2() [test.c:22]{ 1 : 20.5996% }
17     - Using 1 of size 128Ko [ 16 , 16 ] {+1}
18     - Using 0 of size 128Ko [ 16 , 16 ] {+0}
19     - Using 2 of size 128Ko [ 16 , 16 ] {+1}
20  test3() [test.c:12]{ 1 : 8.24696% }
21     - Using 2 of size 128Ko [ 16 , 16 ] {+0}
22     - Using 1 of size 128Ko [ 16 , 16 ] {+0}
23  test1() [test.c:32]{ 1 : 47.7775% }
24     - Using 0 of size 128Ko [ 16 , 16 ] {+0}

```


Avec les données collectées, il est possible d’extraire un résumé d’utilisation des tableaux pour chacune des fonctions. Sur un simple programme test, il est par exemple possible d’obtenir la sortie du code 3.6. Cette sortie fournit les informations suivantes :

Section ALLOC ligne 1-5 : Liste les allocations effectuées par les différentes fonctions. Chaque allocation est alors associée à un ID unique utilisé pour suivre son historique et caractérisé par sa taille. Les nombres entre crochets donnent respectivement les alignements relatifs aux pages de 4Ko et 2Mo, permettant une analyse rapide des problèmes d’alignement.

Section FREE ligne 6-10 : Donne les points de libérations des tableaux.

Section MEM USAGE ligne 11-24 : Cette section liste les tableaux utilisés par les différentes fonctions du programme en donnant le nombre d’appels de ces fonctions et le pourcentage de temps consommé par ces dernières. Sont alors listés chacun des tableaux utilisés identifiés par leur ID et en rappelant leurs paramètres (taille, alignement). Ces informations sont complétées par le décalage du premier élément utilisé pouvant potentiellement, permettre une détection de certains problèmes de conflits lecture/écriture décrits en section 3.4.4.

On peut alors générer un graphique plaçant les points de chaque fonction dans un repère construit sur le nombre de tableaux accédés et le temps relatif des fonctions. Il est ainsi possible de repérer rapidement la présence de zones à problème ayant un impact potentiellement significatif sur les performances (nombre de tableaux supérieur à l’associativité avec un temps relatif important). Sur EulerMHD on constate ainsi aisément la présence d’une fonction exploitant 9 tableaux et occupant plus de 50% du temps d’exécution (figure 3.3.4). Cette visualisation peut également permettre de retrouver rapidement les fonctions impactées en comparant deux exécutions superposées sur le même graphique, l’une utilisant des alignements standards, l’autre des alignements aléatoires. Il pourrait bien sûr être envisagé de construire un greffon de ce type pour Valgrind en bénéficiant des bases fournies par cet outil. L’important dans notre approche est de profiter des *adresses de base* et *tailles de bloc* extraites de l’allocateur mémoire. Ces informations permettent d’associer une certaine sémantique aux adresses accédées par le programme en ne pas les considérer comme de simples données éparses.

3.7 Application de règles de décalage

Nous avons remarqué que certains alignements devaient être évités au niveau de l’allocateur notamment pour les gros tableaux. De plus, nous avons montré dans la section précédente qu’il était possible de détecter (malgré de possibles faux positifs) les fonctions à problème. On peut donc se demander s’il n’est pas possible d’exploiter les données fournies par l’outil précédent pour générer une politique d’allocation exploitant ces informations en considérant une première étape de profilage pour les générer. En théorie, il pourrait être possible de considérer qu’une allocation établie dans une fonction particulière aura par la suite un cycle de vie relativement déterministe. Ceci est notamment vrai pour les gros tableaux habituellement présents en nombre limité. La génération de règles d’alignements à éviter semble donc possible à partir des profils précédents. Ceci est d’autant plus vrai que les points à éviter sont rares en rapport de l’espace des alignements possibles.

D’un point de vue pratique, on se heurte toutefois à la difficulté de trouver un moyen simple et efficace d’associer un tableau à une information extraite d’un profil d’une exécution précédente. Il ne paraît pas raisonnable de remonter la pile d’appels pour chaque allocation. Nous avons donc préféré tester diverses heuristiques de génération d’alignements ne nécessitant pas de profil. À titre de comparaison, sur un programme simple nous avons tout de même mis en place une méthode basée sur l’ordre d’allocation des tableaux. Cette méthode basique permet

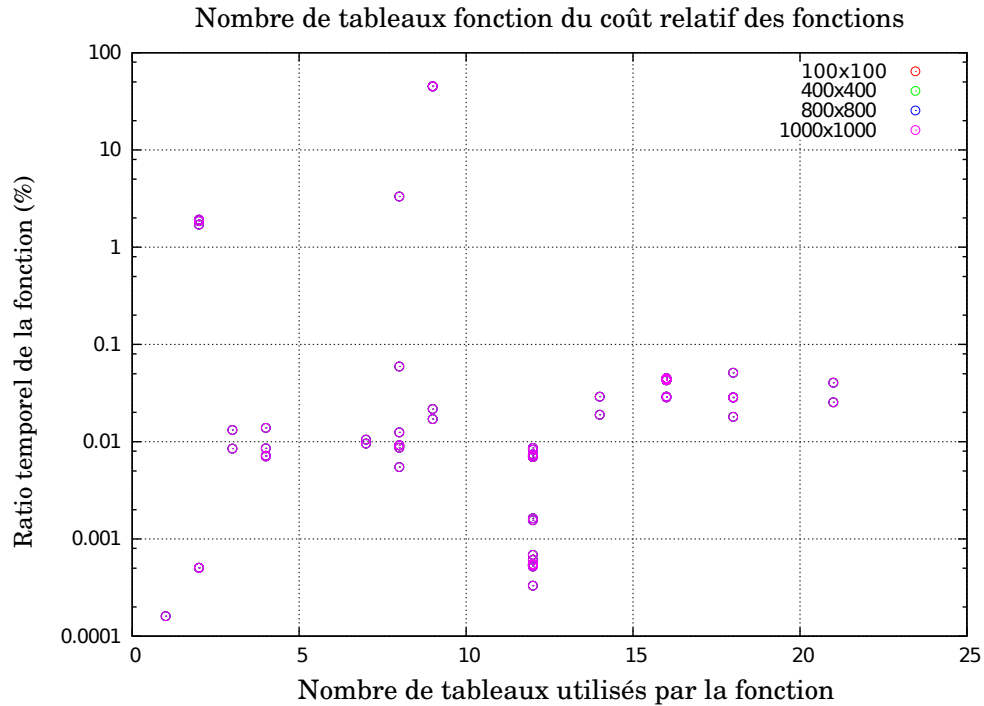


FIGURE 3.15 – Distribution de l'utilisation mémoire des fonctions du programme EulerMHD sur la base du temps d'exécution relatif des fonctions et du nombre de tableaux manipulés. Remarquons que la distribution est pour ce programme indépendante de la taille du problème.

ainsi d'évaluer les gains potentiels d'un apport d'information extérieur. L'alignement de l'adresse de base du tableau est choisi parmi un intervalle de $[0, 4Ko[$ selon les règles qui suivent :

Aléatoire : Les valeurs sont choisies aléatoirement avec un pas de 16 ou 64 octets.

Incrément : Les valeurs sont choisies selon une méthode tourniquet en incrémentant une variable globale de 16 ou 64 octets.

Groupe : L'algorithme génère des groupes rassemblant les tableaux utilisés simultanément dans les différentes fonctions comptant pour plus que 1% du temps d'exécution total. Les groupes ayant des tableaux communs sont fusionnés de sorte à agréger les tableaux en relation pour le programme dans son ensemble. Chaque groupe est alors associé à un décalage global et les alignements des tableaux sont répartis de sorte à être dispersés au sein de l'espace $[0, 4Ko[$ avec un pas minimum de 16 ou 64 octets.

Les résultats expérimentaux montrent toutefois (figure 3.16) que le choix d'une politique strictement aléatoire conduit au meilleur gain, obtenant des performances aussi bonnes qu'une méthode beaucoup plus lourde basée sur l'exploitation de profils. Remarquons qu'empiriquement l'utilisation d'une discrétisation par pas de 64o de l'espace des alignements possible semble apporter les meilleurs résultats. Cette observation est cohérente avec l'utilisation de lignes de cache de 64 octets par le matériel.

3.8 Conclusion

Dans ce chapitre, nous avons étudié en détail les problèmes d'interférences pouvant survenir entre les politiques de gestion de la mémoire au niveau de l'OS et celles mises en place au niveau de l'allocateur. Nous avons rappelé le problème de fuite de cache observé sous Linux et traité par des méthodes de coloration de pages par d'autres OS. Ces approches sont souvent décrites dans les théories des mécanismes de pagination. Nous avons toutefois montré que le choix

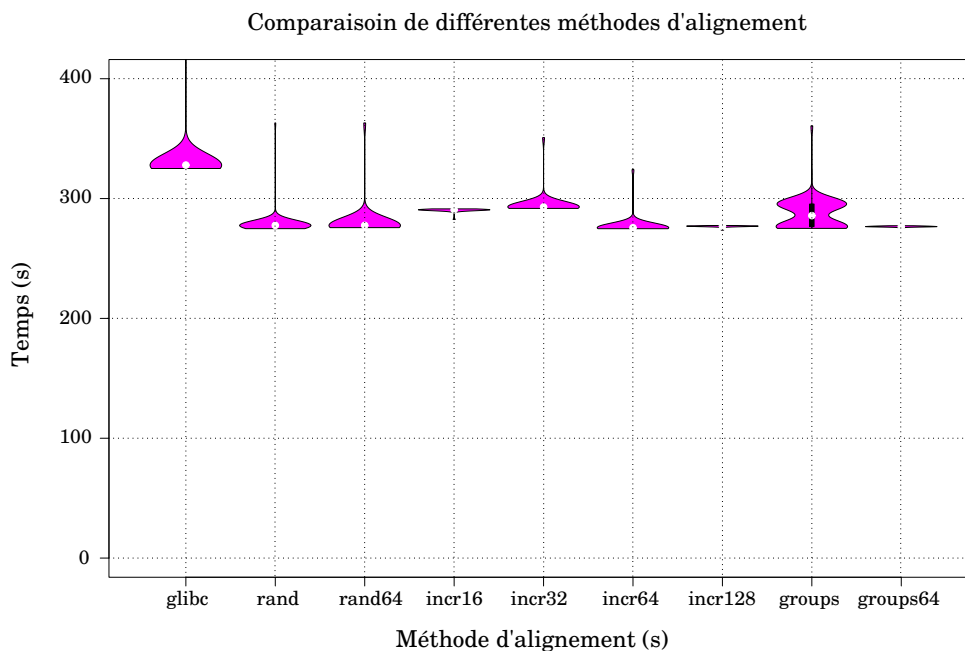


FIGURE 3.16 – Distribution de différentes exécutions du programme EulerMHD pour différentes règles d'alignement.

d'une méthode de coloration n'était pas sans conséquence et que les techniques habituellement employées impliquaient une prise en compte explicite du problème au niveau de l'allocateur mémoire en espace utilisateur (*malloc*).

Nous avons également observé que les allocateurs actuels tendent à prendre des décisions conduisant par défaut aux pertes de performances observées dans ce chapitre. Il en ressort que la méthode aléatoire exploitée par Linux évite ce type de problème montrant notamment en parallèle une plus grande résistance à ces cas pathologiques. Certaines améliorations ont donc été proposées tant au niveau des politiques de coloration qu'au niveau de l'allocateur, les deux pouvant être utilisées de manière complémentaire. Remarquons que l'étude [ADM11] décrit le problème de placement des macros blocs de manière générique, mais ne traite pas de leur interaction avec la politique de pagination sous-jacente.

Dans la suite, nous allons discuter les problématiques entourant le développement d'un allocateur parallèle avec support NUMA. Nous nous intéresserons alors aux performances des fautes de pages pour l'allocation de grands tableaux nécessaires en HPC.

Chapitre 4

Mécanismes d'allocations parallèles et contraintes mémoires

Dans le chapitre précédent, nous nous sommes attachés à étudier les interactions pouvant s'établir entre la politique de l'OS et l'allocateur mémoire en terme de choix des adresses. Ici, nous allons décrire la mise en place d'un allocateur parallèle dans le cadre du projet MPC. Nous décrirons donc dans un premier temps les contraintes prises en compte et l'état des lieux des allocateurs disponibles. Une fois de plus nous nous intéresserons principalement à la problématique de gestion de grands volumes de données, notamment vis-à-vis des performances de leur allocation par l'OS.

Nous verrons notamment qu'avec un nombre croissant de cœurs, Linux est affecté par un problème d'extensibilité des performances de ses fautes de pages. Le problème sera traité ici du point de vue de l'allocateur en visant à réduire l'interaction de ce dernier avec l'OS. Nous discuterons notamment la possibilité d'établir un compromis entre consommation mémoire et efficacité d'allocation des grands segments. Pour ce faire, nous considérerons un environnement dans lequel nous serons amenés à faire plus attention aux problématiques de consommation.

Nous développerons également les points concernant le support des architectures NUMA. Ces architectures nécessitent une prise en charge explicite par l'allocateur lorsque l'on fonctionne en mode multithread. Nous verrons à cette occasion que MPC nous permet d'exploiter des informations utiles, non disponibles en temps normal.

4.1 Approche générale

Comme nous l'avons discuté en section 2.4, un allocateur mémoire a pour principal objectif de maintenir à jour une liste de blocs non utilisés et d'offrir un algorithme efficace pour répondre aux requêtes de l'utilisateur. Il gère également les échanges de mémoire avec l'OS. Rappelons toutefois qu'il n'est pas possible de prédire les propriétés des requêtes à venir. Les algorithmes sélectionnés se construisent donc sur la base d'heuristiques. Un algorithme sera considéré comme bon, s'il est affecté par un nombre réduit de cas problématiques (explosion de la consommation mémoire ou du temps de décision) et qu'il offre de bonnes performances.

À ce titre, de nombreuses stratégies ont été évaluées au fil du temps. On pourra notamment se référer à une étude bilan réalisée par Wilson en 1995 [WJNB95] pour obtenir une vue des différentes techniques mises en place dans les allocateurs de l'époque. Cette étude traite notamment le problème de fragmentation mémoire. Willson y décrit le problème sous la forme de trois niveaux conceptuels. La *stratégie* tente d'exploiter les régularités du flux de requête. La *politique* est un choix de procédure implémentable pour placer les blocs en mémoire. La *mécanique* est un

ensemble d’algorithmes et structures de données permettant d’implémenter la politique. Cette séparation peut s’appliquer de manière générale à tout algorithme, mais prend une tournure toute particulière dans la conception d’un allocateur mémoire. Ce dernier doit en effet résoudre simultanément le problème d’allocation des éléments demandés et de ses propres métadonnées. Cette double problématique contraint largement les algorithmes en les couplant fortement à la structure des métadonnées.

Nous commencerons donc par décrire les besoins particuliers auxquels on s’intéresse afin de pouvoir construire une stratégie adaptée. Nous étudierons également les différents allocateurs disponibles allant dans le sens de nos besoins. Nous entrerons finalement dans la description de nos politiques et mécaniques d’allocation propres.

4.2 Description du besoin

Comme discuté au chapitre 1, nos travaux s’insèrent dans le cadre du développement de l’environnement de programmation MPC (section 1.7.9). Cet environnement vise à fournir un support exécutif pour les grappes de noeuds multicœurs de type NUMA avec pour objectif un fonctionnement à l’échelle sur les supercalculateurs actuels disposant de centaines de milliers de cœurs. Dans ce contexte, la bibliothèque MPC a besoin d’un allocateur efficace pour son mode de fonctionnement canonique, à savoir : un nombre de processus réduit (typiquement un par nœud) et une gestion des tâches internes (MPI, OpenMP...) à base de threads. Les points clés guidant notre développement peuvent donc être énumérés comme suit :

Parallèle : Les architectures actuelles nous orientent vers une programmation multithread, les allocateurs mémoires doivent donc être conçus pour être appelés en parallèle en minimisant l’utilisation de synchronisations. Depuis le milieu des années 2000, certains allocateurs sont disponibles en respectant cette caractéristique. Nous en étudierons certains en section 4.4.

NUMA : MPC vise un fonctionnement sur architecture NUMA. Or, en contexte multithread, l’allocateur peut être sollicité de ce point de vue pour deux raisons : l’une, liée aux impacts des primitives de synchronisation potentiellement utilisée par l’allocateur lui-même ; l’autre, liée au recyclage de blocs notamment pour les petites allocations. Or, en contexte NUMA, il peut être important d’éviter les recyclages transversaux, conduisant à l’échange de blocs entre threads distants (au sens NUMA). Remarquons qu’il n’existe pas d’allocateurs génériques de niveau production prenant explicitement en compte cette problématique. Ce type de problème est par exemple discuté dans [DSR12]. Nous montrerons que MPC nous permet d’exploiter des informations habituellement non disponibles pour faciliter ce support (section 4.11).

Problématique des allocations moyennes/grosses : Les allocateurs mémoires sont très souvent comparés sur la base de leurs performances brutes d’allocation. Or, les benchmarks utilisés pour ces comparaisons se concentrent majoritairement sur un nombre important de petites allocations et des volumes relativement faibles de données. Dans un contexte HPC, il importe de remarquer que l’on exploite aujourd’hui des ensembles de données de plusieurs giga-octets, bien loin des ensembles de quelques Mo traités dans nombre de benchmarks. D’autre part, les architectures actuelles requièrent de limiter les petites allocations pour ne pas réduire l’efficacité du processeur. Les simulations numériques ont donc une certaine tendance (ou devraient idéalement tendre) à favoriser l’exploitation de segments moyens ou grands. C’est donc dans cette optique que vont s’orienter nos travaux, considérant que les problèmes des petites allocations sont très largement traités dans la littérature actuelle. Nous verrons que le traitement des gros segments est différent, notamment, parce que les coûts d’allocation ne se limitent pas à la fonction d’allocation

elle-même. Ceci du fait des politiques de pagination paresseuses utilisées par les OS modernes.

Compromis économie/performances : Nous discuterons la problématique de choix entre une politique d'économie mémoire et d'orientation vers la performance. Nous discuterons également la possibilité de rendre ce choix dynamique afin de s'adapter aux différentes phases des programmes.

Segments utilisateurs : Nous décrirons la méthode annexe retenue par notre allocateur afin d'inclure nativement la gestion de segments spécialement préparés par l'utilisateur. Cette intégration devra, si possible, être intégrée de manière compatible avec les routines d'allocation standards.

4.3 Aspects génériques des allocateurs

Au-delà des caractéristiques listées ci-dessus, un allocateur mémoire doit répondre à certaines contraintes techniques nécessaires à leur bon fonctionnement. Nous discutons ici les quatre points centraux dans la construction d'un allocateur mémoire : la gestion des blocs libres, la fusion et scission de blocs, les alignements et le placement des métadonnées.

4.3.1 Gestion des blocs libres

Un allocateur mémoire remplit essentiellement trois fonctions internes :

1. Demander et rendre de la mémoire au système d'exploitation sous-jacent.
2. Maintenir un suivi des blocs libres non rendus à l'OS pour réutilisation futur.
3. Trouver le meilleur choix de réutilisation parmi les blocs libres disponibles.

Au titre des points 2 et 3, l'allocateur consiste essentiellement à fournir une liste de blocs libres. L'organisation de cette dernière vise alors à optimiser la capacité à trouver rapidement un bloc réutilisable en réponse à une requête de l'utilisateur. La méthode retenue doit également minimiser les problèmes de fragmentation afin de limiter la consommation mémoire. On distingue habituellement les algorithmes suivants pour cela :

Meilleur ajustement (*best-fit*) : L'algorithme cherche le bloc libre ayant la taille la plus proche de la requête. Appliquée de manière stricte cette méthode peut nécessiter un parcours de l'ensemble des blocs libres. Il est généralement appliqué de manière partielle en donnant une borne sur le nombre d'éléments parcouru.

Premier suffisant (*first-fit*) : L'algorithme retient le premier élément de taille suffisante dans la liste parcourue. Cette méthode est l'une des plus employées. Certaines études montrent d'ailleurs qu'elle peut tendre vers l'efficacité de l'algorithme *best-fit* [WJNB95]. Pour ce type d'algorithme, on a le choix d'ordre de la liste de sorte à favoriser la réutilisation des blocs récents (*LIFO* : *Last In, Last Out*) ou plus anciens (*FIFO* : *First In, First Out*). Une alternative intéressante peut également consister à trier les éléments par adresse de sorte à favoriser la localité spatiale.

Prochain suffisant (*next-fit*) : Simple évolution de l'algorithme *first-fit*, ce mode se rappelle de la dernière position dans la liste et reprend la recherche à cette position. Cette approche tend toutefois à générer plus de fragmentation et une moins bonne localité du fait de l'étalement des blocs dans l'espace d'adressage.

Plus grand disponible (*worst-fit*) : A l'opposé de l'algorithme *best-fit*, l'algorithme cherche le bloc le plus grand disponible, ceci afin de maintenir des *résidus* de taille suffisante lors des scissions.

Une amélioration importante peut s'obtenir en construisant des listes distinctes (*ségrégation*) pour différentes tailles de blocs. On obtient alors un algorithme plus proche d'un algorithme de type *best-fit* sans avoir le surcoût d'un parcours complet. L'allocateur peut alors fonctionner en imposant des tailles précises, impliquant toutefois une augmentation de la fragmentation interne, ou bien, préférer travailler par classes de tailles approximatives.

4.3.2 Fusion et scission de blocs

Au-delà de cette sélection d'éléments libres, il importe de remarquer qu'un bloc trop grand peut être *scindé* pour satisfaire une requête plus petite. De la même manière, deux blocs libres voisins peuvent être *fusionnés* pour donner un bloc plus grand. Sur ce point, les allocateurs peuvent décider d'appliquer les fusions et scissions de manière immédiate, différée ou de ne pas en effectuer. L'approche immédiate permet d'ajuster au plus près la consommation mémoire du programme, mais induit un surcoût si les blocs fusionnés doivent être à nouveau scindés pour répondre aux requêtes suivantes. De nombreux allocateurs introduisent donc un cache et maintiennent ainsi un certain nombre de blocs non fusionnés accessibles rapidement[Eva06, SG].

Les fusions de blocs nécessitent la connaissance de taille des blocs voisins. Ce critère important doit être pris en compte dans la construction des métadonnées de l'allocateur de manière à permettre une implémentation efficace de ces actions. À ce titre, dans le cadre d'une approche sur base de liste chaînée, l'utilisation d'algorithme *first-fit* avec un tri par adresse peut permettre de mettre en commun les métadonnées de fusion et recherche de blocs libres.

Une autre approche introduite par Knuth[Kno65, PN77] dite *buddy allocator* consiste à n'autoriser qu'un nombre réduit de tailles construites sur la base d'une suite numérique. Cette structure permet des fusions et scissions rapides. On utilise habituellement des puissances de deux (*binary buddies*) permettant des calculs rapides sur la base d'opérations binaires. Il est aussi possible d'utiliser des suites plus complexes (Fibonacci...). Cette approche permet des algorithmes performants et réduit la fragmentation externe. Il augmente toutefois significativement la fragmentation interne pour les grandes tailles.

La méthode extrême consiste à mettre en place une ségrégation au niveau du stockage lui-même en interdisant le mélange de blocs de différentes tailles. Différentes régions sont alors mises en place pour générer un nombre prédéterminé de tailles de blocs. Dans ce cas de figure, l'allocateur n'effectue que les fusions et découpages par grands ensembles. Cette approche est retenue dans un certain nombre d'allocateurs récents, dont les trois principaux que nous allons étudier.

4.3.3 Contraintes d'alignements

Sur les architectures x86 (et de nombreuses autres), tout élément doit être aligné sur sa propre taille jusqu'à 8 octets. Un allocateur permettant le voisinage de blocs de tailles différentes doit donc allouer des blocs d'un minimum de 8 octets afin d'assurer que le voisin suivant

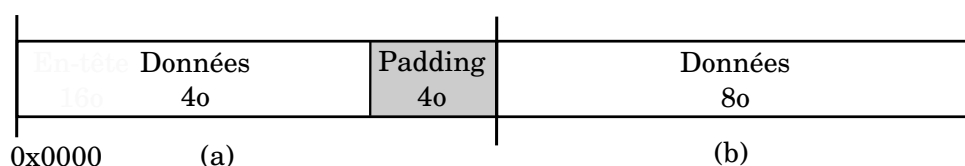


FIGURE 4.1 – Illustration du remplissage rendue obligatoire par la règle d'alignement mémoire. On suppose ici l'allocation d'un bloc (a) de 4 octets suivi d'une allocation (b) de 8 octets ou plus.

vérifie bien lui aussi cet alignement. Le cas est illustré dans la figure 4.1. Certains allocateurs choisissent donc d'appliquer des politiques de ségrégation pour éviter le mélange des petits blocs et ainsi permettre de ne pas grossir artificiellement ces derniers. Cela n'a toutefois un intérêt que pour les blocs entre un et huit octets. Toutes les allocations de taille supérieure doivent forcément respecter cet alignement. L'utilisation des unités vectorielles (Intel SSE/AVX...) nécessite aujourd'hui des alignements supérieurs (32 octets pour AVX), nous remarquerons toutefois que les allocateurs génériques ne prennent pas en charge cette spécificité qui est laissée au soin des utilisateurs au travers des fonctions de type *memalign* et *posix_memalign*.

4.3.4 Placement des métadonnées

Rappelons que les fonctions de libération (*free*) et redimensionnement (*realloc*) utilisent l'adresse de base des segments comme identifiant. Lors de ces appels, l'allocateur doit donc retrouver les métadonnées (en général la taille, l'état libre ou alloué et des informations sur les blocs voisins) du bloc à partir de cette simple adresse. La méthode usuelle consiste à placer ces informations juste avant le segment afin de pouvoir retrouver cet en-tête par simple décrétement de l'adresse fournie.

Cette approche simple à implémenter a toutefois trois effets néfastes. En cas de dépassement de tableaux, le programme a de fortes chances de corrompre les en-têtes de l'allocateur conduisant à un plantage. Ce type d'erreur est en général difficile à déboguer sans outils tels que valgrind[NS07] ou des approches par bibliothèques telles que electric-fence ou similaires[LLC10]. Le second impact est lié à la contrainte d'alignement discutée plus tôt. Les en-têtes doivent eux aussi satisfaire certains alignements pour être manipulés, en général huit octets pour les adresses et tailles encodées en 64 bits. Ceci augmente donc la fragmentation interne pour les petites allocations. Enfin, toujours dans le cas des petites allocations, l'espace occupé par les en-têtes génère une réduction d'efficacité des caches en occupant une place, qui idéalement serait utilisée par les données utilisateurs (rappelons que les échanges vers les caches sont effectués sur la base d'éléments de 64 octets).

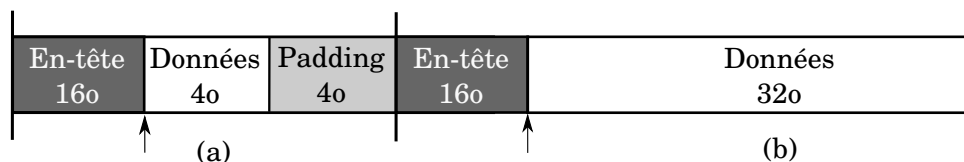


FIGURE 4.2 – Illustration d'exploitation d'en-têtes placés en début de segments. Le segment (a) illustre un cas de fragmentation interne importante lié aux contraintes d'alignement et occupation de place de l'en-tête.

Une deuxième école consiste donc à externaliser les en-têtes en fournissant un moyen efficace de retrouver ces derniers à partir de l'identifiant constitué par l'adresse de base du bloc. Ce type d'allocateur doit toutefois résoudre le problème supplémentaire de gestion de l'espace de stockages des métadonnées. Nous verrons un exemple avec l'allocateur Jemalloc de FreeBSD.

4.4 Allocateurs disponibles

L'étude de Wilson a pris place dans un contexte essentiellement séquentiel, considérant les allocations comme un problème de *consommation mémoire* (fragmentation) et d'*efficacité séquentielle* des algorithmes de décision. Entre-temps, l'arrivée du multicœur pose une problématique nouvelle en requérant des allocateurs mémoires dans un premier temps *thread-safe* (impliquent habituellement des synchronisations coûteuses), mais de préférence parallèles. Dans ce

contexte, ce domaine a connu une reprise d’activité au milieu des années 2000 avec le développement de nouveaux allocateurs prenant en compte cette problématique.

Afin de familiariser le lecteur avec leurs caractéristiques propres, nous allons décrire ici le fonctionnement de certains des allocateurs parallèles disponibles que nous utiliserons tout au long de cette étude. Il s’agit notamment de l’allocateur `Dlmalloc/Ptmalloc`[Lea, Glo] fourni par la libc Linux, `Jemalloc`[Eva06] fournit par FreeBSD, `TCMalloc`[SG] développé par Google et `Hoard`[BMBW00] un allocateur ayant inspiré FreeBSD. Nous discuterons également succinctement un point intéressant de l’allocateur `MAMA`[KK06]. Notons que nous ne décrirons pas ces allocateurs dans tous les détails, mais nous focaliserons sur les aspects intéressants pour nos travaux.

4.4.1 Linux : `dlmalloc` et `ptmalloc`

Linux fournit un allocateur dit `dlmalloc`[Lea] implémenté à l’origine par Doug Lea. Pour le parallèle, cet allocateur nécessite la mise en place de synchronisations globales pénalisant les applications intensives en terme d’allocation. Il a aujourd’hui été dérivé par Wollfram Glober (`ptmalloc`[Glo]) en lui ajoutant la notion d’arène. Cette approche fournit des allocateurs distincts en segmentant l’espace virtuel. Lors d’une allocation, un thread tentera d’utiliser l’arène qu’il a précédemment utilisée. En cas de conflit (verrou actif) il migrera à la suivante. En cas d’épuisement, une nouvelle arène est créée. Les arènes sont définies par leur plage d’adresse et nécessitent donc un grand espace virtuel pour fonctionner avec un nombre important de threads. Cette méthode est parfois critiquée[BMBW00, SG, KK06] comme pouvant conduire à une surconsommation mémoire et générant du faux partage entre les threads.

D’un point de vue caractéristique, cet allocateur utilise des listes doublement chaînées pour gérer les blocs libres. Les métadonnées sont placées en bordure de bloc (début et fin). Ce type d’approche implique une taille minimale d’allocation de seize octets, les métadonnées occupant vingt-quatre octets (deux tailles et des bits d’états) pour des adresses en 64 bits. Les blocs libres sont maintenus dans des listes par groupe de taille dans lesquelles l’algorithme applique une recherche par meilleure correspondance afin de limiter la fragmentation. Cet allocateur prend également en compte certaines considérations de localité en testant les blocs proches de l’allocation précédente s’ils répondent à la requête de manière exacte afin de favoriser le voisinage de blocs ayant des chances d’avoir des durées de vie similaires et accès simultanés.

Les versions séquentielles de l’allocateur utilisent `brk` pour réserver la mémoire et `mmap` pour les segments au-delà d’un certain seuil (en général 128Ko ou 256Ko). L’utilisation de `mmap` pour les gros segments permet d’éviter l’apparition de fragmentation externe sur des segments de grandes tailles. Ces blocs sont libérés avec `munmap`. Les trous formés restent donc virtuels et n’ont pas de conséquence sur la consommation de mémoire physique.

4.4.2 Hoard

`Hoard`[BMBW00] est un allocateur parallèle développé en 2000 par Emeri Berger à la suite de sa thèse sur les allocateurs mémoires. Dans ses travaux de thèse[Ber02], il s’est surtout intéressé à la capacité à composer rapidement et efficacement des allocateurs mémoires personnalisés pour les applications. Il propose alors des briques (algorithmes) de bases fournies sous forme de *templates* C++. Il a ainsi construit un allocateur plus général à partir des observations obtenues pendant ses travaux.

Cet allocateur apporte certaines notions importantes reprises par l’allocateur `Jemalloc` de FreeBSD. On notera essentiellement un fonctionnement à deux niveaux avec des tas locaux et

un tas global permettant une meilleure extensibilité. Les échanges entre tas locaux et globaux se font par “super-blocs”, des segments de grandes tailles qui seront ensuite découpés localement. Le problème de contention sur le tas global est supposé faible du fait de la taille importante des super-blocs échangés (de l’ordre du Mo). Concernant le découpage des super-blocs, il introduit également un principe de ségrégation du contenu des super-blocs en forçant un découpage de ces derniers en blocs de tailles uniques. Les super-blocs sont échangés avec le tas global s’ils franchissent un seuil de blocs libres de manière à limiter la surconsommation mémoire des tas locaux. Les gros segments sont toujours traités directement par appels à *mmap/munmap*. Bien qu’intéressant, nous allons voir que la version actuelle de cet allocateur supporte mal la grosse simulation numérique utilisée pour nos tests.

4.4.3 Jemalloc

Cet allocateur a été implémenté par JASON EVANS pour FREEBSD et NETBSD, il est aujourd’hui intégré par défaut dans FIREFOX et repris par FACEBOOK. Il reprend les grands principes mis en place dans Hoard : les super-blocs et leur découpage en objets de tailles uniques. JEMALLOC, découpe ses super-blocs en *runs* eux-mêmes découpés selon une taille fixe. Lors des allocations, les *run* d’une classe de taille donnée sont utilisés jusqu’à leur remplissage total. Puis, un nouveau *run* est choisi parmi ceux partiellement vides en suivant des règles de priorité fonction du taux de remplissage. Cette remarque de la part de l’auteur est intéressante pour assurer une purge et compacité maximale des super-blocs. Il peut ainsi rendre les pages inutilisées à l’OS. Comme nous le verrons, cet allocateur est relativement efficace en terme de mémoire avec une bonne maîtrise de la fragmentation, argument qui a notamment intéressé l’équipe de Firefox.

Sur le plan technique, il est également intéressant d’observer l’utilisation de champs de bits en début de *run* pour notifier l’état d’occupation des sous-blocs. Cette approche permet de ne pas placer d’en-têtes au milieu des blocs alloués tout en garantissant un accès en temps constant à ces derniers. Cela permet de plus, de se prémunir d’un écrasement des métadonnées en cas de dépassement. Cette méthode originale est décrite comme rarement utilisée dans l’étude de Wilson. Les grandes allocations sont indexées par un arbre bicolore en les supposant peu nombreuses du fait de leur grande taille (supérieur à 1Mo). Contrairement à Hoard, les super-blocs sont échangés directement avec l’OS, il n’y a donc pas d’échange direct entre les tas locaux. En ce qui concerne l’organisation topologique, l’allocateur crée $4P$ arènes (P le nombre de processeurs) et les associe par tourniquet à chaque nouveau thread afin d’obtenir un nombre constant de threads par tas. Ce problème particulier est tiré de discussion de la part de E. Berger et J. Bonwick[BMBW00, BA01] dans les années 2000. Le but est de prendre en compte les programmes créant et détruisant de nombreux threads tout en assurant une répartition homogène des threads sur les différents tas. Notons que contrairement à Ptmalloc, cette association est fixe. L’organisation générale de Jemalloc est visible sur la figure 4.3 extraite de leur documentation. On y observe l’utilisation fréquente d’arbres colorés rouge/noir ainsi que la notion de cache locale à chaque thread similaire à TCMalloc.

Jemalloc tend à avoir une politique de libération agressive en renvoyant régulièrement la mémoire vers l’OS. Ce point, couplé à de bonnes heuristiques, lui permet de maintenir une faible consommation mémoire. Nous verrons toutefois dans la suite que cela donne aussi la limite de Jemalloc. Cet allocateur utilise des super-blocs de 2 ou 4Mo et force leurs alignements mémoires sur cette taille. Au vu des discussions de la section 3.5.3, il est clair que ce choix peut poser problème notamment s’il est appliqué sur grosses pages (cas de FreeBSD). Une correction possible serait donc d’utiliser des super-blocs de tailles non multiples de la taille des voies du cache, de ne pas forcer d’alignement particulier ou d’introduire un décalage aléatoire à l’intérieure des blocs comme discuté en section 3.7.

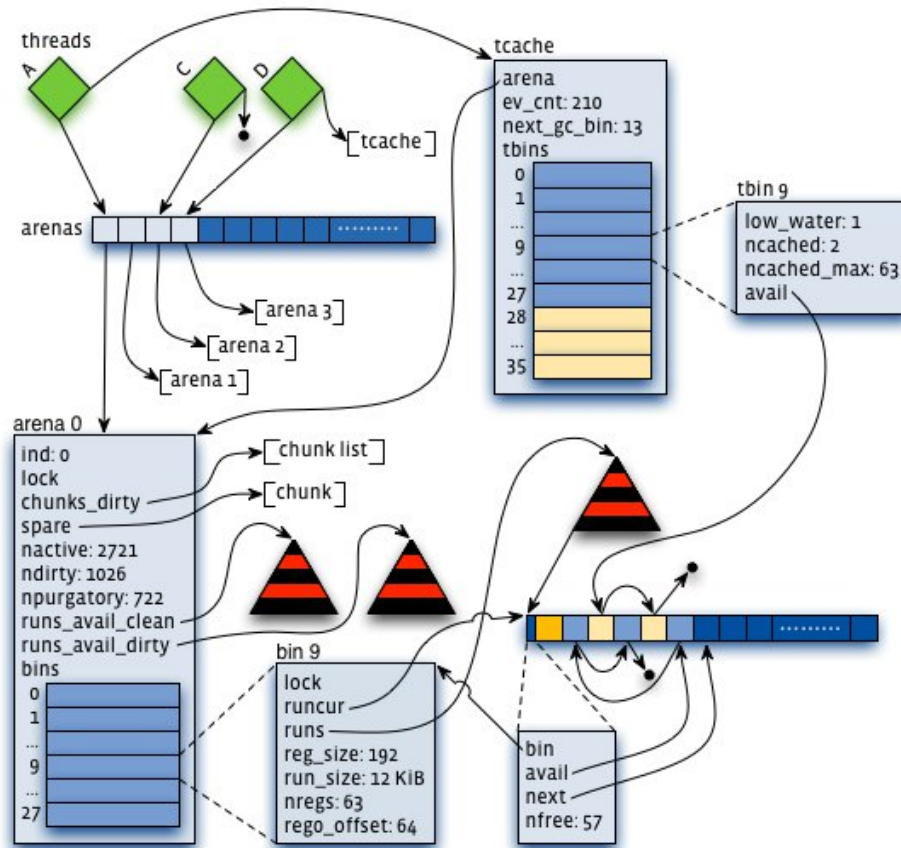


FIGURE 4.3 – Structure interne de Jemalloc extraite de la documentation de l'allocateur[Eva06]. Il donne les liens entre les structures principales : les arènes, les caches locaux aux threads et les runs.

4.4.4 TCMalloc

TCMalloc[SG] est un allocateur développé par Google. Cet allocateur reprend principalement la critique sur PTMalloc vis-à-vis du non-échange de blocs entre les différents tas locaux. Il reprend donc les approches précédentes avec un cache de bloc local et un tas global. Contrairement à jemalloc, TCMalloc génère des purges du cache local au-delà d'une certaine taille, et compte donc plus fréquemment sur son tas global. Cette approche est certainement intéressante en cas de flux d'allocation/libération régulier et contrôle mieux la mémoire maintenue dans les threads. En contrepartie, il semble possible de perdre l'efficacité de l'approche si les libérations se font par paquets suivis de paquets d'allocations. Dans cette situation, les caches locaux vont en effets tendre à se vider pour finalement devenir inefficaces lors du flot d'allocations suivant.

Les versions récentes de TCMalloc introduisent un mécanisme de libération lente de la mémoire vers l'OS en procédant de manière graduelle avec un plafond de *débit de libération* (TCMALLOC_RELEASE_RATE). Sous Linux, ces libérations sont effectuées à l'aide d'un appel `MADVISE(MADV_DONTNEED1)` sur les pages non utilisées. Son tas principal est donc construit comme un allocateur général de page en fusionnant et découpant des ensembles contigus de gros segments dits *span* et balayé régulièrement par des fonctions de libération mémoire.

Cet allocateur est relativement efficace en terme de performance, nous verrons toutefois que

1. Le drapeau `MADV_DONTNEED` permet de rendre les pages physiques à l'OS tout en laissant le segment projeté dans l'espace virtuel.

cela se fait au prix d'une surconsommation mémoire. L'allocateur montre également ses limites sur nœud NUMA du fait d'un non-support explicite de ces architectures et de sa méthode de maintien de la mémoire en espace utilisateur. Des développeurs d'AMD ont effectué quelques travaux pour intégrer un support explicite des architectures NUMA dans cet allocateur[Kam]. Les sources ne sont toutefois pas disponibles pour test.

4.4.5 MAMA

L'allocateur MAMA[KK06] développé par Cray reprend la discussion sur les allocateurs parallèles en critiquant les approches précédentes. Ils critiquent la création d'autant de tas que de threads nourris par un tas centralisé (potentiellement l'OS). Les auteurs partent du principe que ces approches ne peuvent pas passer à l'échelle au-delà d'un certain seuil. Ils remarquent en effet que l'augmentation du nombre et du coût des synchronisations sur le tas central impose l'échange de blocs toujours plus gros pour réduire le nombre d'échange. Pour eux, cette approche induit une augmentation de la mémoire retenue par les tas locaux qui peut à terme devenir inacceptable.

Les auteurs proposent donc une approche intéressante visant à faire collaborer les différents threads lors des allocations. Si plusieurs threads sont en attente d'un verrou, alors rien n'interdit que ces derniers ne se mettent d'accord pour que l'un d'entre eux récupère les blocs pour l'ensemble. Les blocs peuvent ensuite être distribués à tout le monde. Cette approche réduit les prises de verrous auprès des structures centrales et maintient un certain nombre d'opérations en cache. Les sources de cet allocateur ne sont pas disponibles, nous n'avons donc pas pu le tester. Nous discuterons toutefois une intégration potentiellement intéressante de ce type d'approche au vu de nos travaux dans une partie discussions en fin de chapitre.

4.5 Impact des allocateurs

Afin d'appuyer notre construction, nous donnons ici quelques résultats préliminaires obtenus à l'aide des allocateurs précédemment cités. Nous prendrons comme base de test l'application Hera décrite en section 1.10. Afin de stresser l'allocateur en parallèle, nous avons utilisé cette dernière couplée avec MPC en utilisant son mode canonique : utilisation de tâches MPI sous forme de processus légers.

L'application Hera est très intensive en terme d'allocation mémoire avec une génération de l'ordre d'un million d'allocation mémoire pour 5 minutes d'exécution sur 12 threads. Les allocations se répartissent en trois groupes principaux (voir figure 4.4). On observe un groupe de petites allocations de courtes durées de vies liées à la structure C++ de l'application. S'ajoute à cela, des allocations et libérations régulières de blocs moyens (de 1 à 4 Mo) liés à la structure AMR du code, dont certains avec une courte durée de vie. Nous remarquerons également un grand nombre d'appels à *realloc*, notamment sous la forme de réallocations croissantes. Ces dernières tendent à générer une forte fragmentation au niveau de l'allocateur et à poser des problèmes importants de performances suivant les implémentations retenues. La gestion des gros blocs à courte durée de vie représente le problème majeur de cette application. Ces blocs impliquent en effets des échanges systématiques avec l'OS mettant en exergue les limites de passage à l'échelle de ce dernier, notamment vis-à-vis des fautes de pages.

La table 4.1 donne les performances obtenues avec les différents allocateurs sur 12, 32 et 128 cœurs. On y remarquera l'évolution des tendances. Sur 12 cœurs (calculateur A) TCmalloc apporte un léger gain de performance (2 %) alors que jemalloc offre de bonnes performances couplées à des gains mémoires importants (2.3 Go économisés sur 3.3 Go utilisés par la

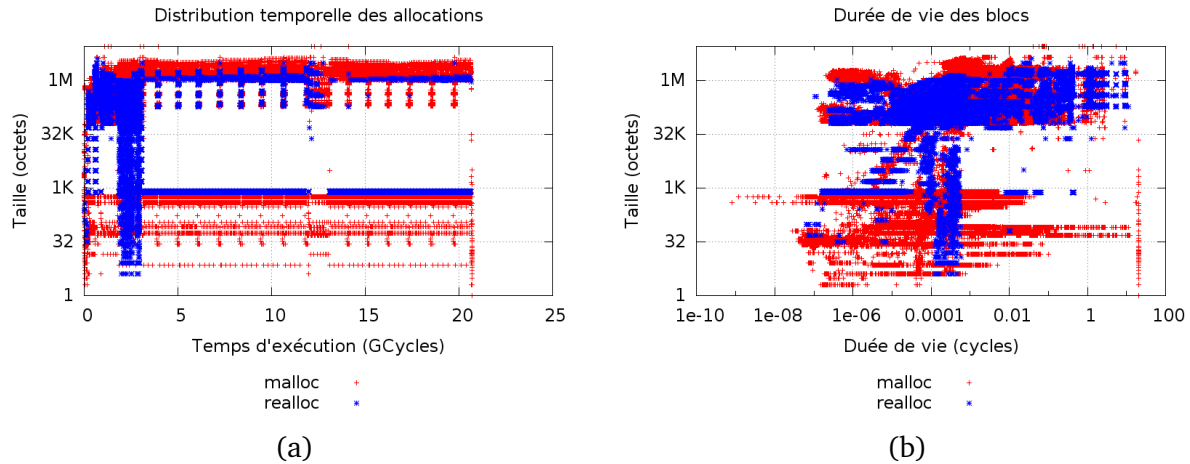


FIGURE 4.4 – Distribution en taille et temporelle des allocations mémoires de l'application Hera sur 12 cœurs pour un problème à 1.4 million de mailles. Sont données, en fonction de la taille des blocs : (a) la répartition temporelle et (b) la durée de vie de ces allocations.

glibc). L'allocateur de la glibc offre des performances similaires et une consommation mémoire moyenne. On s'intéresse principalement à l'évolution des performances sur 32 et 128 cœurs. Sur ces architectures on observe une nette dégradation associée à jemalloc et TCMalloc. Le premier est impacté par une forte augmentation du temps système que l'on impute au nombre trop important d'échanges avec l'OS. Nous supposons pour l'instant que la perte de performance de TCMalloc est pour partie liée à un non-support du NUMA. Nous discuterons et confirmerons ce point dans la suite. L'allocateur Hoard supporte très mal notre application dans l'ensemble des configurations testé très probablement à cause d'un support plus faible des appels à *realloc*, point sensible de cette application. Certains tests avec cet allocateur conduisent à une explosion mémoire empêchant la terminaison de la simulation. Nous n'avons donc pas inclus les résultats partiels obtenus avec cet allocateur.

A : Nœuds 12 cœurs Cassard (2 * 6)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (Go)
1	Standard glibc	143.89	130.10	8.53	3.3
2	Jemalloc	143.05	128.07	14.53	1.9
3	TCMalloc	141.14	139.98	0.65	6.9
B : Nœuds 32 cœurs Tera 100 (4 * 8)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (Go)
1	Standard glibc	101.11	67.43	9.41	8.1
2	Jemalloc	145.73	70.49	57.32	6.7
3	TCMalloc	106.28	82.97	1.96	8.6
C : Nœuds 128 cœurs Tera 100 (4 * 4 * 8)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (Go)
1	Standard glibc	284.06	170.94	15.9	14.1
2	Jemalloc	351.49	214.54	123.99	12.2
3	TCMalloc	438.42	396.59	27.57	14.4

TABLE 4.1 – Mesure préliminaire des performances de l'application Hera sur différents calculateurs NUMA en fonction de l'allocateur retenu. Les tests sont effectués dans le mode canonique de MPC, à savoir, un processus par nœud et un thread par cœur physique. Pour être comparables, les temps utilisateurs et systèmes sont donnés par thread.

4.6 Structure de l'allocateur

Dans cette section nous allons décrire les concepts clés utilisés pour construire notre allocateur mémoire. Pour ce faire, nous tenterons de répondre aux contraintes décrites en section 4.2 et de prendre en compte les observations de la section précédente.

4.6.1 Organisation générale

Comme on peut le voir sur le schéma 4.5 notre allocateur se construit sur la base de deux éléments principaux :

Tas local : Tout comme Hoard, Jemalloc ou TCMalloc, ce composant prend en charge la gestion locale des allocations. Il maintient essentiellement une liste de blocs libres générés par la découpe de *macro blocs* plus gros. C'est à ce niveau qu'interviennent les algorithmes de décision, fusion et découpage de blocs. Une instance est créée pour chaque thread. Le tas local du thread est retrouvé par le biais d'un pointeur de type TLS² initialisé lors de la première allocation.

Source mémoire : Ce composant offre une manière générique de demander de la mémoire et offre la possibilité d'implémenter un cache entre la source réelle et les tas locaux. Les échanges avec les tas locaux se font par *macro-blocs* d'une taille supérieure ou égale à 2 Mo. D'une manière générale la source mémoire peut être vue comme un cache placé entre un allocateur standard et l'OS en surchargeant les fonctions standards *mmap/munmap*.

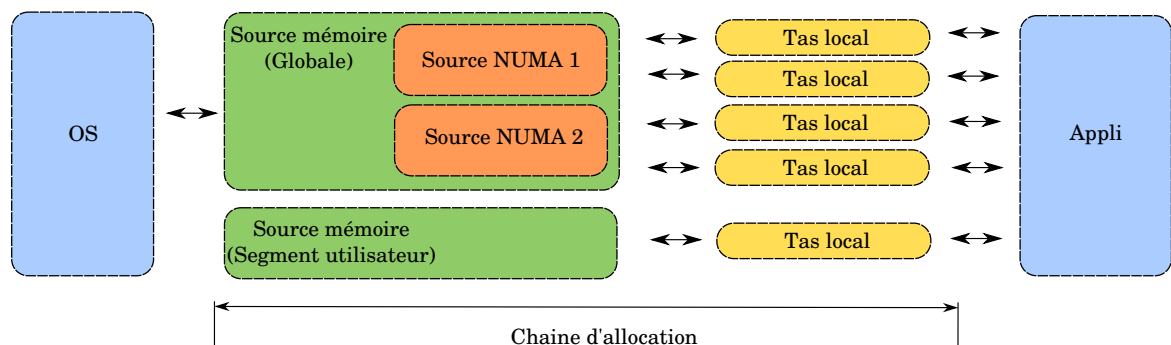


FIGURE 4.5 – Schéma d'organisation générale de l'allocateur faisant intervenir deux composants principaux, des sources mémoires et tas locaux assemblés pour former la chaîne complète d'allocation entre l'OS et l'application.

Cette approche à deux composants permet de redéfinir sa propre source mémoire, il est ainsi possible de gérer des segments utilisateurs au sein de l'allocateur principal, comme cela sera discuté en section 4.13. Les tas locaux peuvent fonctionner sans verrous s'ils sont créés par thread, les contentions se reportant alors sur la source mémoire de manière limitée par la taille importante des échanges. Remarquons également que les allocations de gros segments (supérieurs à 1Mo) sont transférées directement à la source mémoire.

La séparation par thread a été mise en place afin de permettre la migration des tâches MPI (de leur mémoire associée) permise par MPC vers des nœuds distants. Ce support des migrations a toutefois été retiré dans les dernières versions, levant la contrainte pour notre allocateur. Sans cette dernière, il pourrait être intéressant d'évaluer des méthodes de mise en commun entre nombres réduits de threads pour limiter la surconsommation engendrée par un trop grand nombre de tas locaux. Cette remarque est surtout valable si l'on considère le fait que les threads

2. Les **TLS** : *Thread Local Storage* sont des variables gérées par le compilateur et permettant d'associer une valeur dépendant du thread courant.

utilisateurs de MPC fonctionnent en mode non *préemptif*. On peut ainsi maintenir des tas locaux sans verrous si l'on crée un tas local par thread système.

4.6.2 Gestion des blocs

Les blocs retournés à l'utilisateur sont formés par découpe de *macros-blocs*. L'en tête de ces derniers est placé en début de bloc et contient les informations représentées dans la figure 4.6. On remarquera l'emploi de taille de stockage sur la base de 56 bits. Ceci permet de disposer d'informations supplémentaires sur l'en-tête tout en maintenant une taille de 16 octets pour cette dernière en remarquant que les architectures actuelles n'offrent en réalité qu'un adressage pratique sur 48 bits. L'en-tête contient une partie commune définissant le type et l'état du bloc. Ceci permet ensuite d'exploiter plusieurs types de blocs. On remarquera la présence de la taille du bloc courant et précédent afin de pouvoir se déplacer à double sens dans la liste de blocs. Ceci permet de fusionner facilement les blocs voisins lors des libérations à la manière de DLMalloc.

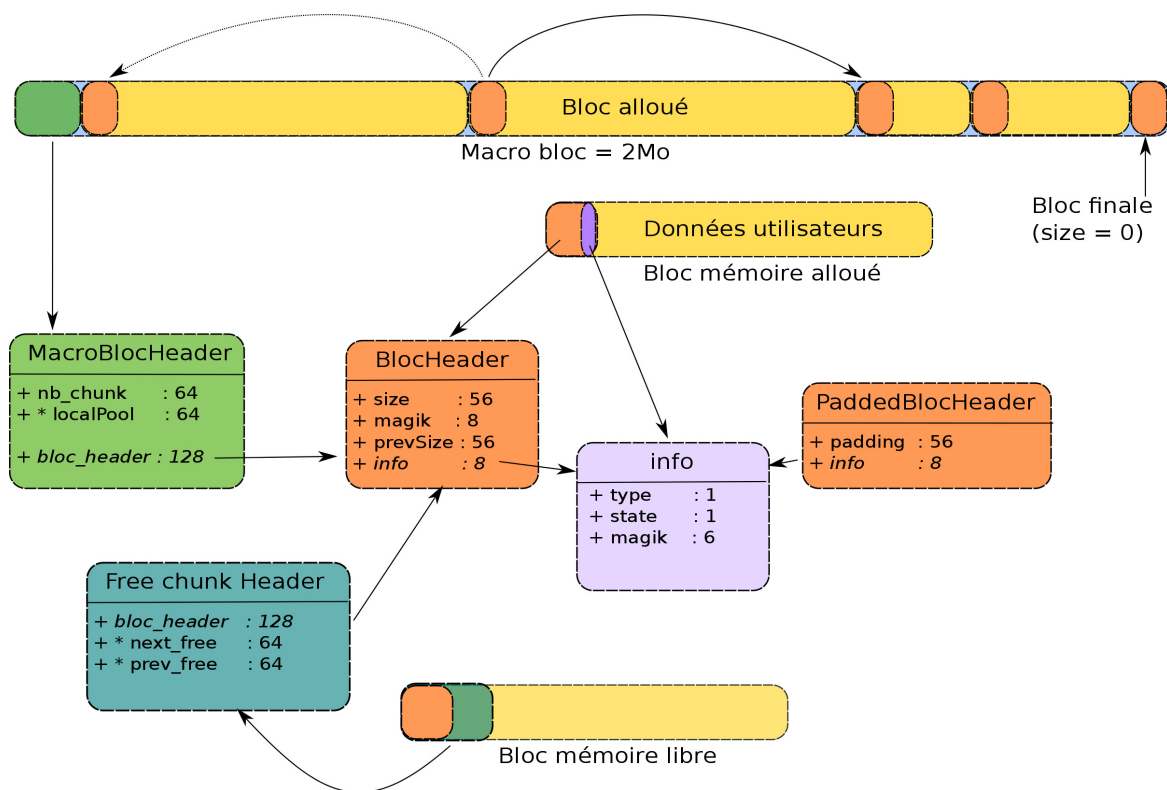


FIGURE 4.6 – Organisation des en-têtes de macro-blocs, blocs alloués et libres. Les tailles des champs sont données en bits considérant l'architecture x86_64.

Les tas locaux maintiennent des listes doublement chaînées de blocs libres pour différentes classes de tailles afin de trouver rapidement les blocs adaptés. Ces listes sont ordonnées par ordre *FIFO*³ de manière à réutiliser les blocs les plus anciens. Ceci afin de permettre aux fusions de libérer des macro-blocs. En cas de scission, le bloc restant est inséré en tête de liste de sorte à être réutilisé rapidement et favoriser des durées de vie proches pour des blocs voisins. En ce qui concerne la gestion des blocs libres, ces derniers sont maintenus sous la forme de listes doublement chaînées en stockant les deux pointeurs à l'intérieur du segment à suivre. Ceci impose une contrainte interdisant la gestion de blocs de taille utile inférieure à 16 octets sur architecture 64 bits.

3. **FIFO** : First In, First Out : système de queue impliquant une réutilisation prioritaire des entrées les plus anciennes.

4.6.3 Discussion à propos des petits blocs

En implémentant le support des blocs, on peut s'interroger sur l'intérêt de fournir une structure spécialisée pour les petites allocations (inférieures à 16 octets) de manière à réduire l'impact des en-têtes. Dans un contexte HPC, on peut toutefois remarquer que les applications vont avoir tendance à consommer de grandes quantités de mémoire. Si l'application alloue toute sa mémoire sous la forme de petits blocs elle sera largement pénalisée par un nombre trop important d'indirections de pointeurs nuisibles aux performances sur les processeurs actuels. À l'exception des parties infrastructure des programmes, ce type d'application doit donc tendre à allouer des tableaux de taille raisonnable. On peut donc supposer que les petits blocs ne sont pas une priorité (bien qu'importants) pour le domaine qui nous concerne. On remarquera par exemple qu'avec des en-têtes génériques de 16 octets, l'application Hera, bien qu'en C++, génère un surcoût d'en-tête de l'ordre de 50 Mo pour une consommation mémoire totale de plus de 4 Go. Les observations précédentes nous ont montrés des problèmes importants liés aux échanges avec l'OS. Nous allons donc nous concentrer sur ce point particulier. Nous considérerons que le problème des petites allocations est déjà bien maîtrisé par les allocateurs décrits précédemment. Cette remarque est notamment valable si l'on retient l'approche des *runs* utilisée par Jemalloc qu'il nous faudrait intégrer à notre gestion. Remarquons à l'occasion qu'avec le recul, l'approche de limitation de taille d'en-tête discuté dans la section précédente pose plus de problèmes d'implémentation (surtout portabilité) qu'elle n'en résout. En pratique, l'approche de la glibc avec des en-têtes de 24 octets semble donc plus raisonnable surtout si elle est couplée à l'approche des runs de jemalloc pour les petits blocs.

4.6.4 Suivi des macro-blocs alloués

Lorsqu'un tas local a besoin de mémoire, il effectue une requête auprès de sa source mémoire pour obtenir un nouveau macro-bloc. Il est toutefois important de remarquer que les macro-blocs ne sont pas de simples segments mémoires. Ils hébergent à leur base un en-tête permettant à tout moment de retrouver le tas auquel appartient le macro-bloc et ses sous-segments. Cette notion est importante si l'on souhaite éliminer les synchronisations au niveau des tas locaux en distinguant les libérations locales des libérations distantes (voir section 4.6.6). Lors d'une libération, il importe donc de pouvoir retrouver la position de cet en-tête à partir de l'adresse du segment à libérer. Trois solutions sont envisageables à ce niveau :

Aligner les segments : Tout comme Jemalloc, il est possible de forcer un alignement des macro-blocs sur leur taille. Cette méthode offre un moyen très simple de retrouver l'en-tête par application d'un masque. Elle a toutefois l'inconvénient de forcer à ré-implémenter un allocateur complet pour gérer les macro-blocs et forcer les adresses des segments demandés à *mmap*. Ce dernier point peut poser des problèmes en cas de mauvaise coopération entre l'allocateur et toute autre bibliothèque travaillant avec *mmap* (CUDA...). L'assurance du fonctionnement passerait donc par une surcharge des appels à *mmap* pour assurer un support valide. Nous ne retiendrons pas cette solution d'autant qu'il est tentant tout comme pour Jemalloc de choisir des alignements à problèmes vis-à-vis des caches. Cette approche a été exploitée en début de thèse, mais conduisait à certaines limitations qui sont discutées dans la suite.

Mise en place d'arènes : L'un des principes des arènes consiste à associer une bande d'adresse bien définie à ces dernières. Cette approche cause toutefois les problèmes habituellement reprochés à ptmalloc en rendant impossible l'échange de mémoire entre les arènes et en nécessitant un grand espace virtuel.

Indexer les éléments : Cette solution consiste à construire un index permettant de retrouver rapidement l'adresse de base d'un macro-bloc à partir d'adresse d'un de ses sous-segments. Nous retiendrons cette solution.

L'indexation peut se construire sur la base d'arbres binaires équilibrés, comme le fait Jemalloc. Remarquons toutefois que la détermination d'appartenance locale ou distante d'un bloc à libérer nécessite l'accès à cet en-tête. Ceci implique donc une lecture de l'index à chaque libération. L'utilisation d'un arbre pourrait pénaliser les performances s'il devenait trop profond. Nous avons donc opté pour une structure assurant un accès en temps constant. À la manière de TCMalloc, l'indexation de nos segments se fera par la création d'une table globale. L'espace d'adressage est ainsi découpé en segments de 2 Mo associés à des pointeurs vers les en-têtes des macro-blocs qu'ils couvrent. Cette table est construite en deux niveaux, à la manière de la table des pages. Le premier niveau adresse des *régions*, de 1 To. Chaque région est associée à une table de deuxième niveau occupant 4 Mo et adressant les sous-segments de 2 Mo.

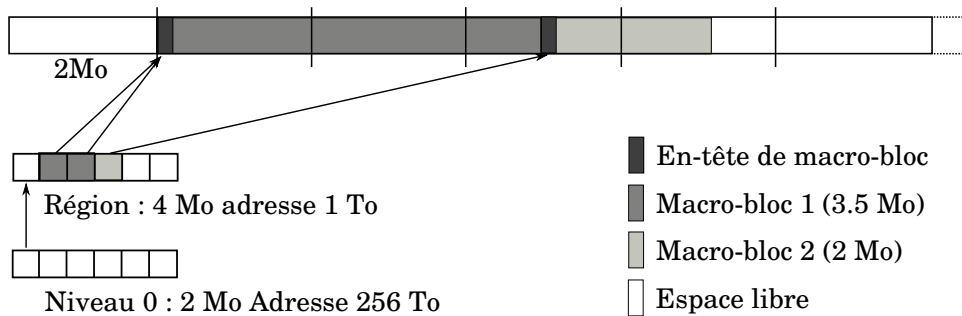


FIGURE 4.7 – Mécanisme d'indexation en régions des macro-blocs pour retrouver les adresses de leurs en-têtes. Sont représenté ici l'indexation de macro-blocs de 3.5 Mo et 2 Mo.

Pour des macro-blocs de taille supérieure à 2 Mo, l'unicité des adresses renvoyées par *mmap* implique l'unicité des entrées d'index à mettre à jour. Cette structuration impose la contrainte d'utiliser des macro-blocs d'une taille minimale de 2 Mo afin d'éviter d'avoir plus que deux macro-blocs à indexer dans la même entrée. De plus, des chevauchements sont possibles si l'on ne force pas un alignement des tailles et adresses sur 2 Mo. Dans cette situation, nous indexons systématiquement le bloc ayant l'adresse la plus élevée. Le bloc précédent est nécessairement enregistré par l'index précédent, une simple comparaison d'adresse permet alors de retrouver les en-têtes en lisant au plus deux entrées dans la table (code 4.1).

Grâce à ces règles, l'index peut être mis à jour sans prise de verrous, sauf lors des modifications de la table de niveau 0. Ceci n'est vrai qu'à la condition de ne pas supprimer de régions. On obtient ici un avantage sur la méthode à base d'arbre qui implique généralement des verrous globaux. Un exemple d'indexation avec chevauchement est donné dans la figure 4.7. Du fait de l'allocation paresseuse, l'essentiel de cette table reste virtuelle tant que la bande mémoire associée n'a pas été utilisée. Remarquons que cette technique peut poser problème si les *mmap* successifs vont en adresse croissante sans réutiliser les zones préalablement libérée par *munmap*. Ce cas de figure semble se rencontrer par exemple sous Windows.

Code 4.1– Levée d'ambiguïté sur les entrées de régions.

```
1 macro_bloc = read_region_entry(chunk_addr);
2 if (macro_bloc > chunkAddr)
3     macro_bloc = read_region_entry(chunk_addr - REGION_SPLITTING);
```

4.6.5 Surcharge possible de la fonction de libération

L'accès à l'en-tête du macro-bloc permet également d'étendre ce dernier en y stockant la fonction de libération à appeler. Ceci peut être utile dans le cadre de MPC pour notifier les couches

infiniband ou gestionnaires de tâches CUDA lors des libérations (*munmap*) de ces segments. Ceci est notamment vrai si ces implémentations nécessitent l’utilisation de pages punaisées⁴ et offrent des optimisations à base d’un suivi précis des libérations mémoires.

La possibilité de surcharger la fonction de libération pourrait également permettre d’exploiter plusieurs allocateurs au sein de la même application tout en reposant sur les procédures standards de libération (*free*). Ceci à la condition que ces derniers reposent tous sur l’exploitation de macros-blocs standards. Le mélange d’allocateurs pose habituellement le problème de libération des blocs si leur libération est déléguée à d’autres bibliothèques non conscientes de l’allocateur utilisé pour les allouer. Avec notre approche, nous levons cette restriction en définissant une forme d’arène sans contraintes d’adresse et à granularité plus faible. Chacune de ces “*mini-arènes*” peut alors être gérée par son propre allocateur. Nous verrons en section 4.13 que cette méthodologie est la base du support des segments utilisateurs proposé comme extension de notre allocateur.

4.6.6 Libérations distantes

Nous avons choisi de fournir des tas locaux sans synchronisation. Cette approche est permise dès lors que l’on assure qu’un unique thread accède à un tas donné. Or, il est tout à fait possible qu’un bloc alloué par un thread soit libéré plus tard par un autre thread. Nous dirons alors qu’il y a *libération distante*. Ce type de libération nécessite la réintégration du segment dans son tas d’origine. Or, si le travail est réalisé par un autre thread, il devient nécessaire d’utiliser des primitives de synchronisation, telle qu’exploité par *Jemalloc*. Afin de ne pas en introduire dans l’ensemble des opérations des tas locaux, nous avons décidé de séparer totalement la gestion des libérations distantes. Leur détection est réalisée grâce aux en-têtes de macro-blocs en comparant le tas courant au tas d’appartenance du bloc.

Lorsque ces libérations surviennent, le bloc est enregistré auprès d’une liste simplement chaînée (File de Libération Distante : FLD). Les blocs sont alors libérés en groupe lorsque le thread propriétaire effectue une opération avec l’allocateur. Cette liste de blocs distants est accédée en insertion par plusieurs threads et en extraction par un seul. Il est donc possible de la construire de manière atomique. On peut par exemple reprendre l’algorithme de liste disponible dans la bibliothèque *OpenPA*[Ope] initialement tiré du projet *MPICH2*[BMG06] pour des fils de message réseau. Remarquons toutefois que l’extraction n’a pas à se faire élément par élément, nous modifions donc légèrement l’algorithme pour vider l’ensemble de la liste en un appel. On pourra trouver en annexe C le détail des modifications apportées sur cette liste que l’on peut dire à insertion multiple et purge unique.

Au-delà de la suppression des verrous, cette méthode permet de maintenir les structures du tas local au niveau du cache du thread auquel il est associé. Ce point peut-être particulièrement important sur les nœuds NUMA ayant de fortes pénalités en cas de déplacement d’un élément mémoire d’un nœud NUMA à l’autre.

4.6.7 Realloc

La fonction *realloc* vise à redimensionner un bloc mémoire. L’implémentation naïve consiste en une nouvelle allocation, une copie des données et libération de l’ancien segment. Cette méthode est toutefois sous-efficace si elle s’applique aux grands segments. Or, nous avons vu en section 4.5 que l’application *Hera* utilise un grand nombre de ces appels. Pour notre allocateur,

4. Pages spécialement marquées pour indiquer à l’OS qu’il ne doit pas déplacer leur emplacement physique. Ce type de page est habituellement utilisé lors de l’emploi des mécanismes dit DMA pour optimiser les échanges avec les périphériques d’entrées/sorties.

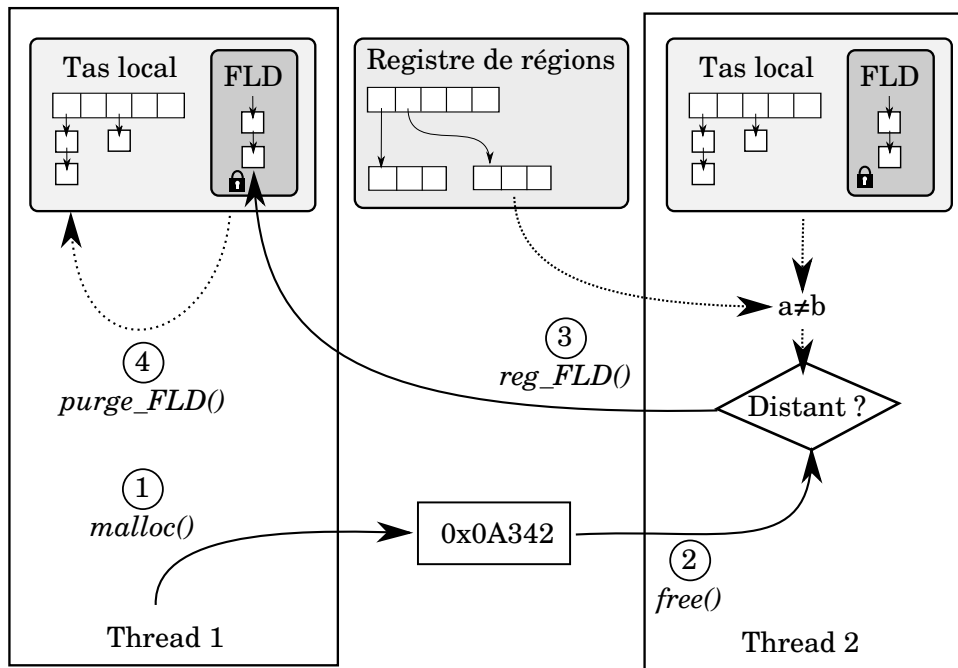


FIGURE 4.8 – Mécanisme de libération à base de File de Libération Distant (FLD). Un bloc est alloué par un thread (1) et transmis pour libération à un autre thread (2). La comparaison d'adresse du tas local avec l'entrée du registre de région permet de distinguer les allocations distantes. Le bloc est alors enregistré dans la queue distante de manière atomique (3). Le thread d'origine purgera cette liste de manière atomique lors de la prochaine allocation (4).

nous avons donc pris soin d'optimiser cette fonction. Les réallocations sont donc traitées par catégorie avec les règles suivantes :

1. Les gros segments sont redimensionnés directement à l'aide de `mremap` si disponible comme cela est fait dans d'autres allocateurs. Ceci permet de simplement migrer les pages dans l'espace virtuel sans effectuer des copies.
2. Pour les blocs standards, un seuil est défini, les redimensionnements décroissants ne génèrent donc un changement que si la mémoire perdue est supérieure à un seuil défini par le paramètre `SCTK_REALLOC_THRESHOLD`.
3. Les autres cas basculent directement sur la méthode allocation/copie/libération. Ceci est surtout indispensable pour les réallocations distantes qui posent un problème de définition sémantique quant à l'attente de l'utilisateur en terme de placement (voir section 4.12). Nous choisirons donc arbitrairement un placement NUMA attaché au thread chargé de la réallocation pour les petits blocs et un maintien pour les gros blocs.

4.7 Réutilisation des gros segments

Dans le cadre de cette thèse, nous nous sommes surtout concentrés sur la gestion de gros blocs. Ceci, notamment pour limiter les interactions avec l'OS et éviter les problèmes observés en section 4.5. Pour ce faire, la source mémoire constitue elle-même un allocateur en maintenant ses segments pour une réutilisation future. On remarquera que pour les gros segments, le coût d'allocation se trouve concentré principalement sur les fautes de pages du fait de la pagination paresseuse. C'est donc ces dernières que l'on vise à réduire. Pour cela, on peut comparer le temps d'allocation d'un élément de 1 Mo à son temps de premier accès. Le résultat est présenté par la figure 4.9 en effectuant les allocations à l'aide de l'allocateur de la glibc ou directement à partir

de *mmap*. Ces graphiques mettent clairement en évidence l'existence d'une source différente de surcoût d'allocation. Les petites allocations sont essentiellement impactées par le temps de la fonction *malloc*. Les grosses sont majoritairement impactées par les fautes de pages.

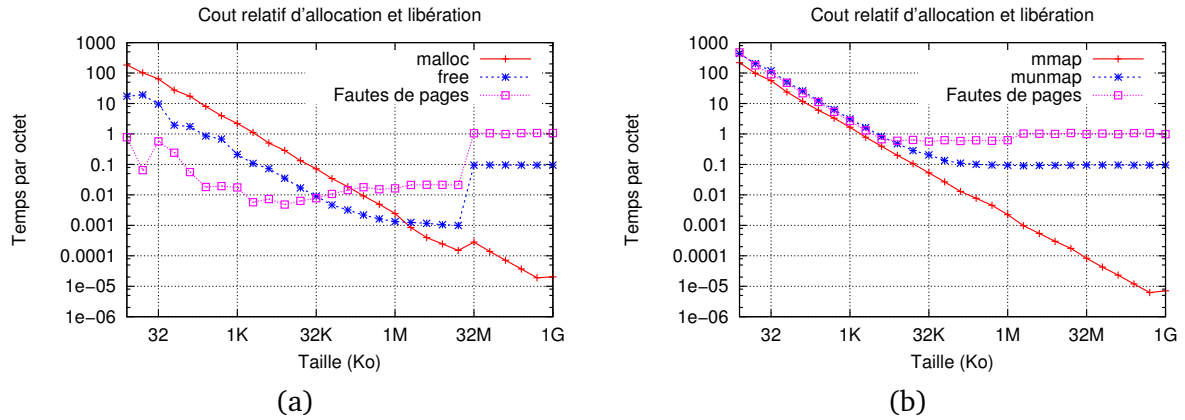


FIGURE 4.9 – Source des différents surcoûts d'allocation en fonction de la taille. L'impact des fautes de pages est mesuré à partir de la différence de temps entre un premier et un second accès aux données en ayant vidé les caches entre temps. Les coûts sont donnés par unité de taille (cycles par octet).

4.7.1 Méthode de réutilisation de TCMalloc

TCMalloc dispose à sa manière d'une méthode de réutilisation. Cet allocateur ré-implémente toutefois l'équivalent complet du couple *mmap/munmap* avec la gestion de ses *span*. Leur approche est intéressante du point de vue de leur contrôle du débit de libération basé sur l'utilisation de *madvise*. Mais leur implémentation limite toutefois les chances de réutilisation des gros segments en introduisant des problèmes de fragmentation à grande échelle. On remarquera que le critère de performance tient plus au contrôle des débits d'allocation que de libération. On peut toutefois montrer que ces derniers sont pour partie reliés sous certaines conditions. Supposons la libération d'un tableau de p pages à un instant t_0 et une allocation de ces mêmes p pages à t_1 . Avec une consigne de c pages maximum libérées par secondes, à t_1 on aura libéré $\Delta t * c$ pages, ce nombre étant limité à p . Sur cet intervalle de temps, on a un débit d'allocation correspondant aux pages libérées, d'où un débit d'allocation inférieur ou égal à la consigne.

Avec un temps t_f de faute de page on montre alors que le surcoût lié à l'OS est au maximum de $s = t_f * \Delta t * c$, on a donc un ratio temps effectif sur temps système borné par $\frac{s}{\Delta t} = t_f * c$. Remarquons toutefois que l'on ne contrôle pas le débit instantané d'allocation (contrairement à la libération), mais moyen. Or, le débit instantané est celui qui peut avoir une influence sur t_f si l'OS ne passe pas à l'échelle. Nous verrons que c'est malheureusement le cas de Linux. De plus, on notera que la proposition précédente n'est vérifiée que si l'on est en capacité de réutiliser 100% des pages en attente. Or, du fait de la fragmentation à grande échelle, ce n'est pas le cas de TCMalloc. Cette approche, bien qu'intéressante, ne permet donc de contrôler les débits d'allocation que de manière approximative.

4.7.2 Méthode proposée

Nous avons vu avec l'application Hera que nous avons beaucoup d'allocations de grandes tailles (de l'ordre de quelques Mo) ainsi que de nombreuses réallocations. Dans ce contexte, nous avons plutôt choisi de centrer notre politique de réutilisation sur les fonctions *mmap*,

munmap, *mremap* en ne contraignant pas le placement des blocs à des adresses précises. Le cache de macro-bloc est contrôlé sur la base de deux paramètres :

Taille maximum de segment : Cette limite permet d'ignorer tous les segments au-delà d'une certaine taille. Les segments trop grands ont en effet peu de chances d'être facilement réutilisables tout en générant une surconsommation mémoire importante.

Mémoire maximum : Ce paramètre permet de borner la quantité de mémoire totale que l'on autorise à maintenir en attente sur chaque nœud NUMA, ceci afin d'éviter tout risque d'explosion inopiné de la mémoire.

Si une requête est en dessous du seuil de réutilisation, l'algorithme recherche le bloc disponible ayant la taille la plus proche de la requête. Si la taille ne correspond pas, alors le segment est redimensionné à l'aide de *mremap*. Les trois sémantiques de recyclage de blocs de taille inadapté sont décrites dans la figure 4.10. Remarquons principalement la présence du dernier cas qui n'est gérable que sur la base d'un emploi de la sémantique *mremap*. Cette approche permet d'assurer la réutilisation de segment même pour des tailles très variables. On assure de la sorte la propriété de réutilisation à 100%. Ceci peut à permettre à terme d'obtenir un meilleur contrôle des débits moyens via une méthode basée sur *madvise()*. Ceci, si l'on dispose d'un OS passant à l'échelle. D'autre part, notre méthode à l'intérêt est de parvenir de limiter le nombre de fautes de pages en maintenant un contenu minimal en page physique pour les segments renvoyés même s'ils ne sont pas entièrement projetés physiquement.

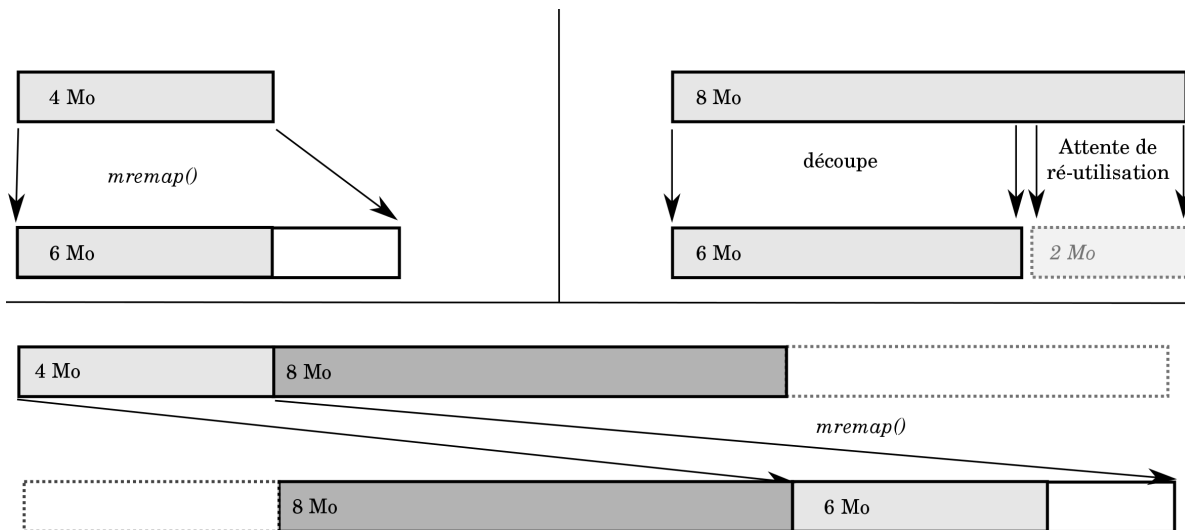


FIGURE 4.10 – Méthode de réutilisation des gros segments à base de *mremap* pour une requête de 6 Mo. À gauche un exemple de réutilisation d'un segment plus petit impliquant une fraction du segment final non paginé. À droite la réutilisation d'un segment trop grand conduisant à une scission pour réutilisation futur. En bas, un cas nécessitant un déplacement éliminant le problème de fragmentation puisqu'il est toujours possible de réutiliser les pages du segment.

4.7.3 Recomposition de gros segments

L'utilisation de *mremap* sous Linux permet de déplacer volontairement des blocs de pages à une adresse déterminée. Ceci peut permettre de recomposer des segments plus grands à partir de segments plus petits. Cette voie n'a pour l'instant pas été explorée du fait des risques d'écrasement de données induit par la méthode et des limites de portabilité de cette dernière. Il n'est d'ailleurs pas sur que cela apporte de réels gains. On remarquera en effet que l'utilisation de *mremap* pour grandir un segment suffit à pouvoir disposer d'une part non négligeable fournie en pages physiques. Les échanges avec l'OS même si non nuls, s'en trouvent donc déjà fortement

réduits. Cette approche d'allocation non totalement physique assure également un minimum de recyclage de la mémoire. Ceci est bénéfique vis-à-vis des applications demandant plus de mémoire que nécessaire et délègue plus de gestion du NUMA à l'OS qui a plus d'information que l'allocateur.

4.7.4 Problème de surallocation

Du fait de la présence de la pagination paresseuse, il est possible pour une application de réclamer des segments virtuels plus grands que nécessaire sans générer une surconsommation de mémoire physique. Cette méthode est parfois appliquée sur des segments croissants dont la taille exacte n'est pas connue par avance. Ces modèles d'allocation posent quelques difficultés en cas de réutilisation de gros segments. Ces derniers sont en effet déjà fournis en pages physiques, la supposition de non-réservation de l'espace non utilisé est donc annulée. La réutilisation de gros segments tend donc à augmenter beaucoup plus la consommation mémoire de ces applications. Comme discuté dans la section précédente, assurer un rafraîchissement suffisant des blocs permet de limiter ces effets néfastes en limitant la fourniture de segments entièrement physiques. L'effet est toutefois statistique et non contrôlé.

4.8 Remise à zéro pour *calloc*

Avec la fonction *calloc*, l'appelant attend une mémoire pré-initialisée à zéro. On remarquera que notre source mémoire avec recyclage introduit deux cas de figure et pose un problème propre à la réutilisation des gros segments. Le premier cas correspond au comportement habituel, à savoir un cache vide impliquant une requête de mémoire via *mmap* auprès de l'OS. Dans cette situation, la zone allouée est automatiquement pré-initialisée à zéro. L'autre situation survient lors du recyclage de segments, qui, par définition ont précédemment été exploités. Dans cette situation, une remise à zéro est nécessaire. Cette dernière pourrait être effectuée par le cache de macro-bloc, mais impliquerait nécessairement un accès complet à la mémoire en forçant sa projection physique. Cette méthode peut engendrer une surconsommation mémoire et un surcoût inutile dans le cas où l'ensemble du segment n'est pas utilisé par l'appelant. Il peut ici s'agir du gestionnaire de blocs standard qui va découper le macro-bloc en sous-éléments.

La fonction appelante de plus haut niveau doit donc prendre en charge elle-même cette remise à zéro. Elle est en effet la seule à connaître la taille exacte à initialiser en ignorant les éventuelles fusions/scissions des fonctions sous-jacentes. Elle doit toutefois disposer d'un retour de ces fonctions pour connaître l'état de pré-initialisation du bloc demandé. La méthode consiste donc à transmettre un booléen par pointeur dans toute la hiérarchie d'appel tel que cela est fait dans *Jemalloc*. La sémantique peut alors être interprétée comme suit :

Pointeur nul ou valeur faut : Aucune remise à zéro n'est attendue, peu importe la méthode d'allocation. L'utilisation de la modification noyau discutée dans le prochain chapitre (5) est alors possible.

Valeur vrai : L'appelant a besoin de mémoire remise à zéro. Dans le cas où les mécanismes de bas niveaux ne permettraient pas d'obtenir une telle mémoire (recyclage au niveau de la source mémoire...), le booléen est remis à faux. L'appelant sait ainsi que la mémoire reçue n'est pas pré-initialisée.

L'approche précédente améliore le procédé, mais ne résout toujours pas le problème, qui du fait du recyclage des segments tend à demander une remise à zéro systématique de la mémoire reçu par la fonction *calloc*. Or, il est à remarquer que les pages non encore accédées (purements virtuelles) seront automatiquement remises à zéro lors de leur premier accès. On peut donc proposer d'analyser la projection de la zone mémoire pour déterminer les zones physiquement

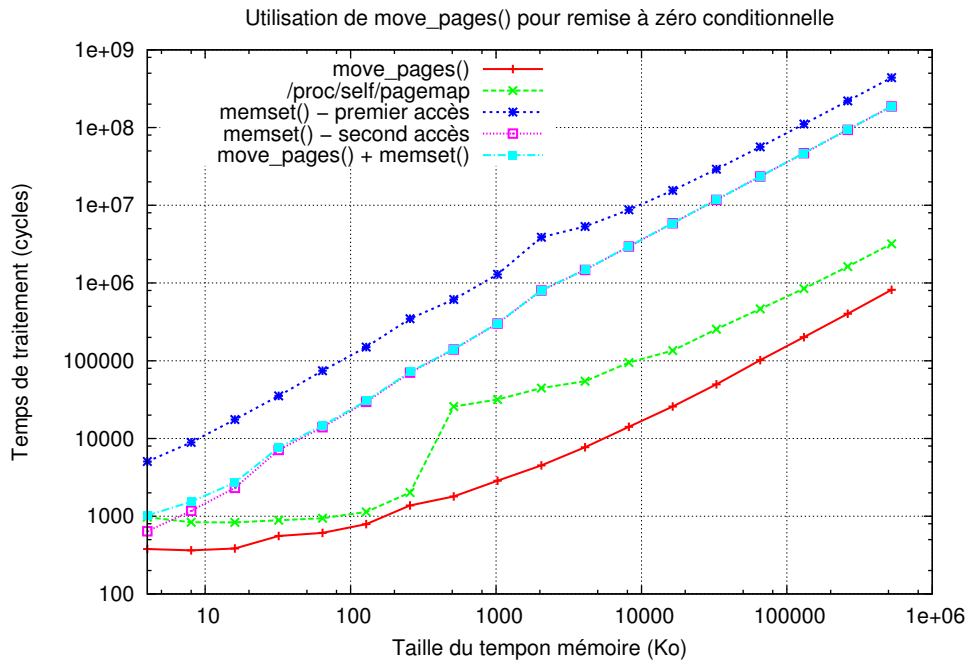


FIGURE 4.11 – Évaluation du coût de détection de présence des pages physique en comparaison du coût de remise à zéro de la mémoire avec ou sans fautes de pages.

présentes. On ne remet alors à zéro que ces dernières. Sous Linux, la requête peut-être faite via l'appel `move_pages` utile aux migrations NUMA ou en lisant la table des pages au travers du fichier `/proc/self/pagemap` s'il est présent. Le graphique 4.11 montre les surcoûts engendrés par ces deux méthodes comparés au coût de remise à zéro par `memset`. On observe clairement l'intérêt de la méthode avec un surcoût mineur à comparer aux gains potentiels. En considérant le cas défavorable d'une zone entièrement physique le surcoût ne dépasse pas 5% pour 64 Ko et 1% pour 1 Mo. Cette méthode a été étudiée en cours de rédaction et n'a pas encore été intégrée dans l'implémentation de l'allocateur faute de temps pour valider sa portabilité. Une alternative plus portable peut aussi consister à ne pas exploiter le cache de la source mémoire dans le cas d'allocation nécessitant une remise à zéro.

4.9 Destruction du tas

Lorsqu'un thread est détruit, le tas local doit également l'être. Or, ceci est impossible, car ce dernier peut encore contenir des segments exploités par d'autres threads et non encore libérés. Notre allocateur n'applique pour l'instant pas une politique de réutilisation des tas locaux (du fait de l'ancien besoin de migration dans MPC). Lorsqu'un thread est détruit, le tas local est marqué comme orphelin et son mode synchronisé est activé afin de ne plus recourir aux *libérations distantes*, qui, sinon, n'ont plus de parent pour effectuer la tâche de libération. Lorsque le dernier macro-bloc est libéré et que le tas local orphelin devient vide, il est détruit par le thread ayant libéré son dernier bloc mémoire.

4.10 Adaptation consommation versus performances

La réutilisation de gros segments permet de limiter les interactions avec l'OS et donc les coûts associés. Cela implique toutefois une augmentation de la consommation mémoire qui au-delà d'un certain point, peut devenir un problème. Ceci d'autant plus dans un contexte où la puis-

sance de calcul tend à croître plus vite que les capacités de stockages mémoires. On remarquera toutefois que les applications n'utilisent pas toujours toute la mémoire du nœud et vont très souvent faire apparaître des pics de consommation lors de certaines étapes de calcul. Il est donc intéressant de remarquer que l'on peut dans ces conditions se permettre d'utiliser la mémoire lorsque disponible pour obtenir de bonnes performances et réduire les performances pour un mode économique dans le cas contraire.

Ce type d'adaptation dynamique est envisageable dans un contexte HPC où l'utilisateur est en général seul sur un nœud et vise à exploiter au mieux les ressources. Dans cet objectif, nous avons construit notre implémentation de façon à offrir les deux politiques extrêmes et pouvoir contrôler un gradient entre ces dernières à l'aide de paramètres. Pour ce faire, il nous a fallu obtenir une capacité à économiser au mieux la mémoire au niveau des tas locaux. Ces derniers renvoient la mémoire de manière agressive vers les sources mémoires. Le contrôle de consommation peut ainsi s'établir à leur niveau en décidant d'y garder ou non les macro-blocs en transit. Remarquons que la politique est ainsi réversible puisque les macro-blocs en attente peuvent être libérés brutalement en cas de pic de consommation. Une politique de contrôle au niveau des tas locaux impliquerait la collaboration de plus de composants et pourrait induire des effets non réversibles si la méthode impacte les décisions de sélection des blocs.

Comme nous allons le montrer en section 4.14 nous avons ainsi atteint la capacité de passer d'un profil de type TCMalloc (rapide, mais consommateur) à un profil Jemalloc (plus économe, mais pénalisé par l'OS). Reste à mettre en place le composant de prise de décision dynamique. Non implémenté pour l'instant, ce dernier sera discuté en section 4.17.2.

4.11 Aspect NUMA

Comme discuté en section 1.5, les architectures cibles sont composées de nœuds NUMA et doivent idéalement être exploitées à l'aide de threads. Dans ce contexte, nous avons vu que l'allocateur devient un des maillons de la gestion NUMA. Pour ces architectures, il conviendrait que l'allocateur sache à l'avance comment le segment mémoire va être utilisé. Cette connaissance n'est malheureusement pas exprimée dans l'interface actuelle définie par l'API POSIX. Dans le cadre d'un allocateur générique, les choix sont donc relativement limités. Nous remarquerons donc essentiellement qu'un segment précédemment alloué par un thread d'un nœud NUMA donné ne doit être réutilisé que dans ce même nœud NUMA.

Cette propriété fondamentale n'est toutefois pas vérifiée sur les allocateurs tels que la glibc, Jemalloc et TCMalloc. Ces allocateurs utilisent en effet une association aléatoire des threads aux différents tas ou génèrent des échanges non contrôlés entre ces derniers. Dans le cadre de notre allocateur, l'association d'un tas à chaque thread permet donc d'assurer la propriété de réutilisation locale pour les petits blocs. La question se pose toutefois pour les gros segments que l'on s'efforce de réutiliser. Ces derniers circulent en effet entre les threads au travers de la *source mémoire*. Afin d'éviter un mélange indésirable, nous construisons donc une *source mémoire* par nœud NUMA.

On remarquera toutefois que les threads ne sont pas nécessairement figés et peuvent migrer d'un nœud à l'autre sans que l'allocateur n'en soit notifié. Il n'est pas raisonnable (trop coûteux) de demander la position du thread à chaque allocation. Nous appliquerons donc différents *niveaux de confiances* entre les threads sur la base des règles suivantes :

1. Lors de la première allocation, les positions autorisées pour le thread sont analysées. Si le thread est fixé sur un nœud NUMA déterminé, alors il est associé à une source mémoire de confiance pour ce nœud NUMA.

2. Si le thread n'est pas fixé sur un nœud particulier, il est associé à une source mémoire générique considérée comme non fiable.
3. En cas de migration du thread, il faut notifier l'allocateur. Ce problème est résolu dans le cadre de MPC par la gestion de threads utilisateurs dont il est facile de suivre les déplacements explicites. Dans le cas général, il nous faut toutefois compter sur la coopération de l'utilisateur qui doit appeler une fonction de migration ou surcharger les fonctions de placement de thread. Certains tel que [DG02] proposent des modifications de l'OS pour permettre ce type de suivi. Cela rend toutefois la méthode non portable sur les OS conventionnels ne disposant pas de ces mécanismes expérimentaux.

Il est également important de noter que les pages des grands segments sont placées par l'OS en fonction du premier accès. Ceci pose un problème de confiance sur ces derniers lors des libérations. Il est en effet possible qu'une partie d'un segment soit sur un nœud NUMA distant. Il n'est toutefois pas raisonnable de demander le placement de chacune des pages à chaque libération, on postulera donc un niveau de confiance envers l'utilisateur. Dans le cas où l'utilisateur voudrait obtenir des segments pour des usages critiques et nécessitant un placement fiable, nous fournissons en supplément un tas partagé de haute confiance (placement forcé de manière explicite) pour chaque nœud NUMA disponible. L'utilisateur peut y allouer de la mémoire en utilisant un appel à `sctk_malloc_on_node()`. La libération de ces segments se faisant de manière standard à l'aide de la fonction `free`.

4.12 NUMA et initialisation

D'un point de vue pratique, on remarquera globalement les difficultés induites par l'étape d'initialisation de l'allocateur. La construction des structures topologiques nécessite l'obtention de la structure NUMA de l'hôte au travers de `Hwloc`[BCOM⁺10] ou de la `LibNUMA`[Kle05]. Or, ces deux bibliothèques effectuent des allocations lors de leur utilisation. Ceci mène rapidement à l'apparition de boucles d'appels infinis que nous avons rompue en initialisant l'allocateur en deux étapes. La première permet d'obtenir un allocateur fonctionnel de type UMA dont la source mémoire est utilisée pour les threads non fixés. La mise en place du NUMA peut alors se dérouler normalement en exploitant cet allocateur pour détecter la topologie et construire les sources de confiance.

Remarquons que sur ces architectures, l'utilisation de *realloc* pose la question de l'attente de l'utilisateur. Attend-il que le bloc réalloué maintienne l'ancienne association NUMA ou la nouvelle en fonction du thread demandeur ? Dans notre cas, les réallocations distantes sont traitées par recopie pour les segments inférieurs au Mo. Les gros blocs sont redimensionnés par *mremap*. Cela pose toutefois la question de fiabilité de positionnement NUMA du nouveau segment s'il est agrandi avec un risque d'association non uniforme pour un nœud donné. Nous avons vu en section 4.7.2 que nous maintenons un certain taux de pages nouvelles par notre méthode de réutilisation, nous comptons donc sur cette dernière pour limiter l'impact de ce problème en libérant tout de même régulièrement une partie de la mémoire.

4.13 Gestion de segments utilisateurs

Dans certaines situations, l'utilisateur peut avoir à établir des segments mémoires aux propriétés particulières (pages punaisées, mémoire partagée, projection mémoire d'un fichier...). L'établissement de ces segments passe souvent par l'utilisation de fonctions systèmes dédiées. Or, une fois établis, ces segments doivent être gérés à la main, obligeant l'utilisateur à construire son propre allocateur.

Afin de prendre en compte cette problématique, nous avons construit notre allocateur de sorte à pouvoir aider à gérer ces segments particuliers. La méthode la plus simple consiste à construire un tas local sans source mémoire et à y placer le segment mémoire. Les fonctions d'allocation internes sont alors utilisables comme le montre le code 4.2. Ce support a notamment été utilisé de manière expérimentale comme base d'implémentation d'un module SHM par un stagiaire (Antoine Capra) pour les communications entre processus d'un même nœud dans MPC.

Code 4.2– Allocation sur un segment utilisateur via les méthodes d'allocation spécifiques.

```

1 //preparation du buffer
2 struct alloc_chain chain;
3 void * buffer = setup_buffer(SIZE);
4
5 //mise en place de l'allocateur
6 user_chain_init(&chain,buffer,SIZE,
7     CHAIN_FLAGS_STANDALONE);
8
9 //utilisation de l'allocateur
10 void * ptr = chain_alloc(&chain,32);
11 ptr = chain_realloc(&chain,64);
12
13 //libération par la methode standard
14 free(ptr);

```

Remarquons que l'interface standardisée des chaînes d'allocation permet de capturer toutes les allocations au travers des appels standards de l'allocateur en faisant pointer temporairement la TLS vers le tas spécialisé. Le code 4.3 montre l'exemple de cette manipulation. Cela permet également de capturer les opérateurs *new* et *delete* du langage C++ ou un ensemble de sous-fonctions d'une bibliothèque. On peut par exemple isoler un segment de code que l'on sait bogué pour qu'il ne pollue pas l'allocateur principal.

Code 4.3– Allocation sur un segment utilisateur via les méthodes d'allocation standards.

```

1 //remplacement de la chaine standard
2 //pour le thread courant
3 old_chain = set_default_chain(&user_chain);
4
5 //capture
6 MyObject * obj = new MyObject;
7 functionWhichUseMalloc();
8
9 //restauration de la chaine d'origine
10 default_chain(&old_chain);
11
12 //utilisation de delete
13 delete obj;

```

Il est également possible de construire sa propre source mémoire si l'on désire permettre l'extension automatique de l'espace mémoire du tas. Sans cela, l'allocateur renvoie un code d'erreur lorsque le segment est épuisé.

4.14 Méthode d'implémentation

L'implémentation d'un allocateur mémoire est une tâche délicate du fait des choix cruciaux intervenants dans la conception et difficiles à changer par la suite. Il faut également ajouter la difficulté à déboguer les éventuels problèmes de corruption mémoire survenant lors de l'exécution de grosses applications. Afin d'éviter au maximum la survenue de tels problèmes chronophages lors des tests applicatifs, nous avons développé l'allocateur en appliquant une méthode

de type développement dirigée par les tests (TDD : Test Driven Développement[Bec03]) typiques des méthodes agiles. L’allocateur dispose donc d’une base de tests unitaires ayant permis d’éliminer la majorité des problèmes avant de passer à une phase de test avec des applications réelles. Cette approche a également permis la modification d’une structure centrale de l’allocateur en vue d’optimisation. Cette modification qui a apporté un gain d’un facteur 2 n’aurait pas été possible ou très difficile sans tests. Elle impliquait en effet un changement sur une structure impactant l’ensemble du code. Cette base de 160 tests a également été utile en permettant à un alternant (Julien Adam) de réaliser en grande partie le portage de l’allocateur sous Windows.

4.15 Évaluation

Dans cette section, nous allons donner une partie des résultats obtenus avec notre allocateur. Ce dernier sera comparé aux allocateurs disponibles discutés en section 4.4. Pour cela, nous commencerons par observer certains micro-benchmarks validant le comportement du code et progresserons ensuite avec divers benchmarks pour terminer l’évaluation avec l’application Hera.

4.15.1 Micro-benchmarks

Le premier micro-benchmark correspond à un accès peu réaliste avec des allocations/libération par lot et locales à chaque thread. Les résultats sur les nœuds larges de Curie sont présentés dans la figure 4.12. Ces résultats montrent que nos performances sur ce type de benchmark sont proches de Tcmalloc pour les grosses allocations. Nos petites allocations sont plus proches du comportement de la glibc même si en retrait du fait d’un manque d’optimisation de cette partie en séquentiel. Ceci est notamment dû aux fusions immédiates actuellement utilisées dans notre allocateur. Notons toutefois que l’écart de performance se gomme avec l’augmentation du nombre de cœurs. Les allocateurs Hoard et Jemalloc rencontrent de gros problèmes de passage à l’échelle sur ce benchmark. La Glibc rencontre un problème similaire pour les grosses allocations.

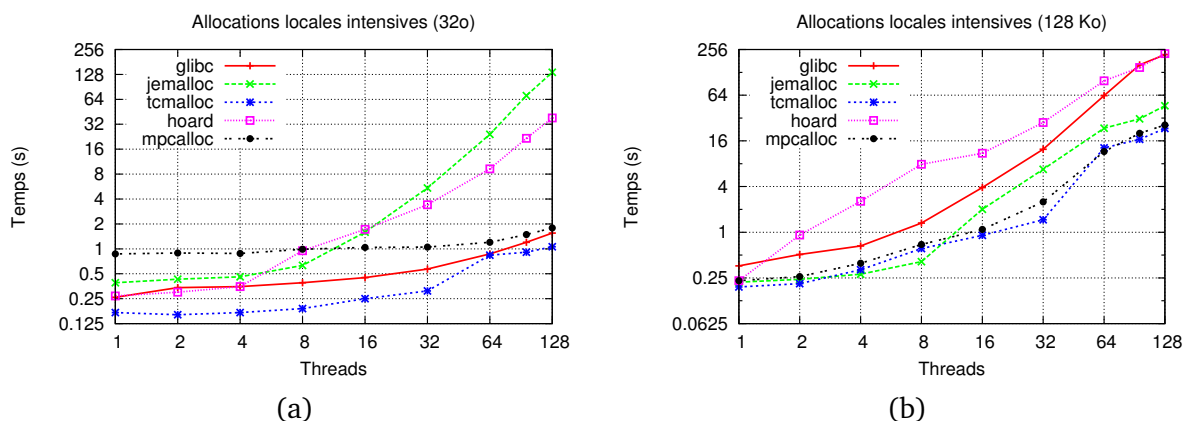


FIGURE 4.12 – Benchmark d’allocations locales de (a) 32 octets ou (b) 128 kilo-octets. Les allocations sont réalisées par lot de 1000 éléments avant libération par le même thread. Lors de ce benchmark, les threads ont été punaisés avec l’outil likwid-pin[THW10].

Le second micro-benchmark correspond à une évaluation d’opération en mode producteur consommateur afin d’évaluer l’impact de la stratégie retenue à cet effet au travers des *files de libération distantes*. Aucun travail n’est effectué sur les données afin de stresser l’allocateur au maximum et le nombre d’allocations total est maintenu constant. Les résultats présentés par la figure 4.13 ont été obtenus sur les nœuds 128 cœurs du calculateur Curie en utilisant un

nombre variable de threads. Le thread 0 génère des ensembles d’allocations de 32 octets qui sont transmis aux différents threads pour libération. Les différents allocateurs montrent clairement des comportements divergeant au-delà de 8 cœurs en fonction des algorithmes retenus. Notre allocateur offre dans ce cas une performance proche de Jemalloc et plus efficace que la Glibc. Remarquons toutefois l’écart de performance pour le mode séquentiel du fait d’un léger manque d’optimisation de notre part dans la gestion spécifique des petits blocs qui ne sont pas l’objectif principal de notre allocateur. Il en résulte une meilleure extensibilité de notre algorithme comparé à l’ensemble des autres allocateurs. Cette dernière s’explique par la prise en compte explicite de la structure NUMA et l’utilisation des FLD atomiques. Les performances pourraient certainement être améliorées en reprenant pour partie la notion de cache local exploité par Jemalloc. On notera que sur ce cas test Hoard multiplie par 10 la consommation des autres allocateurs pour 128 threads.

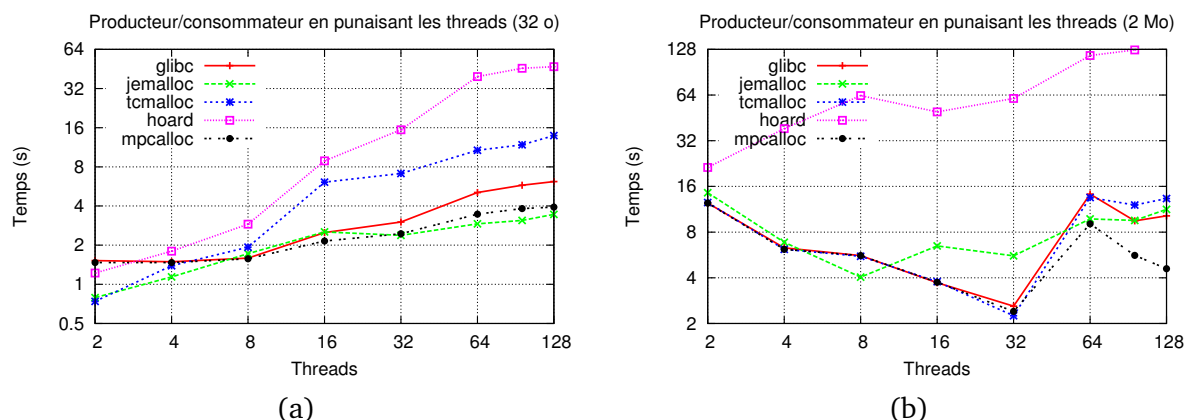


FIGURE 4.13 – Benchmark d’allocation en mode producteur (1 thread) consommateur (autres threads) pour un total constant de (a) 6 millions d’allocation de 32 octets ou (b) 1600 allocations de 2 Mo. Lors de ce benchmark, les threads ont été punaisés avec l’outil likwid-pin.

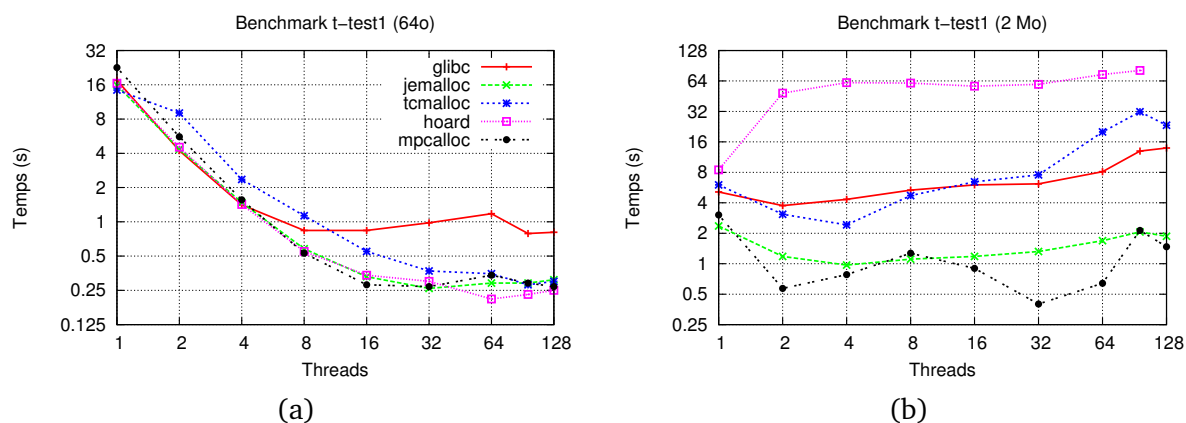


FIGURE 4.14 – Benchmark d’allocation *t-test1* provenant d’Hoard et Lockless. Le benchmark génère des allocations de tailles aléatoires bornées par la consigne (a) 64 octets ou (b) 2 Mo. Les blocs sont ensuite échangés entre les threads.

La figure 4.14 donne un extrait de performance obtenu sur le benchmark *t-test1* utilisé par un certain nombre d’allocateurs, dont Hoard et Lockless[[MH](#)]. Ce dernier reprend le cas précédent, mais en exploitant des blocs de taille aléatoire inférieure à une borne. Notre allocateur fournit de bonnes performances sur ce benchmark qu’il s’agisse des petites ou grosses allocations. Sur ce benchmark, notre allocateur offre des performances dans la moyenne pour les petites tailles

et comme précédemment des gains sur les grosses allocations.

4.15.2 Résultats sur sysbench

Afin de valider les performances sur une application plus concrète, on peut exploiter le benchmark sysbench[Kop] basé sur MySQL en générant un grand nombre d'allocations en contexte parallèle. Ce benchmark également exploité par d'autres allocateurs a toutefois l'inconvénient de ne pas passer à l'échelle sur les nœuds NUMA dont nous disposons. Il est néanmoins possible d'observer les gains apportés par notre allocateur. Le benchmark est exécuté sur les nœuds 128 cœurs du calculateur Curie en punaisant les threads avec l'outil likwid-pin[THW10] pour éviter des migrations inopinées. Les performances sont présentées sur le graphique 4.15 et montrent clairement l'apport de notre allocateur y compris au pic (8 cœurs) de performance avec un gain de 13% comparé à la glibc. Sur 128 cœurs, le gain apporté est de 34% bien que le benchmark ne soit pas extensible à cette échelle. Sur ce benchmark l'allocateur Hoard fournit des performances raisonnables, bien que moins régulières.

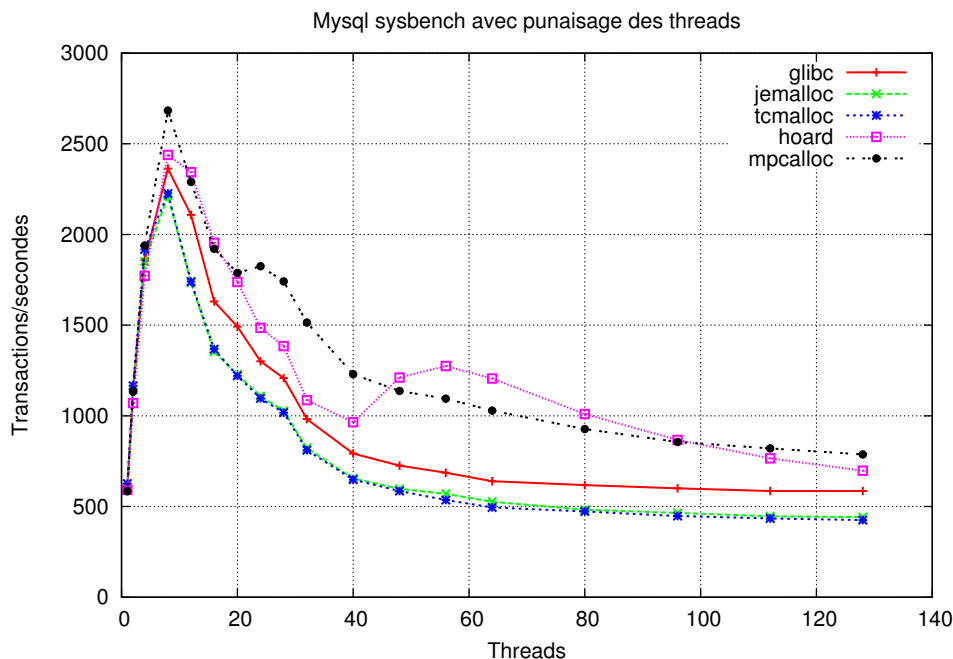


FIGURE 4.15 – Sysbench avec MySQL en fonction de l'allocateur sur les nœuds larges Curie. Lors de ce benchmark, les threads ont été punaisés avec l'outil likwid-pin.

4.15.3 Résultats sur la simulation numérique Hera

Nous terminons ici les résultats liés à l'allocateur avec un test sur un code imposant de simulation numérique : Hera. Nous pouvons ainsi éprouver la méthode sur une application réelle et exploitée à grande échelle sur les supercalculateurs du CEA. Les résultats sont présentés dans la table 4.2. Ces essais sont réalisés avec trois profils de configuration pour notre allocateur. Ces trois profils permettent de décorréler les différents problèmes rencontrés et non séparables avec les autres allocateurs. Pour ce faire, nous avons défini les profils UMA et NUMA qui permettent un maintien dans les sources mémoires d'un maximum de 500 Mo de mémoire. Ces deux profils activent ou non le support des architectures NUMA. Le dernier profil (ECO) est configuré pour limiter la consommation mémoire en ne gardant aucun segment dans les sources mémoires.

A : Nœuds 12 cœurs Cassard (2 * 6)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (GB)
1	MPC-NUMA	135.14	132.63	1.79	4.3
2	MPC-UMA	146.11	143.50	1.86	4.3
3	MPC-ECO	162.96	130.98	16.20	2.0
4	Glibc	143.89	130.10	8.53	3.3
5	Jemalloc	143.05	128.07	14.53	1.9
6	TCMalloc	141.14	139.98	0.65	6.9
B : Nœuds 32 cœurs Tera 100 (4 * 8)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (GB)
1	MPC-NUMA	89.33	64.34	2.39	15
2	MPC-UMA	94.82	71.41	2.58	15
3	MPC-ECO	248.17	74.19	87.21	6.7
4	Glibc	101.11	67.43	9.41	8.1
5	Jemalloc	145.73	70.49	57.32	6.7
6	TCMalloc	106.28	82.97	1.96	8.6
C : Nœuds 128 cœurs Tera 100 (4 * 4 * 8)					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (GB)
1	MPC-NUMA	120.07	100.44	5.64	16.9
2	MPC-UMA	229.38	207.25	5.88	16.5
3	MPC-ECO	762.47	460.53	56.13	14.1
4	Glibc	284.06	170.94	15.9	14.1
5	Jemalloc	351.49	214.54	123.99	12.2
6	TCMalloc	438.42	396.59	27.57	14.4

TABLE 4.2 – *Mesure de performance de la simulation numérique Hera avec différents allocateurs sur les nœuds NUMA disponibles au CEA. Les exécutions sont réalisées dans le mode de fonctionnement canonique de MPC à savoir un processus par nœud et un thread par cœur physique. Pour être comparables, les temps utilisateurs et systèmes sont donnés par thread. Ces tables montrent les gains importants de notre allocateur sur les nœuds NUMA 128 cœurs.*

Le mode ECO de notre allocateur montre des consommations mémoires proches des résultats de Jemalloc sur une partie des machines, mais au prix d'un surcoût en temps induit par les appels trop nombreux à destination de l'OS. En comparant ces résultats au mode UMA, nous pouvons extraire l'impact de l'interaction avec l'OS. Ce mode apporte une réduction du temps système non négligeable (facteur 4 à 10) qui se traduit également en gain sur le temps d'exécution total avec des performances proches (architecture A) ou meilleures que le plus performant des allocateurs (architectures B et C). Sur 128 cœurs, les gains apportés par ce mode atteignent 20% comparé à la Glibc. On peut alors comparer ces résultats avec l'activation du support NUMA qui permet de dépasser les performances de tous les allocateurs testés en doublant les performances de la glibc sur 128 cœurs. Ces gains sont toutefois obtenus au prix d'une augmentation de la consommation mémoire d'environ 2 Go.

Avec ces mesures, on montre l'intérêt d'introduire un support explicite des architectures NUMA au niveau de l'allocateur et de prendre en compte le problème d'échange avec l'OS en ce qui concerne les allocations de grands segments. On remarque également que nous sommes parvenus à obtenir une implémentation permettant de passer d'un profil plus économe de type Jemalloc à un profil plus performant, mais consommateur de mémoire de type TCMalloc. Ces résultats ouvrent donc la porte à une adaptation dynamique entre ces profils extrêmes en fonction de la mémoire disponible.

4.16 Bilan général

Nous avons vu avec les résultats précédents qu’il était possible d’améliorer les performances des allocateurs actuels sur les nœuds NUMA dont nous disposons. Nous avons notamment vu sur un cas concret de simulation numérique que l’OS pouvait devenir un problème majeur avec un coût associé à l’OS de 20% du temps total d’exécution. Pour ce type d’application, le contrôle des échanges avec l’OS devient plus important que l’optimisation propre de la gestion des petits blocs. À ce titre, nous avons notamment proposé une méthode de réutilisation des gros segments éliminant les problèmes de fragmentation par utilisation de la fonction *mremap* comme élément central de l’approche. L’utilisation de threads sur architecture NUMA nécessite également la mise en place d’allocateurs supportant explicitement ces dernières. On notera les efforts fournis pour maintenir l’utilisation de l’API standard d’allocation afin obtenir le support NUMA sur la base d’établissement de niveau de confiance envers les différents threads. Toujours pour les architectures NUMA, nous avons mis en place des procédés éliminant au maximum le besoin de synchronisation de sorte à obtenir des tas locaux sans verrous. Ne subsistent donc que les synchronisations nécessaires pour les *libérations distantes* et l’accès aux *sources mémoires* communes. Remarquons que cette approche est permise par le mécanisme d’indexation à base de *région* lui-même essentiellement sans verrous. Grâce à ce travail, nous avons obtenu des gains importants sur la grosse simulation numérique Hera avec un double des performances de l’allocateur de la glibc. Du fait d’un manque de prise en compte de l’OS et du NUMA, les autres allocateurs parallèles montrent des performances très dégradées dans ces mêmes conditions.

4.17 Discussion d’améliorations possibles

Dans cette section nous allons discuter certaines améliorations qu’il serait bon d’intégrer et déduites du recul pris lors de nos travaux sur l’allocateur. Les remarques qui suivent prennent en compte les problèmes et erreurs rencontrés lors de nos travaux sur l’allocateur.

4.17.1 Niveaux topologiques

La version actuelle de notre allocateur mémoire met en place un tas local pour chaque thread. Cette stratégie peut s’avérer mauvaise pour des programmes très dynamiques en terme de création de threads. Dans ce contexte, il serait bon de retenir une politique de réutilisation des tas existants pour limiter leur nombre au lieu de les détruire en même temps que les threads.

À ce titre, on peut proposer de contrôler l’instanciation des allocateurs suivant la topologie de la machine. Avec l’aide de *hwloc*, il serait ainsi intéressant de pouvoir définir le niveau d’association des tas locaux (cœur, socket, nœud NUMA, niveau de cache...). Ce type d’approche pourrait permettre de réduire la consommation mémoire des différents tas en mettant en commun les blocs libres de ces derniers tout en maintenant des synchronisations locales.

Remarquons également que MPC crée autant de threads système que de cœurs puis met en place des threads utilisateurs à l’intérieur de ces derniers. Dans ce contexte, il peut être intéressant de créer un tas local par thread système et non utilisateur. Ceci est d’autant plus intéressant que les threads utilisateurs de MPC sont non interruptibles. Par conséquent, les tas locaux de cette configuration peuvent être maintenus sans primitive de synchronisation. Il est donc possible de gagner sur le plan de la consommation mémoire sans perdre sur le plan de la performance.

4.17.2 Politique de consommation dynamique

Nous avons vu avec les résultats précédents que nous avons obtenus un allocateur capable de reproduire en partie les résultats d'économie mémoire de Jemalloc ou bien un profil plus gourmand en mémoire évitant les pertes de performances liées à l'OS. Nous avons notamment pris soin de rendre les décisions de ces profils réversibles. Nous pouvons donc désormais mettre en place une politique d'adaptation dynamique entre les profils extrêmes en cours d'exécution. De cette manière, il est possible de s'adapter à la consommation mémoire des différentes phases de l'application. On remarque en effet que certaines applications tendent à produire des pics localisés de consommation. Ces pics déterminent les limites de taille de problème exploitable par le programme. Lors des pics de consommation, il serait donc intéressant de basculer l'allocateur dans un mode économe pour passer le cas et pouvoir compter sur des gains de performances pour les étapes moins consommatrices. Au niveau de notre allocateur, un basculement vers le profil économe peut être réalisé en suivant les étapes :

1. Changement des paramètres de contrôle de consommation en n'autorisant plus la source mémoire à maintenir des macro-blocs pour usage futur.
2. Purge des macro-blocs en attente dans les sources mémoires.
3. Pour diminuer plus la consommation mémoire, il est aussi possible de parcourir les blocs libres et rendre au système les pages non utilisées à l'aide d'appels à *madvise*. Ce traitement peut se faire au prix d'un surcoût lors du changement de politique.

Cette approche n'a pas encore pu être évaluée d'un point de vue pratique, mais semble toutefois rencontrer deux difficultés qu'il est nécessaire de lever. La première correspond au choix de la consommation à contrôler : *virtuelle* ou *physique*. Dans le cas où l'application ne fait pas trop d'allocations *spéculatives*, il n'y a peu de différence, mais les deux peuvent être amenées à diverger dans le cas contraire. Le contrôle de la mémoire virtuelle peut surestimer la consommation réelle de l'application. Les limites d'exécution sont liées à la mémoire physique, c'est idéalement cette dernière que l'on doit suivre ou estimer. Le second problème est lié au choix du seuil limite. Si l'on considère un unique processus sur le nœud, il est possible d'analyser la mémoire disponible au lancement de l'application et de fonctionner en estimant l'évolution de cette dernière en fonction des allocations reçues. Afin de s'assurer d'un bon suivi, il serait certainement important de mettre à jour cette information au-delà d'une certaine quantité de mémoire échangée avec l'OS. Dans le cadre de notre allocateur, ce suivi doit se faire au niveau des sources mémoires. Ces dernières étant multiple (une par nœud NUMA) il serait important de mettre en place un composant central chargé du suivi et interroger périodiquement par les sources mémoires NUMA.

En pratique il est important de noter que le seuil limite doit être choisi en deçà de la quantité de mémoire disponible sous peine de trop pénaliser le cache disque qui peut avoir un impact important si l'application effectue beaucoup d'entrées/sorties. Si l'on souhaite considérer un environnement multi-processus, il serait important d'utiliser une augmentation de la consigne de consommation lente et lissée. On peut ainsi espérer permettre aux différents processus d'aboutir à un état stable, évitant le passage brutal d'un mode (économe ou vitesse) à l'autre de manière alternative pour chacun. Pour ce faire, on pourrait asservir l'augmentation du seuil de mémoire maintenu dans les sources mémoires à une fonction dépendante du débit moyen d'échange mémoire avec l'OS (inspiré du paramètre seuil de TCMalloc) et de la mémoire disponible. La configuration s'exprimerait alors sous la forme de deux paramètres : le taux minimal de mémoire devant être laissé libre et un débit maximum d'échange moyen de mémoire avec l'OS. Ces deux paramètres sont mesurables, le dernier pouvant être estimé par un micro-benchmark en fonction des limites d'extensibilité de l'OS.

4.17.3 Surcharge de mmap/munmap ?

Lors de notre étude, nous avons choisi de ré-implémenter entièrement un allocateur. Nous nous sommes toutefois principalement consacrés à la réutilisation des macro-blocs. Avec le recul, il serait peut-être intéressant d’extraire le travail réalisé dans cette partie et le mettre en place sous la forme d’une surcharge des fonctions *mmap/munmap/mremap*. Il serait ainsi possible d’appliquer notre approche de réutilisation sur les divers allocateurs disponibles. Ce choix n’a pas été fait initialement, car un support explicite du NUMA bien que pris en charge au niveau de nos sources mémoires nécessite l’assurance d’une association fiable de ces dernières aux différents tas locaux. Nous ne pouvions toutefois pas garantir facilement ce point sur les allocateurs usuels qui peuvent (jemalloc, TCMalloc et hoard) associer plusieurs threads à un tas, et ce de manière aléatoire. Lors du retour des blocs (surcharge de *munmap*) il serait également indispensable d’utiliser des en-têtes externes pour savoir quel nœud NUMA a alloué le macro-bloc, les allocateurs usuels pouvant potentiellement avoir fait des échanges entre les tas locaux, notamment lors de la libération des gros segments réalisés par un appel direct à *munmap* depuis le thread de libération. Cette approche mériterait d’être évaluée même si l’on peut s’attendre à des gains plus limités sur architecture NUMA.

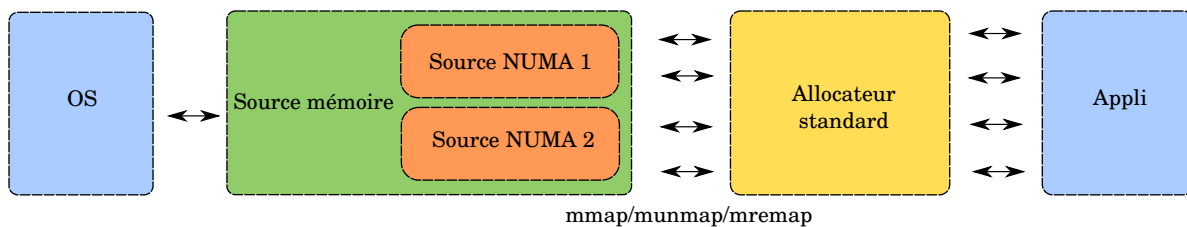


FIGURE 4.16 – Exemple d’exploitation de la méthode de cache de gros segment en interceptant *mmap*, *munmap* et *mremap* pour des allocateurs standards.

4.17.4 Modification de Jemalloc ?

Si l’on considère la réalisation de l’idée précédente, on dispose d’un cache mémoire générique à placer sous un allocateur standard. Partant de cet outil, il serait certainement intéressant et plus robuste de construire un allocateur en reprenant entièrement Jemalloc. Ce dernier dispose en effet d’une implémentation efficace à faible consommation. L’ajout du cache sous-jacent permettrait de combler ses lacunes en terme d’appels trop intensifs à l’OS tout en offrant la possibilité de contrôler efficacement le compromis consommation performance comme nous l’avons fait dans notre allocateur.

Un support efficace du NUMA nécessiterait toutefois quelques modifications au niveau de cet allocateur en reprenant certaines de nos observations. Il serait tout d’abord essentiel de modifier la méthode d’association aléatoire des threads aux tas locaux. Il faut en effet limiter ces dernières par groupes NUMA avec application des règles de confiance (fiable, non fiable) appliquées dans notre approche et décrite en section 4.11. Si Jemalloc ne supporte pas la notion de libération distante de manière spécifique, il serait probablement important de réintroduire la notion de *file de libération distante* (FLD) que nous avons utilisé en section 4.6.6.

Jemalloc emploie également une politique de maintien des derniers blocs libérés pour une réutilisation rapide. Ce mécanisme devrait également être modifié pour ne pas maintenir des blocs provenant d’un thread distant (au sens NUMA) dans un cache local.

4.17.5 API, sémantique NUMA ?

L'API POSIX actuelle ne permet pas à l'allocateur de disposer de suffisamment d'information pour traiter de manière fiable les architectures NUMA en mode thread. Dans notre étude nous avons observé des gains non négligeables en nous restreignant à cette API. Il nous faut toutefois remarquer que ces gains sont obtenus sur des applications limitant les échanges d'éléments entre les threads. L'allocateur peut donc raisonnablement supposer les intentions de l'utilisateur. Nous avons en effet exploité une approche MPI implémentée sur base de thread, mais utilisant un travail local au threads et des échanges inter-threads sous la forme de messages. Dans un modèle type OpenMP ou modèle à base de tâches, il importe de remarquer que les intentions de l'utilisateur peuvent devenir radicalement différentes de ce qui est supposé par l'allocateur. Ceci notamment pour des tableaux non uniformes (au sens NUMA) dont les accès sont répartis entre les threads. Dans un tel contexte, il peut être important de proposer des extensions d'interface permettant de transmettre des déclarations d'intention à l'allocateur.

Pour ne pas trop modifier l'API actuelle, le plus simple serait certainement d'introduire des *pragmas*. On pourrait ainsi notifier l'allocateur de l'attente pour l'allocation ou groupe d'allocations suivant. Cette approche de pré-définition d'intention est la seule valide si l'on considère le C++ avec son opérateur `new` qui peut difficilement être étendu en terme de sémantique par ajout de paramètres. En terme d'information, si l'on considère le maintien d'un allocateur générique, il serait important de pouvoir distinguer quatre catégories d'allocation :

Locale (*local*) : C'est la supposition par défaut faite par l'allocateur, à savoir que l'élément alloué doit être utilisé par le thread chargé de l'allocation.

Distante (*remote*) : Le bloc est alloué pour être utilisé sur un autre nœud NUMA dont on connaît l'ID. Nous fournissons actuellement un tel support au travers de la fonction `malloc_on_node()`.

Motif (*map*) : Une fonction de placement peut être donnée pour définir un motif de position de chacune des pages du segment. Plusieurs fonctions élémentaires peuvent alors être offertes en standard : mode entrelacé, aléatoire et répartition par zones contiguës.

Inconnue (*unknown*) : L'utilisation n'est pas connue à l'avance, le segment ne peut donc être placé avec confiance. Idéalement l'allocateur doit laisser la main à l'OS en fournissant des segments virtuels. À la libération, l'allocateur ne peut garder ces segments dont il ne connaît pas le placement ou dont il sera trop difficile de trouver une utilisation s'il ne sont pas uniformes.

Pour des allocateurs spécifiques, il est possible de produire une sémantique plus riche, mais dans le cadre d'un allocateur généraliste cette dernière doit être limitée, car aucune supposition ne peut être faite sur l'application. Remarquons que notre implémentation prend en charge les deux premiers points si l'on considère la politique par défaut et la présence de la fonction `sctk_malloc_on_node()`. Pour les deux suivants, il est nécessaire d'ajouter un drapeau sur les blocs concernés pour les libérer directement vers l'OS, leurs réutilisations étant trop délicates au sens général. On peut proposer comme exemple de syntaxe C :

Code 4.4– Proposition de pragma pour fournir de l'information à l'allocateur

```
1 #pragma numaalloc local [strict]
2 #pragma numaalloc remote(id) [strict]
3 #pragma numaalloc map(func) [strict]
4 #pragma numaalloc unknown
5 void * ptr = malloc(SIZE);
```

Des améliorations sont possibles si l'on obtient un moyen efficace d'obtenir le placement des pages auprès de l'OS au moment des libérations. Ces informations peuvent aisément être

exploitées pour la gestion des petits segments internes aux pages. Dans le cas d'allocateurs généralistes, ces informations ne pourront toutefois être exploitées que de manière parcellaire en considérant le placement majoritaire des pages d'un grand segment. Pour ces segments, il est certainement préférable de se reposer sur l'OS à condition de lever les limitations qui nous ont obligés à mettre en place du recyclage pour cette catégorie d'allocation.

Chapitre 5

Problématique de remise à zéro de la mémoire

Dans la section précédente, nous avons montré que l'application Hera pouvait être largement pénalisée sur les nœuds à 128 cœurs. Au-delà du non-support NUMA des allocateurs testés, ces résultats mettent en avant une augmentation du temps système sur les gros nœuds. Nous avons donc construit une politique d'allocation permettant de limiter les échanges avec l'OS et obtenu une nette amélioration des performances. Ces gains sont toutefois obtenus au prix d'une augmentation de la consommation mémoire.

Dans ce chapitre, nous allons observer le système lui-même et tenter de comprendre une part du problème en travaillant directement sur ce dernier. Nous allons notamment nous concentrer sur le problème de remise à zéro des pages fraîchement allouées qui représentent une part importante des coûts d'allocation. Nous discuterons les problèmes de passage à l'échelle rencontrés au niveau de l'OS, que nous tenterons dans un premier temps de résoudre par emploi de grosses pages. Nous montrerons que cette méthode a également ses limites. Nous proposerons donc une amélioration des performances, basée sur une extension de la sémantique d'échange de mémoire entre l'OS et les applications et montrerons qu'elle est surtout efficace avec les grosses pages.

5.1 Évaluation du problème de performance

Rappelons que lors des allocations mémoires, l'allocateur peut être amené à demander la projection dans l'espace virtuel de nouveaux segments. Ces échanges passent par l'appel *mmap* déjà discuté précédemment. On rappelle également que l'utilisation de cet appel fournit à l'utilisateur un segment purement virtuel, ceci, du fait de la politique de pagination paresseuse décrite en section 2.3.2. Lors d'une nouvelle allocation de ce type, le premier accès génère des fautes de pages pour obtenir des pages physiques. On s'intéresse ici à l'extensibilité de ces fautes de pages, considérant le cas d'une application parallèle basé sur les *threads*. Pour cela, on peut construire un micro-benchmark en procédant comme suit :

- Création de N *thread* OpenMPI ou N *processus* MPI.
- Chaque tâche alloue un grand segment mémoire pour un total de 10Go de mémoire alloué sur le nœud.
- Le temps du premier accès en écriture est mesuré à l'aide du compteur RDTSC pour chacune des pages accédées, afin d'obtenir une distribution des temps de fautes de pages.

Nous avons également considéré une mesure de confirmation par un simple accès de type *memset*, afin d'éliminer un risque de biais, par notre méthode de mesure page par page entrelacée avec la construction en vol d'un histogramme. De la même manière, la procédure de mesure est rejouée sur des segments physiquement alloués afin d'évaluer le biais de mesure qui, sur Nehalem, est de l'ordre de 200 cycles. La figure 5.1 donne la distribution détaillée pour un

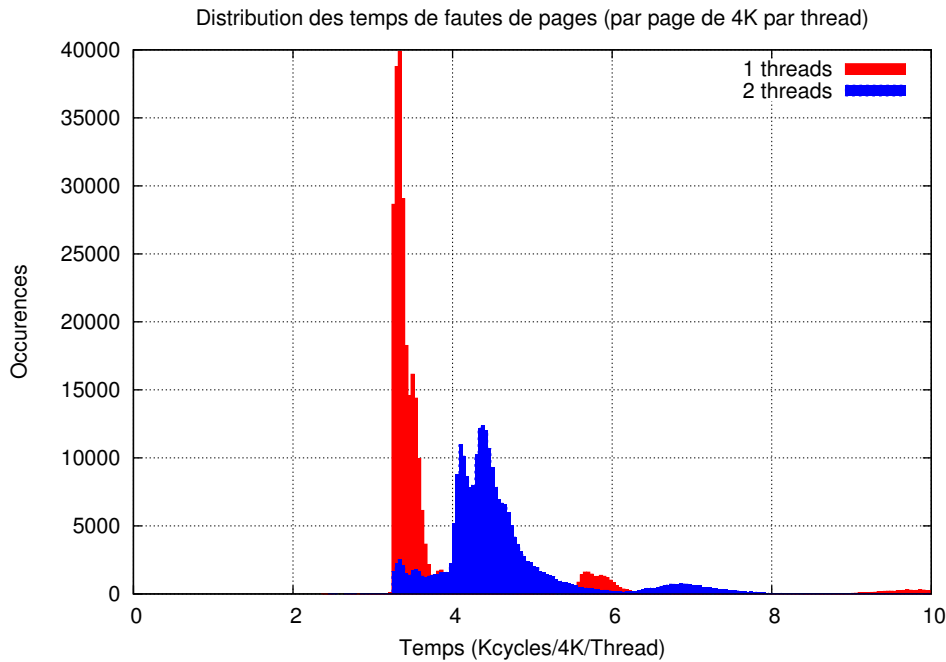


FIGURE 5.1 – Distribution des temps de fautes de pages sur les nœuds Cassard pour 1 ou 2 threads. Cette mesure est réalisée sur la base d’une allocation d’un segment de 10Go.

ou deux threads. Les distributions prennent la forme d’une gaussienne associée à une queue se prolongeant dans les valeurs hautes, liées à l’impact de l’ordonnancement. Cette figure montre clairement un décalage du pic de la distribution vers les valeurs hautes lorsque le nombre de threads augmente. On peut ainsi obtenir les courbes de la figure 5.2, donnant l’extensibilité des fautes de pages sur les nœuds de 128 cœurs de Tera 100 ou sur Xeon Phi. Les mesures sont données pour des allocations parallèles en mode *thread* ou *processus*. Sur Xeon Phi notre méthode de mesure introduisait un biais trop important, du fait d’un manque d’optimisation de notre code pour cette architecture. Les résultats de cette architecture sont donc donnés par la méthode de confirmation à base d’un accès complet par *memset*. Ceci explique l’absence de barres d’erreurs sur ce graphique. Les temps sont donnés par page par thread ; on attend donc idéalement un temps constant.

Sur les nœuds de 128 cœurs, la mesure montre une relative extensibilité pour les processus, donc pour des approches types MPI. Un léger effet est observable au-delà de 8 cœurs et induit par la structure NUMA du nœud. À l’opposé, l’utilisation de threads conduit à une augmentation proportionnelle au nombre de flux utilisés. Ce problème se comprend bien si l’on considère que la table des pages d’un processus est commune à l’ensemble de ses threads. Toute modification de cette dernière nécessite donc, une prise de verrous conduisant au problème observé. Ce manque d’extensibilité peut devenir bloquant pour toute application ayant tendance en contexte parallèle à libérer/allouer des segments de grandes tailles comme nous l’avons vu avec l’application Hera. Remarquons que les simulations numériques tendent à fonctionner par phase et donc à grouper les allocations. Ce comportement tend donc à favoriser les contentions sur les allocations. Ce problème a été étudié en 2011 par l’équipe de Clements Austin[CKZ12] en pointant les contentions sur ces verrous. Ils ont ainsi proposé une modification des algorithmes du noyau pour maximiser l’utilisation des RCU[MS98] à la place des verrous conventionnels. Ce type de verrous permet des accès en lecture sans blocage. Ceci introduit toutefois des contraintes supplémentaires sur les méthodes de mise à jour qui impliquent des restrictions sur les algorithmes candidats.

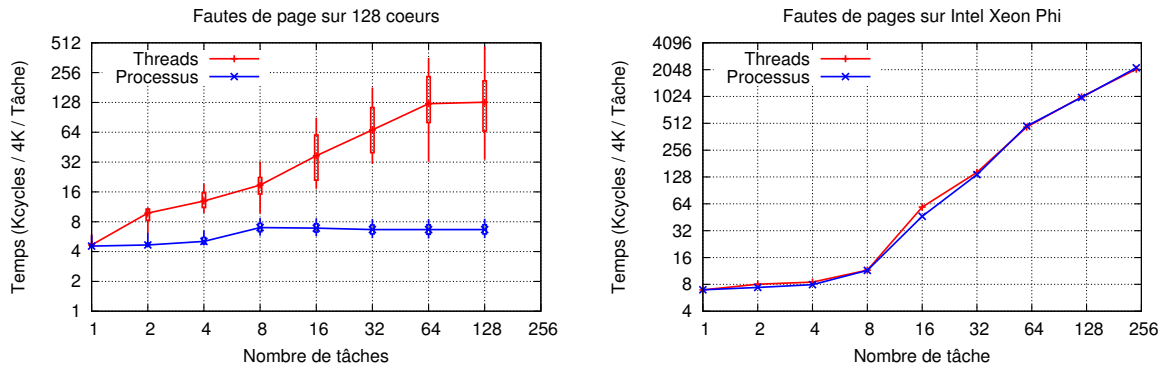


FIGURE 5.2 – Temps des fautes de pages sur les nœuds larges Tera 100 et sur Xeon Phi. Les barres d'erreurs donnent les quartiles 50% et 80%.

Sur Xeon Phi, même observation pour les threads. Toutefois, les processus sont cette fois-ci impactés par le problème. Nous n'avons pour l'instant pas de preuves formelles de la source de ce nouveau problème. Nous supposons qu'il s'agit d'un problème de contention sur certaines structures globales du noyau (compteurs, listes...). Sous Linux, la mémoire est découpée en *ré-gions* correspondant aux différents nœuds NUMA. Le Xeon Phi n'est pas vu comme NUMA par l'OS. La seule région mémoire mise en place est donc accédée par les 240 threads exploitables contre une limite à 8 pour notre nœud 128 cœurs. On remarquera également que les instructions atomiques sont plus pénalisantes sur Xeon Phi que sur les processeurs conventionnels. Des problèmes peuvent donc apparaître sur certains compteurs communs (mémoire consommé par l'utilisateur...).

5.2 Utilisation de grosses pages

En section 2.5.3 nous avons décrit la présence d'un support de *grosses pages* de 2 Mo dans les architectures types x86_64. Ces dernières sont habituellement mises en place pour améliorer les performances des TLB. Ces gains s'obtiennent notamment par une réduction du nombre de pages nécessaires à l'adressage, par un facteur 512 comparé à une base de 4 Ko. On peut alors se demander si cette réduction du nombre de pages ne peut pas également réduire la contention que nous observons sur les fautes de pages de LINUX.

Le graphique 5.3 donne le résultat des mesures obtenues avec l'implémentation THP de Linux. Les résultats avec grosses pages sont divisés par un facteur 512 pour normaliser ces derniers sur la base de segment de 4 Ko. Cette normalisation permet ainsi de comparer le coût relatif aux pages standards de 4 Ko sur une base commune. Sur ce graphique, on remarque une amélioration des performances séquentielles avec une réduction du coût de 3400 à 2000 cycles, soit une réduction de 40% que l'on doit comparer au facteur 512 attendu dans l'idéal. Avec 24 threads, les coûts deviennent similaires aux pages standards. On observe de plus, l'apparition du même problème d'extensibilité avec une dégradation plus rapide des performances. Les grosses pages apportent donc un léger gain en séquentiel, mais souffrent du même problème que les pages standards en parallèle sur 24 threads.

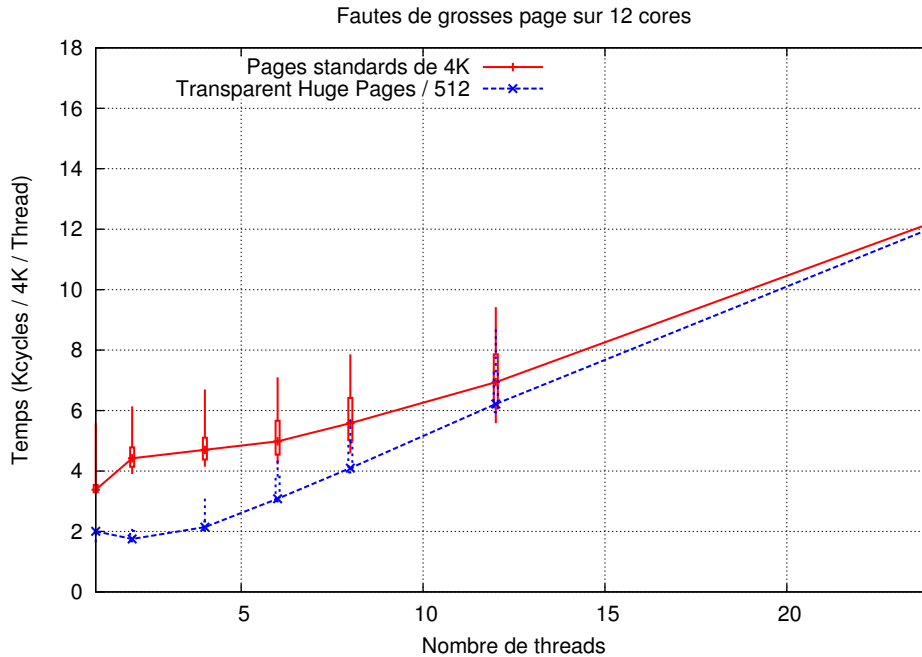


FIGURE 5.3 – Mesure des temps des fautes de pages en utilisant l'implémentation THP de Linux sur les nœuds Bi-Westmere du calculateur Cassard. La mesure est effectuée en accédant à un élément de chacune des pages de 2 Mo. Les temps obtenus sont divisés par 512 pour donner un temps normalisé par segments de 4 Ko.

5.3 Le problème de la remise à zéro

Lorsqu'une faute de page survient, il importe de remarquer que les pages physiques fournies à un processus étaient précédemment exploitées par le noyau ou par un autre processus. Pour des raisons de sécurité, il importe donc d'effacer le contenu de ces dernières afin d'interdire toute fuite d'information d'une entité vers une autre. Pour ce faire, chaque faute de page implique un appel à la fonction `clear_page()` du noyau afin de remplir les pages de zéro. Nous remarquerons toutefois que l'écriture de ces zéros peut représenter un coût non négligeable. Comme vu précédemment, une faute de page coûte en séquentiel de l'ordre de 3400 cycles. Or, la remise à zéro peut être évaluée à près de 1400 cycles, soit 40% du coût total d'une faute de page. Les grosses pages sont, elles, proportionnellement dominées par la remise à zéro avec un impact supérieur à 97%. Dans ce contexte, on explique pourquoi les grosses pages ne permettent pas d'obtenir des gains, puisqu'elles ne peuvent réduire que le coût constant de manipulation des structures correspondant aux 60% des pages standards. La part du coût lié à la remise à zéro, elle, augmente de manière proportionnelle à la taille des pages et finit par devenir dominante. Dans le noyau Linux, on observe de plus, que la fonction `clear_page()` est appelée dans une section critique protégée par des verrous en lecture. L'utilisation de grosses pages tend donc à augmenter la taille de cette section critique empêchant à minima l'utilisation en parallèle des méthodes de manipulation de l'espace virtuel `mmap`, `mremap` et `munmap`. La remise à zéro de plusieurs pages par les différents cœurs rencontre également une limite liée à la bande passante mémoire du processeur. Ce facteur peut devenir le point limitant au-delà d'un certain seuil.

5.4 Solutions existantes

Nous remarquerons qu'au-delà du coût associé, cette opération implique des transferts mémoires et une purge potentielle d'une partie du cache. Une observation similaire a déjà été faite

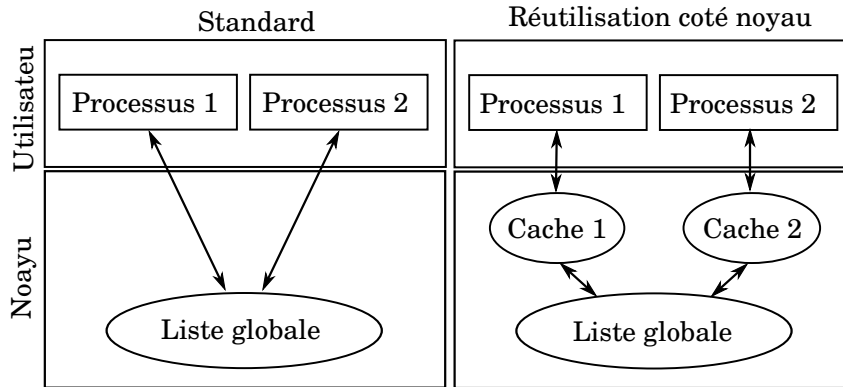


FIGURE 5.4 – Principe de réutilisation des pages au niveau noyau permettant d’éliminer le besoin de remise à zéro du contenu des pages.

en espace utilisateur dans une publication [YBF⁺11] en considérant les ramasses miettes pour des langages tels que java. Cette étude discute notamment d’un choix de libération différée. Ils analysent également l’utilisation d’accès mémoires non temporels (*non-temporal store*) permettant d’outrepasser les caches et de ne pas évincer des données utiles pour la suite. Les auteurs discutent également une technique employée dans certaines machines virtuelles Java en effectuant des remises à zéro par lots au moment de la libération plutôt que sur le chemin critique lors de l’allocation.

D’une manière similaire les développeurs de Windows[RS09] ont intégré dans leur noyau un système permettant d’effectuer la remise à zéro des pages depuis un thread système de faible priorité. Les listes de pages libres distinguent donc les pages remises à zéro des autres. Cette approche permet d’éliminer le coût de remise à zéro sur le chemin critique des fautes de pages. Cette dernière a également l’avantage, en contexte virtualisé, de générer des pages fusionnables. Ces pages peuvent alors être fusionnées par des techniques de type KSM que nous discuterons dans le prochain chapitre.

5.5 Proposition : réutilisation des pages

La section précédente vient de lister quelques modifications possibles, afin d’améliorer les performances des remises à zéro des pages, en changeant leur position d’appel ou en améliorant leur impacte sur les caches. Or, dans ces deux solutions, on maintient le coût de manipulation de la mémoire, donc, des contraintes de bande passante et de consommation d’énergie associées. De plus, dans un contexte HPC, la méthode utilisée par Windows nécessiterait certainement l’emploi d’un thread système par nœud NUMA, afin de ne pas saturer le lien entre ces derniers. Sur les nœuds 128 cœurs ceci représente 16 threads, qui tendraient à introduire du bruit système gênant pour les simulations numériques, problème déjà discuté dans l’étude [NMM⁺07]. Les simulations ont, de plus, tendance à accaparer la ressource CPU, pouvant ne pas laisser assez de temps libre pour que ces tâches de priorité faible puissent accéder suffisamment à la ressource.

Nous nous sommes donc orientés vers une approche radicalement différente, en partant du constat que la mémoire est très souvent initialisée juste après son allocation. On note de plus que la fonction *malloc*, de par sa définition, ne garantit pas de remise à zéro de la mémoire allouée. C’est de fait, ce qui arrive lorsqu’elle réutilise des segments mémoires de petite taille. On se propose donc de supprimer autant que possible le besoin de remise à zéro. Au niveau de l’OS, ces dernières sont toutefois nécessaires pour interdire toute fuite d’information entre les

entités en exécution. On remarquera toutefois qu'une page précédemment utilisée par un processus, peut sans problème être réutilisée par ce même processus, puisqu'elle ne contient que des données qui lui sont propres. Comme le montre la figure 5.4, ce fonctionnement peut-être obtenu en créant un cache de pages en espace noyau et attaché à chaque processus. Les pages peuvent alors y être capturées lors des libérations (*munmap*) et réutilisées sans remise à zéro lors des fautes de pages suivantes.

Sur le principe, cette approche revient à pousser plus loin la réutilisation que nous avons mis en place au sein de l'allocateur pour les gros segments, mais en descendant le cache au niveau du noyau. Cette modification apporte deux intérêts majeurs en comparaison du travail en espace utilisateur :

Maîtrise de la consommation : Nous avons vu que la réutilisation de gros segments au niveau utilisateur pouvait générer une surconsommation dans le cas d'applications allouant plus de mémoire virtuelle que nécessaire. Avec une approche noyau, ce cas de figure est automatiquement pris en charge en maintenant l'efficacité du mécanisme d'allocation paresseuse.

Réclamation : La méthode utilisateur tend à augmenter la consommation mémoire de l'application par rétention de segments au niveau de l'allocateur. En cas de besoin mémoire de la part d'autres processus ou de l'OS, il est toutefois impossible ou difficile de réclamer cette mémoire. Avec l'approche noyau, l'OS peut très facilement réclamer les pages en attentes dans les caches locaux des processus.

Support NUMA : Au niveau utilisateur, il n'est possible d'effectuer un contrôle NUMA qu'à la granularité d'un segment et de supposer le placement des pages de ce dernier. Ce problème est absent du côté noyau avec une gestion qui se place à la granularité de la page. Ce niveau d'abstraction profite en effet de toutes les informations de placement du thread générant la faute de page.

Réduction de contention : Cette approche permet de réduire les contentions sur les listes globales de pages pour un fonctionnement par processus. En mode processus léger, il est par contre nécessaire de travailler l'implémentation pour ne pas introduire une contention sur le cache mis en place.

5.6 Extension de la sémantique *mmap*/*munmap*

Code 5.1– Extension de la sémantique de *mmap*

```
1 //Allocation standard, segments pre-initialise a zero
2 void * ptr = mmap(NULL, SIZE, PROTECTION,
3                  MAP_ANON | MAP_PRIVATE, 0, 0);
4 //Pas de capture pour raison de securite si dasactive
5 munmap(ptr, SIZE);
6
7 //Allocation sans remise a zero force.
8 void * ptr = mmap(NULL, SIZE, PROTECTION,
9                  MAP_ANON | MAP_PRIVATE | MAP_PAGE_REUSE, 0, 0);
10 //Capture des pages lors de la liberation
11 munmap(ptr, SIZE);
```

La sémantique POSIX de *mmap* impose par sa définition le renvoi d'une mémoire initialisée à zéro, il n'est donc pas possible d'inclure notre approche dans cette interface sans une extension de sa sémantique. Par défaut, le comportement de *mmap* est maintenu. Nous avons donc ajouté un drapeau permettant d'informer *mmap* que la mémoire allouée n'a pas besoin d'être initialisée. L'expression de cette information par l'appelant permet ainsi de faire cohabiter les deux

sémantiques. Cette nouvelle expression peut être exploitée dans les fonctions *malloc* et *realloc* qui au contraire de *calloc* n'ont pas à assurer une mise à zéro de la mémoire.

5.7 Détails d'implémentation

5.7.1 Modification de Linux

On peut dans un premier temps se demander s'il n'est pas possible d'implémenter la modification proposée sous la forme d'un module noyau. Il convient toutefois de remarquer qu'un rapide test montre que le coût des fautes de pages est doublé lorsque l'on tente d'exploiter ce type d'approche. Or, pour les pages standards, nous attendons un gain de l'ordre de 40% correspondant à la fraction de temps associée à la fonction *clear_page*. Pour les pages standards, il nous faut donc évaluer l'implémentation au niveau du code noyau principal. Concernant les grosses pages, il n'est pour l'instant pas possible de les manipuler depuis l'interface offerte aux modules, il faudrait donc recourir à une approche de type VHP (Virtual Huge Pages) développée lors du stage de fin d'étude ayant conduit à cette thèse[SM]. Il a donc été décidé de modifier directement le noyau pour évaluer le réel intérêt de l'approche.

Les modifications ont été réalisées sur les versions 2.6.32 modifiée par Redhat et 2.6.36 officielle. Au niveau du noyau, nous avons dans un premier temps défini une nouvelle structure de gestion des pages en attentes. Cette structure est alors injectée dans la structure de description de l'espace mémoire virtuel de chaque processus (*mm_struct*). Elle suit ainsi le même cycle de création/libération que le processus associé. Lors d'un *fork*, le nouveau processus réinitialise sa structure de sorte qu'il ne partage aucune page avec le précédent. Les implémentations noyau des appels *madvise* et *mmap* sont alors modifiées pour prendre en compte l'activation du cache local en fonction du drapeau décrit précédemment (*MAP_PAGE_REUSE*).

Les modifications délicates sont alors mises en place pour dérouter les chemins d'appels standards de faute de page et de libération. Lors des fautes de pages, le chemin standard est dérouté au niveau de la fonction *do_anonymous_page* dédiée aux fautes de pages anonymes qui nous intéressent. Si le segment touché dispose du drapeau, une page est cherchée dans le cache local. Si le cache est vide ou que le drapeau de réutilisation n'est pas actif, le noyau poursuit le chemin standard et renverra une page remise à zéro comme à l'origine. De manière identique, les grosses pages sont supportées en modifiant la fonction spécifique *do_huge_pmd_anonymous_page*.

Les problèmes principaux ont été rencontrés sur le choix du point de capture qui idéalement doit se situer au niveau des fonctions *tlb_remove_page* ou *zap_pte_range*. Ce choix s'est toutefois avéré difficilement praticable du fait de problèmes de cohérence avec les mécanismes d'invalidation des TLB. La page ne doit en effet pas être enregistrée dans notre cache tant que le processeur la croit projetée dans l'espace du processus. La version finale modifie donc la fonction *zap_pte_range* et *zap_huge_pmd* pour activer un drapeau sur la page considérée et la marquer en attente de capture. L'opération de capture prend alors place plus en aval de la pile d'appels au niveau des fonctions *free_hot_cold_page*, au point où elle s'insère normalement dans des listes pour réutilisation rapide dans le noyau d'origine. Cette approche bien qu'élégante n'avait pas été retenue initialement, car elle consomme un bit de drapeau supplémentaire sur les pages qui en utilise déjà un grand nombre. Une capture à ce point nécessite également plus d'efforts de validation en terme de sécurité pour assurer une capture limitée aux pages désirées. Certaines subtilités sont également à prendre en compte quant à la mise à jour des compteurs d'utilisation des pages, afin de ne pas aboutir à des valeurs négatives. Ces problématiques techniques ont occupé l'essentiel du temps de travail sur cette modification.

On remarque qu'en pratique les modifications à l'intérieur du noyau sont relativement limitées avec un différentiel représentant au total 720 lignes dont la moitié sont dédiées à l'implé-

mentation des fonctions de manipulation de la structure elle-même. Il s'agit donc d'une modification qui peut raisonnablement être vérifiée et portée sur les différentes versions du noyau.

5.7.2 Capture des zones sans réutilisation ?

On remarquera que le cache capture les pages libérées par l'application, et renvoie ces pages à l'application lors des allocations. La taille du cache est donc bornée par la taille mémoire maximale utilisée par le processus au cours de son exécution. Toutes les allocations suivant ce pic mémoire n'induisent donc plus qu'un jeu de vases communicants entre le cache en espace noyau et l'espace virtuel du processus. Deux approches peuvent être retenues à ce niveau. Le noyau peut capturer et réutiliser les pages du cache pour toutes les allocations, avec remise à zéro si le drapeau n'est pas actif pour la zone considérée. La borne mémoire est ainsi assurée de manière passive. En terme de sécurité, on introduit toutefois un changement puisqu'une bibliothèque de cryptographie pourrait voir ses clés réutilisées à une autre adresse du processus. Ce comportement est celui par défaut de la fonction *free*, mais nous l'étendons ici à *munmap*.

Ce point peut être résolu en nettoyant le contenu de la page au moment de la capture pour les zones sans réutilisation, ou bien, en ne capturant que les pages des zones avec ce support. Nous avons retenu la dernière méthode dans notre prototype. Il faut toutefois remarquer que dans ce mode, un logiciel malicieux pourrait faire grossir sans limite le cache en usant de l'appel *madvise* pour y libérer les pages, mais ne jamais les ré-utiliser en effectuant plus d'allocations avec le drapeau activé. Ce problème se résout toutefois de manière automatique, si l'on dispose d'une intégration dans le système de réclamation de pages, comme discuté dans la suite, ou en supprimant l'utilisation de *madvise* pour ce drapeau.

Le choix d'une capture limitée aux zones marquées permet également de n'activer le cache que lorsqu'il a une réelle utilité. On évite ainsi d'impacter le système dans son fonctionnement standard.

5.7.3 Limite de consommation et réclamation

Comme nous venons de la voir, la taille du cache est limitée à la taille mémoire maximale du processus de manière mécanique, si les bons choix sont faits au niveau de la sémantique. Dans le cas contraire, il peut être intéressant d'introduire un mécanisme de contrôle actif de la taille du cache. Il faut toutefois remarquer qu'il est alors nécessaire d'introduire un compteur permettant de suivre en temps réel la taille de ce dernier. Ce compteur ajouterait de la contention sur la structure de gestion. En contexte HPC, nous considérons habituellement un unique gros programme, nous avons donc décidé de ne pas recourir à cette approche dans notre prototype.

En cas de saturation mémoire, l'OS peut avoir besoin de réclamer des pages. Dans ce contexte, une fonction de réclamation est amorcée par le noyau et cherche à vider certains caches, notamment les caches disques, avant de poursuivre avec les fonctions de pagination disque (*swap*). Il serait intéressant de s'intégrer dans ces méthodes pour vider en priorité le nouveau cache que nous avons introduit. Ce dernier n'a en effet pas d'incidence directe sur les processus en exécution et n'implique pas de potentiels accès disques supplémentaires. Afin de ne pas pénaliser un unique processus, il serait alors intéressant de réclamer une fraction de page en choisissant le processus ayant le plus gros cache. Il est aussi possible d'utiliser un tourniquet pour ne pas pénaliser systématiquement le même. En cas de nécessité d'effectuer de la pagination disque, il semble judicieux de rendre inactif les caches des processus jusqu'à un retour à une situation normale.

Remarquons qu'une intégration dans les mécanismes de réclamation permet automatiquement de régler les problèmes de limitation de la taille du cache. Si ce dernier tend à grossir de

trop, l'OS viendra naturellement ponctionner des pages dans ce dernier, limitant de fait sa taille. Ceci justifie notre choix de ne pas intégrer de compteurs spécifiques nécessaires à un contrôle actif. Ce choix est fait dans un contexte HPC, il peut ne pas être valide dans un contexte général avec divers processus de consommation équivalente en cas de restriction mémoire. Le support de la réclamation n'est pas indispensable à une première évaluation d'intérêt. Il a donc été laissé pour travaux futurs bien qu'il a été vérifié qu'une telle modification est possible dans la structure actuelle de Linux.

5.7.4 Intégration dans les allocateurs

L'allocateur en espace utilisateur est le point d'entrée principale pour exploiter cette nouvelle sémantique dans les applications. Une grande majorité des allocateurs actuels base leur support des gros segments sur l'appel *mmap*. Le drapeau ajouté par notre proposition peut être activé pour les fonctions *malloc* et *realloc* si l'on assure que *calloc* renvoie des espaces mis à zéro, afin de maintenir la sémantique officielle. Ce type d'intégration peut être aisé si l'allocateur ne repose pas sur des remises à zéro implicites de la mémoire, dans la gestion de ses structures internes. Dans ces conditions, l'intégration peut se faire en propageant un booléen jusqu'aux appels à *mmap*, afin de l'informer du besoin de remise à zéro comme décrit en section 4.8.

Cette intégration a donc été faite dans l'allocateur de MPC développé par nos soins. Nous avons également modifié Jemalloc qui introduisait déjà nativement la propagation de ce type d'information. Cet allocateur est un bon candidat expérimental du fait de sa politique agressive de libération stressant l'OS.

5.8 Résultats expérimentaux

Cette section fournit les résultats expérimentaux préliminaires obtenus avec notre premier prototype noyau et montre l'intérêt de la méthode sur les nœuds 12 cœurs du cluster Cassard. Ces résultats restent toutefois à évaluer sur les nœuds 128 cœurs des calculateurs Tera 100 et Curie.

5.8.1 Micro-benchmark

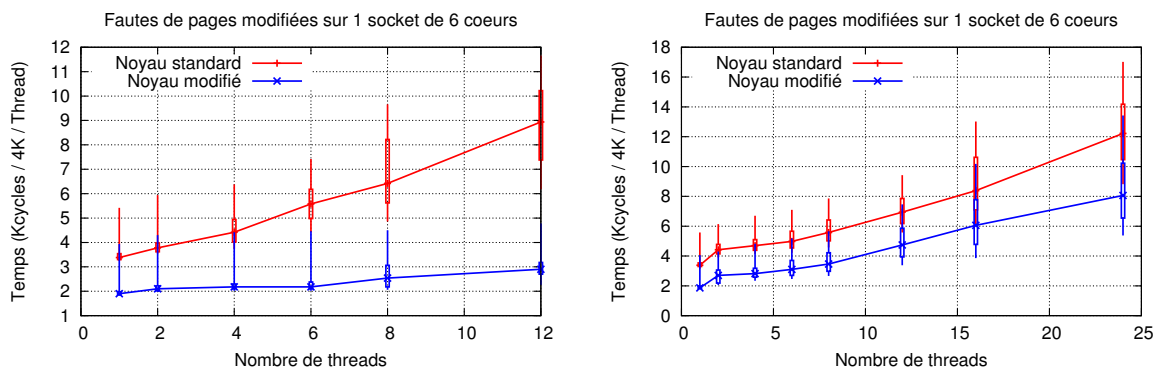


FIGURE 5.5 – Évaluation de l'impact de notre proposition sur micro-benchmark en exploitant un ou deux sockets des nœuds Cassard.

Le graphique 5.5 donne les résultats d'application du micro-benchmark précédent sur les fautes de pages. Les mesures montrent une nette amélioration des performances séquentielles

avec un temps de faute de page passant de 3400 à 1900 cycles soit un gain de 45% correspondant à l'ordre de grandeur attendu. L'amélioration est également notable en fonctionnant sur un unique processeur (6 cœurs hyperthreadés, donc 12 processus légers) avec un temps passant de 8950 à 2900 cycles, soit un gain de 66%. Cette amélioration de passage à l'échelle peut s'expliquer en considérant une réduction de la saturation des accès mémoires et une réduction de la taille de la section critique entre les verrous en lecture. Les effets NUMA deviennent toutefois dominants, dès lors, que l'on exploite l'ensemble de la machine avec 24 threads. Les gains se limitent alors à la réduction du coût constant de remise à zéro, soit, 33%. La réduction des contentions sur les verrous en mode NUMA reste donc un problème prioritaire pour les pages de 4 Ko.

Les résultats obtenus avec les grosses pages sont fournis dans la figure 5.6. Dans ce mode, les gains obtenus sur la remise à zéro deviennent prépondérants et permettent d'obtenir une réduction des coûts par un facteur 57. Le coût relatif est donc ramené à 35 cycles par page équivalente de 4 Ko. Remarquons que la réduction de bande passante et de taille de la section critique permettent également une nette amélioration de l'extensibilité. Le ralentissement observé lors du passage de 1 à 24 threads était d'un facteur 3.1, il devient 1.4 avec notre patch.

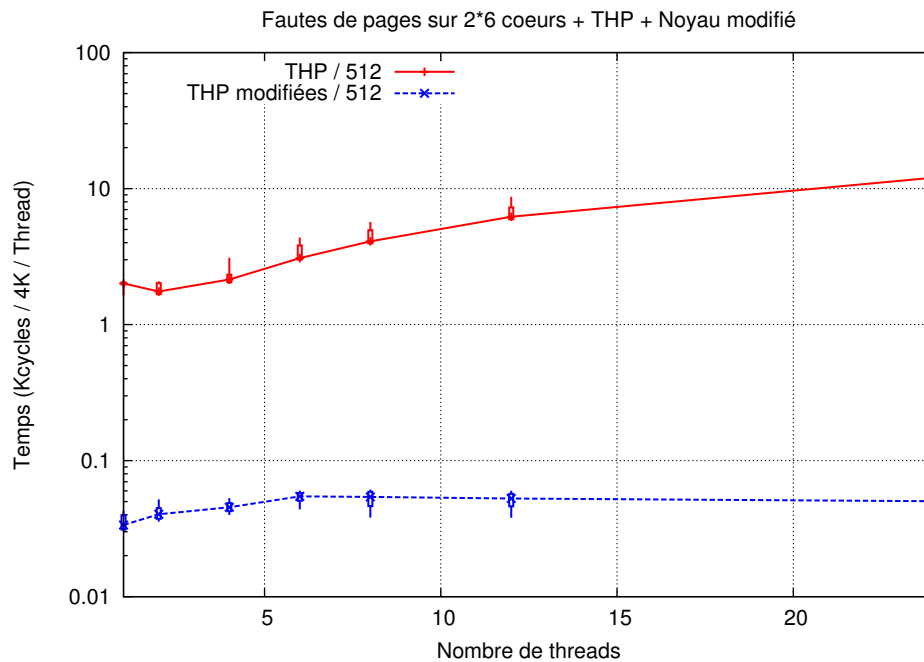


FIGURE 5.6 – Évaluation de l'impact de notre proposition sur micro-benchmark en exploitant des grosses pages sur les nœuds Cassard.

5.8.2 Application HydroBench

HydroBench[dV] est un benchmark d'hydrodynamique 2D basé sur une programmation hybride MPI + OpenMP et disponible sur Github. Cette application a été utilisée en mode OpenMP sur les 12 cœurs du calculateur Cassard. Les anciennes versions de ce benchmark souffraient d'allocations/libérations répétitives de grands segments mémoires à chaque pas de temps. Cette version stressait la chaîne de gestion mémoire de manière importante. C'est donc cette version que nous avons utilisée. Remarquons, que les nouvelles versions éliminent ces motifs d'allocations répétitifs. Sur cette application, les différents allocateurs offrent une consommation mémoire quasi identique du fait d'un motif d'allocation très simple, basé sur quelques grands

Pages standards. (S)							
	Application	Noyau	Allocateur	Total	Utilisateur	Système	GFlops
1	Originale	Original	Glibc	1 :29	543.3	30.7	1.7
2	Originale	Original	MPC	1 :28	532.9	31.5	1.7
3	Originale	Modifié	MPC	1 :19	534.9	19.7	1.7
4	Originale	Original	MPC Perf	0 :59	528.6	0.5	2.6
5	Modifiée	Original	Glibc	0 :43	475.0	0.4	3.6

Grosses pages (THP). (G)							
	Application	Noyau	Allocateur	Total	Utilisateur	Système	GFlops
1	Originale	Originale	Glibc	1 :13	533.3	18.8	2.1
2	Originale	Originale	MPC	1 :18	550.7	17.8	2.0
3	Originale	Modifié	MPC	1 :11	557.0	7.0	2.2
4	Originale	Originale	MPC Perf	1 :05	566.3	1.0	2.4
5	Modifiée	Originale	Glibc	0 :50	539.2	0.4	3.5

TABLE 5.1 – Mesure avec la version OpenMP du benchmark HydroBench sur les 12 cœurs du calculateur Cassard. La version Perf. (performance) de notre allocateur correspond à l'activation d'un politique de réutilisation maintenant les gros segments de l'application en espace utilisateur pour réutilisation rapide.

tableaux.

La table 5.1 montre des gains de 12% sur le temps d'exécution et 37% sur le temps système en employant notre modification de l'OS (S2,S3). Il est bien sûr possible d'obtenir de meilleurs gains (44%) en travaillant au niveau de l'allocateur mémoire grâce à une réduction importante du nombre de fautes (S2,S4). Les grosses pages améliorent par elles-mêmes, les performances de 12% (S1,G1). Notre modification du noyau apporte, elle, une division par 2.5 du temps système (G2,G3) rendant le cache en espace utilisateur moins efficace (G3,G4). Des gains plus importants, jusqu'à 52% sont logiquement obtenus en modifiant les schémas d'allocation au niveau du code de l'application (S4,S5 et G4,G5). Nous remarquerons toutefois, que ce type de correction n'est pas forcément possible dans une application plus complexe, limitée par sa consommation mémoire.

5.8.3 Application Hera

Nous pouvons également évaluer l'amélioration apportée sur l'application Hera exploitée de la même manière qu'en section 4.15.3. La table 5.2 peut être comparée aux résultats d'origine de la table 4.2 (sur le calculateur Cassard) de la section 4.15.3. Ces résultats montrent que, dans ce cas particulier, les pages de 2 Mo tendent à être trop grandes. Ces grosses pages augmentent la consommation mémoire de l'ensemble des allocateurs à l'exception de TCMalloc qui maintient une consommation fixe. Cet effet d'accroissement peut s'expliquer par la présence de zones purement virtuelles qui, dans le cas des grosses pages, deviennent physiquement allouées. D'autre part, du fait des problèmes de passage à l'échelle et du surcoût de remise à zéro, les grosses pages dégradent les performances de l'application.

Les résultats obtenus avec notre modification du noyau sont donnés dans la table 5.3 pour les nœuds Cassard. Les tests sont réalisés avec des versions modifiées des allocateurs Jemalloc et MPC. Avec les pages standards, la modification du noyau permet d'obtenir un gain global de performance de 2% (S2,S3 et S4,S5). Le temps système, lui, est impacté par une réduction de 33%, correspondant aux gains observés sur le micro-benchmark précédent. Sur les grosses

Grosses pages sur l'application Hera.					
	Allocateur	Total (s)	Utilisateur (s)	Système (s)	Mémoire (Go)
1	MPC-NUMA	137.89	135.13	1.86	6.2
2	MPC-UMA	147.15	144.38	1.97	6.2
3	MPC-Lowmem	196.51	140.39	28.24	3.9
4	jemalloc	144.72	129.62	14.66	2.5
5	std	149.77	130.08	12.92	4.5
6	Tcmalloc	150.13	149.03	0.51	6.5

TABLE 5.2 – Temps d'exécution de l'application Hera sur les nœuds 12 cœurs de Cassard. Les temps sont donnés par thread.

pages, notre modification induit une amélioration des performances sur le temps total de 30% et 5% pour les deux allocateurs ayant une faible consommation (H2,H3 et H4,H5). Les temps systèmes sont eux réduits respectivement d'un facteur 9.7 et 2.4. Dans le cadre de MPC, cette modification permet de rendre le profil à basse consommation aussi efficace que le mode NUMA (H1,H3). En d'autres termes, cette modification du noyau permet de compenser le surcoût induit par les libérations agressives de mémoire.

Noyau modifié et pages standards de 4 Ko (S)						
	Allocateur	Noyau	Total	Utilisateur	Système	Mémoire (go)
1	MPC-NUMA	Original	135.14	132.63	1.79	4.3
2	MPC-Economique	Original	161.58	131.00	15.97	2.0
3	MPC-Economique	Modifié	157.62	132.70	10.60	2.0
4	Jemalloc	Original	143.05	128.07	14.53	1.9
5	Jemalloc	Modifié	140.65	130.80	9.32	3.2
Noyau modifié et grosses pages de 2 Mo (H)						
	Allocateur	Noyau	Total	Utilisateur	Système	Mémoire (Go)
1	MPC-NUMA	Original	137.89	135.13	1.86	6.2
2	MPC-Economique	Original	196.51	140.39	28.24	3.9
3	MPC-Economique	Modifié	138.77	131.70	2.90	3.8
4	Jemalloc	Original	144.72	129.62	14.66	2.5
5	Jemalloc	Modifié	138.47	130.44	6.40	3.2

TABLE 5.3 – Benchmark de notre modification noyau avec l'application Hera sur les nœuds 12 cœurs du calculateur Cassard. Hera est exécuté avec 12 processus légers en un seul processus. Les temps utilisateurs et systèmes sont donnés en secondes par thread.

5.9 Conclusion

Ce chapitre reprend le problème de performance observé sur les architectures parallèles modernes et s'intéresse aux surcoûts introduits par l'OS lui-même. Dans le cadre de cette étude, nous avons montré que les fautes de pages étaient impactées par un manque de passage à l'échelle. En complément des travaux d'autres équipes, nous nous sommes intéressés à la fraction de surcoût (40%) induite par les mécanismes de remise à zéro des pages renvoyées par l'OS.

Nous avons vu que certains tentent au niveau allocateur d'optimiser ces remises à zéro en utilisant des écritures intemporelles. Du côté OS, nous avons rappelé que des OS comme Win-

dows pouvaient déplacer ces dernières afin de les sortir du chemin critique d'allocation. Au contraire de ces deux méthodes nous avons choisi de chercher à éliminer le besoin de recourir à ces remises à zéro. On peut ainsi espérer un gain de bande passante, d'utilisation du processeur et probablement d'énergie. Pour ce faire, il a été remarqué qu'une réutilisation locale au processus pouvait permettre de maintenir les contraintes de sécurité, tout en éliminant ces nettoyages mémoires. Ce comportement a été introduit en étendant la sémantique associée à *mmap*. Nous permettons ainsi à l'appelant de notifier l'OS du besoin ou non de mémoire pré-initialisé à zéro.

Les micro-benchmarks montrent une amélioration de 40% sur les fautes de pages standards séquentielles et une amélioration en parallèle en l'absence d'effet NUMA. Les observations montrent toutefois que ces effets sont à prendre en compte et nécessitent toujours des améliorations au niveau des verrous de l'OS. Les gains observés restent toutefois de l'ordre de 32% sur 24 threads. L'approche retenue a également donné des résultats intéressants et novateurs pour les grosses pages, proportionnellement plus pénalisées par ces remises à zéro. Grâce à notre approche, ces dernières peuvent également être utilisées avec la motivation d'augmenter les performances d'allocation, les gains représentant une réduction des coûts par un facteur 57 en terme de temps système.

Sur des simulations numériques, cette méthode a montré des améliorations de 1% (Hera) à 12% (HydroBench) sur les temps d'exécution globaux avec pages standards, mais dans les deux cas, une réduction de 30% du temps système associé. Les grosses pages modifiées ont notamment montré leur intérêt sur l'application Hera en permettant une réduction du temps d'exécution de 30%. Ce gain est comparable à la méthode de rétention mémoire en espace utilisateur. Cette méthode permet donc de compenser les libérations agressives de mémoire.

Ces résultats sont donnés sur un calculateur 12 cœurs moins impactés par l'OS que les nœuds larges à 128 cœurs. Il faut donc s'interroger sur la projection de des résultats sur ces gros nœuds qui devrait montrer un plus grand intérêt pour cette technique. Ces tests n'ont pour l'instant pas pu être réalisés faute d'accès en temps voulu à ce type de machine avec droits administrateurs.

Chapitre 6

Étude complémentaire sur le problème de consommation : KSM

6.1 Introduction

De nos jours, la quantité de mémoire disponible n'augmente pas aussi rapidement que le nombre de cœurs. Il devient donc de plus en plus important de réduire au maximum la consommation mémoire des applications. Les approches décrites précédemment ont eu pour objet principal la prise en compte de la problématique de l'allocateur mémoire notamment en terme de performances. Dans le chapitre précédent, nous avons cherché à améliorer les performances de l'OS pour permettre une libération plus agressive de la mémoire et ainsi améliorer le ratio performance/consommation mémoire. En complément, nous allons ici évaluer une technique actuellement offerte par le noyau Linux pour réduire l'empreinte mémoire des applications.

Dans les simulations, il arrive que certaines données soient dupliquées. Dans un contexte MPI, c'est par exemple le cas pour les tables de constantes physiques si l'on utilise plusieurs processus par nœud. On peut également citer les maillages contenant des données très répétitives (beaucoup de 0...). Ces problèmes de données dupliquées sont également rencontrés dans le cadre des offres de serveurs mutualisés. Dans ce cadre, les serveurs doivent offrir une grande quantité de mémoire pour gérer le fonctionnement simultané des différentes machines virtuelles (VM). Or, il est fréquent que le même système d'exploitation soit installé sur chacune des VM. Il en résulte une duplication d'un grand nombre de données en mémoire (codes exécutables, ressources diverses, images...).

Dans ce contexte, les développeurs de KVM¹[Hab08] ont mis au point un composant noyau pour Linux : KSM (*Kernel Samepage Merging*)[AEW09] dont l'objectif est de fusionner les pages mémoires ayant des contenus identiques. Il est ainsi possible de réduire la mémoire consommée par les machines virtuelles gérées par KVM. Ces travaux sont en partie associés, à A. Arcangeli à qui l'on doit le support actuel des grosses pages dans Linux. Ce module est présent dans la branche officielle du noyau depuis la version 2.6.32. Des mécanismes similaires (TPS : *Transparent Page Sharing*) existent également dans les solutions de virtualisation propriétaires tels que VMWare[Wal02].

Dans ce document, nous allons décrire succinctement le fonctionnement de KSM et montrer qu'il peut être utilisé pour réduire la consommation mémoire de certaines simulations numériques. On notera que l'équipe de développement de KSM a collaboré avec le CERN autour d'applications pouvant présenter des problématiques similaires à celles développées dans le cadre de

1. KVM : *Kernel Virtual Machine* est un module du noyau Linux offrant des mécanismes de mise en place de machines virtuelles.

nos activités[AEW09]. Leur analyse s'intéresse surtout au problème de duplication des modèles de détecteurs équivalents aux tables de constantes de nos programmes MPI. Au contraire, nous évaluerons ici l'application de la méthode sur des données plus dynamiques pour identifier les redondances au sein du maillage actif. Dans un premier temps, nous rappellerons les principes de base concernant la gestion des mémoires partagées. Puis nous décrirons le fonctionnement de KSM. Enfin, nous donnerons quelques résultats obtenus sur l'application Hera en montrant qu'il est possible avec KSM d'éviter de recourir à de trop coûteuses indirections. Nous exposerons à l'occasion certaines limites de l'implémentation actuelle de KSM (version datée de début 2011).

6.2 Mémoire partagée

Rappelons que la gestion de la mémoire est basée sur la notion fondamentale de *mémoire virtuelle* mise en place par les mécanismes de *pagination* et décrite en section 2.2.3. Notons toutefois que ces mécanismes n'imposent aucune restriction quant au nombre de *pages virtuelles* pointant vers les *pages physiques*. Il devient alors possible de mettre en place un système de *mémoire partagée* en faisant pointer plusieurs *pages virtuelles* vers les mêmes *pages physiques*. L'empreinte mémoire d'un programme peut donc être réduite en évitant de dupliquer physiquement des pages ayant le même contenu.

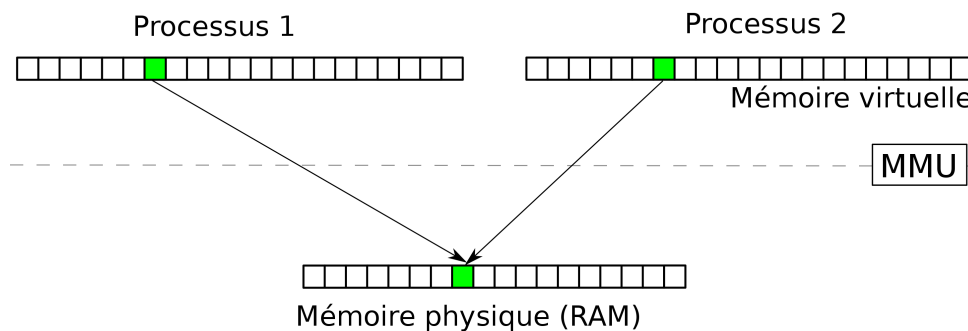


FIGURE 6.1 – Exemple de partage de pages physiques entre deux instances d'une application par le mécanisme de pagination. Ici par exemple, le segment de code de deux instances d'une même application.

La figure 6.1 illustre un tel partage entre deux processus utilisant le même exécutable. Ce type de partage reste totalement transparent pour l'application, car géré entièrement par le matériel et le système d'exploitation. Bien qu'ils reposent sur les mêmes mécanismes, on peut distinguer deux cas particuliers pour ces partages. Ils correspondent essentiellement à deux manières d'interagir avec l'OS pour demander leur mise en place :

Mémoire partagée (SHM²) : l'objectif est de partager de manière définitive un segment mémoire. Dans ce cas, les modifications apportées par l'un des utilisateurs du segment seront immédiatement répercutées sur les autres puisque le même segment physique est utilisé. Cette approche peut, par exemple, être utilisée pour faire communiquer des processus au sein d'un même nœud.

Copie sur écriture (COW³) : la copie sur écriture a pour but de partager temporairement des segments mémoires en lecture. Dans ce cas, lors de la première écriture, l'OS va dupliquer la page afin que la modification n'impacte pas les autres utilisateurs du segment. Ce cas de figure est illustré sur la figure 6.2. Cette approche est utilisée par le système pour éviter de dupliquer les codes exécutables et bibliothèques dynamiques. C'est également sur la base de ce mécanisme que KSM se construit.

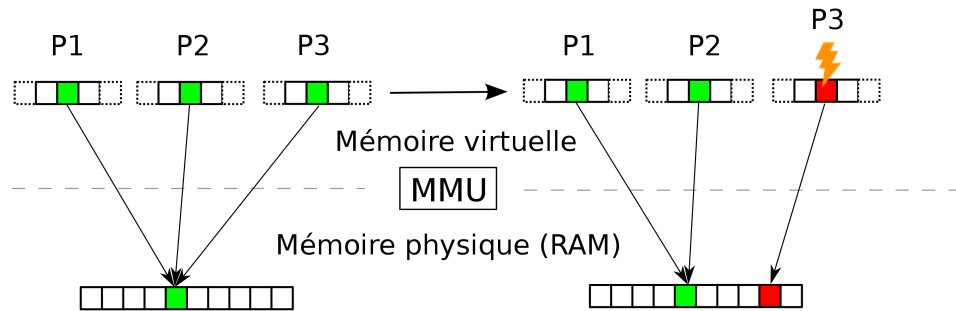


FIGURE 6.2 – Exemple de copie sur écriture (COW) entre trois processus P1, P2 et P3.

6.3 Principe de KSM

Dans cette section, nous allons décrire succinctement la manière dont KSM fonctionne afin d'évaluer ce que cette approche peut apporter dans le cadre des simulations numériques.

6.3.1 L'idée maîtresse

Nous avons vu que le mécanisme de *copie sur écriture* permettait d'éviter la duplication de certaines données jusqu'à leur première modification. Actuellement, ce mécanisme est essentiellement utilisé en phase d'initialisation lorsque l'on duplique des segments mémoires. Par exemple lors du *fork*⁴. La mémoire va ensuite tendre à augmenter au gré des modifications, mais ne sera jamais réduite même si les données écrites sont identiques. C'est à ce point qu'intervient KSM.

L'objectif de KSM est de fournir un mécanisme actif permettant de fusionner dynamiquement en mode *copie sur écriture* un groupe de pages identiques. Ce mécanisme fonctionne à l'intérieur d'un processus ou entre processus, et ce, de manière transparente pour l'application.

6.3.2 Fonctionnement interne

KSM utilise une méthode active pour détecter et fusionner les pages similaires. Pour ce faire, un démon noyau (*ksmd*) se charge d'analyser régulièrement les pages pour trouver les copies et les fusionner en mode *copie sur écriture*. KSM concentre son activité autour d'un sous-ensemble de la mémoire. Une surveillance complète de la mémoire serait en effet trop coûteuse et peu rentable. KSM préfère donc limiter ses efforts autour des pages identifiées par le développeur comme contenant potentiellement des redondances.

Pour trouver les pages identiques, KSM procède à une comparaison bit à bit du contenu des pages (avec un *memcmp*) en faisant une recherche dans un arbre équilibré. Cette méthode est potentiellement plus coûteuse comparée à l'utilisation d'une méthode de hachage, mais permet de ne pas entrer en conflit avec un brevet de VMWare et d'éviter tout risque de collision. Lorsque deux pages similaires sont identifiées, la table des pages est mise à jour pour ne garder qu'une des deux pages physiques et libérer la seconde. Si par la suite l'application modifie le contenu de la page, le mécanisme de *copie sur écriture* va dupliquer automatiquement la page.

4. Le *fork* est une opération permettant de créer une copie d'un processus. Lors de cette copie, la mémoire des deux instances est entièrement initialisée en mode copie sur écriture.

6.3.3 Marquage des pages

KSM impose le marquage des zones mémoires pouvant contenir des redondances. Au niveau de l'application, ce marquage est réalisé via l'appel *madvise()* avec le flag *MADV_MERGEABLE*. À noter que KSM travaille au niveau des pages, il lui faut donc trouver des similarités sur des tailles et adresses multiples de la taille d'une page (4 Ko en général).

Code 6.1– Exemple d'utilisation de KSM.

```
1 void * ptr = mmap(NULL, size, PROT_WRITE | PROT_READ, MAP_ANONYMOUS | MAP_PRIVATE
    , 0, 0);
2 madvise(ptr, size, MADV_MERGEABLE);
```

La contrainte précédente rend délicat le marquage des segments alloués au préalable par la fonction *malloc* car leur alignement ne peut être garanti. Il faut donc passer directement par un appel à *mmap*, ou disposer d'un allocateur modifié se chargeant lui-même d'effectuer le marquage. La seconde méthode est très certainement préférable, car elle n'impacte pas directement le code de l'application et maintient l'isolation des responsabilités. Au niveau de l'allocateur, il suffit d'intercepter les appels à *mmap* et *brk* afin de marquer la mémoire allouée. C'est par exemple trivial avec l'allocateur actuel du framework MPC (patch d'une ligne). En contrepartie, cette méthode risque de marquer trop d'éléments inintéressants. Nous noterons toutefois que pour les simulations numériques, la majeure partie de la mémoire est habituellement dédié au maillage que l'on cherche à réduire dans nos tests.

6.3.4 Activation et configuration

KSM se paramètre à l'aide des fichiers de configuration disponibles dans */sys/kernel/mm/ksm/*, on y trouve notamment des statistiques et les paramètres :

Nom	Signification
run	Permet d'activer (1) et désactiver (0) le démon <i>ksmd</i> .
pages_to_scan	Nombre de pages à scanner à chaque réveil du démon.
sleep_millisecs	Délais entre deux scans.

TABLE 6.1 – Paramètres de contrôle de KSM accessibles dans */sys/kernel/mm/ksm*

L'agressivité du démon se règle à l'aide des deux paramètres *pages_to_scan* et *sleep_millisecs*. Ces accès nécessitent les droits administrateurs par défaut. On remarquera que Red-Hat, Fedora et Debian proposent un démon (*ksmtuned*) en espace utilisateur pour activer ou désactiver automatiquement KSM en fonction de la charge mémoire. Cet outil adapte également l'agressivité de KSM en fonction du nombre de pages enregistrées auprès de KSM. La version actuelle applique une simple croissance ou décroissance par paliers entre deux bornes définies dans la configuration.

6.4 Test sur Hera

Hera est une plateforme de simulation multi-physique multi-matériau opérant sur maillages de type AMR (Adaptive Mesh Refinement). Les grandes classes de solveurs disponibles accèdent aux données AMR multi-matériau par une représentation de type $\rho[imat][nc]$, où *imat* est l'indice de matériau et *nc* le numéro de maille AMR. En pratique, il est extrêmement rare qu'une maille contienne l'ensemble des matériaux. La plateforme gère donc deux implémentations de tableaux AMR multi-matériaux :

- une implémentation *adressage directe* avec dimensionnement au nombre de mailles AMR multiplié par le nombre total de matériaux.
- une implémentation *chunk* assurant une compression par blocs des tableaux contenant des zéros.

Le langage C++ utilisé pour la plateforme Hera permet un codage *unique* des solveurs pour ces deux représentations mémoires (surcharge d'opérateurs et templates). Le choix de l'implémentation peut changer dynamiquement en cours de calcul, selon le sous-domaine considéré, en fonction du nombre de mailles AMR et du nombre courant de matériaux présent dans le sous-domaine (passage du mode *chunk* au mode *adressage direct* et vice-versa).

Cette approche permet de réduire la consommation mémoire de l'application, mais ceci au prix d'une complexification notable de la plateforme (implémentation des *chunk*) et des performances. Outre le coût de l'indirection, le compilateur ne peut en effet plus appliquer les optimisations habituellement valides pour des parcours de tableaux, même si l'implémentation est réalisée par une surcharge de l'opérateur crochet du C++. Dans ce cas, il est en effet impossible de prédire si deux éléments consécutifs sont contigus en mémoire. Il est donc par exemple impossible de vectoriser les opérations. L'impact de cette technique est un gain de l'ordre de 25% en mémoire, mais jusqu'à un doublement du temps d'exécution sur les sous-domaines où un grand nombre de matériaux sont présents. Le débogage d'un tel système peut aussi être une source de problème.

KSM a été testé sur cette application afin d'évaluer les gains qu'il peut apporter en comparaison des *chunks*. Le maillage ayant beaucoup de zones similaires, on peut supposer que KSM parviendra à fusionner les pages et à compenser la désactivation de ces indirections. Ceci permettrait au code de fonctionner de manière plus efficace, plus propre, tout en réalisant des gains mémoires.

6.4.1 Méthode de test

Pour tester KSM sur Hera, nous avons utilisé l'allocateur de MPC, modifié de manière à intercepter les appels à *mmap* et marquer les pages associées. On marque ainsi l'essentiel de la mémoire dynamique de l'application auprès de KSM. L'application Hera a également été modifiée pour intercepter les quelques appels directs à *mmap* qu'elle réalise. Concernant les mesures de consommation mémoire, la version testée de KSM ne met pas à jour la taille de la mémoire résidente du processus. Nous avons donc dû utiliser la mémoire libre sur le système pour évaluer la consommation en mémoire physique de l'application. Ce problème est discuté plus en détail dans la section donnant les limites actuelles de KSM.

6.4.2 Résultats

La méthode a été appliquée à Hera en considérant un problème Air-Alu à 6 matériaux en déclarant trois matériaux virtuels pour l'air et pour l'aluminium. Dans un premier temps, les tests ont été réalisés en fixant l'agressivité de KSM avec *pages_to_scan* = 2000 et *sleep_millisecs* = 20. Ces valeurs correspondent plus ou moins au choix optimal pour Hera compte tenu de nos évaluations de l'espace des paramètres. Sur la figure 6.3 on observe clairement les gains apportés par KSM : 16% lorsque les indirections sont désactivées et 12% lorsqu'elles sont actives. On remarque également que l'impact sur les performances est négligeable dans ce cas-ci. En utilisant les 8 threads, on observe des gains mémoires du même ordre. On remarquera que l'on disposait de 8 hyper-threads sur lesquels KSM pouvait tourner seul.

On remarque sur la figure 6.4 que les gains mémoires ne sont pas immédiats, KSM fusionne petit à petit le contenu de la mémoire. Il en résulte une limite. En effet, le pic mémoire n'est pas

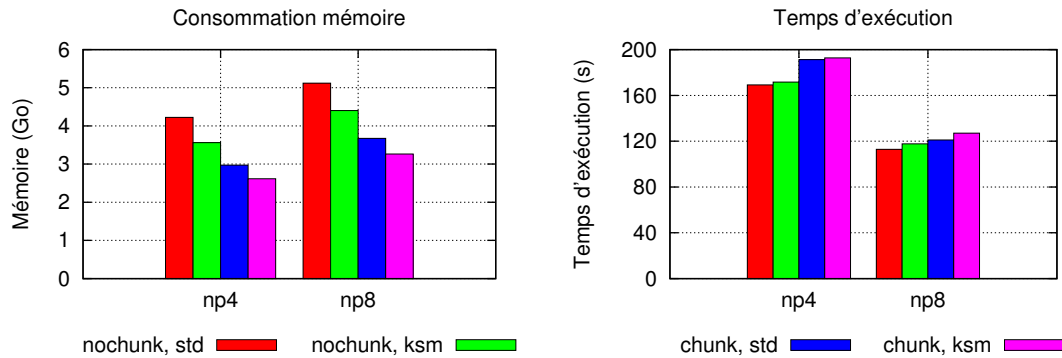


FIGURE 6.3 – Gains observés avec KSM sur une exécution d'Héra utilisant 4 ou 8 processus MPI sur 8 cœurs disponibles avec 2.3 millions de mailles. La mémoire utilisée est donnée sous la forme d'une moyenne sur l'ensemble de l'exécution.

réduit en proportion du gain moyen. Ici, KSM n'analyse donc pas les pages assez rapidement. On peut rappeler à l'occasion que KSM ne dispose que d'un unique thread noyau pour scanner la mémoire, ce qui est très certainement beaucoup trop faible pour réaliser des fusions très agressives avec des applications utilisant plusieurs cœurs.

La figure 6.5 montre un test réalisé avec un problème consommant 14Go de mémoire et utilisant 8 processus MPI. On observe des gains mémoires moyens plus faibles avec un temps d'exécution plus impacté, bien que restant très largement inférieur au temps d'exécution obtenue en laissant les indirections actives. Le détail est en annexe, mais sur ce test, par moment, KSM parvient à réduire la consommation mémoire autant que les indirections. Il lui faut toutefois du temps et il perd ses gains lors des pics de consommation.

Un de nos tests a été réalisé par erreur en laissant active une option d'instrumentation du code à la compilation. Hera était donc exécuté avec des performances dégradées. Lors de ce test, nous avons remarqué que le gain mémoire apporté par KSM était bien plus important que les cas précédents. La figure 6.6 donne les résultats obtenus lors de ce test. Clairement, sur la figure 6.6 KSM dispose de plus de temps pour fusionner les pages et permet d'atteindre des gains mémoires bien plus importants. Cette fois, le pic de consommation est lui aussi diminué. Un test réalisé en mode séquentiel sur un cas plus petit montre même des gains mémoires supérieurs à ceux obtenus avec les indirections. On est donc a priori limité par le côté non optimal de l'implémentation actuelle de KSM qui ne fusionne pas les pages assez rapidement.

D'après ces résultats, KSM apporte des gains non négligeables sur l'application Hera. L'implémentation actuelle montre toutefois ses limites avec une cadence de fusion limitée. Mais comme nous l'avons vu avec la version non optimisée d'Hera, il est probablement possible d'obtenir des gains supérieurs si KSM pouvait fusionner les pages plus rapidement. Une approche de ce type représente donc un bon candidat pour éliminer les indirections présentes dans certaines applications pour peu que KSM soit amélioré.

6.5 Limitations de KSM

Au vu des expériences réalisées et à la lumière des documentations et codes sources de KSM, on peut identifier les limites de l'implémentation actuelle comme suit :

Taille des blocs : KSM repose sur la pagination pour fusionner les blocs identiques, cela suggère que les données présentent des similarités sur des segments de la taille d'une page et

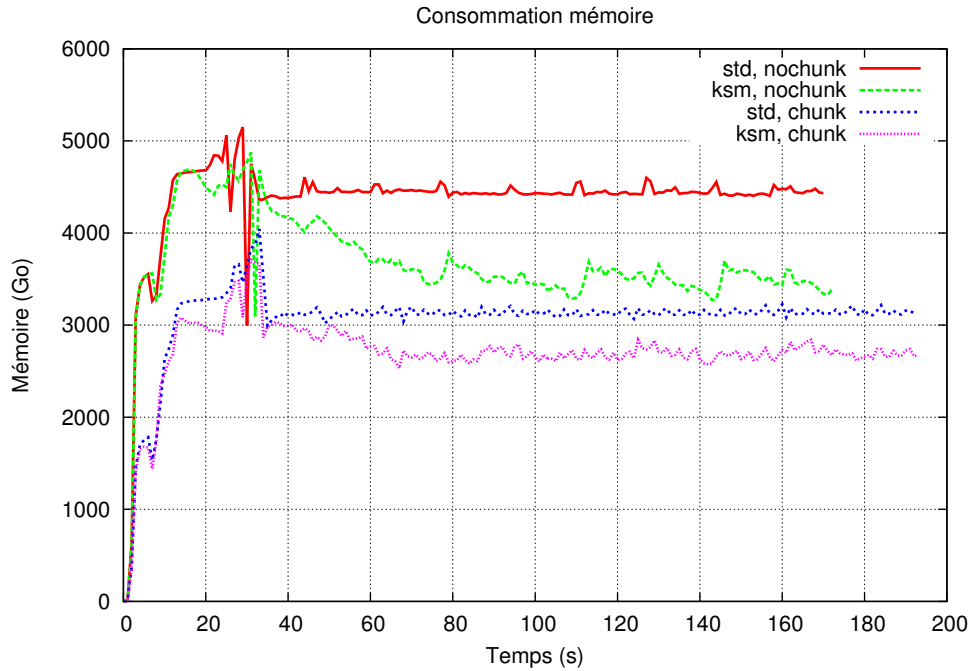
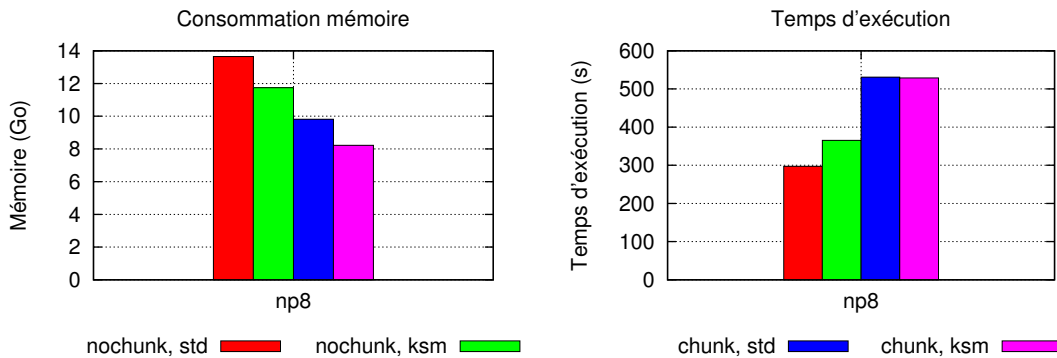


FIGURE 6.4 – Détail de l'utilisation de la mémoire au cours du temps de Hera sur 4 cœurs.

FIGURE 6.5 – Test avec Hera sur un problème occupant 14Go et résolu avec 8 processus MPI avec les paramètres de KSM `pages_to_scan = 200000` et `sleep_milliseconds = 100`.

alignés sur cette même taille (4Ko en général). Cette limitation n'est pas contournable dans le cadre de cette approche. Il semble toutefois qu'il soit tout de même possible d'obtenir des gains avec cette limitation comme le montrent nos résultats.

Démon séquentiel : Le démon `ksmd` chargé d'analyser les pages périodiquement est actuellement implémenté de manière séquentielle. Nous avons vu sur Hera que cela représentait une limite quant à la quantité de fusion que l'on peut attendre de KSM. Sur des nœuds disposant d'un grand nombre de cœurs il serait certainement très souhaitable de pouvoir rendre ce démon parallèle.

Support NUMA : Pour l'instant, KSM ne tient absolument pas compte des aspects NUMA. Le démon va en effet tenter de fusionner toutes les pages même si ces dernières proviennent de nœuds NUMA distincts. Cela peut être souhaitable en situation de forte consommation mémoire, mais pas en permanence.

Contrôle de KSM : KSM fonctionne en tâche de fond avec un nombre très restreint de paramètres. Dans le cadre d'une application, il serait probablement intéressant de pouvoir

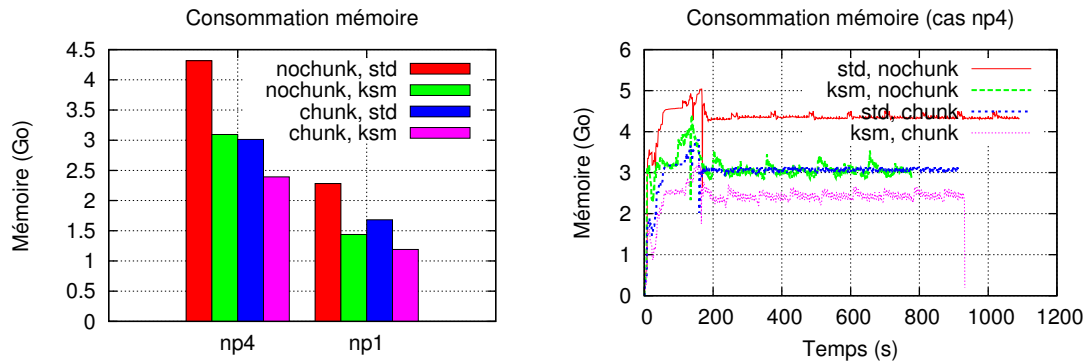


FIGURE 6.6 – Gains observés avec KSM sur une exécution d’Héra utilisant 1 ou 4 processus sur 8 coeurs disponibles. Hera a été compilé par erreur avec l’option `-finstrument-function` de gcc donc avec des performances très réduites. À droite, détail temporel de la consommation mémoire du cas à 4 processus.

interagir avec le démon. Pour demander explicitement l’analyse des pages lors de certaines phases, par exemple. En cas de pic mémoire, l’application pourrait être prête à céder de son temps de calcul (via l’allocateur) pour fusionner des pages et ainsi éviter d’enclencher la pagination disque ou se faire interrompre.

Observable mémoire : L’implémentation actuelle de KSM ne met pas à jour le nombre de pages physiques enregistrées auprès du processus (RSS : Resident Segment Size). Il n’est donc pas possible d’observer le taux de fusion en suivant cette observable via `ps` ou `top`. Les gains sont visibles sur la mémoire libre globale du système. La mise à jour de RSS n’étant pas réalisée, il faudrait s’assurer que les gestionnaires de tâche ne tentent pas tuer l’application en détectant une utilisation mémoire supérieure à la réalité.

Passage du pic : Nous l’avons vu sur Hera, bien que l’application consomme en moyenne moins de mémoire, la réduction du pic est moins importante. Les nouvelles allocations ne sont en effet pas réalisées en mode pré-fusionné. Il faut donc du temps à KSM pour analyser les pages et les fusionner. Une solution serait de pouvoir notifier KSM pour le réveiller de manière interactive lorsque l’on arrive à un pic critique ; quitte à ce que l’allocateur fasse attendre l’application le temps que la mémoire soit fusionnée.

6.6 Bénéfices potentiels de KSM

Au-delà d’une simple réduction de la mémoire, on peut attendre de KSM quelques bénéfices en terme de développement et d’exécution :

Utilisation des caches : Les derniers niveaux de cache des processeurs actuels sont adressés physiquement. On peut donc espérer obtenir une amélioration de leur efficacité.

Réduire les indirections : Nous avons vu avec Hera qu’il est possible de compenser en partie ou totalement l’utilisation des indirections introduites pour réduire la consommation mémoire de l’application. Contrairement à une méthode implémentée en logiciel, les indirections sont ici gérées par le matériel avec un surcoût nul en termes d’accès. De plus, leur mise en place ne déborde pas sur le code de l’application. Il faut toutefois que les données présentent des redondances sur des segments d’une taille minimum de 4Ko.

Fusion des tableaux de constantes physiques : En MPI, les tables de constantes physiques doivent être dupliquées pour chaque processus. Cette duplication peut conduire à une surconsommation importante et en général inutile de mémoire. KSM permet dans ce cas

de fusionner automatiquement et de manière transparente ces répliques. D'autres techniques existent, mais celle-ci a l'avantage de ne pas nécessiter de modifications importantes de l'application.

6.7 Piste non évaluée : extension de la sémantique de `mmap`.

Comme décrit, KSM a pour but de rechercher les pages au contenu identiques dans l'espace d'adressage et de les fusionner en mode *copie sur écriture*. La section 6.5 donne les limites de l'implémentation actuelle de ce procédé, notamment lié aux contraintes de performance des coûteuses analyses de contenu. On peut ainsi se demander s'il ne serait pas préférable de permettre au développeur de lui même mettre en place des projections en mode *copie sur écriture* depuis l'espace utilisateur.

La sémantique attendue correspondrait à pouvoir demander la mise en place d'un segment copie d'un autre segment du même espace virtuel. Remarquons que la sémantique actuelle permet d'établir ce type de projection uniquement au travers de l'appel système *fork* ou par projection de mémoire partagée et en ré-implémentant le support de copie en espace utilisateur sur la base de signaux de types *faute de segmentation*. Ces approches sous-optimales gagneraient à être supportées directement par le noyau qui intègre déjà ces mécanismes de manière optimisée. Il ne manque pour cela que le moyen de demander leur mise en place. Le code 6.2 donne un exemple d'extension en considérant la mise en place d'un appel dit *mcow*, mais qui pourrait également s'implémenter par extension de *mmap*.

Code 6.2– Proposition de sémantique *mcow*

```
1 //Segment existant
2 char * cow_target;
3 //Segment pre-existant a ecraser ou NULL
4 char * map_on_addr;
5
6 //Nouvel appel systeme
7 char * ptr = mcow(map_on_addr, cow_target, size);
8
9 //Ou par modification de mmap
10 char * ptr = mmap(map_on_addr, size, PROTECTION,
11                  MAP_PRIVATE | MAP_ANON | MAP_COW, 0, cow_target);
12
13 //modification d'un des segments entrainants une
14 //copie automatique par le noyau
15 ptr[0] = 0;
```

6.8 Conclusion

Nous avons vu que KSM proposait un mécanisme actif permettant de fusionner les pages physiques identiques en mode copie sur écriture. Cette approche à la base développée dans le cadre de KVM, peut également être appliquée aux applications pour réduire leur empreinte mémoire, tout en maintenant des performances décentes. Dans certains cas, KSM peut permettre d'éliminer les indirections mises en place au niveau logiciel pour réduire la consommation mémoire de certaines applications. Au contraire des approches logicielles, KSM a l'avantage de ne pas dégrader l'optimisation du code par le compilateur et de profiter des mécanismes matériels.

Sur Hera, bien que la version actuelle de KSM montre ses limites, nous avons pu observer des gains mémoires de l'ordre de 15%. Dans un cas, les gains ont pu atteindre 35%, et ce, en gardant des performances proches du mode sans indirections et sans modifications lourdes de l'application. La version actuelle de KSM étant séquentielle, les fusions ne sont toutefois pas assez rapides pour réduire certains pics de consommation.

KSM est actuellement conçu principalement pour les machines virtuelles et pour interagir le moins possible avec ces dernières. Pour une utilisation plus générale, il serait probablement intéressant de lever certaines limitations de l'implémentation actuelle, notamment la limite liée à l'aspect séquentiel du démon *ksmd* et le non-support des nœuds NUMA. Une collaboration entre le démon *ksmd* et l'application via l'allocateur pourrait également être souhaitable pour un usage en contexte HPC.

Troisième partie

Conclusion et perspectives

Chapitre 7

Conclusion

Dans ce manuscrit, nous avons dans un premier temps rappelé l'évolution des supercalculateurs massivement parallèles atteignant aujourd'hui l'ordre du million de cœurs. La structuration de ces calculateurs doit aujourd'hui répondre aux problématiques croissantes de consommation d'énergie, d'accès à la mémoire et de stockage de l'information (mémoire et systèmes de fichiers). Les solutions actuellement retenues conduisent à la construction d'architectures très hiérarchiques composées de grappes de nœuds NUMA multicœurs. Ces architectures impliquent une programmation à mémoire partagée (*threads*) en plus de l'approche historique à base de mémoire distribuée (MPI). Dans ce contexte, avec un nombre croissant de threads à exploiter, le système d'exploitation et les bibliothèques sont soumis à un besoin croissant de support du parallélisme et de prise en compte de ces hiérarchies.

Nous nous sommes donc intéressés aux problématiques liées à la gestion de la mémoire du point de vue HPC. L'analyse préliminaire réalisée en début de thèse a ainsi mis en évidence un problème d'interférence possible entre les différents composants de la chaîne de gestion mémoire (caches processeurs, OS, allocateur, application). Cette étude s'est surtout intéressée au problème de placement des données vis-à-vis des caches en évaluant les politiques de coloration de pages des OS. À l'opposé du discours conventionnel, nous avons ainsi montré l'intérêt habituellement négligé d'une politique de coloration aléatoire telle qu'employée dans Linux. Il a été montré que cette approche offre une plus grande résistance aux cas pathologiques en comparaison des méthodes de coloration dites "régulières". Ces méthodes trop régulières conduisent à des effets de résonances qui dépendent des décisions de l'allocateur mémoire et du schéma d'accès à la mémoire par l'application. Les dégradations observées peuvent alors représenter des facteurs entiers tels que cela a par exemple été observé sur EulerMHD (facteur 3 sur 8 cœurs). L'augmentation du nombre de cœurs couplée à l'exploitation de caches partagés tendent à augmenter l'impact de ce problème. Ces pertes sont à comparer aux gains maximums de 60% sur les NAS ou aux 5% observés sur Linpack. Il a ainsi été montré que la politique de Linux représentait un bon choix en évitant ce travers. Cette observation peut être exploitée pour améliorer les techniques de coloration actuelles afin de supprimer leur aspect trop régulier et bénéficier des avantages des deux méthodes. Nous avons également montré que ces problèmes touchaient notablement les grosses pages de 2 Mo du fait de leur définition matérielle. Cette problématique doit donc être prise en compte au niveau de l'allocateur mémoire en espace utilisateur.

La seconde problématique abordée concerne l'étude de l'allocateur mémoire en espace utilisateur. Dans le cadre du développement de MPC, il est apparu important de traiter la question de la gestion de ces allocations en prenant en compte les architectures massivement parallèles actuelles et structurées sous forme NUMA. Certains travaux débutés dans les années 2000 ont permis l'émergence d'allocateurs mémoires parallèles efficaces. Nous avons toutefois observé que les allocateurs disponibles (Jemalloc, TCMalloc et Hoard) rencontraient de grandes difficul-

tés à fournir de bonnes performances sur les nœuds 128 cœurs dont nous disposons désormais. Les observations montrent des pertes de performances comparées à l’allocateur de la Glibc pouvant atteindre 20%. Les problèmes rencontrés tiennent en deux points essentiels : un manque de prise en compte explicite des aspects NUMA et un trop grand nombre d’échanges avec le système d’exploitation qui est affecté par des problèmes d’extensibilité. Concernant le second point, nous avons été amenés à repenser le fonctionnement de l’allocateur avec un effet tampon pour parer à l’impossibilité de changer rapidement les algorithmes centraux de l’OS. Nous avons donc étendu le fonctionnement normal de l’allocateur pour les petits segments afin d’obtenir un recyclage des grands segments (au-delà du mégaoctet). Cette approche demande toutefois une prise en main spécifique du problème, les méthodes de réutilisation de ces grands segments devant satisfaire des contraintes différentes des petits segments.

Avec un support du NUMA, nous avons ainsi obtenu sur une simulation numérique conséquente, des gains qui peuvent atteindre 50% du temps d’exécution total sur des nœuds 128 cœurs. Ces gains sont toutefois obtenus au prix d’une augmentation de la consommation mémoire (de l’ordre de 15%) non nécessairement acceptable pour certaines applications limitées par la mémoire disponible. Nous avons donc travaillé pour obtenir une méthode supportant une politique complémentaire économe au sein de la même implémentation. L’activation de ce profil économe dégrade les performances, mais permet d’obtenir des gains mémoires équivalents à ce que propose Jemalloc (18% avec Hera sur les nœuds 32 cœurs). L’obtention de cette possibilité de contrôle ouvre désormais la porte à une migration dynamique d’une politique à l’autre en fonction des phases de l’application et de la disponibilité de la ressource mémoire. Les concepts fondamentaux mis en avant dans notre démarche peuvent être résumés comme suit :

Tas locaux et source mémoire : L’allocateur est globalement structuré sur la base de deux composants principaux. Les tas locaux associés à chaque thread gèrent la réutilisation locale des petits segments. Ces derniers ont un fonctionnement sans verrous pour optimiser les performances en contexte parallèle. En complément, les sources mémoires sont chargées de fournir des macro-blocs (taille typique de 2 Mo) aux tas locaux.

Contrôle de recyclage : Les tas locaux mettent en place des politiques de retour agressif de la mémoire vers les sources mémoires. Ceci permet de concentrer la politique de contrôle de consommation au niveau des sources mémoires. Cette approche implique de fait l’emploi de décisions réversibles ouvrant la possibilité d’exploitation de choix dynamiques de la politique à suivre. Ceci limite également la rétention de mémoire au niveau des différents threads, point qui pourrait éventuellement poser problème.

Ré-utilisation des gros segments : Nous avons discuté l’intérêt d’exploiter une méthodologie de recyclage des gros segments basé sur la sémantique *mremap* offerte par Linux. Il est ainsi possible d’éliminer les risques de fragmentation de la mémoire à grande échelle pouvant intervenir sinon.

Distinction des libérations distantes : Les tas locaux fonctionnent sans verrous, il a donc été nécessaire de distinguer les libérations locales des libérations distantes afin d’éliminer tous les conflits potentiels. Cette approche a été mise en place en construisant une file locale atomique d’accumulation de blocs à libérer. Cette liste est alors vidée par le thread parent lors des opérations mémoires suivantes.

Source mémoire NUMA : La réutilisation des macro-blocs implique un suivi de leur appartenance NUMA afin d’éviter tout échange involontaire en réponse à une requête d’allocation. En pratique, il est impossible (ou difficile) d’assurer un contrôle générique pour les threads non fixé sur un nœud particulier. Nous avons donc mis en place une politique de confiance isolant les threads “fiabiles” des threads “non fiable”. Il est ainsi possible de réduire les risques de pollution mémoire des threads pour lesquels l’utilisateur fait des efforts de support NUMA.

Registre par région : La mise en place d'un registre sous forme de région nous a permis d'obtenir une structure d'indexation des macro-blocs afin d'attacher l'appartenance de ces derniers à leur tas de rattachement. Cette approche permet ainsi de distinguer les allocations locales et distantes. Elle permet également d'étendre la possibilité de mixer différents allocateurs s'intégrant dans les opérations *free* et *realloc* standards. Remarquons que la structure retenue permet un accès essentiellement sans verrous à cet index.

Support de segments utilisateurs : La structure actuelle de l'allocateur permet également de réutiliser ses composants internes afin de gérer les allocations sur un segment mis en place par l'utilisateur avec des propriétés particulières (pages punaisées, mémoire partagée...).

Le recyclage de gros segments a été introduit pour compenser une faiblesse de l'OS. Nous nous sommes donc intéressés à ce problème et avons observé qu'au-delà des problèmes principaux d'extensibilité, 40% du temps des fautes de pages pouvait être consommé par des besoins de remise à zéro de la mémoire. Nous avons également montré que les grosses pages étaient affectées par ce problème de manière beaucoup plus importante. Sous Windows, le point d'effacement du contenu des pages est déplacé pour le sortir du chemin critique. Dans ce document, nous avons proposé de supprimer ce dernier. Les remises à zéro sont toutefois nécessaires pour des raisons de sécurité. Nous avons donc proposé une modification de la sémantique d'interaction avec *mmap* permettant une réutilisation locale de la mémoire par l'OS. Cette sémantique n'impose plus l'effacement des données tout en maintenant le niveau de sécurité initial. Ceci nous a permis de réduire de 45% les temps de fautes de pages tout en réduisant la consommation de bande passante, cycle processeur et purge des caches. En contexte non NUMA, des gains de passage à l'échelle ont également été observés. Sur les grosses pages, les gains obtenus peuvent atteindre un facteur 57 sur micro-benchmarks, ouvrant un nouvel intérêt pour ce type de pagination. Sur application réelle, nous avons montré sur 12 cœurs qu'il était ainsi possible de compenser les surcoûts de libération des allocateurs mémoires exploitant des profils économes. Ces travaux ont fait l'objet d'une publication en 2013[VSW13].

Le dernier point abordé a traité la problématique d'économie de l'espace mémoire par fusion des pages au contenu identique à l'aide du module KSM (Kernel Samepage Memory) offert par le noyau Linux. Cette approche a montré un certain intérêt pour la simulation numérique multi-physiques multi-matériaux testés en réduisant l'intérêt d'un recours à un système d'indirections logicielles. L'implémentation actuelle de ce module montre toutefois certaines limites du fait de son manque de parallélisme en ne permettant pas de fusionner assez rapidement les pages sur un grand nombre de cœurs. Le principe général de ce type d'approche reste toutefois intéressant à reprendre même si l'implémentation du mécanisme nécessite d'être revisitée plus en profondeur avec un point de vue HPC.

Pour résumer, tout au long de ce manuscrit, nous nous sommes attachés à étudier les problèmes de gestion mémoire en considérant le contexte HPC avec le nombre croissant de cœurs organisés sous forme de hiérarchie NUMA. Au cours de ces analyses, nous avons mis en évidence quelques points problématiques en terme de performance notamment au niveau de l'interaction des OS / allocateur mémoire / matériel. Nous avons montré que ces composants doivent être considérés comme un tout, afin de mettre en cohérence leurs politiques internes, qu'il s'agisse de la problématique de placement vis-à-vis des caches ou du taux d'échange de mémoire entre ces deux composants. Nous rappelons que la consommation de la mémoire est actuellement un problème reprenant de l'importance. Nous avons donc également cherché à trouver un équilibre entre cette diminution des échanges et le surcoût mémoire que cela engendre dans notre approche. Dans cette optique il a été montré qu'une meilleure coopération de l'allocateur et de l'OS pouvait permettre par une extension sémantique d'éliminer les coûteux effacements mémoires nécessaires à la politique de sécurité de l'OS. Un résumé de la spécificité et de l'orientation générale de nos travaux peut être décrit comme une étude de la bonne coopération allocateur /

OS sur architecture NUMA en prenant en compte l'utilisation de grands volumes mémoires.

Il est reconnu que l'accès à la mémoire est une problématique montante sur les architectures modernes. Les résultats obtenus montrent que les problèmes de gestion du partage de cette ressource sont également un point important nécessitant d'être ré-investiguer en prenant en compte l'évolution des architectures et des usages qui en sont faits. Lors de notre étude, nous avons parfois observé des écarts de performance sous forme de facteurs entiers, pointant des lacunes qu'il devient important de combler. Les problèmes observés sont en partie induits par un manque de passage à l'échelle des mécanismes internes à l'OS. On rappelle ainsi que le coût de gestion de cette mémoire est nécessairement proportionnel à la taille à gérer. Si les mécanismes ne sont pas rendus parallèles, la gestion de cet espace grandissant finira nécessairement par poser un problème majeur. Ajoutons qu'au vu de nos travaux, l'utilisation d'une taille de page de 4 Ko semble aujourd'hui trop faible au vu des volumes à traiter. À l'opposé, 2 Mo semble pour l'heure une taille trop importante pour les caches. Un meilleur compromis semble se situer proche de 64 Ko ou 128 Ko (une fraction des voies du cache pour permettre un certain aléa) si l'on tient compte de nos résultats couplés à l'étude réalisée lors du stage ayant précédé cette thèse.

D'une manière plus générale, l'évolution des architectures actuelles représente un défi pour la pile logicielle existante en nécessitant un besoin de passage à l'échelle toujours plus grand et la prise en compte de nombreux paramètres. Par ailleurs, l'augmentation de la complexité des problèmes traités et des volumes de données associés tend à faire croître le nombre de composants logiciels en interaction. Dans un contexte où les architectures évoluent rapidement, il semble de plus en plus important de garder une capacité d'adaptabilité en limitant les adhérences non réversibles à une architecture propre ou de découpler les parties adhérentes des parties qui peuvent être abstraites. À ce titre, les méthodes de développement logiciel sont actuellement soumises à rude épreuve, particulièrement dans le domaine HPC où la performance reste un point clé, parfois, semble-t-il, mis trop en avant avec une vue à trop court terme. Cet objectif se réalise alors au détriment d'une capacité de maintenance et d'adaptabilité à long terme du code et de sa performance. Ceci est particulièrement vrai si les optimisations fines se font au prix d'une dégradation de la vue d'ensemble. Le point clé n'étant pas d'abandonner la performance, bien au contraire, mais de lui assurer une expression maintenable.

Chapitre 8

Perspectives

Une partie des travaux de cette thèse mettent en lumière les problèmes d'interaction entre logiciel et structure fine du matériel. Nous avons notamment discuté la problématique des caches du fait de leur associativité. Ces problèmes délicats à prendre en compte au niveau logiciel peuvent toutefois conduire à une perte importante de performance. Nous avons ainsi proposé des pistes d'amélioration des méthodes de coloration de pages, mais n'avons pas encore prototypé ces modifications au niveau noyau. Sur ce point particulier, il paraît important de suivre l'évolution de ces interactions, notamment pour évaluer l'impact de l'arrivée des nouveaux caches partagés en forme d'anneau tels qu'employés dans le Xeon Phi. Il faudrait en effet vérifier si leur implémentation particulière limite ou maintient les problèmes de conflits inter-cœurs observés dans notre étude pour les grosses pages ou les paginations régulières. Un prototype de détection de ces cas pathologiques a également été mis au point lors de ces travaux. Ce dernier ne supportait toutefois pas les applications multithreadées. Il pourrait donc être intéressant de reprendre les points clés de ce prototype et d'implémenter l'analyse dans un outil tel que Valgrind.

Au niveau des allocations, nos travaux nous ont permis d'obtenir une méthode offrant des profils performants ou économes sur une base unique. Il est ainsi possible de reproduire les extrêmes observés au travers des allocateurs Jemalloc et Tcmalloc. À ce titre, il reste désormais à évaluer l'intérêt d'application d'une politique dynamique s'adaptant aux phases de l'application. Dans nos travaux, nous avons pour partie laissé de côté la problématique des petites allocations en les considérant comme suffisamment discutées dans la littérature. L'implémentation de notre prototype mériterait toutefois d'intégrer un support plus performant reprenant les idées fondamentales de Jemalloc pour ces allocations. Comme discuté en fin de section 4.17.4, il serait certainement intéressant d'évaluer l'application des concepts mis en avant dans cette thèse au sein de l'implémentation de Jemalloc. Il serait ainsi possible de lui apporter un support NUMA et les mécanismes de recyclage tout en profitant de sa gestion efficace des allocations petites et moyennes.

Toujours au niveau de l'allocateur, nous avons également discuté l'intérêt de fournir (par exemple au travers de directives de type *pragma*) un supplément d'information à l'allocateur vis-à-vis des problèmes de projection NUMA. Deux des sémantiques proposées ont été supportées dans le prototype actuel. Sans ces informations, nous avons montré que certaines ambiguïtés pouvaient subsister, notamment au niveau de l'usage de la fonction *realloc*. Ce point serait intéressant à lever si les architectures NUMA s'installent de manière durable. Sur le plan topologique, il serait intéressant de choisir le niveau de placement (approche de type HLS) des composants de l'allocateur (source mémoire, tas local). Ceci peut offrir un levier supplémentaire pour contrôler le rapport entre contention mémoire et trop grande dissémination des tampons d'allocations conduisant à une augmentation de la consommation mémoire. Cette remarque est

notamment liée à la structure non NUMA exposée par les architectures de type Xeon Phi. Avec un grand nombre de cœurs, il serait en effet intéressant d'utiliser une source mémoire pour x cœurs et par exemple exploiter un tas local par cœur physique et non thread.

Au niveau OS, l'étude préliminaire sur le problème de remise à zéro des pages a montré un intérêt certain pour l'approche. Cette dernière reste toutefois à évaluer sur les nœuds 128 cœurs et sur Xeon Phi afin de confirmer les gains potentiels de cette méthode sur ces architectures. Si l'intérêt est confirmé, il reste à implémenter l'intégration aux mécanismes de réclamation mémoire du noyau et à éventuellement discuter une intégration au noyau officiel. Cette méthode induit une réduction des transferts mémoires et du nombre d'opérations, elle pourrait donc être étudiée sur le plan énergétique afin d'évaluer les gains éventuels sur ce plan. Il ne faut toutefois pas oublier que le problème fondamental d'extensibilité de l'algorithme de gestion de la table des pages demeure et nécessitera d'être surveillé et probablement corrigé. Toujours vis-à-vis de l'extensibilité, nous avons vu que les mécanismes KSM du noyau pouvaient très certainement bénéficier d'améliorations, voir, d'extensions de la sémantique, pour profiter des informations dont dispose le programmeur en lui permettant d'établir lui même des segments en copie sur écriture.

Cette thèse s'est focalisée sur les aspects logiciels. Nous pouvons toutefois proposer d'éventuelles idées à étudier au niveau matériel. Sur ce plan, nous pouvons proposer les trois pistes suivantes. En cas de généralisation de l'emploi des grosses pages, il peut être intéressant d'évaluer l'applicabilité du concept de brassage par masque discuté en section 3.5.2. Ceci permettrait de ne pas trop corréliser les décisions de l'allocateur avec les caches. Dans la même orientation, on pourrait s'interroger sur les méthodes de remise à zéro de la mémoire, par exemple, en déléguant ce travail directement à la RAM. Ceci éviterait l'exécution, par le processeur, de transferts mémoire coûteux et peu productifs. Il pourrait en résulter un gain en terme d'énergie bien qu'il faille régler les problèmes de cohérences. Enfin nous avons discuté (section 2.5.2) la méthode d'invalidation des TLB sur l'ensemble des cœurs. L'impact de la méthode actuelle sur architecture x86 mériterait certainement d'être ré-étudiée vis à vis du nombre croissant de cœurs.

Pour terminer, nous avons au cours de cette étude observé de nombreux phénomènes complexes liés à la structure du matériel ou de l'OS. À ce titre, les applications doivent adapter leur code et surtout leurs méthodes d'accès à la mémoire pour tenir compte de ces problèmes. Il ne semble toutefois pas raisonnable de prendre explicitement en compte ces trop nombreux paramètres lors de l'écriture d'un code de calcul. Si une méthode d'organisation des données est implémentée de manière trop statique et trop contrainte par les algorithmes, il semble difficile d'imaginer que l'on puisse raisonnablement aboutir à un code optimal. Les choix ont en effet tendance à être mauvais et ne pourront être corrigés s'ils ne sont pas réversibles. D'autre part, l'incapacité à tester efficacement d'autres organisations des données nuit aux méthodes de recherche. Ceci conduit à des comparaisons souvent délicates, voire biaisées, des différentes approches par le nombre de changements trop importants nécessaires à leur mise en place. À ce titre, il semblerait intéressant d'obtenir une méthodologie d'abstraction entre organisation mémoire des données et algorithme. Le cas extrême peut consister à évaluer l'intérêt des DSL pouvant aider à la séparation des problèmes, au moins en phase de prototypage.

Bibliographie

Bibliographie

- [ABHS89] Melvin C. August, Gerald M. Brost, Christopher C. Hsiung, and Alan J. Schiffleger. [Cray X-MP : The Birth of a Supercomputer](#). *Computer*, 22(1) :45–52, jan 1989.
- [ABI⁺09] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. [An Extension of the StarSs Programming Model for Platforms with Multiple GPUs](#). In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.
- [ADM11] Yehuda Afek, Dave Dice, and Adam Morrison. [Cache index-aware memory allocation](#). *SIGPLAN Not.*, 46(11) :55–64, June 2011.
- [AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. [Increasing memory density by using KSM](#). In *OLS '09 : Proceedings of the Linux Symposium*, pages 19–28, July 2009.
- [AHH89] A. Agarwal, J. Hennessy, and M. Horowitz. [An analytical cache model](#). *ACM Trans. Comput. Syst.*, 7(2) :184–215, May 1989.
- [Amd67] Gene M. Amdahl. [Validity of the single processor approach to achieving large scale computing capabilities](#). In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.
- [AN09] Cédric Augonnet and Raymond Namyst. [Euro-Par 2008 Workshops - Parallel Processing](#). chapter A Unified Runtime System for Heterogeneous Multi-core Architectures, pages 174–183. Springer-Verlag, Berlin, Heidelberg, 2009.
- [Arc10] Andrea Arcangeli. [Transparent Hugepage Support](#), KVM Forum, 2010.
- [ASBC09] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. [Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches](#). In *HPCA*, 2009.
- [BA01] Jeff Bonwick and Jonathan Adams. [Magazines and Vmem : Extending the Slab Allocator to Many CPUs and Arbitrary Resources](#). In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 15–33, Berkeley, CA, USA, 2001. USENIX Association.
- [BAM⁺96] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. [Compiler-directed page coloring for multiprocessors](#). *SIGOPS Oper. Syst. Rev.*, 30(5) :244–255, September 1996.
- [BB99] Mark Baker and Rajkumar Buyya. [Cluster computing : the commodity supercomputer](#). *Softw. Pract. Exper.*, 29(6) :551–576, May 1999.
- [BBB⁺91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrisnan, and S. K. Weeratunga. [The nas parallel benchmarks](#). Technical report, 1991.
- [BCD69] A. Bensoussan, C. T. Clingen, and R. C. Daley. [The multics virtual memory](#). In *Proceedings of the second symposium on Operating systems principles*, SOSP '69, pages 30–42, New York, NY, USA, 1969. ACM.
- [BCOM⁺10] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. [hwloc : A Generic Framework for Managing Hardware Affinities in HPC Applications](#). In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.
- [BCRJ⁺10] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Kolai, and Cédric Valensi. [Performance Tuning of x86 OpenMP Codes with MA-QAO](#). In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010.
- [BDH⁺08] Kevin J. Barker, Kei Davis, Adolfo Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. [Entering the petaflop era : the architecture and performance of Roadrunner](#). In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1 :1–1 :11, Piscataway, NJ, USA, 2008. IEEE Press.
- [Bec03] K. Beck. *Test Driven Development : By Example*. Pearson Education, 2003.
- [Ber02] Emery David Berger. *Memory management for high-performance applications*. PhD thesis, 2002. AAI3108460.
- [BGW93] Amnon Barak, Shai Geday, and Richard G. Wheeler. *The MOSIX Distributed Operating System : Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
- [Bha13] Srivatsa S. Bhat. [mm : Memory Power Management](#), 2013.
- [BKW98] Satyendra Bahadur, Viswanathan Kalyanakrishnan, and James Westall. [An empirical study of the effects of careful page placement in Linux](#). In *ACM 36th Southeast Conference*, 1998.
- [BLRC94] Brian N. Bershad, Dennis Lee, Theodore H. Rømer, and J. Bradley Chen. [Avoiding conflict misses dynamically in large direct-mapped caches](#). *SIGPLAN Not.*, 29(11) :158–170, November 1994.
- [BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. [Hoard : a scalable memory allocator for multithreaded applications](#). *SIGPLAN Not.*, 35 :117–128, November 2000.
- [BMG06] Darius Buntinas, Guillaume Mercier, and William Gropp. [Design and Evaluation of Nemesi : a Scalable, Low-Latency, Message-Passing Communication Subsystem](#). In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 521–530, Singapour, Singapour, 2006.
- [BMG07] Darius Buntinas, Guillaume Mercier, and William Gropp. [Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesi communication subsystem](#). *Parallel Comput.*, 33(9) :634–644, September 2007.
- [BOP03] Katherine Barabash, Yoav Ossia, and Erez Petrank. [Mostly concurrent garbage collection revisited](#). *SIGPLAN Not.*, 38(11) :255–268, October 2003.

- [BP05] Daniel P. Bovet and Marco Cesati Ph. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 3 edition, November 2005. Paperback.
- [BSL⁺02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. *Scratchpad memory : design alternative for cache on-chip memory in embedded systems*. In *Proceedings of the tenth international symposium on Hardware/-software codesign, CODES '02*, pages 73–78, New York, NY, USA, 2002. ACM.
- [Buc07] Ian Buck. *GPU Computing : Programming a Massively Parallel Processor*. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO '07*, pages 17–, Washington, DC, USA, 2007. IEEE Computer Society.
- [BZ93] David A. Barrett and Benjamin G. Zorn. *Using lifetime predictors to improve memory allocation performance*. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI '93*, pages 187–196, New York, NY, USA, 1993. ACM.
- [CCD⁺05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. *Grid'5000 : A Large Scale and Highly Reconfigurable Grid Experimental Testbed*. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, GRID '05*, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.
- [CD97] Michel Cekleov and Michel Dubois. *Virtual-Address Caches Part 1 : Problems and Solutions in Uniprocessors*. *IEEE Micro*, 17(5) :64–71, September 1997.
- [CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. *Cache-conscious structure layout*. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, PLDI '99*, pages 1–12, New York, NY, USA, 1999. ACM.
- [CKZ12] Austin T. Clements, M. Frans Kaashoek, and Nikolai Zeldovich. *Scalable address spaces using RCU balanced trees*. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII (2012)*, pages 199–210, New York, NY, USA, 2012. ACM.
- [CLT] W. Jalby C. Lemuet and S. Touati. *Improving Load/Store Queues Usage in Scientific Computing*. In *Proceedings ICPP 2004*.
- [CPJ10] Patrick Carribault, Marc Pérache, and Hervé Jourdain. *Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC*. In *Proceedings of the 6th international conference on Beyond Loop Level Parallelism in OpenMP : accelerators, Tasking and more, IWOMP'10*, pages 1–14, Berlin, Heidelberg, 2010. Springer-Verlag.
- [CWHW03] Matthew Chapman, Ian Wienand, Gernot Heiser, and New South Wales. *Itanium Page Tables and TLB*, 2003.
- [DBa11] Jack Dongarra, Pete Beckman, and al. *The International Exascale Software Project roadmap*. *Int. J. High Perform. Comput. Appl.*, 25(1) :3–60, February 2011.
- [DCK07] Robert H. Dennard, Jin Cai, and Arvind Kumar. *A perspective on today's scaling challenges and possible future directions*. *Solid-State Electronics*, 51(4) :518 – 525, 2007.
- [DD68] Robert C. Daley and Jack B. Dennis. *Virtual memory, processes, and sharing in MULTICS*. *Commun. ACM*, 11(5) :306–312, May 1968.
- [DEJ⁺10] Frédéric Duboc, Cédric Enaux, Stéphane Jaouen, Hervé Jourdain, and Marc Wolff. *High-order dimensionally split Lagrange-remap schemes for compressible hydrodynamics*. *Comptes Rendus Mathématique*, 348(1–2) :105 – 110, 2010.
- [Den70] Peter J. Denning. *Virtual Memory*. *ACM Comput. Surv.*, 2(3) :153–189, September 1970.
- [DG02] Dave Dice and Alex Garthwaite. *Mostly lock-free malloc*. In *Proceedings of the 3rd international symposium on Memory management, ISMM '02*, pages 163–174, New York, NY, USA, 2002. ACM.
- [DGR⁺74] R. H. Denard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. *Design of ion-implanted MOSFETs with very small physical dimensions*. *IEEE Journal of Solid-state Circuits*, 98, 1974.
- [DL95] E.R. Dougherty and P.A. Laplante. *Introduction to Real-Time Imaging*. IEEE Press Understanding Science & Technology Series. Wiley, 1995.
- [Don88] Jack Dongarra. *The LINPACK Benchmark : An Explanation*. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK, 1988. Springer-Verlag.
- [DP00] David Detlefs and Tony Printezis. *A Generational Mostly-concurrent Garbage Collector*. Technical report, Mountain View, CA, USA, 2000.
- [Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*, 2007.
- [DSR12] Robert Dobbelin, Thorsten Schutt, and Alexander Reinefeld. *An Analysis of SMP Memory Allocators : MapReduce on Large Shared-Memory Systems*. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops, ICPPW '12*, pages 48–54, Washington, DC, USA, 2012. IEEE Computer Society.
- [dV] Guillaume Colin de Verdière. *Hydrobench*, <https://github.com/HydroBench>.
- [EBSA⁺12] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. *Power Limitations and Dark Silicon Challenge the Future of Multicore*. *ACM Trans. Comput. Syst.*, 30(3) :11 :1–11 :27, August 2012.
- [EK93] Rüdiger Esser and Renate Knecht. *Intel Paragon XP/S - Architecture and Software Environment*. In *Anwendungen, Architekturen, Trends, Seminar, Supercomputer '93*, pages 121–141, London, UK, UK, 1993. Springer-Verlag.
- [EKTb99] Michael Eberl, Wolfgang Karl, Carsten Trinitis, and Andreas Blaszczyk. *Parallel Computing on PC Clusters - An Alternative to Supercomputers for Industrial Applications*. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 493–498, London, UK, UK, 1999. Springer-Verlag.
- [Eva06] Jason Evans. *A Scalable Concurrent malloc(3) Implementation for FreeBSD*, 2006.
- [FK99] Ian Foster and Carl Kesselman, editors. *The grid : blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
- [Fly66] M. Flynn. *Very high-speed computing systems*. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.
- [FM90] Marc Feeley and James S. Miller. *A parallel virtual machine for efficient scheme compilation*. In *Proceedings of the 1990 ACM conference on LISP and functional programming, LFP '90*, pages 119–130, New York, NY, USA, 1990. ACM.

- [FYA⁺97] Hiroaki Fujii, Yoshiko Yasuda, Hideya Akashi, Yasuhiro Inagami, Makoto Koga, Osamu Ishihara, Masamori Kashiyaama, Hideo Wada, and Tsutomu Sumimoto. *Architecture and Performance of the Hitachi SR2201 Massively Parallel Processor System*. In *Proceedings of the 11th International Symposium on Parallel Processing, IPPS '97*, pages 233–241, Washington, DC, USA, 1997. IEEE Computer Society.
- [GCO65] E. L. Glaser, J. F. Couleur, and G. A. Oliver. *System design of a computer for time sharing applications*. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, AFIPS '65 (Fall, part I), pages 197–202, New York, NY, USA, 1965. ACM.
- [GFLMR13] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. *XKaapi : A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures*. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis, May 2013.
- [GH12] Mel Gorman and Patrick Healy. *Performance characteristics of explicit superpage support*. In *Proceedings of the 2010 international conference on Computer Architecture, ISCA'10*, pages 293–310, Berlin, Heidelberg, 2012. Springer-Verlag.
- [Glo] Wolfram Gloger. *PTMalloc* : <http://www.malloc.de/en/>.
- [Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
- [Hab08] Irfan Habib. *Virtualization with KVM*. *Linux J.*, 2008(166), February 2008.
- [Han73] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
- [HBHR11] Jorg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. *CAMA : A Predictable Cache-Aware Memory Allocator*. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems, ECRTS '11*, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.
- [Hen06] John L. Henning. *SPEC CPU2006 benchmark descriptions*. *SIGARCH Comput. Archit. News*, 34(4) :1–17, September 2006.
- [HK] Michal Hocko and Tomas Kalibera. *Reducing performance non-determinism via cache-aware page allocation strategies*. In *Proceedings of WOSP/SIPEW 2010*, pages 223–234.
- [HLC09] Kim Hazelwood, Greg Lueck, and Robert Cohn. *Scalable Support for Multithreaded Applications on Dynamic Binary Instrumentation Systems*. In *Proceedings of the 2009 International Symposium on Memory Management, ISMM '09*, pages 20–29, New York, NY, USA, 2009. ACM.
- [HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [Int10a] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A : System Programming Guide, part 1*, June 2010.
- [Int10b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B : System Programming Guide, Part 2*, June 2010.
- [JJF⁺99] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*. Technical report, 1999.
- [Jou05] Hervé Jourden. *HERA : A Hydrodynamic AMR Platform for Multi-Physics Simulations*. In Tomasz Plewa, Timur Linde, and V. Gregory Weirs, editors, *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 283–294. Springer Berlin Heidelberg, 2005.
- [JW98] Mark S. Johnstone and Paul R. Wilson. *The memory fragmentation problem : solved ?* In *Proceedings of the 1st international symposium on Memory management, ISMM '98*, pages 26–36, New York, NY, USA, 1998. ACM.
- [JYV12] Albert Cohen Jean-Yves Vet, Patrick Carribault. *Multigrain Affinity for Heterogeneous Work Stealing*, 2012.
- [Kam] Patryk Kaminski. *NUMA aware heap memory manager (AMD)*.
- [KELS62] T. Kilburn, D. B G Edwards, M. J. Lanigan, and F. H. Sumner. *One-Level Storage System*. *Electronic Computers, IRE Transactions on*, EC-11(2) :223–235, 1962.
- [KH92] R. E. Kessler and Mark D. Hill. *Page placement algorithms for large real-indexed caches*. *ACM Trans. Comput. Syst.*, volume 10 :338–359, November 1992.
- [Kje10] Henrik Kjellberg. *Partial Array Self-refresh in Linux*, 2010.
- [KK06] Simon Kahan and Petr Konecny. *"MAMA !" : a memory allocator for multithreaded architectures*. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '06*, pages 178–186, New York, NY, USA, 2006. ACM.
- [Kle05] Andi Kleen. *A NUMA API for LINUX*. Technical report, April 2005.
- [KLS86] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. *VAXcluster : a closely-coupled distributed system*. *ACM Trans. Comput. Syst.*, 4(2) :130–146, May 1986.
- [Kno65] Kenneth C. Knowlton. *A fast storage allocator*. *Commun. ACM*, 8(10) :623–624, October 1965.
- [KNTW93] Yousef A. Khalidi, Michael N. Nelson, Madhusudhan Talluri, and Dock Williams. *Virtual Memory Support for Multiple Pages*. Technical report, Mountain View, CA, USA, 1993.
- [Kop] Alexey Kopytov. *SysBench : a system performance benchmark*.
- [KPH61] T. Kilburn, R. B. Payne, and D. J. Howarth. *The Atlas supervisor*. In *Proceedings of the December 12-14, 1961, eastern joint computer conference : computers - key to total systems control*, AFIPS '61 (Eastern), pages 279–294, New York, NY, USA, 1961. ACM.
- [KPKZ11] Mahmut Kandemir, Ramya Prabhakar, Mustafa Karakoy, and Yuanrui Zhang. *Multilayer cache partitioning for multiprogram workloads*. *EuroPar'11*, 2011.
- [LBF92] William L. Lynch, Brian K. Bray, and M. J. Flynn. *The effect of page allocation on caches*. In *Proceedings of the 25th annual international symposium on Microarchitecture, MICRO 25*, pages 222–225, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [Lea] Doug Lea. *A Memory Allocator*.
- [LLC10] Books LLC. *Free Memory Management Software : Valgrind, Memcached, Mtrace, Lebl128, Splint, Duma, Electric Fence, Memory Pool System, Mpatrol, Memwatch*. Books Nippan, 2010.

- [Lor72] H. Lorin. *Parallelism in Hardware and Software : Real and Apparent Concurrency*. Prentice-Hall Series in Automatic Computation. Pearson Education, Limited, 1972.
- [LPMZ11] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. *Flikker : saving DRAM refresh-power through critical data partitioning*. *SIGARCH Comput. Archit. News*, 39(1) :213–224, March 2011.
- [LRW91] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. *The cache performance and optimizations of blocked algorithms*. *SIGPLAN Not.*, 26(4) :63–74, April 1991.
- [McC60] John McCarthy. *Recursive functions of symbolic expressions and their computation by machine, Part I*. *Commun. ACM*, 3(4) :184–195, April 1960.
- [McG65] W. C. McGee. *On dynamic program relocation*. *IBM Syst. J.*, 4(3) :184–199, September 1965.
- [MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. *Producing wrong data without doing anything obviously wrong!* ASPLOS, 2009.
- [MH] Rahul Manghwani and Tao He. *Scalable Memory Allocation*.
- [MM06] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
- [Moo65] G. E. Moore. *Cramming More Components onto Integrated Circuits*. *Electronics*, 38(8) :114–117, April 1965.
- [Moo75] Gordon E. Moore. *Progress in digital integrated electronics*. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.
- [MPI94] Forum MPI. *MPI : A Message-Passing Interface*. Technical report, 1994.
- [MS98] PAUL E. MCKENNEY and JOHN D. SLINGWINE. *READ-COPY UPDATE : USING EXECUTION HISTORY TO SOLVE CONCURRENCY PROBLEMS*. In *Parallel and Distributed Computing Systems*, 1998.
- [NIDC02] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. *Practical, transparent operating system support for superpages*. In *Proceedings of the 5th symposium on Operating systems design and implementation, OSDI '02*, pages 89–104, New York, NY, USA, 2002. ACM.
- [NMM⁺07] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. *The ghost in the machine : observing the effects of kernel operation on parallel application performance*. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07*, pages 29 :1–29 :12, New York, NY, USA, 2007. ACM.
- [NS07] Nicholas Nethercote and Julian Seward. *Valgrind : a framework for heavyweight dynamic binary instrumentation*. *SIGPLAN Not.*, 42(6) :89–100, June 2007.
- [Ope] *OpenPA : Open Portable Atomics*.
- [PCJ09] Marc Pérache, Patrick Carribault, and Hervé Jourden. *MPC-MPI : An MPI Implementation Reducing the Overall Memory Consumption*. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.
- [Per12] *H4H, PERF CLOUD*. http://www.teratec.eu/activites/projetsR_D_H4H_Perfcloud.html, 2012.
- [PJO8] Marc Pérache, Hervé Jourden, and Raymond Namyst. *MPC : A Unified Parallel Runtime for Clusters of NUMA Machines*. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing, Euro-Par '08*, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.
- [PN77] James L. Peterson and Theodore A. Norman. *Buddy systems*. *Commun. ACM*, 20(6) :421–431, June 1977.
- [PTH11] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. *Controlling cache utilization of HPC applications*. In *Proceedings of the international conference on Supercomputing, ICS '11*, pages 295–304, New York, NY, USA, 2011. ACM.
- [RB03] J. Howker R. Bryant. *Linux scalability for large NUMA systems*, 2003.
- [RF92] B. Ramakrishna Rau and Joseph A. Fisher. *Instruction-Level Parallel Processing : History, Overview and Perspective*, 1992.
- [Riz97] Luigi Rizzo. *A very fast algorithm for RAM compression*. *SIGOPS Oper. Syst. Rev.*, 31(2) :36–45, April 1997.
- [RLBC94] Theodore Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. *Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware*. In *In 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–266, 1994.
- [RM09] G. Ruetsch and P. Micikevicius. *Optimizing matrix transpose in cuda*, 2009.
- [Rob04] L. Robertson. *Anecdotes*. *Annals of the History of Computing, IEEE*, 26(4) :71–73, 2004.
- [Rob12] Les Robertson. *Computing Services for LHC : From Clusters to Grids*. In René Brun, Federico Carminati, and Giuliana Galli Carminati, editors, *From the Web to the Grid and Beyond*, The Frontiers Collection, pages 69–89. Springer Berlin Heidelberg, 2012.
- [RS09] Mark Russinovich and David A. Solomon. *Windows Internals : Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009.
- [Rus78] Richard M. Russell. *The Cray-1 Computer System*. *Communications of the ACM*, 21(1) :63–72, 1978. <http://www.odysci.com/article/1010112982782683>.
- [SCE99] Timothy Sherwood, Brad Calder, and Joel Emer. *Reducing cache misses using hardware and software page placement*. In *Proceedings of the 13th international conference on Supercomputing, ICS '99*, pages 155–164, New York, NY, USA, 1999. ACM.
- [SG] Paul Menage Sanjay Ghemawat. *TCMalloc : Thread-Caching Malloc*, <http://goog-perftools.sourceforge.net/>.
- [SGS10] John E. Stone, David Gohara, and Guochun Shi. *OpenCL : A Parallel Programming Standard for Heterogeneous Computing Systems*. *IEEE Des. Test*, 12(3) :66–73, May 2010.
- [SHcF06] Sushant Sharma, Chung-Hsing Hsu, and Wu chun Feng. *Making a case for a Green500 list*. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)/ Workshop on High Performance - Power Aware Computing*, 2006.
- [SKR⁺04] Byeong Hag Seong, Donggook Kim, Yangwoo Roh, Kyu Ho Park, and Daeyeon Park. *TLB Update-Hint : A Scalable TLB Consistency Algorithm for Cache-Coherent Non-uniform Memory Access Multiprocessors*. *IEICE Transactions*, 87-D(7) :1682–1692, 2004.

- [SM] Valat S. and Pérache M. [Optimisation de l'utilisation des caches L2/L3 et meilleure distribution des pages : prototype d'un module noyau Linux.](#)
- [SSC96] L. M. Silva, J. G. Silva, and S. Chapple. [Implementing Distributed Shared Memory on Top of MPI : The DSMPI Library.](#) In *Proceedings of the 4th Euro-micro Workshop on Parallel and Distributed Processing (PDP '96)*, PDP '96, pages 50–, Washington, DC, USA, 1996. IEEE Computer Society.
- [Sun90] V. S. Sunderam. [PVM : a framework for parallel distributed computing.](#) *Concurrency : Pract. Exper.*, 2(4) :315–339, November 1990.
- [SZ98] Matthew L. Seidl and Benjamin G. Zorn. [Segregating heap objects by reference behavior and lifetime.](#) In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pages 12–23, New York, NY, USA, 1998. ACM.
- [Tan05] Andrew S. Tanenbaum. *Structured Computer Organization (5th Edition)*. 2005.
- [TC12] M. Tolentino and K.W. Cameron. [The Optimist, the Pessimist, and the Global Race to Exascale in 20 Megawatts.](#) *Computer*, 45(1) :95–97, 2012.
- [TCP12] Marc Tchiboukdjian, Patrick Carribault, and Marc Perache. [Hierarchical Local Storage : Exploiting Flexible User-Data Sharing Between MPI Tasks.](#) In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 366–377, Washington, DC, USA, 2012. IEEE Computer Society.
- [TH94] Madhusudhan Talluri and Mark D. Hill. [Surpassing the TLB performance of superpages with less operating system support.](#) *SIGOPS Oper. Syst. Rev.*, 28(5) :171–182, November 1994.
- [Tho80] James E. Thornton. [The CDC 6600 Project.](#) 2(4) :338–348, October/December 1980.
- [THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. [LIKWID : A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments.](#) In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.
- [Top10] Top500. [Top 500 Supercomputer Sites.](#) <http://www.top500.org/>, 2010.
- [VDGR96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. [Operating system support for improving data locality on CC-NUMA compute servers.](#) *SIGPLAN Not.*, 31(9) :279–289, September 1996.
- [VJYA13] Carribault Patrick Vet Jean-Yves and Cohen Albert. [Parallélisme de tâches et localité de données dans un contexte multi-modèle de programmation pour super-calculateurs hiérarchiques et hétérogènes](#), 2013.
- [VSW13] Pérache Marc Valat Sébastien and Jalby William. [Introducing Kernel-Level Page Reuse for High Performance Computing.](#) *MSPC '13*, 2013.
- [Wal02] Carl A. Waldspurger. [Memory resource management in VMware ESX server.](#) In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 181–194, New York, NY, USA, 2002. ACM.
- [Wie08] Ian Wienand. [Transparent Large-Page Support for Itanium Linux](#), 2008.
- [wik] [Flynn's taxonomy.](#)
- [WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. [Dynamic Storage Allocation : A Survey and Critical Review.](#) In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, 1995. Springer-Verlag.
- [WM95] Wm. A. Wulf and Sally A. McKee. [Hitting the memory wall : implications of the obvious.](#) *SIGARCH Comput. Archit. News*, 23(1) :20–24, March 1995.
- [Wol] Marc Wolff. [Analyse mathématique et numérique du système de la magnétohydrodynamique résistive avec termes de champ magnétique auto-généré.](#)
- [YBF+ 11] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. [Why nothing matters : the impact of zeroing.](#) In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 307–324, New York, NY, USA, 2011. ACM.
- [YDLC10] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. [High-performance operating system controlled online memory compression.](#) *ACM Trans. Embed. Comput. Syst.*, 9(4) :30 :1–30 :28, April 2010.
- [YIN+ 11] Kazutomo Yoshii, Kamil Iskra, Harish Naik, Pete Beckman, and P. Chris Broekema. [Performance and Scalability Evaluation of 'Big Memory' on Blue Gene Linux.](#) *Int. J. High Perform. Comput. Appl.*, 25 :148–160, May 2011.
- [ZLHM09] Panyong Zhang, Bo Li, Zhigang Huo, and Dan Meng. [Evaluating the Effect of Huge Page on Large Scale Applications.](#) In *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, NAS '09, pages 74–81, Washington, DC, USA, 2009. IEEE Computer Society.

Annexes

Annexe A

Détail structurel des machines tests

Cette annexe fournit le détail des architectures tests utilisé pendant cette thèse. Pour ces travaux, nous rappelons que nous avons essentiellement exploité les nœuds des calculateurs de classe pétaflopiques Curie et Tera 100. Ces deux calculateurs sont conçus sur la base de nœuds exploitant respectivement 2 ou 4 processeurs octocœurs Intel. Une fraction de nœuds dits larges exploitent quant à eux 16 processeurs grâce à la technologie BCS (figure A.1) développée par Bull et permettant ainsi d'obtenir 128 cœurs en mémoire partagée. Ont également été utilisés les nœuds d'un petit ordinateur d'expérimentation Cassard et une station autonome à base de processeur Nehalem.

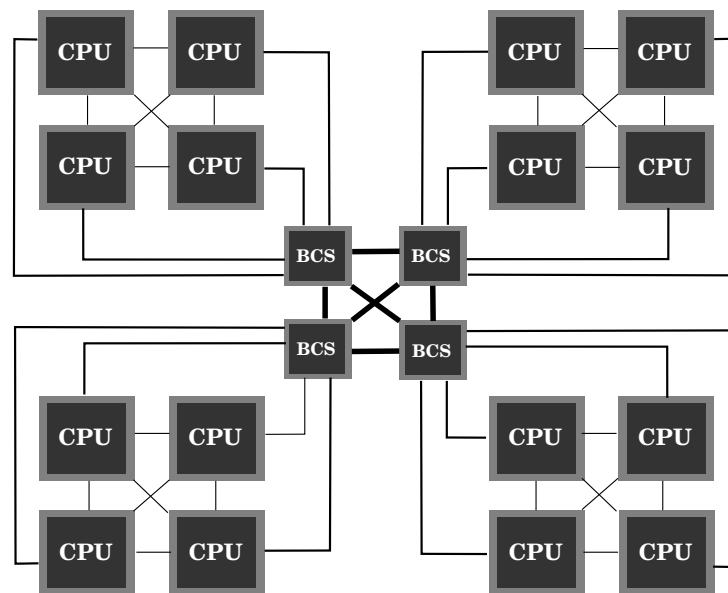


FIGURE A.1 – Structure à deux niveaux des nœuds larges 128 cœurs agrégés par la technologie BCS.

Calculeur Tera 100		
	Noeuds fin	Noeuds large
Performance	1.05 Pflops	
Mise en service	2009	
Noeuds	4370	55
Processeurs	17480	880
Coeurs	138368	7040
Processeurs / noeud	4	16
Coeurs / noeud	32	128
Mémoire / noeud	64 Go	512 Go
Stokage disque	8 Po	
Débit stockage disque	300 Go/s	
architecture	Intel Nehalem EX	Intel Nehalem EX
Processeur	Xeon X7560	Xeon X7560
Fréquence	2.27 GHz	2.27 GHz

Calculeur Curie		
	Noeuds fin	Noeuds large
Performances	1.36 Pflops	
Mise en service	2010	
Noeuds	5040	90
Processeurs	10080	1440
Coeurs	80640	11520
Processeurs / noeud	2	16
Coeurs / noeud	16	128
Coeurs / CPU	8	8
Mémoire / noeud	64 Go	512 Go
Stokage disque	5 Po	
Débit stockage disque	100 Go/s	
architecture	Intel Sandy Bridge EP	Intel Nehalem EX
Processeur	Xeon E5-2680	Xeon X7560
Fréquence	2.7 GHz	2.27 GHz

TABLE A.1 – Table d'information détaillée des architectures utilisées lors des tests.

Annexe B

Complémenté sur l'interférence des mécanismes d'allocations

Chapitre fournit quelques mesures complémentaires obtenues lors de l'étude des interactions entre OS, allocateur et caches discutés dans le chapitre 3. Sont essentiellement fourni ici des résultats obtenus sur un matériel différent : station à base de Core 2 Duo en employant la même démarche.

B.1 SpecCPU 2006

Les résultats des benchmarks SPEC CPU 2006 sont donnés dans la figure B.1. Seuls les benchmarks flottants (SPEC FP) ont été testés. Les résultats ont été générés en mode *base* avec les données de références.

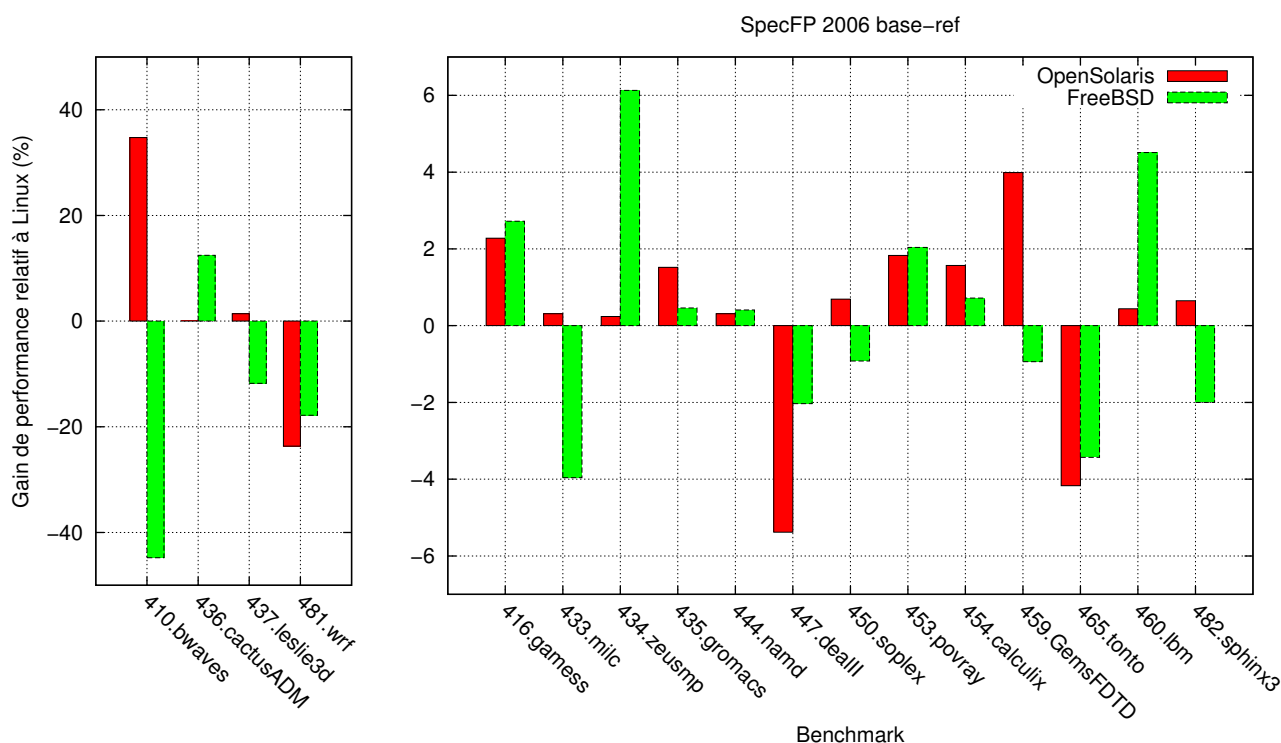


FIGURE B.1 – Gains de temps relatifs à Linux pour les benchmarks de SpecFP2006 en mode *base-ref*. Plus grand est meilleur.

Ici les résultats sont très inégaux entre les différents benchmarks, *410.bwaves* sort du lot avec un comportement proche de celui obtenu pour le programme MHD étudié en section 3.3.4. Sous OpenSolaris, les gains et pertes se compensent pour donner en moyenne un gain de 1%, contre -3.4% sous FreeBSD. En ne tenant pas compte des benchmarks *bwaves* et *bwrf* les deux moyennes deviennent quasi nulles. Certains benchmarks montrent toutefois des écarts allant de 2% à 10%.

B.2 Linpack

Des tests ont été réalisés sur le Linpack en faisant varier les paramètres N (taille du problème) et Bs (la taille des blocs). Les valeurs utilisées sont données dans le tableau B.1.

N	1000	2000	3000	6000	10240	17920	19328
Bs	32	64	128	256	512		

TABLE B.1 – Valeurs de certains des paramètres utilisés pour configurer le Linpack. Toutes les combinaisons ont été testées.

La figure B.2 donne les gains obtenus en comparant les performances maximums obtenues sur chaque OS pour les différentes tailles de problèmes. On y remarque des gains presque systématiques de 0.5% à 2% sous FreeBSD et OpenSolaris.

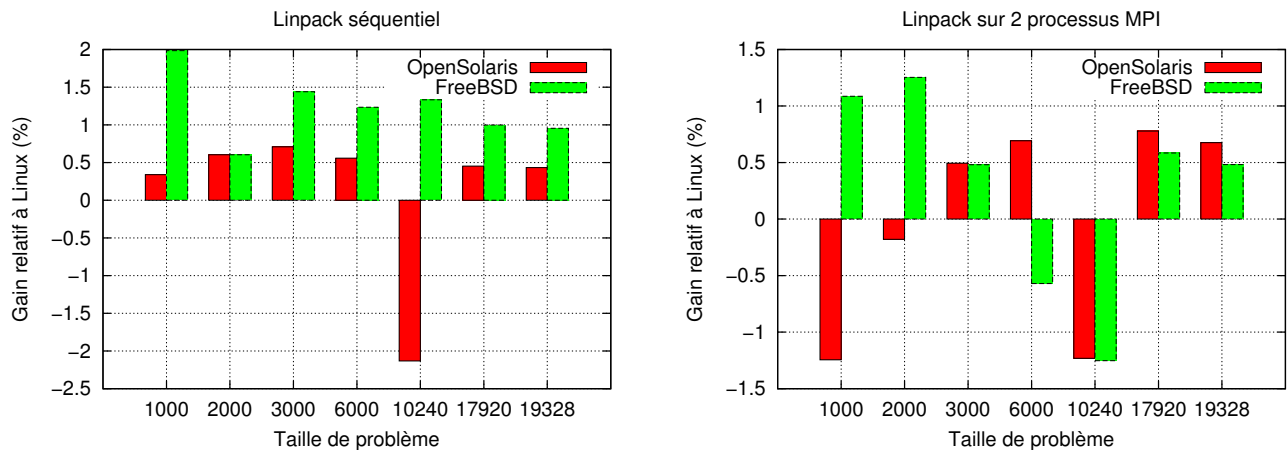


FIGURE B.2 – Gains de performances relatifs à Linux en gardant la taille de bloc (Bs) donnant la meilleure performance sous Linux pour chaque taille de problème (N) ; voir table B.2. Plus grand est meilleur.

N	1000	2000	3000	6000	10240	17920	19328
Bs séquentiel	128	256	256	256	256	256	256
Bs avec 2 processus MPI	64	64	64	128	128	256	256

TABLE B.2 – Tailles de bloc (Bs) retenues pour avoir donné les meilleures performances sous Linux, pour chaque taille de problème (N). À part pour $N=3000$ avec 2 processus, le maximum est atteint avec le même Bs sur tous les systèmes testés.

Les écarts étant relativement faibles, de multiples exécutions ont été réalisées pour chacune des tailles en choisissant le paramètre Bs ayant donné la meilleure performance sous Linux (table B.2). On obtient ainsi pour chaque taille une distribution des performances qui nous

informe sur la stabilité de la mesure. La figure B.3 donne quelques exemples de distributions obtenues. Les gains moyens obtenus par cette méthode sont résumés dans la figure B.4.

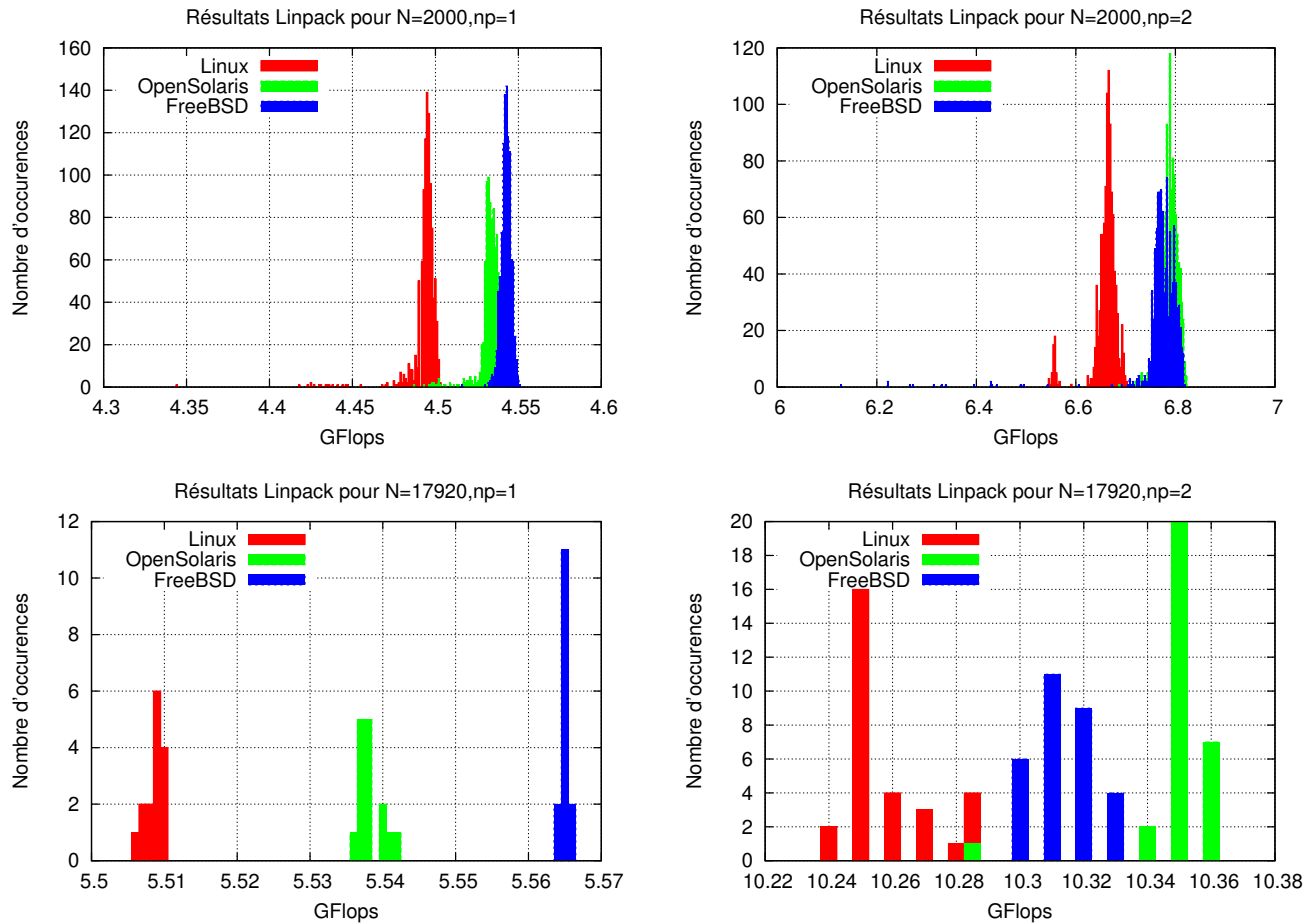


FIGURE B.3 – Distribution des mesures de performances pour les problèmes de tailles 2000 et 17920. Attention, les échelles ne commencent pas à 0.

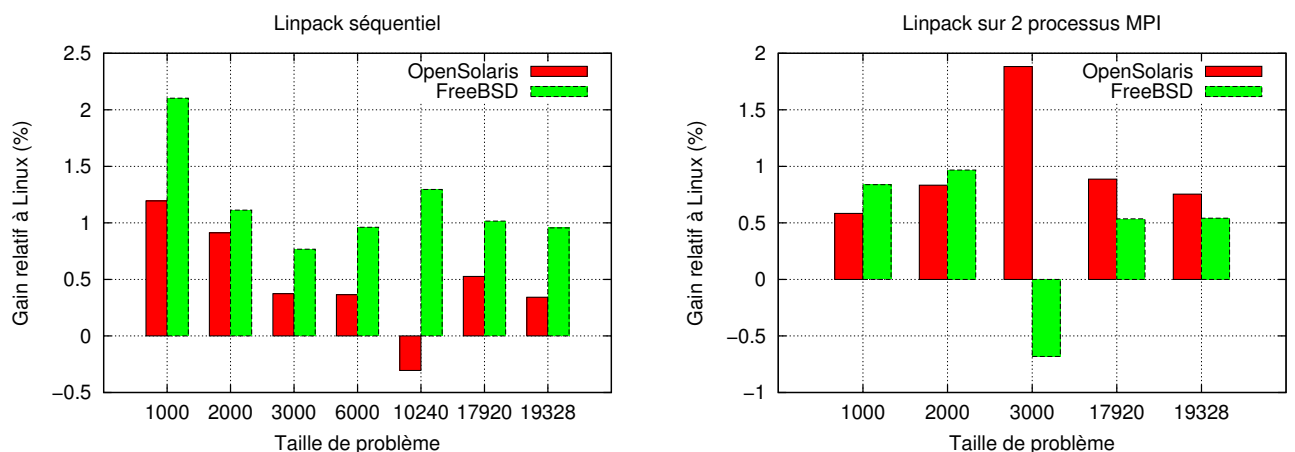


FIGURE B.4 – Gains de performances relatifs à Linux en utilisant les moyennes des distributions obtenues en figure B.3, toujours pour les couples $\{N, B_s\}$ listés en table B.2. Plus grand est meilleur.

On confirme donc bien l'observation de gains allant de 0.5% à 2% sous FreeBSD et OpenSolaris. Remarquons que ces résultats sont compatibles avec les observations faites par l'étude de

Zang en 2009[ZLHM09].

B.3 Alignements des tableaux de l'application MHD

En appliquant un modulo de 64 octets sur les adresses de chacun des grands tableaux de l'application MHD on obtient sur les différents systèmes :

Linux :

Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
Alignements :	16, 16, 16, 16	16, 16, 16, 16
...		

OpenSolaris :

Alignements :	16, 0, 48, 32	16, 0, 48, 32
Alignements :	16, 0, 48, 32	16, 0, 48, 32
Alignements :	48, 32, 16, 0	48, 32, 16, 0
Alignements :	16, 0, 48, 32	16, 0, 48, 32
Alignements :	48, 16, 48, 16	48, 16, 48, 16
Alignements :	16, 48, 16, 48	16, 48, 16, 48
Alignements :	16, 48, 16, 48	16, 48, 16, 48
Alignements :	48, 16, 48, 16	48, 16, 48, 16
...		

FreeBSD :

Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
Alignements :	0, 0, 0, 0	0, 0, 0, 0
...		

B.4 Résumé des effets d'alignements

Cette section fournit un résumé sous forme de table des problèmes d'alignements mis en évidence dans le chapitre 3. La table B.3 identifie les paramètres principaux pouvant générer les problèmes discutés. La table B.5 liste les problèmes et résume les conditions nécessaires à leur apparition. Cette table donne également les solutions envisageables pour réduire ou éliminer le problème en question, ces solutions sont résumées dans la table B.4.

1. Page Directory Entry
2. NBARR * NBTH

Nom	Description	Core 2 Duo	Core i7
Paramètres matériels			
<i>LL</i>	Dernier niveau de cache (partagé).	L2	L3
<i>L1SS</i>	Taille des voies du cache L1.	4 Ko	4 Ko
<i>LLSS</i>	Taille des voies du dernier niveau de cache.	256 Ko	512 Ko
<i>LLASSO</i>	Associativité du dernier niveau de cache	4 (pour 1 Mo) 8 (pour 2 Mo)	16
<i>CORES</i>	Nombre de cœurs physiques.	2	4
<i>HT</i>	Niveau d'Hyper-threading.	1	2
<i>CPTH</i>	Nombre de threads physiques.	$HT * CORES = 2$	$HT * CORES = 8$
<i>TLBASSO</i>	Associativité des DTLB	4 pour DTLB1	4 pour DTLB0 et STL
<i>TLBSIZE</i>	Taille mémoire adressée par une voie du DTLB	256 Ko pour DTLB1	64 Ko pour DTLB0 512 Ko pour STL
<i>PDEASIZE</i>	Taille mémoire adressée par PDE ¹	1 Go	1 Go
Paramètres logiciels			
<i>NBTH</i>	Nombre de threads utilisés.	2	4 or 8
<i>NBARR</i>	Nombre de tableaux utilisés par thread.	-	-
<i>NBS</i>	Nombre de flux mémoires simultanés ²	-	-
<i>SBA</i>	Adresse de base des flux mémoires.	-	-

TABLE B.3 – Identification des paramètres impliqués dans les problèmes d'alignement mémoire.

Solution	Niveau	Description
color	kernel	Fournir une coloration de page.
huge		Fournir un support de grosse page.
nrcolor		Casser les régularités en appliquant un décalage aléatoire entre chaque VMA par exemple. De manière générale, éviter d'exploiter un unique modulo.
16bp	malloc, application	Appliquer un décalage basé sur des multiples de 16o (ex. : $a_i = a_i + i * 16 o$)
4kp		Appliquer un décalage base sur des multiples de 4 Ko (ex. : $a_i = a_i + i * 4 Ko$)
nrsplit	libomp, application	Changer la manière de découper le travail, uniquement pour le problème lié à la directive "omp for".
chacc	application	Changes le schéma d'accès aux données.
chnbs		Réduire le nombre de flux simultanés.
smcache		Optimiser pour un dernier niveau de cache plus petit que physiquement disponible.

TABLE B.4 – Liste des solutions envisageables pour résoudre les différents problèmes listés dans la table B.5. Les solutions sont détaillées dans la section 3.5.

Impacte	Nom	Alignement	OMP	OS	Pages	Condition	Solutions	Probabilité
LL	Fuite dernier niveau de cache	-	-	Oui	4kB	- Utilisation de l'ensemble du dernier cache.	color, nrcolor, huge ou smcache	Élevé : Linux, Faible : SunOS
	OpenMP sur coloration régulière	LLSS	Oui	Oui	4 Ko	- <i>SBA</i> aligné relativement à <i>LLSS</i> - <i>NBS</i> > <i>LLASSO</i> - <i>NBTH</i> <= <i>CPUTH</i>	16bp, 4kp, nrcolor, nrsplit ou chnbs	Élevé : SunOS, Null : Linux
			Non	Non	>= <i>LLSS</i>		16bp, 4kp, nrsplit ou chnbs	Moyen
L1 ? ,LL	Pagination régulière	LLSS, L1SS ?	Non	Oui	4 Ko	- <i>NBS</i> > <i>LLASSO</i> - <i>SBA</i> aligné sur <i>LLSS</i> (ou à <i>L1SS</i> ?)	16bp, 4kp, nrcolor ou chnbs	Élevé : SunOS, Null : Linux
				Non	>= <i>LLSS</i>		16bp, 4kp ou chnbs	Moyen
L1	Conflits Load/Store	4 Ko	Non	Non	?? ?	- Utilisation d'accès de type a[i] = b[i-1]. - Tableaux alignés sur 4 Ko.	16bp ou chacc	Élevé
TLB, L1 hline TLB	Limite des PDE	PDEASIZE	Non	Non	4 Ko	- <i>NBS</i> > <i>TLBASSO</i> - <i>BSA</i> aligné sur <i>TLBSSASIZE</i> - <i>BSA</i> distants de plus que <i>PDEASIZE/NBS</i>	16bp, 4kp ou chnbs	Faible
	Limit d'associativité du DTLB	TLBSSASIZE	Non	Non	4 Ko	- <i>BSA</i> aligné sur <i>TLBSSASIZE</i> - <i>NBS</i> > <i>TLBASSO</i>	16bp, 4kp ou chnbs	Moyen

TABLE B.5 – Résumé des impacts de performance d'accès induit par les politiques d'allocation.

Annexe C

File atomique pour l’allocateur

Dans le cadre du développement de notre allocateur, nous avons eu besoin d’une file atomique ayant des propriétés d’accès synchronisés en considérant les contraintes d’insertion multiples et de purge par un thread unique. Pour ce faire nous avons repris l’implémentation de queue atomique fournie par la bibliothèque OpenPA et utilisée pour les échanges de message en mémoire partagé de Mpich2. Cette structure à toutefois l’inconvénient de considérer la récupération des éléments un a un. Dans le cadre des libérations distantes de notre allocation, nous considérons plutôt le cas d’une purge de l’ensemble de la liste en une opération. La liste a donc été modifiée de sorte que l’opération de récupération renvoie l’ensemble du contenu de la liste.

Code C.1– Queue atomique à purger unique

```
1  /** This method is thread-safe and can be used concurrently **/
2  static __inline__ void sctk_mpscf_queue_insert(struct sctk_mpscf_queue *
3      queue, struct sctk_mpscf_queue_entry * entry)
4  {
5      /* vars */
6      struct sctk_mpscf_queue_entry * prev;
7
8      /* errors */
9      assert(queue != NULL);
10     assert(entry != NULL);
11     assert(((unsigned long long)queue % SCTK_ALLOC_POINTER_ATOMIC_ALIGN == 0));
12     assert(((unsigned long long)entry % SCTK_ALLOC_POINTER_ATOMIC_ALIGN == 0));
13
14     /* this is the new last element, so next is NULL */
15     entry->next = NULL;
16
17     /* update tail with swap first */
18     prev = (struct sctk_mpscf_queue_entry *)sctk_atomics_swap_ptr(&queue->
19         tail, entry);
20
21     /* Then update head if required or update prev->next
22        This operation didn't required atomic ops as long as we are aligned in
23        memory*/
24     if (prev == NULL) {
25         /* in theory atomic isn't required for this write otherwise we can do
26            atomic write */
27         sctk_atomics_store_ptr(&queue->head, entry);
28     } else {
29         prev->next = entry;
30     }
31 }
32
33 /** Loop until last element and spin on it until next was setup to the given
34     expected tail. **/
```

```

30 static __inline__ void sctk_mpscfc_queue_wait_until_end_is(struct
    sctk_mpscfc_queue_entry * head, struct sctk_mpscfc_queue_entry *
    expected_tail)
31 {
32     /* vars */
33     volatile struct sctk_mpscfc_queue_entry * current = head;
34
35     /* errors */
36     assert(current != NULL);
37     assert(expected_tail != NULL);
38
39     /* loop until we find tail */
40     while (current != expected_tail)
41     {
42         if (current->next != NULL)
43             current = current->next;
44         else
45             sctk_atomics_pause();
46     }
47
48     /* check that we have effectively the last element otherwise it's a bug.
        */
49     assert(current->next == NULL);
50 }
51
52 /** CAUTION, this method must be called in critical section. */
53 static __inline__ struct sctk_mpscfc_queue_entry *
    sctk_mpscfc_queue_dequeue_all(struct sctk_mpscfc_queue * queue)
54 {
55     /* vars */
56     struct sctk_mpscfc_queue_entry * head;
57     struct sctk_mpscfc_queue_entry * tail;
58
59     /* errors */
60     assert(queue != NULL);
61
62     /* read head and mark it as NULL */
63     head = (struct sctk_mpscfc_queue_entry *)sctk_atomics_load_ptr(&queue->
        head);
64
65     /* if has entry, need to clear the current list */
66     if (head != NULL)
67     {
68         /* Mark head as empty, in theory it's ok without atomic write here.
69            At register time, head is write only for first element.
70            as we have one, produced work only on tail.
71            We will flush tail after this, so it's ok with cache coherence if
72            the two next
73            ops are not reorder.*/
74         sctk_atomics_store_ptr(&queue->head, NULL);
75         //OPA_write_barrier();
76
77         /* swap tail to make it NULL */
78         tail = (struct sctk_mpscfc_queue_entry *)sctk_atomics_swap_ptr(&queue
            ->tail, NULL);
79
80         /* we have head, so NULL tail is abnormal */
81         assert(tail != NULL);
82
83         /* walk on the list until last element and check that it was
84            tail, otherwise, another thread is still in registering the tail
85            entry
86            but didn't finish to setup ->next, so wait unit next became tail*/

```

```
85         sctk_mpscfc_queue_wait_until_end_is(head,tail);
86     }
87
88     /* now we can return */
89     return head;
90 }
```
