



Vers des supports exécutifs et des outils de profilage adaptés aux architectures many/multi-cœur dans le cadre du calcul hautes performances

Marc Pérache

► To cite this version:

Marc Pérache. Vers des supports exécutifs et des outils de profilage adaptés aux architectures many/multi-cœur dans le cadre du calcul hautes performances. Calcul parallèle, distribué et partagé [cs.DC]. Université de Versailles Saint-Quentin-en-Yvelines, 2015. tel-03544457

HAL Id: tel-03544457

<https://cea.hal.science/tel-03544457>

Submitted on 26 Jan 2022

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.



Vers des supports exécutifs et des outils de profilage adaptés aux architectures many/multi-cœur dans le cadre du calcul hautes performances

Thèse d'Habilitation à Diriger les Recherches

pour l'obtention de

l'Habilitation à Diriger les Recherches de l'université de Versailles
Saint-Quentin-en-Yvelines
(spécialité informatique)

par

Marc Pérache

Composition du jury

<i>Président :</i>	Malony Allen	- Université de l'Oregon
<i>Rapporteurs :</i>	Cappello Frank	- Argonne National Laboratory
	Cohen Albert	- INRIA/Ecole Normale Supérieure
	Krajecki Michaël	- Université de Reims
<i>Examineurs :</i>	Jalby William	- Université de Versailles Saint-Quentin-en-Yvelines

Table des matières

1	Introduction	1
1.1	Contexte	1
1.2	Organisation du document	2
2	Supports exécutifs multithread en contexte d'exécution calcul hautes performances	3
2.1	Le contexte calcul hautes performances	3
2.1.1	Environnement matériel	3
2.1.2	Environnement logiciel	5
2.2	Le support exécutif MPC	6
2.2.1	MPC en 2006	6
2.2.2	Le nouveau MPC	7
2.3	Environnement d'exécution / machine centre de calcul	9
2.3.1	Virtualisation	9
2.3.2	Profilage à grande échelle	9
2.4	Les challenges des supports exécutifs en contexte calcul hautes performances	10
3	L'allocation mémoire dans le contexte calcul hautes performances	11
3.1	Introduction	11
3.2	Optimisation de l'allocation mémoire en contexte multithread	13
3.2.1	Espace utilisateur	13
3.2.2	Espace noyau	15
3.3	Localité des données en contexte multithread	19
3.3.1	Gestion de la localité	19
3.3.2	Maintien de la localité	20
3.4	Optimisation de la consommation mémoire	20
3.4.1	Espace utilisateur	21
3.4.2	Espace noyau	21
3.5	Résultats sur la simulation numérique Hera	24
3.6	Conclusion/travaux futurs	25
3.6.1	Conclusion	25
3.6.2	Travaux futurs	27
4	Communication réseau rapide en contexte multithread	29
4.1	Gestion des accès réseau en contexte massivement multithread sur nœuds de grosse taille	29
4.1.1	Impact du multithreading sur les performances du réseau	29
4.1.2	Contribution : Multi-Threaded Virtual Rails	29
4.1.3	Gestion des tampons réseau en contexte Multithread	33
4.1.4	Évaluation de la méthode	33
4.1.5	Conclusion	36
4.2	Gestion de la progression des messages en contexte multithread	36
4.2.1	La progression des messages non-bloquants	36
4.2.2	Notre contribution : le Collaborative Polling	37
4.2.3	Évaluation de la méthode	37
4.2.4	Conclusion	39
4.3	Gestion de l'empreinte mémoire	39
4.3.1	Le protocole <i>eager</i> en contexte RDMA	39
4.3.2	Contribution : Auto-ajustement des tampons Eager RDMA	39

4.3.3	Évaluation de la méthode	42
4.3.4	Conclusion	43
4.4	Conclusion/Travaux futurs	43
4.4.1	Conclusion	43
4.4.2	Travaux futurs	43
5	Un support exécutif en contexte virtualisé	45
5.1	Passage de messages efficace entre machines virtuelles	45
5.1.1	Un périphérique virtuel pour le passage de messages	46
5.1.2	Des bibliothèques de communication en contexte virtualisé	46
5.2	Un périphérique virtuel pour le passage de messages	46
5.2.1	Contraintes supplémentaires en environnement virtualisé	46
5.2.2	Spécification du périphérique virtuel	48
5.3	Évaluation	52
5.3.1	Éléments d'implémentation	52
5.3.2	Évaluation sur des logiciels de calcul	52
5.3.3	Migrations de machines virtuelles	54
5.3.4	Allocation mémoire en contexte virtualisé	56
5.3.5	Bilan des évaluations	56
5.4	Conclusion/Travaux futurs	57
5.4.1	Conclusion	57
5.4.2	Travaux futurs	57
6	Profilage à grande échelle	59
6.1	Problématique	59
6.2	L'outil MALP	59
6.2.1	De la trace à l'analyse in situ	60
6.2.2	Des nœuds de calcul vers les nœuds d'analyse	60
6.2.3	Profilage multi-applications	61
6.2.4	Blackboard	61
6.3	Évaluation	62
6.3.1	Analyses mises en œuvre	62
6.3.2	Surcoût de la méthode d'analyse in situ	64
6.4	Conclusion/Travaux futurs	65
6.4.1	Conclusion	65
6.4.2	Travaux futurs	66
7	Conclusion	67
7.1	Travaux réalisés	67
7.1.1	Supports exécutifs multithread	67
7.1.2	... dans l'environnement d'un centre de calcul	69
7.2	Projets et perspectives	70
7.2.1	Supports exécutifs, localité des données et équilibrage de charge	70
7.2.2	Consommation mémoire	74
7.2.3	Virtualisation en contexte calcul hautes performances	76
7.2.4	Conclusion	80
	Bibliographie	87

Table des figures

2.1	Architecture des nœuds de calcul de la machine Curie “nœuds larges”	4
2.2	MPC en 2006	6
2.3	MPC aujourd’hui	7
3.1	Distribution en taille et temporelle des allocations mémoire 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.	12
3.2	Schéma d’organisation générale de l’allocateur faisant intervenir deux composants principaux, des sources mémoire et tas locaux assemblés pour former la chaîne complète d’allocation entre le système d’exploitation et l’application.	13
3.3	Organisation des en-têtes de macro-blocs, blocs alloués et libres. Les tailles des champs sont données en bits considérant l’architecture x86_64.	14
3.4	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).	15
3.5	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%.	16
3.6	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 segment de 4 Ko.	17
3.7	Principe de réutilisation des pages au niveau noyau permettant d’éliminer le besoin de remise à zéro du contenu des pages.	18
3.8	<i>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.</i>	22
3.9	<i>Détail de l’utilisation de la mémoire au cours du temps de Hera sur 4 cœurs.</i>	22
4.1	Micro-benchmark de bande passante MPI en contexte MPC avec deux nœuds de calcul et 16 tâches MPI avec le protocole <i>eager</i> spécifique aux messages de petite taille. Chaque tâche MPI communique avec une tâche sur le nœud distant comme décrit sur la figure (a). La figure (b) illustre la contention sur les accès aux files de messages Infiniband.	30
4.2	Estimation du nombre de <i>vrails</i> (ou <i>virtual subchannels</i>) dans une configuration multirails.	31
4.3	Comparaison des stratégies de sélection du <i>vraïl</i> . Dans les deux cas, un message est envoyé de la tâche MPI 1 vers la tâche MPI 6.	32
4.4	Il y a trois configurations possibles pour une couche de communications multithread sur les nœuds 128 cœurs. La figure (a) utilise une carte réseau par nœud. Dans la figure (b), nous utilisons un <i>vraïl</i> par nœud NUMA de niveau 1. La figure utilise un <i>vraïl</i> par nœud NUMA de niveau 2 (un <i>vraïl</i> par processeur).	34
4.5	Temps d’exécution et consommation mémoire sur le benchmark IMB AllToAll. Toutes les valeurs sont normalisées par rapport à MPC 1 <i>vraïl</i> 1 HCA.	34
4.6	Temps d’exécution sur le benchmark IMB AllToAll monorail	35
4.7	Extensibilité faible du code Athena sur les supports exécutifs MPC, Intel MPI, Bull MPI et AMPI	35
4.8	Recouvrement calcul/communications en contexte MPI	36
4.9	Implémentation MPI sans Collaborative-Polling (gauche) et implémentation MPI avec Collaborative-Polling (droite)	37

4.10	Implémentation du Collaborative-Polling dans le contexte MPC pour Infiniband	38
4.11	Illustration du protocole de <i>rendez-vous</i> à gauche avec le Collaborative-Polling et à droite sans. Le Collaborative-Polling permet à une tâche MPI inactive de faire progresser la totalité des étapes du protocole.	38
4.12	Évaluation du temps d'exécution de l'application EulerMHD	39
4.13	Le protocole <i>eager</i> RDMA	40
4.14	Protocole d'ajustement des tampons RDMA. L'émetteur initie la requête.	40
4.15	Exécution HERA[64] sur 64 nœuds avec 1 024 tâche MPI en contexte MPC. Le maillage est composé de 256^3 mailles et 300 itérations sont réalisées. La figure reporte la mémoire physique allouée sur les nœuds 13 et 21.	41
5.1	Projection des tampons de communication en espace utilisateur invité	49
5.2	Utilisation d'accès distants en lecture pour recouvrir les communications par du calcul . .	50
5.3	Transfert de messages à l'aide des tampons de communication	51
5.4	Transfert de messages par accès mémoire distant	51
5.5	Temps d'exécution du LINPACK	53
5.6	Temps d'exécution de HERA	54
5.7	Performances des communications pendant les migrations	55
5.8	Impact des migrations sur le temps d'exécution de HERA	55
5.9	Patch noyau en contexte virtualisé	56
6.1	Modèle de profilage à base de prise de traces.	60
6.2	Modèle de profilage à base de couplage réseau.	60
6.3	Architecture des flux dans MALP.	61
6.4	Implémentation du profilage à base de <i>blackboards</i>	61
6.5	Implémentation de notre architecture de <i>blackboard</i>	62
6.6	Exemple de topologie de communications pondérée.	63
6.7	Comparaison du niveau d'asynchronisme des applications EulerMHD et lbm.	64
6.8	Surcoût relatif de notre approche sur les NAS benchmarks et l'application EulerMHD (exécutions sur la machine TERA 100).	64
6.9	Surcoût relatif de différents outils de profilage sur le benchmark NAS SP.D (exécutions réalisées sur la machine Curie).	65
6.10	Surcoût de MALP en extensibilité forte sur le benchmark NAS LU.D.	66
7.1	Principe de l'algorithme de transport de particules Monté-Carlo	71
7.2	Mécanisme d'ajustement dynamique de la consommation mémoire	77

Liste des tableaux

3.1	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.	12
3.2	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.	25
3.3	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.	26
4.1	Évaluation des temps MPI pour l'application EulerMHD	39
4.2	NAS Fourier Transform (FT) classe D sur 512 tâches MPI, 32 nœuds. La taille des tampons est définie à 16Ko. Les résultats sont par processus MPI.	40
4.3	Exécution HERA sur 32 nœuds et 512 tâche MPI. Le maillage est composé de 256^3 et l'exécution composée de 40 itérations. Le tableau présente les temps d'exécution en fonction du nombre de d'ajustements de la taille des tampons <i>eager</i> RDMA.	42

Chapitre 1

Introduction

Ce document présente les activités de recherche que j’ai menées depuis la fin de ma thèse en 2006. Ces activités se sont déroulées au CEA DAM Île de France au sein du département Sciences de la Simulation et de l’Information où j’occupe actuellement le poste d’ingénieur/chercheur.

1.1 Contexte

Durant ma thèse, qui s’est effectuée au CEA DAM Île de France au sein du département Sciences de la Simulation et de l’Information, j’ai conçu et implémenté la première version de l’environnement *Multi-Processor Computing* (MPC). Cet environnement avait pour but principal de fournir un environnement de programmation de type mémoire distribuée par passage de messages basé sur les threads pour les codes de calcul déséquilibrés. Pour parvenir à cet objectif avec un code parallélisé en décomposition de domaines, nous avons choisi une approche de *surcharge* où nous avons plus de domaines de calcul que de cœurs de calcul (jusqu’à un facteur 10 entre le nombre de sous-domaines et le nombre de cœurs). L’ordonnanceur de threads au cœur de MPC devait donc gérer ces threads/sous-domaines et les répartir dynamiquement sur les cœurs. Très vite, nous nous sommes heurtés au problème des communications collectives et nous avons mis en place une technique de communications collectives intégrées à l’ordonnanceur qui avait de bonnes performances en contexte de surcharge.

Durant ces travaux, nous avons diagnostiqué un certain nombre de problèmes de performances autour de l’allocation mémoire et du support du réseau rapide. En effet, MPC disposait d’une première implémentation d’allocateur mémoire qui gérant uniquement les aspects contention en contexte multithread. Cette première version n’était donc pas optimisée pour la localité des données. En ce qui concerne le support des réseaux rapides, nous avons délégué cela temporairement à MPI. Cette délégation était limitée en performances car nous ne disposions pas de version MPI_THREAD_MULTIPLE. De plus, le standard MPI ne permettait pas, en contexte MPC, de tirer la quintessence du réseau sous-jacent. Nous avons aussi une difficulté d’utilisation de MPC du fait qu’il ne proposait pas d’interface standard. Enfin, notre approche n’anticipait pas l’évolution des modèles de programmation vers la programmation hybride MPI + Thread.

Ces dernières années, j’ai eu l’opportunité de m’intéresser à ces aspects, c’est-à-dire, l’étude des mécanismes et fonctionnalités contribuant à l’obtention de bonnes performances sur des super-calculateurs. Je me suis aussi intéressé à l’environnement dans lequel évoluaient ces supports exécutifs au sein des calculateurs. Ces travaux ont pu être menés de concert avec l’évolution des centres de calcul ouvert et classifié présents sur le site de Bruyères-le-Châtel.

Le problème de l’accès efficace aux couches réseau rapide en contexte multithread sur des machines ayant une hiérarchie mémoire de plus en plus marquée s’est posé à nous. J’ai donc commencé à étudier comment accéder avec le moins de contention possible à la couche réseau *elan* de la machine Tera10. Ces travaux préliminaires nous ont permis de poser les bases pour une analyse plus poussée des ces problématiques qui commença en 2010 (voir chapitre 4).

Dès 2007 et grâce à ce premier retour d’expériences sur l’utilisation des réseaux rapides, nous nous sommes posés la question de leur utilisation dans un contexte qui prenait beaucoup d’ampleur à l’époque : la virtualisation (voir 5). En effet, bien que cette technologie n’était pas utilisée en contexte calcul hautes performances, du fait du surcoût en termes de performances de cette méthode, elle était très prometteuse. La possibilité d’offrir aux utilisateurs des centres de calcul une stabilité de la pile logicielle sans pour autant sacrifier la sécurité a vite séduit les administrateurs des centres de calcul.

En parallèle de ces travaux, nous avons commencé en 2009 à réétudier le problème de l'allocation mémoire. Cette fois-ci, grâce aux travaux sur la virtualisation, nous avons enfin la possibilité de proposer une approche espace noyau + espace utilisateur acceptable par les centres de calculs (voir chapitre 3). En effet, demander l'utilisation d'un noyau modifié en contexte de production n'est pas acceptable. En revanche, avoir une machine virtuelle disposant d'un noyau modifié est tout à fait envisageable.

Enfin, nous nous sommes posés la question de la raison du manque d'utilisateurs du support exécutif MPC. Nous sommes arrivés à la conclusion que le fait que MPC ne dispose pas d'une interface de programmation standard était problématique. Nous avons donc entrepris de transformer MPC en une implémentation MPI à part entière. En 2009, alors que les premiers retours d'utilisateurs commençaient à arriver, nous avons été confrontés au problème de l'adaptation des outils de débogage et de profilage en contexte multithread sur calculateur de grande taille. Pour résoudre ces problèmes, nous avons tout d'abord conçu une approche générique de débogage des supports exécutifs à base de threads utilisateurs. Ces travaux se sont soldés par un transfert industriel et une collaboration avec la société Allinea. Nous nous sommes ensuite attelés au problème du profilage qui a mené à la création du logiciel MALP (voir chapitre 6).

La totalité de mes travaux a été influencée par les machines arrivant sur le site de Bruyères-le-Châtel mais aussi les activités de R&D autour de leur conception, ainsi que les collaborations industrielles et académiques dans le cadre de projets ANR, français ou européens.

1.2 Organisation du document

Dans ce document, nous allons tout d'abord présenter le contexte des supports exécutifs dans le calcul hautes performances. Ce chapitre sera aussi l'occasion de donner une vue globale de notre contribution. Cela permettra également de présenter le support exécutif MPC ainsi que ces évolutions depuis la fin de ma thèse.

Les chapitres 3 et 4 vont présenter les deux principales contributions directes aux supports exécutifs. Ces travaux ont été réalisés dans MPC. Le chapitre 3 détaillera les travaux que nous avons menés dans le cadre de l'allocation mémoire avec comme objectifs les performances en contexte multithreadé sur architecture NUMA. Ce chapitre présentera aussi la première brique de la stratégie globale de réduction de l'empreinte mémoire. Le chapitre 4 présente quant à lui les différentes méthodes mises en œuvre pour obtenir de bonnes performances dans les accès aux ressources réseaux en contexte multithread. Ce chapitre présentera aussi une méthode de recouvrement des communications par le calcul. Enfin, nous terminerons ce chapitre par une méthode de contrôle et d'ajustement dynamique de l'empreinte mémoire des couches réseau.

La suite du document présentera les études réalisées pour optimiser le fonctionnement et l'utilisation des supports exécutifs. Nous commencerons par une étude de la virtualisation. Cette étude nous a amené à concevoir une abstraction des ressources réseau pour les machines virtuelles. Nous verrons aussi en quoi les machines virtuelles peuvent apporter une flexibilité supplémentaire dans l'optimisation des supports exécutifs en contexte de production. Le chapitre 6 va quant à lui présenter notre méthode de profilage in situ sur des nœuds dédiés. L'outil que nous avons conçu permet de limiter au maximum l'impact du profilage tout en permettant un profilage à l'échelle. Nous montrerons que notre outil est par construction très bien adapté pour tenir compte à la fois de la virtualisation et du multithreading qui arrivent sur les nœuds de calcul des super-calculateurs.

Chapitre 2

Supports exécutifs multithread en contexte d'exécution calcul hautes performances

Ce chapitre a pour but de positionner notre contribution dans le contexte général du calcul hautes performances présent et à venir. Il a pour but d'éclairer le lecteur sur la démarche générale en présentant une vue cohérente de nos différents travaux. Nous allons tout d'abord présenter le contexte de nos travaux. Ensuite, nous allons présenter le support exécutif MPC. Puis, nous discuterons des activités que nous avons menées à la périphérie des supports exécutifs pour les rendre plus adaptés au contexte d'exécution envisagé dans les centres de calcul ainsi que les outils que nous avons mis en place pour préparer le passage à l'échelle. Enfin, nous terminerons par une discussion sur les challenges que devront relever les supports exécutifs.

2.1 Le contexte calcul hautes performances

Les travaux que nous présentons dans ce document sont liés au calcul hautes performances et en particulier aux supports exécutifs. Dans ce contexte, nous sommes fortement liés aux applications et architectures matérielles. Dans cette section, nous allons tout d'abord décrire l'évolution des architectures ainsi que les architectures types que nous avons utilisées pour nos évaluations. Dans un deuxième temps, nous présenterons les modèles de programmation disponibles sur ces architectures. Enfin, nous détaillerons deux applications représentatives qui seront nos applications phares pour les évaluations que nous allons présenter.

2.1.1 Environnement matériel

Dans cette section, nous allons maintenant détailler l'environnement matériel sur lequel s'exécutent les codes de calcul. Nous commencerons par une description de l'évolution des super-calculateurs, puis nous détaillerons l'architecture d'un nœud de calcul en particulier.

Évolution des architectures des super-calculateurs

Depuis un peu plus de dix ans, nous avons amorcé un changement marqué dans l'architecture des processeurs qui a eu un impact direct sur l'architecture des super-calculateurs. Avec la stagnation de la fréquence des processeurs, nous avons assisté à une explosion du nombre d'unités de calcul que ce soit au travers des architectures de type many/multi-cœurs ou encore des architectures de types GPUs. Cette évolution a mis une pression accrue sur les supports exécutifs.

En effet, au niveau de la machine complète, le nombre de nœuds de calcul a significativement augmenté. Si l'on prend le cas des machines du CEA : en 2001, la machine TERA1 disposait de l'ordre de 1 000 nœuds de calcul. Aujourd'hui avec TERA100, nous sommes à 4700 nœuds et les projections pour les machines futures sont de l'ordre de 10 000 nœuds de calcul d'ici 3-4 ans. Cette évolution met donc une forte pression sur le réseau d'interconnexion et sur le support exécutif qui en a la charge.

Au niveau du nœud de calcul aussi, nous avons assisté à une évolution. En effet, si l'on prend toujours l'exemple de TERA1, il y avait 4 processeurs mono-cœur par nœud. Aujourd'hui, nous atteignons 16

processeurs octo-cœurs pour un total de 128 cœurs par nœud dans le cas de processeurs “généralistes”. Si on prend en compte les architectures many-cœurs ou les GPUs, il y a plusieurs centaines de cœurs par processeur et un ou plusieurs processeurs par nœud (sous la forme de plusieurs accélérateurs par exemple). Cette augmentation du nombre de ressources par nœud de calcul, s’est faite au prix d’une complexification de ces nœuds qui exhibent maintenant une topologie fortement hiérarchique. Cette évolution a pour effet d’augmenter visiblement la contention sur les supports exécutifs et les systèmes d’exploitation.

Dans ce contexte de nœuds hiérarchiques, les supports exécutifs ont aussi la charge de la localité des données. En effet, cette hiérarchie impacte directement le temps d’accès aux données. Il est donc préférable d’effectuer des accès locaux pour obtenir une latence réduite et donc le maximum de performances. Le support exécutif doit donc à la fois exposer cette hiérarchie à l’utilisateur qui désire en tirer parti mais également fournir une politique transparente efficace aux utilisateurs qui ne veulent pas avoir à gérer cette complexité.

Exemple de l’architecture Curie

Dans les travaux que nous avons menés, nous nous sommes toujours attachés à ce qu’il soit adapté aux architectures présentes dans les grands centres de calcul. L’évolution des super-ordinateurs est depuis plus de dix ans marquée par une augmentation significative du nombre des unités de calcul. Cette augmentation du niveau de parallélisme n’est pas sans conséquence sur toute la chaîne logicielle utilisée (du système d’exploitation à l’application en passant par les supports exécutifs et les outils de profilage/débuggage).

Un grand nombre de nos expériences sera réalisé sur les partitions “nœuds larges” des machines Curie ou Terra100. Cette architecture a la particularité de disposer d’un grand nombre de cœurs par nœud de calcul et d’exhiber de fortes contraintes de topologie. En effet, ces nœuds sont composés des 4 “cartes mères” disposant de 4 processeurs (Nehalem EX 8 cœurs) reliées entre elles par un système conçu par Bull appelé *Bull Coherent Switch* (BCS)[21]. Comme on peut le voir sur la figure 2.1, les nœuds larges vont disposer de 128 cœurs pour 512 Go de mémoire totale. Ils disposent aussi de quatre cartes réseau Infiniband réparties sur les quatre nœuds NUMA de niveau 2.

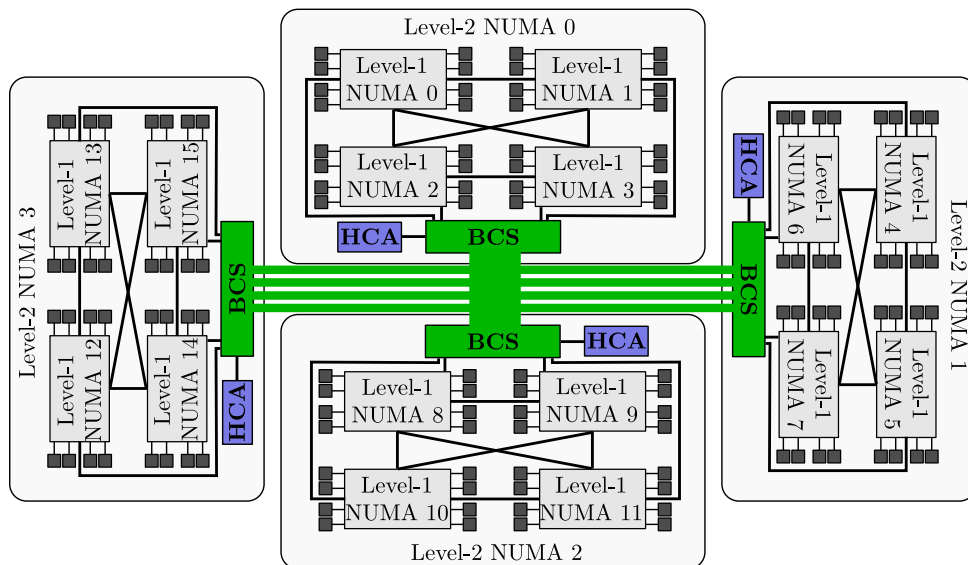


FIGURE 2.1 – Architecture des nœuds de calcul de la machine Curie “nœuds larges”

Cette architecture est assez représentative de l’évolution des architectures. En effet, c’est actuellement l’une des architectures offrant le plus grand nombre de cœurs par nœud et donc les plus hauts niveaux de contention en cas d’accès concurrents. Cette architecture préfigure ce que pourrait être un *System On Chip* (SOC) dans les années futures. Le challenge ne réside pas seulement dans l’utilisation efficace unitaire de ces nœuds de calcul, mais aussi dans leur agrégation en grappe. Les projections d’architectures Exascale proposent des grappes de plusieurs millions de cœurs agrégés en nœuds de calcul comportant plusieurs centaines de cœurs disposant chacun d’une quantité mémoire limitée à une fraction de Go voire quelques Go. Cette évolution des architectures va donc mettre une forte pression sur les applications via les modèles de programmation et les supports exécutifs.

2.1.2 Environnement logiciel

Dans cette section, nous allons décrire rapidement les différents modèles de programmation utilisés dans le contexte du calcul hautes performances. Ensuite, nous détaillerons deux exemples de codes de calculs représentatifs.

Modèles de programmation

Pour tirer parti de ces évolutions architecturales, de nombreux modèles de programmation ont émergé. On peut citer, par exemple, les langages *Partition Global Address Space* (PGAS). Ces modèles de programmation permettent de cibler les architectures de type grappe de multi-processeurs. Dans cette approche, les données sont réparties sur les nœuds de calcul et un mécanisme de type communications unidirectionnelles (*One Sided Communications*) permet d'accéder aux données qui ne sont pas locales. Ce type de langage est très bien adapté pour tirer parti des mécanismes *Remote Direct Memory Access* (RDMA) présents dans les réseaux rapides. Néanmoins, les PGAS mettent une forte pression sur la localité des données pour obtenir de bonnes performances. En effet, une mauvaise localité va se traduire par de nombreux accès réseau et donc les performances de l'application seront directement liées à la latence du réseau qui est très élevée en comparaison d'un accès local à la mémoire. Les PGAS sont des langages ou des extensions du langage qui vont reposer sur des supports exécutifs comme une implémentation MPI ou sur une implémentation spécifique comme GASNET[14].

Une autre approche de modèle de programmation est la programmation à base de tâches. Cette approche maintenant présente dans la norme 3 de OpenMP permet de diviser le travail à effectuer en plusieurs tâches élémentaires qui pourront être ordonnancées de manière astucieuse pour équilibrer la charge de travail mais aussi maximiser la localité des accès aux données. Ce type d'approche convient aussi bien à l'utilisation de processeurs généralistes que des accélérateurs[3, 4, 103]. Ici encore, le support exécutif va avoir un rôle très important car il sera le garant de la localité des données.

Enfin, les modèles de programmation standards comme MPI ou OpenMP sont toujours largement utilisés directement par les applications. En effet, la communauté de la simulation numérique dispose d'un très gros volume d'applications et d'une large expérience sur ces modèles de programmation et ne semble pas enclin à une réécriture complète mais plutôt à une approche incrémentale. C'est pourquoi de nombreuses personnes se tournent vers le modèle MPI + Thread. En effet, une des principales difficultés issue de l'évolution des architectures et en particulier de l'augmentation du nombre de cœurs, est l'emprise mémoire des applications. En effet, le modèle MPI par nature favorise la duplication de données et donc une consommation mémoire linéaire en le nombre de processus mis en œuvre. Passer à un modèle MPI + Thread permet de contenir le nombre de processus MPI tout en étant capable d'utiliser plus de cœurs de calcul.

Les travaux présentés dans ce document vont se focaliser sur les aspects support exécutif MPI dans un contexte multithread MPI via un thread-based MPI ou plus classiquement MPI + OpenMP ou MPI + PThread.

Exemples de codes réels

Pour mieux comprendre les difficultés du calcul hautes performances, il faut aussi se poser la question des applications s'exécutant sur super-calculateur. Nous allons ici décrire deux applications représentatives. La première est un gros code de production multi-physique. La seconde est un code d'étude représentative de travaux très ciblés.

HERA Depuis quelques années, des études sont engagées au CEA/DAM sur des méthodes et algorithmes de calcul utilisant des maillages AMR (Adaptive Mesh Refinement). HERA (pour Hydrodynamique Euler Raffinement Adaptatif) est une plate-forme multi-physique AMR développée au CEA/DAM[64]. Ce logiciel permet de simuler des écoulements compressibles multifluides (domaine de la Dynamique Rapide) en plusieurs dimensions d'espaces, avec couplage à des modèles physiques variés, comme des phénomènes thermiques de diffusion non linéaires. HERA est composé principalement de 300 000 lignes de C++, mais fait également appel à des langages interprétés comme Python ou C#. La méthode de parallélisation choisie est l'approche SPMD par décomposition de domaines. La parallélisation des schémas numériques par cette approche est classique : aux points de synchronisation des algorithmes, pour prendre en compte les contributions des sous-domaines voisins, des échanges de messages s'opèrent pour mettre à jour les bords de chaque sous-domaine. En revanche, l'approche AMR couplée à la décomposition de domaine induit des déséquilibres de charge entre les tâches parallèles. Le maillage s'adaptant au cours de la simulation, les charges de

travail attribuées à chaque tâche évoluent au cours du calcul. Des schémas de communications très agressifs sont donc mis en place pour redistribuer périodiquement les données entre les tâches.

EulerMHD EulerMHD[104, 105, 106] est une application C++ MPI pur résolvant à la fois les équations d'Euler et de la Magneto-hydrodynamique d'ordre élevé sur un maillage cartésien 2D. Ce code simule les équations d'Euler et la magnétohydrodynamique (MHD). Le code résout ces équations sur maillage cartésien en une et deux dimensions d'espaces. Le découpage MPI est un découpage 2D en sous-domaines rectangulaires. A chaque pas de temps, il y a des communications droite-gauche et haut-bas ainsi qu'une réduction sur le pas de temps. C'est un exemple d'application à priori équilibrée avec une parallélisation très courante.

Ces deux applications seront utilisées comme benchmark de référence tout au long de ce document.

2.2 Le support exécutif MPC

Dans cette section, nous allons détailler le support exécutif MPC et en particulier son évolution depuis la fin de ma thèse en 2006.

2.2.1 MPC en 2006

Fin 2006, j'avais réalisé la première implantation du support exécutif MPC[85, 91]. Déjà, MPC était un support exécutif à base de threads utilisateur. Ces derniers étaient gérés par un ordonnanceur de threads utilisateur optimisé pour la gestion efficace des communications en contexte de surcharge (i.e. lorsqu'il y a plus de threads que de cœurs disponibles). Cet ordonnanceur avait été en particulier optimisé pour la gestion des communications collectives. En effet, ces dernières avaient été intégrées à l'ordonnanceur. A cette époque, MPC ne disposait que de sa propre interface de communication haut niveau similaire à MPI. Il était donc nécessaire de modifier tous les codes applicatifs pour les rendre compatibles MPC. De plus, le support de réseaux rapides utilisait l'implantation MPI sous-jacente et ne permettait donc pas une optimisation fine des accès au réseau. En effet, MPI n'est pas le protocole le plus adapté à la gestion de destinataires multiples au sein d'un même processus (contexte d'exécution de MPC). De plus, les implantations MPI ne supportent pas très bien les contextes massivement multithread. La figure 2.2 décrit le MPC de l'époque.

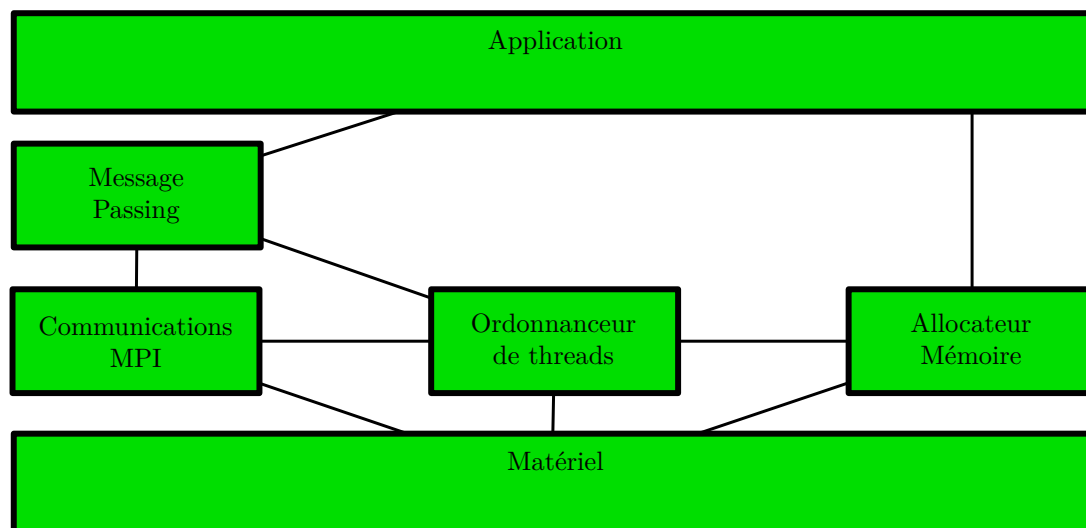


FIGURE 2.2 – MPC en 2006

A l'époque, MPC disposait aussi d'une première version d'allocateur mémoire basé sur le mécanisme des tas locaux. Cet allocateur permettait de maintenir la localité des données mais au prix d'une consommation mémoire élevée comparée à un allocateur standard comme celui de la *libc*. De plus, cet allocateur ne permettait pas d'optimisation fine d'échange de blocs mémoire entre threads en conservant la localité des données. Si cet allocateur était assez bien adapté aux architectures matérielles de 2006, ce n'est pas le cas pour les architectures actuelles et encore moins pour les architectures futures.

2.2.2 Le nouveau MPC

MPC se veut maintenant un support exécutif disposant de toutes les fonctionnalités nécessaires à l'évolution des applications et architectures vers l'Exascale. Pour parvenir à ce but, il a été nécessaire de revoir la modularité de MPC ainsi que les standards proposés. Dans ce document, je ne détaillerai pas les travaux relatifs à OpenMP[76, 27] ou encore de gestion multi-modèles de programmation[25, 99] réalisés conjointement avec Patrick Carribault.

Architecture générale

Un des principaux changements dans l'architecture de MPC est le fait que MPC est maintenant une implémentation MPI, une implémentation OpenMP et une implémentation PThread. Le but de ce changement est de permettre aux applications non modifiées d'utiliser MPC. La figure 2.3 décrit les différents modules présents dans MPC actuellement.

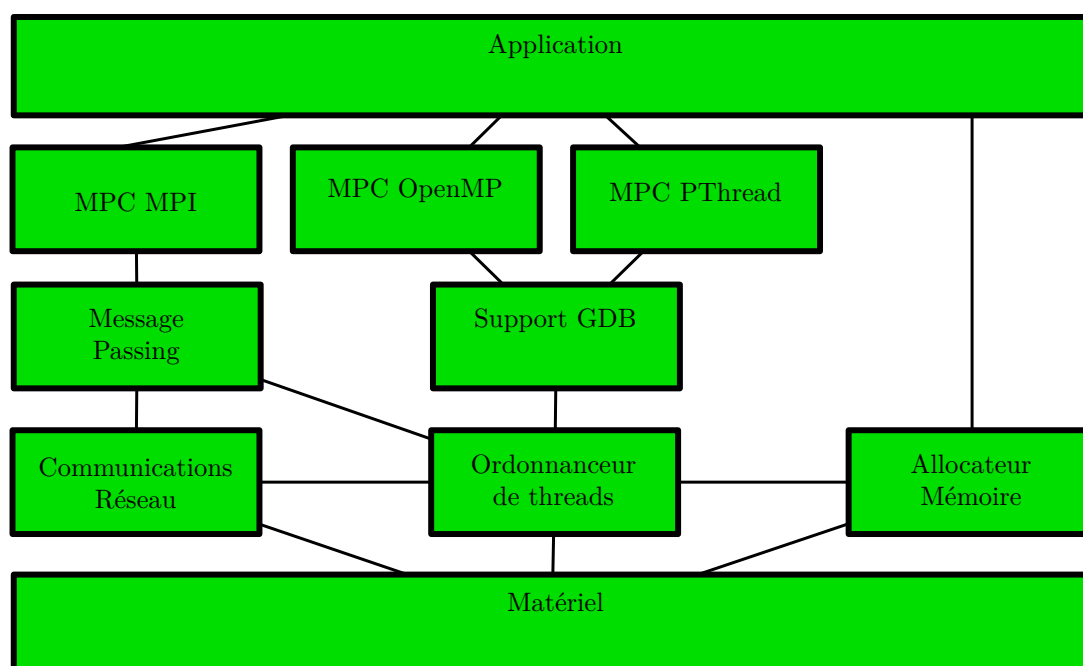


FIGURE 2.3 – MPC aujourd'hui

L'architecture de MPC se décompose en trois grandes parties :

1. Les couches hautes : ces couches sont en charge de fournir les implémentations des standards supportés par MPC. C'est cette partie qui est en lien avec les applications. Nous avons réalisé un effort constant pour être conformes et donc suivre les évolutions des normes. Le but est que toute application MPI, OpenMP et/ou PThread puisse fonctionner au dessus de MPC.

MPC MPI : Ce module est en charge d'implémenter le standard MPI et se base sur le module *message passing*. Actuellement, le support MPI fourni est compatible avec la norme 1.3.

MPC OpenMP : Ce module est en charge d'implémenter le standard OpenMP. Actuellement, le support OpenMP fourni est compatible avec la norme 3.1.

MPC PThread : Ce module est en charge d'implémenter le standard PThread. L'intérêt de disposer d'une implémentation PThread est de pouvoir facilement intégrer/porter des supports exécutifs existant au dessus de MPC. Ceci a par exemple été fait avec TBB, Cilk, UPC.

2. Ces couches intermédiaires fournissent un support générique aux couches hautes.

Message Passing : Ce module est le module historique de communication de MPC. Ce module est en charge des communications inter tâches MPC/MPI ne nécessitant pas une sémantique MPI complexe.

Support GDB : Ce module fournit une implémentation de la `lib.thread_db` nécessaire à la compatibilité avec GDB pour les threads utilisateurs.

3. Ces couches basses sont directement en lien avec le matériel et gèrent les aspects réseau et NUMA.

Allocation mémoire : Ce module est l'allocateur mémoire qui sera détaillé dans le chapitre 3

Communications réseau : Ce module est en charge des commutations via le réseau rapide et sera détaillé dans le chapitre 4.

Ordonnanceur de threads : Ce module est le cœur de MPC. Ce module a la charge de l'ordonnement des threads utilisateurs mais aussi de la gestion de l'équité entre les différents modèles de programmation avec notamment le mécanisme de scrutation intégré à l'ordonnanceur qui permet de ne pas avoir d'attente active. La scrutation intégrée à l'ordonnanceur (fonctionnalité présente dès 2006) est la pierre angulaire du support massivement multithread et multimodèles de programmation de MPC.

Environnement MPC

Fin 2006, MPC était un support exécutif ne fournissant pas toutes les briques nécessaires pour être utilisé de manière intensive ou en contexte de production. Je me suis donc fixé pour objectif de rendre MPC plus robuste tout en faisant de MPC une plateforme de recherches facile à utiliser et à enrichir en particulier pour les doctorants et post-doctorants. En plus des activités de recherche, qui seront décrites plus tard, nous avons mis en place via de la sous-traitance, des stages ou encore des apprentis et ingénieurs, un environnement de débogage de threads utilisateur destiné à la fois à l'utilisateur final ainsi qu'aux développeurs. Nous avons ensuite mis en place un système modulaire qui permet de concevoir de nouveaux algorithmes spécifiques à une ou quelques fonctionnalités tout en conservant MPC totalement fonctionnelle ce qui facilite l'évaluation d'applications réelles. Enfin, nous avons mis en place une procédure de non-régression adaptée au contexte calcul hautes performances. Dans la suite de ce document, nous allons donc décrire succinctement ces développements ainsi que les bénéfices obtenus.

Débogage : Une des premières choses nécessaire à un support exécutif comme MPC est de permettre un débogage aisé aussi bien pour les utilisateurs finaux que les doctorants travaillant sur MPC. L'approche utilisée dans MPC, avec l'utilisation de threads utilisateur, a comme principale contrepartie de rendre difficile, voire impossible, le débogage avec les outils classiques comme GDB ou ses dérivés. Cette limitation ne concerne pas uniquement MPC, mais aussi tous les supports exécutifs à base de threads ou de tâches. En effet, pour pouvoir déboguer dans ce contexte, il est nécessaire que le support exécutif fournisse une implémentation de la `libthread_db` dont l'API a été proposée par SUN. Nous avons donc décidé d'implémenter cette API pour MPC. Comme cette API est lourde à développer, nous avons choisi de mettre en place une approche générique nommée ULDB que nous avons décrite dans [89]. Cette interface permet à tout support exécutif proposant des threads utilisateurs d'avoir un support GDB en implémentant seulement quelques fonctions. Nous avons testé notre approche sur les bibliothèques de threads utilisateurs GNUPth, Marcel et MPC. Pour assurer une plus grande robustesse de cette approche, nous avons mis en place une collaboration industrielle avec la société Allinea au travers du produit DDT. Grâce à cette collaboration, il est possible à partir de DDT 4.3 d'avoir un support complet de MPC. De plus, nous distribuons et maintenons conjointement une version modifiée de GDB pour les bibliothèques de threads utilisateur. Enfin, ces travaux ont grandement facilité le portage d'applications au-dessus de MPC.

Modularité/Configuration : Afin de faciliter l'intégration de nouveaux algorithmes (ordonnement, allocation mémoire, communications, ...), nous avons mis en place un mécanisme de configuration qui permet de remplacer un des composants standards de MPC par une version personnalisée/spécifique via un fichier de configuration XML. Cette méthode permet avec un coût de développement réduit d'évaluer rapidement une nouvelle approche ou de mettre en place des méthodes d'autotuning de MPC en fonction de l'application. Ces modifications se faisant avec un niveau d'abstraction élevé, il n'est pas nécessaire de connaître les détails de fonctionnement de MPC. Toutes ces modifications couches basses n'impactent pas directement les standards proposés par MPC. Il est donc possible à tout moment d'évaluer de nouvelles contributions sur des applications réelles. C'est par exemple cette méthode qui nous permet de gérer facilement les *vraie* qui seront détaillés dans le chapitre 4.1.2.

Validation : Le dernier axe mis en place pour faciliter le travail de nos doctorants/post-doctorants est une batterie de non-régression de plus de 28 000 tests. Le but de cette suite de tests est de permettre aux ingénieurs de maintenir le niveau de fonctionnalité requis pour supporter le maximum d'applications. Ainsi, lors de l'évaluation d'un nouvel algorithme, il est immédiatement possible de savoir quelles sont les applications qui pourront être évaluées. La principale caractéristique de cet

outil est sa bonne intégration au contexte d'un centre de calcul. En effet, son aspect générique lui permet de s'adapter à un grand nombre de gestionnaires de ressources. Cet outil de non-régression a été conçu par un apprenti ingénieur. Cet outil est basé sur un ordonnanceur de tests sous contraintes de dépendances et en lien avec les contraintes imposées par le gestionnaire de ressources (temps d'exécution, ...). Initialement conçu pour MPC, cet outil nommé JCHRONOSS va être étendu et généralisé à d'autres projets au niveau du CEA et mis en OpenSource.

2.3 Environnement d'exécution / machine centre de calcul

Un autre aspect très important est l'environnement des supports exécutifs. Nous avons donc mené des travaux pour une meilleure intégration de MPC et des supports exécutifs en général dans un environnement de type centre de calcul.

2.3.1 Virtualisation

Le premier environnement que nous avons étudié est l'exécution d'un support exécutif en contexte virtualisé. En effet, un nombre croissant de centres de calcul montre un intérêt pour la virtualisation. Ce choix repose essentiellement sur des raisons de facilité d'administration et de flexibilité offertes aux utilisateurs. En effet, les machines virtuelles peuvent permettre de conserver un environnement stable du point de vue utilisateur tout en maintenant une pile logicielle sur l'hôte¹ qui est régulièrement mise à jour. Néanmoins, cet environnement virtualisé dans lequel le support exécutif est amené à s'exécuter peut être très impactant d'un point de vue performances.

Nous nous sommes donc tout d'abord intéressés au problème des communications via le réseau rapide en contexte virtualisé. En effet, les communications réseau ont été particulièrement optimisées pour réduire au maximum la latence. L'utilisation de machines virtuelles peut dégrader ces performances et donc impacter fortement les applications. Nous avons donc conçu un mécanisme permettant de limiter au maximum ces effets indésirables.

Nous nous sommes aussi intéressés à ce que pouvait apporter les machines virtuelles aux concepteurs de supports exécutifs. Dans le cadre des supports exécutifs, il est souvent tentant de passer en espace noyau pour optimiser des appels critiques pour l'exécution en contexte calcul hautes performances. Néanmoins, aucun centre de calcul n'acceptera d'un utilisateur qu'il utilise son propre système d'exploitation modifié. C'est ici que les machines virtuelles peuvent changer la donne. En effet, il est possible sans crainte de problème de sécurité d'avoir un système d'exploitation modifié dans une machine virtuelle. Nous avons donc mené des travaux pour évaluer si l'exécution, en contexte virtualisé, d'optimisation noyau était aussi intéressante que la même optimisation exécutée de manière native.

Les travaux relatifs à la virtualisation seront détaillés dans le chapitre 5.

2.3.2 Profilage à grande échelle

Le dernier aspect que nous traitons dans ce document est l'aspect profilage à grande échelle d'applications massivement multithread. En effet, dans le contexte d'exécution de MPC, il faut des outils de profilage dédiés au calcul hautes performances mais aussi adapté au mode d'exécution de MPC. La majorité des outils de profilage sont basés sur des traces stockées sur le système de fichiers parallèles du super-calculateur. Hors, la ressource entrée/sortie sur les systèmes de fichiers parallèles est de plus en plus critique dans les centres de calcul. Nous avons adressé tous ces problèmes avec une approche de profilage déportée couplée à une analyse in situ.

L'idée maîtresse de ces travaux est de limiter au maximum l'impact sur les entrées/sorties en valorisant au plus tôt les données issues des sondes présentes dans le code. En effet, la valorisation de ces données permet de réduire le volume de données à stocker en fin d'exécution de l'application profilée.

La seconde propriété de notre approche est de déporter ces analyses sur des nœuds dédiés différents de ceux de l'application. Le premier avantage de cette approche est de ne pas impacter l'application en termes de consommation mémoire. L'empreinte mémoire de notre approche est limitée à quelques tampons réseaux utilisés pour stocker les données avant l'envoi sur les nœuds de traitement. Le deuxième avantage est de ne pas requérir en cours de calcul à des ressources processeur sur les nœuds de calcul exécutant le code applicatif. Enfin, cette approche permet aussi, dans le cas d'une machine hétérogène de placer les processus de traitement des données sur les cœurs les plus appropriés à ce traitement.

Les travaux relatifs au profilage à grande échelle seront détaillés dans le chapitre 6.

1. Par hôte, on désigne l'hyperviseur des machines virtuelles.

2.4 Les challenges des supports exécutifs en contexte calcul hautes performances

Une des principales difficultés auxquelles doivent faire face les supports exécutifs est le passage à l'échelle. Par passage à l'échelle, il faut comprendre tout d'abord l'extensibilité du support exécutif lui-même. Ceci passe par une très bonne gestion des aspects contention d'accès aux ressources que ce soit mémoire ou réseau. Mais aussi la gestion d'une topologie complexe (intra-nœud ou inter-nœuds) liée à l'évolution des architectures et qui va de paire avec l'augmentation du nombre de ressources. Le support exécutif doit aussi disposer d'outils à minima aussi extensibles que lui pour le débogage et le profilage d'applications. Comme nous l'avons mentionné plus haut, tout cela doit évoluer dans un contexte de centre de calcul avec ces spécificités comme peut l'être la virtualisation. Enfin, l'extensibilité du support exécutif n'est rien sans l'extensibilité des applications utilisant ce support exécutif. C'est pourquoi, nous nous attacherons à proposer des évolutions facilitant l'extensibilité des applications en contraignant la consommation mémoire du support exécutif par exemple.

Ce document propose et détaille une approche pour permettre l'extensibilité des supports exécutifs.

Chapitre 3

L'allocation mémoire dans le contexte calcul hautes performances

L'allocation mémoire en contexte calcul hautes performances est une problématique qui prend de plus en plus d'importance suite à l'évolution des applications vers le multithreading et celle des architectures vers le multi/many-core. En effet, la mémoire est une ressource centrale critique et partagée entre tous les cœurs/processus/threads d'une application. Dans ce chapitre nous allons décrire les travaux réalisés durant la thèse de Sébastien Valat sur les méthodes d'optimisation de la gestion de la mémoire dans le cadre du calcul hautes performances[102, 101].

3.1 Introduction

L'allocateur mémoire est le mécanisme qui va être en charge de fournir des espaces mémoire à l'application. Il se décompose classiquement en deux parties. La première partie est située en espace utilisateur. On peut citer les allocateurs suivants Dmalloc/Ptmalloc[71, 51] fourni par la libc Linux, Jemalloc[45] fourni par FreeBSD, TCMalloc[94] développé par Google et Hoard[9] un allocateur ayant inspiré FreeBSD. Ces allocateurs servent d'interface entre l'application et le système d'exploitation via les appels système *brk/sbrk* et *mmap/munmap/mremap*. La seconde partie est située en espace noyau. Elle a pour objectif de fournir une page physique à chaque page virtuelle utilisable par l'application.

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 2.1.2. Nous avons utilisé cette application couplée avec MPC en utilisant son mode canonique à base de tâches MPI sous forme de processus légers afin de stresser l'allocateur en parallèle.

L'application Hera est très intensive en terme d'allocation mémoire avec une génération de l'ordre d'un million d'allocations mémoire pour 5 minutes d'exécution sur 12 threads. Les allocations se répartissent en trois groupes principaux (voir figure 3.1). On observe un groupe de petites allocations de courtes durées de vie liées à la structure C++ de l'application. Il 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 effet des échanges systématiques avec le système d'exploitation mettant en exergue les limites de passage à l'échelle de ce dernier, notamment vis-à-vis des fautes de pages.

La table 3.1 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 performances (2 %) alors que Jemalloc offre de bonnes performances couplées à des gains mémoire importants (2.3 Go économisés sur 3.3 Go utilisés par la 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

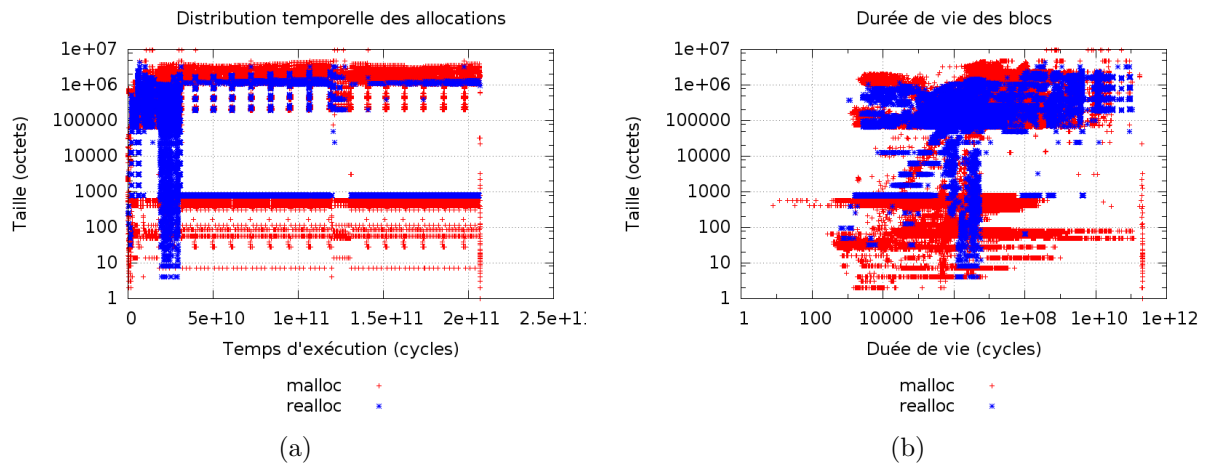


FIGURE 3.1 – Distribution en taille et temporelle des allocations mémoire de l'application Hera sur 12 cœurs pour un problème à 1.4 million de mailles. Sont données, en fonction de la taille des blocs : (a) la répartition temporelle et (b) la durée de vie de ces allocations.

l'on impute au nombre trop important d'échanges avec le système d'exploitation. 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)	Utilisation (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)	Utilisation (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)	Utilisation (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 3.1 – 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.

3.2 Optimisation de l'allocation mémoire en contexte multithread

3.2.1 Espace utilisateur

Comme nous l'avons vu en 3.1, l'allocation mémoire en contexte multithread est sujette en premier lieu au problème de contention lors de l'allocation/désallocation de données. L'allocateur que nous avons développé adresse ce problème particulièrement sur les architectures disposant d'un grand nombre de cœurs¹. En effet, les effets du multithreading se font d'autant plus ressentir que le nombre de cœurs est élevé sur le nœud de calcul.

L'approche retenue dans ces travaux est la conception d'une bibliothèque d'allocation mémoire arborescente. Les travaux précédents sur l'allocation mémoire [45, 51, 9, 94, 65, 10] avaient permis de mettre en évidence que l'utilisation de "tas locaux" permet de réduire significativement la contention au sein de l'allocateur mémoire. Néanmoins, cette approche n'est pas suffisante sur des nœuds de calcul de grandes tailles. En effet les tas locaux peuvent, lors d'allocations massives parallèles, générer de la contention sur les couches systèmes (défaut de page et mise à jour de la table des pages). Pour éviter ce problème, nous avons conçu un algorithme d'allocation arborescent comme illustré sur la figure 3.2 qui permet de hiérarchiser les accès aux ressources partagées et décroître la contention.

Organisation générale

Comme on peut le voir sur le schéma 3.2 notre allocateur se construit sur la base de deux éléments principaux :

Tas locaux : Tout comme Hoard[9], Jemalloc[45] ou TCMalloc[94], 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 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 2Mo. D'une manière générale la source mémoire peut être vue comme un cache placé entre un allocateur standard et le système d'exploitation en surchargeant les fonctions standards mmap/munmap.

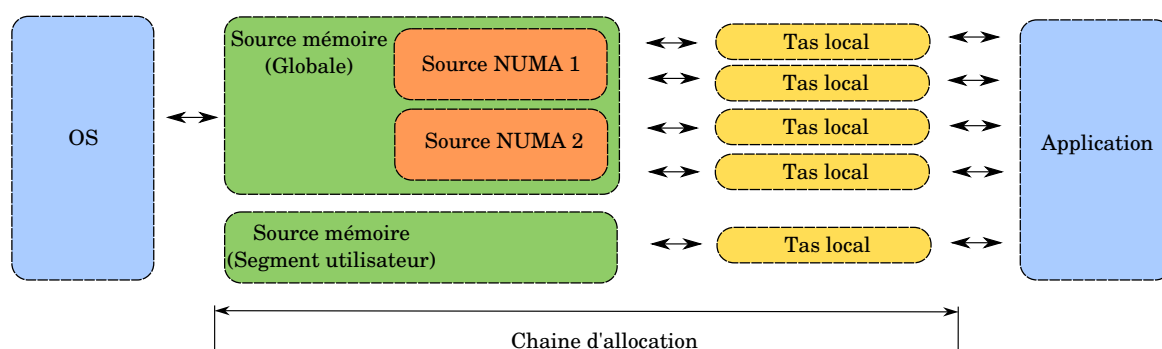


FIGURE 3.2 – Schéma d'organisation générale de l'allocateur faisant intervenir deux composants principaux, des sources mémoire et tas locaux assemblés pour former la chaîne complète d'allocation entre le système d'exploitation et l'application.

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. 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.

1. Les machines TERA 100 et Curie disposent de nœuds dotés de la technologie BCS de Bull qui permet de construire des nœuds de calcul disposant de 16 processeurs (8 cœurs par processeurs dans le cas qui nous intéresse)

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

Le second avantage de cette approche est qu'elle permet l'échange de blocs entre threads directement au niveau de la source mémoire sans nécessiter de passer par un appel système. Cette fonctionnalité est particulièrement intéressante dans le contexte d'un thread-based MPI comme MPC car nous pouvons alors échanger des segments mémoire inutilisés entre tâches MPI.

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 3.3. On remarquera l'emploi de taille de stockage sur la base de *56bits*. Ceci permet de disposer d'informations supplémentaires sur l'en-tête tout en maintenant une taille de *16octets* pour cette dernière en remarquant que les architectures actuelles n'offrent en réalité qu'un adressage physique sur *48bits*. 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.

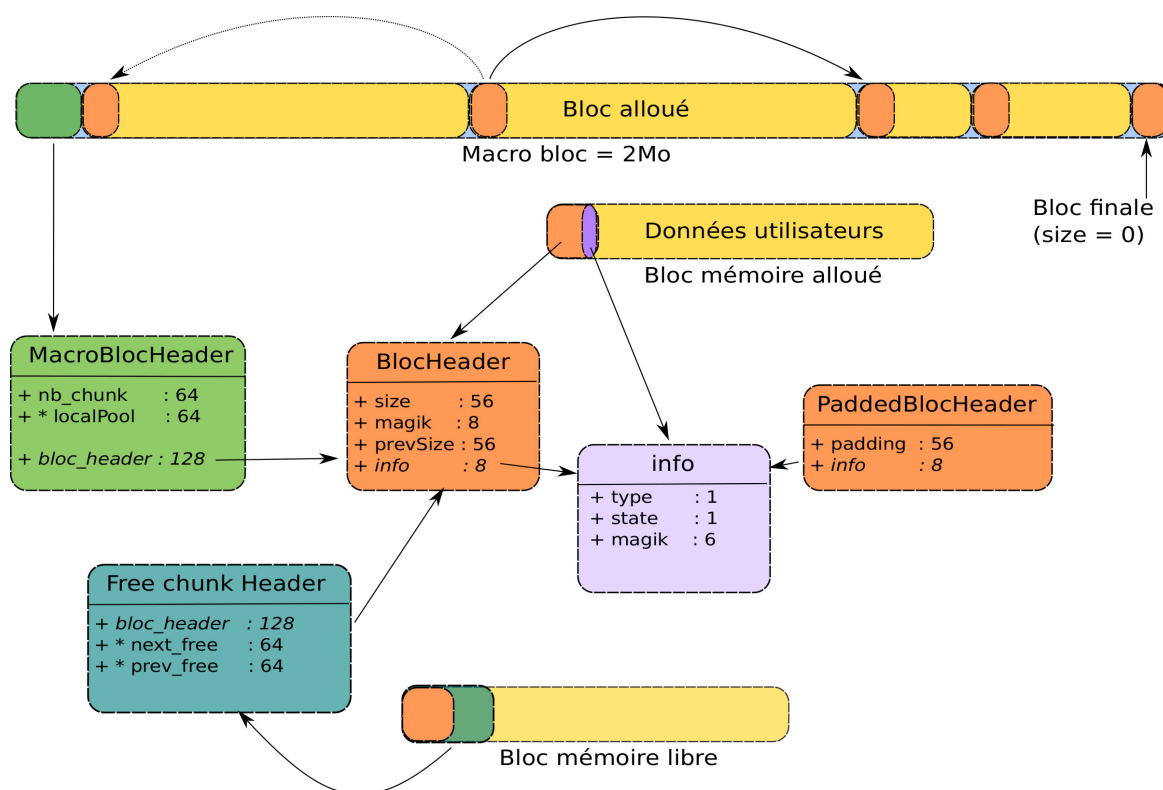


FIGURE 3.3 – Organisation des en-têtes de macro-blocs, blocs alloués et libres. Les tailles des champs sont données en bits considérant l'architecture x86_64.

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 de manière à réutiliser les blocs les plus anciens 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.

Réutilisation des gros segments

Dans le cadre de ces travaux, nous nous sommes surtout concentrés sur la gestion de gros blocs notamment pour limiter les interactions avec le système d'exploitation et éviter les problèmes de contention.

Pour ce faire, la source mémoire constitue elle-même un allocateur en maintenant des 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 comme on peut le voir sur la figure 3.4. Les petites allocations sont essentiellement impactées par le temps de la fonction `malloc`. Les grosses sont majoritairement impactées par les fautes de pages.

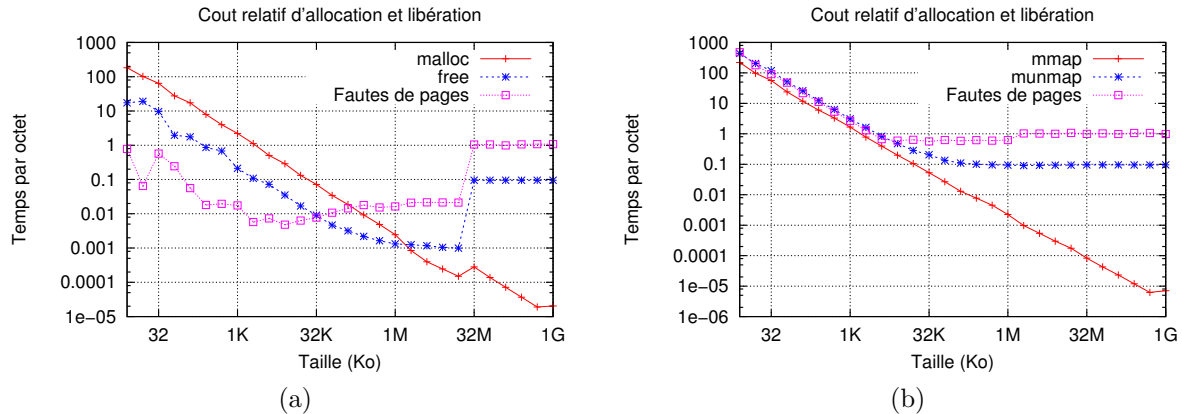


FIGURE 3.4 – Source des différents surcoûts d'allocation en fonction de la taille. L'impact des fautes de pages est mesuré à partir de la différence de temps entre un premier et un second accès aux données en ayant vidé les caches entre temps. Les coûts sont donnés par unité de taille (cycles par octet).

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`. 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 inopinée de la consommation 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`. Cette approche permet d'assurer la réutilisation de segment même pour des tailles très variables.

Avec cette technique d'allocation par source mémoire, il est possible de contrôler finement la contention à chaque niveau de l'arbre, favoriser le "recyclage de pages" entre threads. Néanmoins, cette approche présente quelques défauts :

1. Une augmentation de la latence lorsque l'appel système est nécessaire. Pour résoudre ce problème, nous avons mis en place un système de tampon de macro blocs à chaque niveau de l'arbre pour limiter au maximum le nombre d'appels systèmes. Ceci a deux effets bénéfiques : premièrement, il permet de réduire la contention sur le système. Deuxièmement, cette technique réduit la latence d'allocation mémoire.
2. Une augmentation de la consommation mémoire. La résolution de ce problème sera traitée dans 3.4.

3.2.2 Espace noyau

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 le système d'exploitation. Ces gains sont toutefois obtenus au prix d'une augmentation de la consommation mémoire.

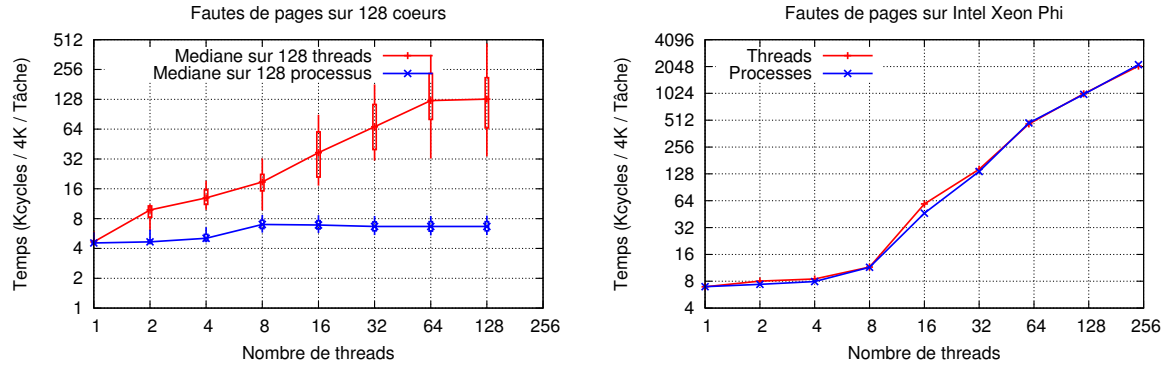


FIGURE 3.5 – Temps des fautes de pages sur les nœuds larges Tera 100 et sur Xeon Phi. Les barres d'erreurs donnent les quartiles 50% et 80%.

Dans cette section, nous allons observer le système lui-même et tenter de déterminer la source du coût des fautes de pages. 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.

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*. On rappelle également que l'utilisation de cet appel fournit à l'utilisateur un segment purement virtuel, ceci, du fait de la politique d'allocation paresseuse. 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 basée sur les threads. Pour cela, on peut construire un micro-benchmark en procédant comme suit :

- création de N threads 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 pour chacune des pages accédées, afin d'obtenir une distribution des temps de fautes de pages.

La figure 3.5, illustre l'extensibilité des fautes de pages sur les nœuds de 128 cœurs de Tera 100 et sur Xeon Phi. Les mesures sont données pour des allocations parallèles en mode thread ou processus. Sur les nœuds de 128 cœurs, la mesure montre une relative extensibilité pour les processus, donc pour des approches type MPI. Un léger effet de contention est observable au-delà de 8 cœurs. Il est sans doute 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 des threads du processus. 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.

Sur Xeon Phi, même observation pour les threads. Toutefois, les processus sont cette fois-ci impactés par le 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 *régions* correspondant aux différents nœuds NUMA. Le Xeon Phi n'est pas vu comme NUMA par le système d'exploitation. 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.

Utilisation de grosses pages

Sur les architectures modernes, il est possible d'utiliser des "grosses pages" (par exemple 2Mo sur X86.64). Ces pages sont habituellement mises en place pour améliorer les performances des TLB. Les gains s'obtiennent notamment par une réduction du nombre de pages nécessaires. Cette réduction est d'un facteur 512 comparé à une base de 4 Ko. On peut alors se demander si cette réduction du nombre

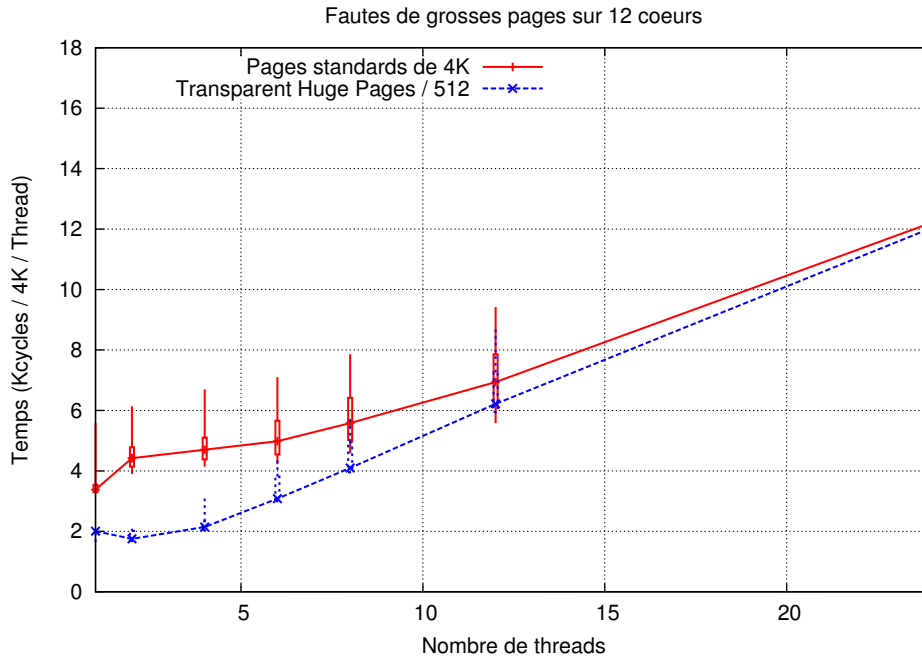


FIGURE 3.6 – Mesure des temps des fautes de pages en utilisant l'implémentation THP de Linux sur les nœuds Bi-Westmere du calculateur Cassard. La mesure est effectuée en accédant à un élément de chacune des pages de 2 Mo. Les temps obtenus sont divisés par 512 pour donner un temps normalisé par segment de 4 Ko.

de pages ne peut pas également réduire la contention que nous observons sur les fautes de pages de LINUX.

Le graphique 3.6 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.

Le problème de la remise à zéro

Lorsqu'une faute de page survient, le système doit fournir une nouvelle page physique. Il est important de noter 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 pages 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 avons remarqué 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, quant à elles, proportionnellement dominées par la remise à zéro avec un impact supérieur à 97%. Dans ce contexte, on comprend bien 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 augmente de manière proportionnelle à la taille des pages et finit par devenir dominante. De plus, dans le noyau Linux, on observe 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 durée 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.

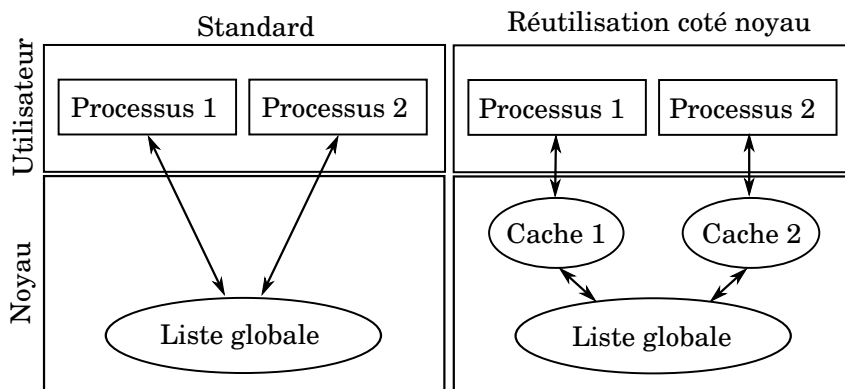


FIGURE 3.7 – Principe de réutilisation des pages au niveau noyau permettant d'éliminer le besoin de remise à zéro du contenu des pages.

Solutions existantes

Nous remarquerons qu'au-delà du coût associé, cette opération implique des transferts mémoire et une purge potentielle d'une partie du cache. Une observation similaire a déjà été faite en espace utilisateur dans une publication [108] en considérant les ramasse-miettes pour des langages tels que Java. Cette étude discute notamment d'un choix de libération différée. Les auteurs analysent également l'utilisation d'accès mémoire 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[93] 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 plus tard.

Notre proposition : la réutilisation des pages

Pour résoudre ce problème, nous nous sommes 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 mémoire de petite taille. On se propose donc de supprimer autant que possible le besoin de remise à zéro. On peut remarquer qu'une page précédemment utilisée par un processus peut sans problème être réutilisée par ce même processus. En effet, elle ne contient que des données qui lui sont propres. Comme le montre la figure 3.7, 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.

Cette approche revient à pousser plus loin la réutilisation que nous avons mise en place au sein de l'allocateur en espace utilisateur 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 :

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 du système d'exploitation, il est difficile voire impossible de réclamer cette mémoire. Avec l'approche noyau, le système d'exploitation peut très facilement réclamer les pages en attente 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 en revanche nécessaire de travailler l'implémentation pour ne pas introduire une contention sur le cache mis en place.

Extension de la sémantique `mmap`/`munmap`

L'approche que nous proposons se base sur une extension de la sémantique POSIX de `mmap`. Par défaut, la fonction `mmap` impose que le segment de mémoire renvoyé soit initialisé à zéro. Il n'est donc pas possible d'inclure notre approche dans cette interface sans une extension de la sémantique. Par défaut, le comportement de `mmap` est maintenu. Nous avons 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.

Listing 3.1 – *Extension de la sémantique de `mmap`*

```
//Allocation standard, segments pre-initialise à zéro
void * ptr = mmap(NULL, SIZE, PROTECTION,
                  MAP_ANON | MAP_PRIVATE, 0, 0);
//Pas de capture pour raison de sécurité si désactive
munmap(ptr, SIZE);

//Allocation sans remise à zéro forcé.
void * ptr = mmap(NULL, SIZE, PROTECTION,
                  MAP_ANON | MAP_PRIVATE | MAP_PAGE_REUSE, 0, 0);
//Capture des pages lors de la libération
munmap(ptr, SIZE);
```

L'approche proposée a été mise en place sous Linux pour les allocateurs MPC et Jemalloc. Les résultats seront présentés en section 3.5.

3.3 Localité des données en contexte multithread

En contexte NUMA la localité des données est primordiale pour tirer la quintessence des architectures. Nous allons maintenant présenter les méthodes que nous avons mise en œuvre pour assurer la localité des données sur architectures NUMA en contexte multithread. Comme précédemment, nous utiliserons une approche duale espace utilisateur/espace noyau.

3.3.1 Gestion de la localité

Comme discuté en introduction, les architectures sont souvent 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 est préférable que l'allocateur sache à l'avance sur quel nœud NUMA 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. Néanmoins, il est possible d'interdire le fait qu'un segment précédemment alloué par un thread d'un nœud NUMA donné ne puisse 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 confiance* 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 tels que [38] proposent des modifications du système pour permettre ce type de suivi. Cela rend toutefois la méthode non portable sur les systèmes conventionnels ne disposant pas de ces mécanismes expérimentaux.

Il est également important de noter que les pages des grands segments sont placées par le système d'exploitation en fonction du premier accès (méthode dit de "first touch"). Ceci pose un problème de confiance sur ce type de blocs 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. 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`.

3.3.2 Maintien de la localité

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` [18] ou de la `LibNUMA` [67]. 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.

Sur les architectures NUMA, 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 3.2.1 que nous maintenions 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.

Un autre outil pour maintenir la localité des données en contexte NUMA peut être l'utilisation de méthode de migration paresseuse des pages qui a donné lieu à un dépôt de conjoint avec la société Bull [77]. Cette méthode utilisée en cours d'exécution ou lors de la migration d'un thread permet de ne migrer que les pages effectivement mal localisées et les rapatrier sur le nœud NUMA sur lequel s'exécute le thread.

3.4 Optimisation de la consommation mémoire

L'empreinte mémoire des applications est assez variable. Il est donc important de pouvoir optimiser le compromis performances/empreinte mémoire en fonction des besoins de l'application. Cette optimisation se doit, de plus, d'être dynamique pour tenir compte des applications aux besoins mémoire variables au cours du temps. Nous allons maintenant détailler les méthodes que nous avons mises en œuvre pour optimiser la consommation mémoire des applications. Comme précédemment, nous utiliserons une approche duale espace utilisateur/espace noyau.

3.4.1 Espace utilisateur

La réutilisation de gros segments permet de limiter les interactions avec le système d'exploitation 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 est d'autant plus vrai dans un contexte où la puissance de calcul tend à croître plus vite que les capacités de stockage mémoire. On remarquera toutefois que certaines applications n'utilisent pas toujours toute la mémoire du nœud et vont faire apparaître des pics de consommation lors de certaines étapes de calcul. Il est donc intéressant de proposer une approche dynamique permettant d'ajuster le compromis performance/consommation mémoire avec comme objectif permettre à l'application de fonctionner le plus rapidement possible.

Ce type d'adaptation dynamique est envisageable dans un contexte calcul hautes performances où l'utilisateur est en général seul sur un nœud et vise à exploiter au mieux les ressources. Pour atteindre cet objectif, nous avons construit notre implémentation de façon à offrir deux politiques extrêmes et pouvoir contrôler un gradient entre ces dernières à l'aide de paramètres. Pour permettre cela, il nous a fallu obtenir une capacité à économiser au mieux la mémoire au niveau des tas locaux. Dans ce contexte, 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 de garder ou non les macro-blocs en transit. Cette politique est réversible puisque les macro-blocs en attente peuvent être libérés brutalement en cas de pic de consommation.

Dans le cadre de la thèse de Sébastien Valat, nous avons mis en œuvre les mécanismes de base de cette adaptation à la demande. L'utilisateur peut en cours de calcul modifier le taux de libération de macro-blocs, mais il manque un modèle de coût pour faire cela de manière automatique. Une extension de ces travaux serait donc la mise en place d'un superviseur mémoire capable de détecter les seuils critiques de consommation mémoire et de déclencher le changement de politique. Ce superviseur pourrait en outre être aussi connecté à d'autres modules comme le module réseau qui dispose lui aussi de plusieurs politiques de consommation mémoire (voir section 4.3).

3.4.2 Espace noyau

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.

L'idée maîtresse

L'idée principale de KSM (Kernel Same Page Merge) est d'utiliser le concept de *copie sur écriture*. Ce concept permet 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*³. 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 moment 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.

Test sur Hera

KSM a été testé sur Hera afin d'évaluer les gains qu'il peut apporter en comparaison des *chunks* (voir 3.5). 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.

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

3. 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.

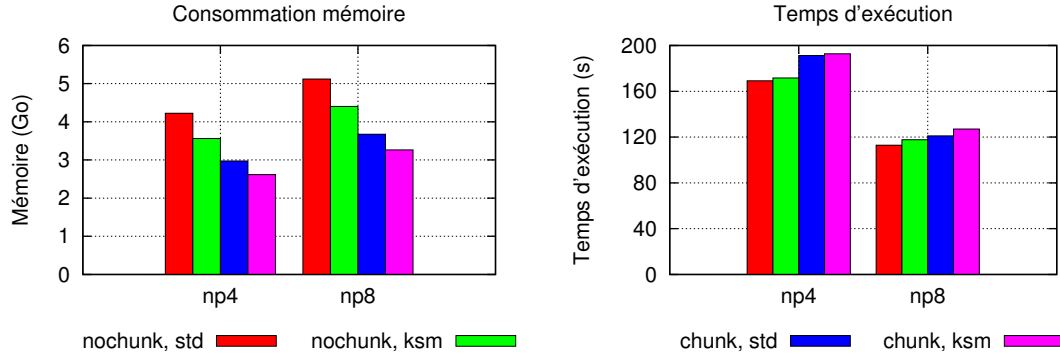


FIGURE 3.8 – Gains observés avec KSM sur une exécution d'Héra utilisant 4 ou 8 processus MPI sur 8 cœurs disponibles avec 2.3 millions de mailles. La mémoire utilisée est donnée sous la forme d'une moyenne sur l'ensemble de l'exécution.

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.

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 3.8 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.

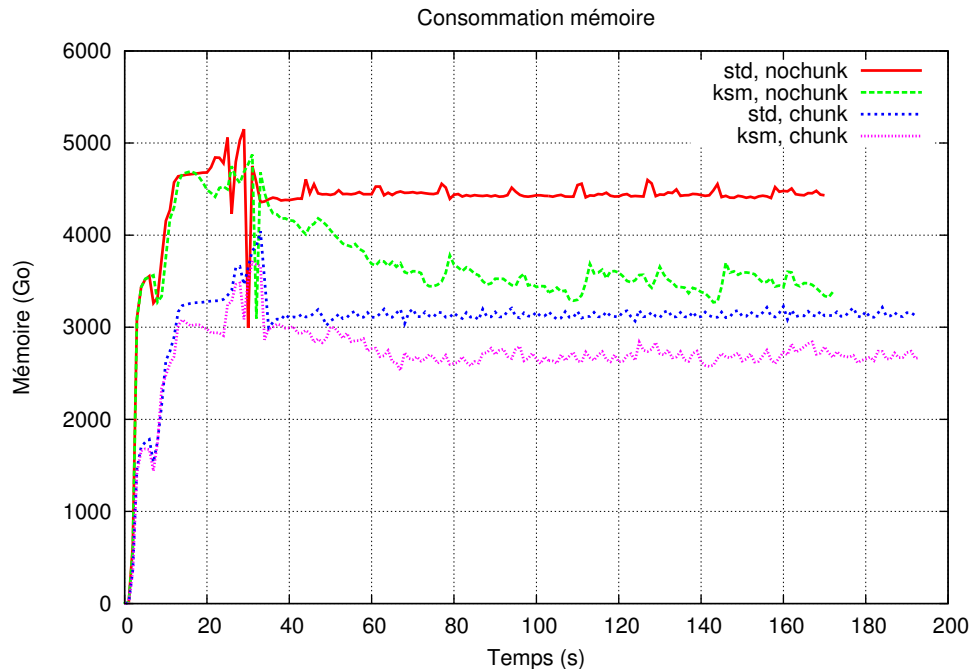


FIGURE 3.9 – Détail de l'utilisation de la mémoire au cours du temps de Hera sur 4 cœurs.

On remarque sur la figure 3.9 que les gains mémoire 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 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.

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.

Limites 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 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 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 de 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 tâches ne tentent pas de 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.

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.

Réduction des 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.

3.5 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.

Hera est une plateforme de simulation multi-physique multi-matériaux 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 mémoire (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 présents 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 *chunks*) 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.

Les résultats sont présentés dans la table 3.3. Ces essais sont réalisés avec trois profils de configuration pour notre allocateur. Ces trois profils permettent de décorréliser 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 mémoire 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 mémoire.

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 du système d'exploitation. En comparant ces résultats au mode UMA, nous pouvons extraire l'impact de l'interaction avec le système d'exploitation. 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 le système d'exploitation 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.

Noyau modifié et pages standards de 4 Ko (S)						
	Allocateur	Noyau	Total	Utilisateur	Système	Mémoire (go)
1	<i>MPC-NUMA</i>	Original	<i>135.14</i>	<i>132.63</i>	<i>1.79</i>	<i>4.3</i>
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	<i>MPC-NUMA</i>	Original	<i>137.89</i>	<i>135.13</i>	<i>1.86</i>	<i>6.2</i>
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 3.2 – 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.

Les résultats obtenus avec notre modification du noyau sont donnés dans la table 3.2 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 est impacté par une réduction de 33%, correspondant aux gains observés sur le micro-benchmark précédent. Sur les grosses 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, quant à 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.

3.6 Conclusion/travaux futurs

3.6.1 Conclusion

Durant les travaux de thèse de Sébastien Valat, nous avons mis en place un allocateur mémoire optimisé pour le contexte multithread. Cet allocateur est aussi adapté au contexte NUMA sans nécessiter d’interaction avec le système d’exploitation (contrairement à l’allocateur de 2006). Nous sommes donc passés d’un allocateur optimisé pour un thread-based MPI comme MPC à un allocateur conçu pour tout les types de multithreading. La particularité de cet allocateur est aussi de limiter au maximum les interactions avec le système d’exploitation. En effet, au sein d’un même processus, de nombreuses structures sont partagées dans le noyau. Limiter les interactions avec le système, c’est donc limiter la contention sur les structures. Une des actions du noyau critique en termes de performances et de contention, est la remise à zéro du contenu des pages mémoire. Pour ce phénomène en particulier, nous avons proposé une approche duale en espace utilisateur et en espace noyau.

Nous avons aussi porté une attention toute particulière à la consommation mémoire. En effet, le passage à l’échelle des supports exécutifs et des applications font qu’il est nécessaire d’être très vigilant sur ce point. C’est pourquoi, nous avons mis en place une politique d’ajustement dynamique de l’empreinte mémoire de l’allocateur. Nous avons aussi évalué avec attention les méthodes de fusion des données identiques (par exemple KSM) pour offrir aux applications le maximum d’espace. Ces dernières techniques bien que prometteuses, ne sont pas tout à fait adaptées au contexte calcul hautes performances. Nous avons donc fait une analyse des limitations de cette approche.

La mise en place de toutes ces méthodes nous a permis d’avoir un allocateur flexible où le compromis performances/consommation mémoire est paramétrable par l’utilisateur. Notre allocateur peut donc au choix être le plus rapide, et donc améliorer significativement les performances d’applications réelles

A : Nœuds 12 cœurs Cassard (2 * 6)					
	Allocateur	Total (s)	Allocateur (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)	Allocateur (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)	Allocateur (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 3.3 – 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 cano- nique 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.

(accélération d'un facteur $\times 2,37$ par rapport à l'allocateur standard sur des nœuds 128 cœurs), ou l'un des plus économes au prix d'une dégradation des performances.

3.6.2 Travaux futurs

Les travaux réalisés jusqu'ici n'ont pas encore permis d'adaptation dynamique *automatique* du compromis performances/consommation mémoire. Ce point serait donc à évaluer avec la mise en place d'un modèle de coût. Ce dernier serait en charge d'ajuster l'empreinte mémoire de l'allocateur mémoire en fonction de la consommation réelle de l'application. L'idée serait de fournir le maximum de performances tant qu'il y a de l'espace mémoire disponible et de réduire la consommation mémoire et donc les performances dans le cas où l'application serait en butée sur la mémoire disponible sur le nœud. La priorité serait : "il faut que l'application puisse passer". Ce type de politique peut aussi être étendue à une gestion efficace de la consommation mémoire. En effet, si une architecture est capable de réduire la consommation mémoire sur les bancs mémoire non utilisés, on peut envisager notre politique de réduction de l'empreinte mémoire comme une réduction de la consommation énergétique.

Le bénéfice de ces méthodes de réduction de l'empreinte mémoire peut être amplifié par les méthodes de factorisation des données communes. Néanmoins, pour que ces méthodes puissent être utilisées en contexte calcul hautes performances, il est nécessaire de les faire évoluer pour être adaptées aux machines de type calcul hautes performances et donc de paralléliser ces méthodes.

Enfin, comme nous le verrons dans la suite du document, il est nécessaire que tous les composants du support exécutif interagissent pour réduire l'empreinte mémoire. Il serait donc nécessaire de mettre en place une politique globale de gestion mémoire, en particulier allocateur mémoire et tampon réseau, pour s'adapter automatiquement et dynamiquement. Cette politique devra proposer aux applications les meilleures performances en fonction du contexte d'exécution.

Chapitre 4

Communication réseau rapide en contexte multithread

Les travaux présentés dans ce chapitre ont pour objectif d’optimiser les performances des supports exécutifs multithread sous contrainte de ressource mémoire en contexte calcul hautes performances. Nous allons porter une attention particulière à l’impact du multithreading sur les communications via les réseaux rapides avec les aspects contention et localité NUMA. Nous verrons ensuite comment le contexte multithread peut aider à la mise en place de méthode de recouvrement des communications par du calcul. Enfin, nous présenterons une méthode d’ajustement dynamique du ratio performances/consommation mémoire.

Les travaux présentés dans ce chapitre résument les travaux de thèse de Sylvain Didelot[39].

4.1 Gestion des accès réseau en contexte massivement multithread sur nœuds de grosse taille

Comme nous l’avons vu en 1.2, les applications utilisent de plus en plus le multithreading pour diminuer la consommation mémoire des applications, mais aussi pour faciliter l’équilibrage de charge. Dans cette section, nous allons décrire comment les supports exécutifs doivent s’adapter pour tenir compte du multithreading sur nœuds de grosse taille. Durant cette section, nous illustrerons nos propositions sur le “Thread-based MPI” MPC.

4.1.1 Impact du multithreading sur les performances du réseau

Dans un contexte multithread, les tâches devant communiquer par le réseau se partagent l’accès aux ressources aussi bien de manière physique (comme dans le contexte MPI-pur) mais aussi de manière logicielle. C’est ce dernier point qui est spécifique au contexte multithread. Dans ce contexte, il est nécessaire de protéger les structures réseau via des verrous[74]. La figure 4.1 illustre le phénomène de contention sur des tests élémentaires de bande passante MPI. Comme nous pouvons le voir sur cette figure, la contention sur une unique structure en contexte InfiniBand participe à l’augmentation de la latence de 20% sur les messages de petite taille. Une autre limitation spécifique au contexte multithread est le nombre maximum d’entrées (pages mémoire) que peut gérer la file de messages (Queue Pair en contexte Infiniband) à un instant donné. Ce type de mémoire est dimensionné pour une utilisation en contexte mono-thread (15 000 entrées pour une carte ConnectX-2 Mellanox). De plus, le nombre de cœurs par nœud augmentant avec les années, il est fréquent d’avoir plusieurs cartes réseau par nœud de calcul [22, 72]. Dans un contexte MPI-pur, ces cartes sont réparties entre les processus MPI. En contexte multithread, utiliser une seule carte est alors une approche sous-optimale. Enfin, il a été démontré dans [47] que même dans un contexte mono-carte, l’utilisation de plusieurs files de messages permet d’augmenter les performances du système.

4.1.2 Contribution : Multi-Threaded Virtual Rails

Dans le cadre de ces travaux, nous avons introduit le concept de “rail virtuel” (*vrail*). Un *vrail* est une représentation abstraite des ressources réseau utilisées pour communiquer entre processus. Plus

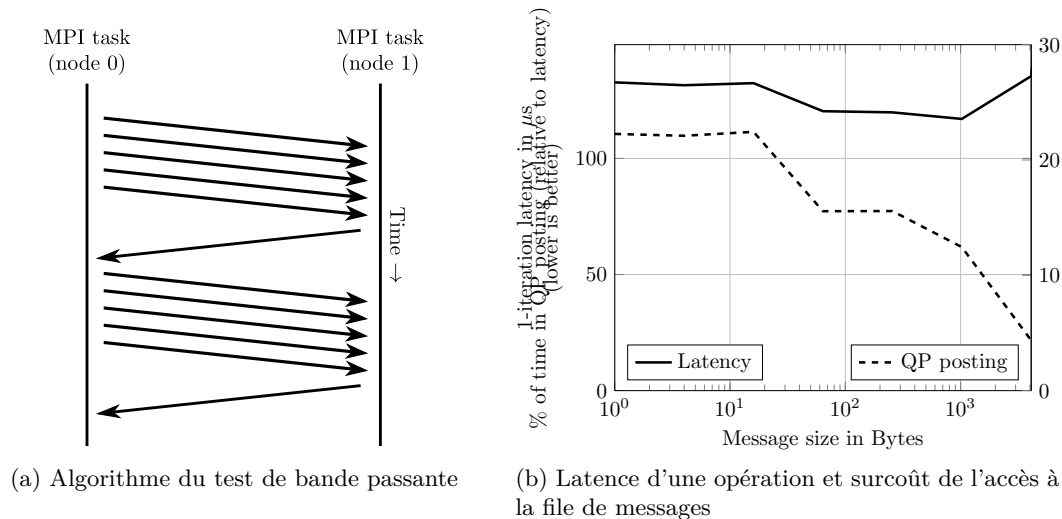


FIGURE 4.1 – Micro-benchmark de bande passante MPI en contexte MPC avec deux nœuds de calcul et 16 tâches MPI avec le protocole *eager* spécifique aux messages de petite taille. Chaque tâche MPI communique avec une tâche sur le nœud distant comme décrit sur la figure (a). La figure (b) illustre la contention sur les accès aux files de messages Infiniband.

précisément, un *vrail* est composé de :

- Une **configuration** : décrit la configuration du *vrail* en termes de nombre de tampons ainsi que leurs tailles.
- Un **device** : définit quel device réseau est utilisé (par exemple Infiniband) ainsi que le port à ouvrir.
- Des **Structures réseau** : structures spécifiques au réseau sous-jacent (QPs, SRQ et CQ pour Infiniband).
- Un ensemble de **tampons réseau** en réception et émission.
- Un **protocole de routage** : définit dans quelles condition doit être utilisé le *vrail*.

Le concept de *vrail* est conçu pour être aussi modulaire que possible. Les *vrails* peuvent être utilisés pour agir sur la contention multipliant les structures réseau. Les *vrails* peuvent aussi être utilisés pour gérer efficacement le multirail dans le cas de machines avec plusieurs cartes réseau. Tous les paramètres des *vrails* sont gérés au travers du fichier de configuration décrits en 2.2.2.

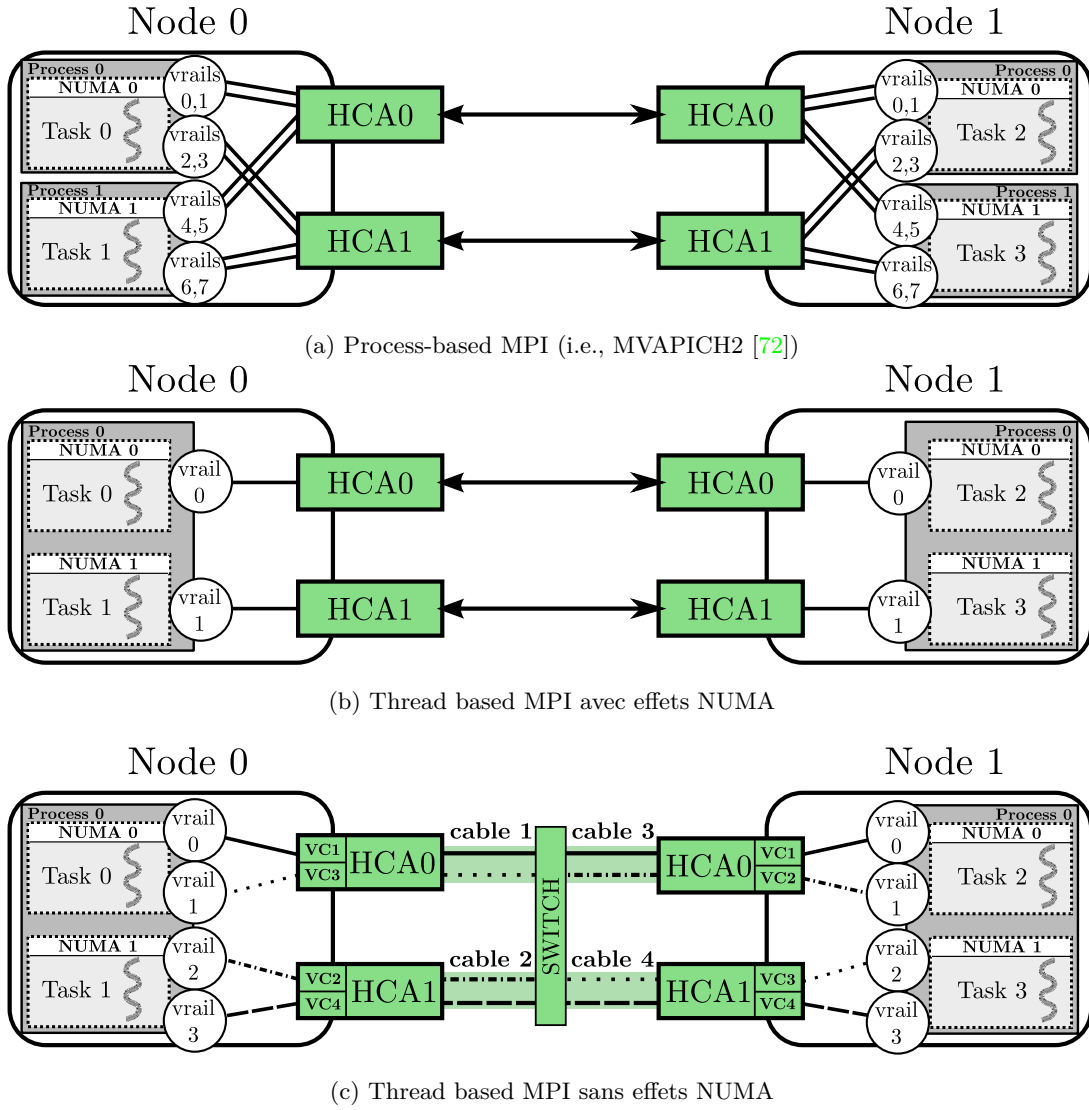
Gestion du multirail

Avec les technologies NUMA comme l'Hyper Transport (AMD), le QPI (Intel) ou le BCS (Bull), il est possible d'aggréger de manière logique un grand nombre de processeurs au sein d'un seul nœud de calcul. Néanmoins, l'utilisation de ces technologies a introduit des accès non uniformes aux entrées/sorties ; c'est ce que l'on nomme les effets NUIOA[78]. Les effets NUIOA complexifient l'optimisation des communications sur des architectures multirail disposant de plusieurs cartes réseau comme les nœuds larges de Curie.

Les aspects multirail ont largement été étudiés dans la littérature. De nombreuses politiques ont été étudiées comme par exemple :

1. *round-robin* : La carte réseau à utiliser est choisie suivant un ordre circulaire entre toutes les cartes réseau disponibles.
2. Découpage de message pondéré ou non pondéré : le message est découpé en multiples fragments répartis sur les différentes cartes réseau. La taille des fragments de message peut être ajustée en fonction des effets NUIOA de l'architecture sous-jacente (méthode pondérée)[79, 72].
3. Routage en fonction de la contention : la carte réseau la moins chargée est choisie pour l'émission du message[5].

Sur certains aspects, notre conception de *vrail* est proche de la notion de *virtual subchannels* décrite dans [72] où chaque tâche MPI peut disposer de plusieurs liens (ou ports) vers les cartes réseau. Comme nous pouvons le voir figure 4.2(a), il est possible de créer un ensemble de *virtual subchannels* par tâche MPI. Néanmoins, il est impossible pour deux tâches MPI de partager des *virtual subchannels*.

FIGURE 4.2 – Estimation du nombre de *vrails* (ou *virtual subchannels*) dans une configuration multirails.

Dans le cas des thread-based MPI, tous les *vrails* sont accessibles par n'importe quelle tâche MPI sur le même nœud. Sur la figure 4.2(b) nous créons un unique *vrail* par carte réseau au sein d'un nœud de calcul. Ainsi, il est possible d'avoir accès à la totalité des capacités réseau disponibles. Avec cette configuration, les tâches 0 ou 1 peuvent indifféremment utiliser les *vrails* 0 ou 1 pour envoyer leurs messages. Notre approche permet donc de réduire le nombre de *endpoints*¹ d'un facteur égal au carré du nombre de cœurs sur le nœud de calcul par rapport à l'approche présentée en 4.2(a). Le principal problème de cette approche survient lorsque la tâche MPI destinatrice est localisée sur un identifiant de nœud NUMA différent de l'identifiant du nœud émetteur. Ce cas de figure est décrit dans la figure 4.2 où la tâche 0 communique avec la tâche 3. De plus, comme les buffers nécessaires aux communications sont souvent situés sur le nœud NUMA contenant la carte réseau, il y a aura nécessairement des accès distants à la mémoire par la carte réseau.

En contexte mémoire partagée, une conception alternative est présentée figure 4.2(c). Dans ce design, le support exécutif interconnecte de manière croisée les différentes cartes réseau. Cette approche ne met plus en évidence d'effet NUMA comme sur la figure 4.2(b) mais nécessite autant de *vrails* que de cartes réseau par nœud. De plus, cette approche ne permet pas dans tous les cas de garantir le débit maximal de la configuration. En effet, dans certaines configurations, une seule carte réseau peut être utilisée en émission pour deux cartes réseau en destination. Dans ce cas, la bande passante maximale disponible est divisée par deux.

1. endpoint : connections point à point entre deux processus MPI.

La consommation mémoire étant directement proportionnelle au nombre de *vrails* par nœud, la consommation mémoire minimum est obtenue avec un seul *vrail* par nœud de calcul. Néanmoins, il a été montré en 4.1.1 que dans cette configuration les performances étaient fortement dégradées. Il est donc logique de penser que le fait d'avoir un *vrail* par carte réseau est un compromis consommation mémoire/performances équilibré pour les raisons suivantes : premièrement, cette configuration permet d'avoir un débit réseau maximal car toutes les cartes réseau du nœud sont utilisées. Deuxièmement, le support exécutif peut optimiser les accès NUIOA aux cartes réseau en utilisant les *vrails*. Enfin, cette approche permet de réduire significativement le nombre de *endpoints* avec un unique *vrail* par nœud NUMA alors que la majorité des implémentations MPI utilisent un *endpoint* par tâche MPI. Dans le reste de ce document, nous nous focaliserons donc sur cette configuration (voir figure 4.2(b)).

Stratégie d'envoi des messages

Dans le contexte un *vrail* par nœud NUMA, deux stratégies de routage sont possibles. La première consiste à choisir le *vrail* en fonction de la localisation de l'émetteur (stratégie *Sender-Driven* présentée sur la figure 4.3(a)). La seconde méthode consiste à avoir une politique basée sur la localisation du destinataire du message (stratégie *Receiver-Driven* présentée sur la figure 4.3(b)).

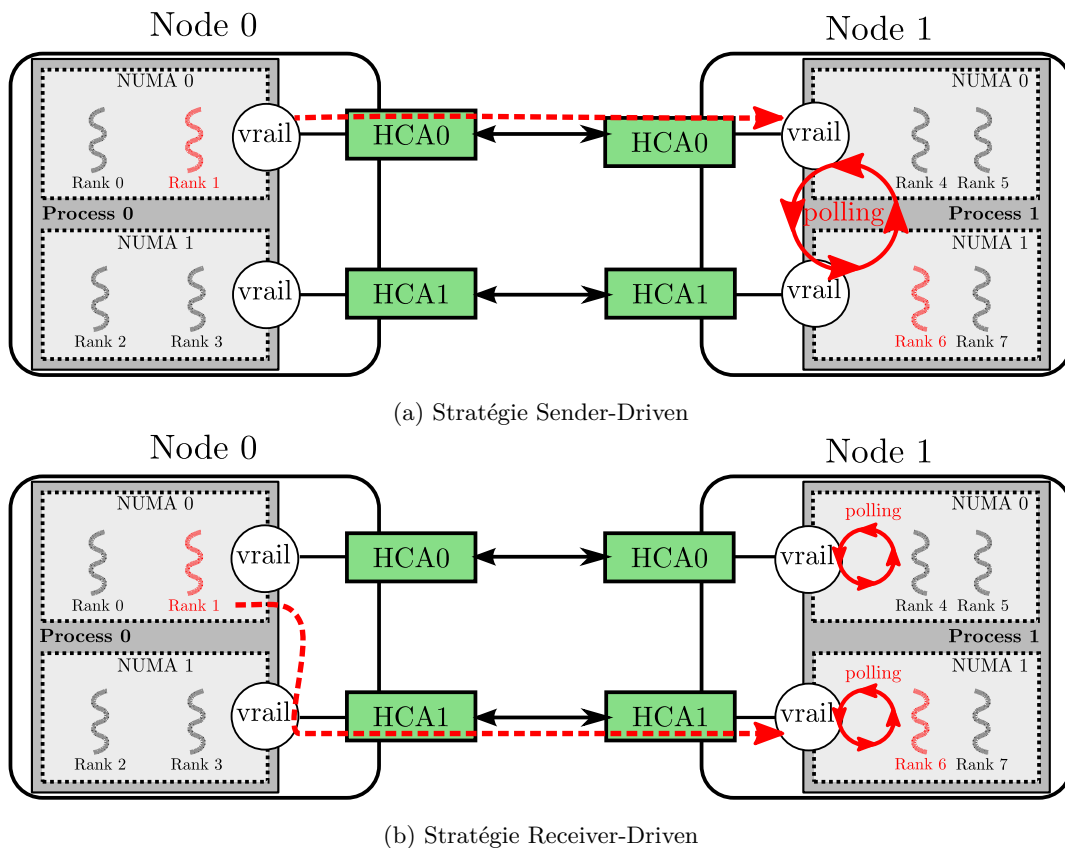


FIGURE 4.3 – Comparaison des stratégies de sélection du *vrail*. Dans les deux cas, un message est envoyé de la tâche MPI 1 vers la tâche MPI 6.

La politique dirigée par l'émetteur est l'approche la plus facile à mettre en œuvre. Lorsqu'une tâche MPI (par exemple la tâche 1 sur la figure 4.3(a)) doit émettre un message, elle va utiliser le *vrail* localisé sur son nœud NUMA. Du côté récepteur, la tâche MPI destinataire (tâche 6 dans notre exemple) va devoir scruter tous les *vrails* car le message peut arriver par n'importe quel *vrail* (cas du `MPI_ANY_SOURCE` par exemple). Bien que le *vrail* utilisé en émission minimise les effets NUIOA, du côté récepteur l'activité NUMA est élevée du fait de la scrutation de tous les *vrails*.

La politique dirigée par le récepteur est plus compliquée à mettre en œuvre mais permet une limitation du trafic NUMA dans toutes les configurations. Avec cette approche, la tâche MPI 1 sur la figure 4.3(b) choisit son *vrail* en fonction de la localité du destinataire. De cette façon, le tâche 6 n'a qu'un seul *vrail* local à scruter. Avec cette configuration, le trafic NUMA est localisé au niveau de l'émetteur à l'émission du message.

4.1.3 Gestion des tampons réseau en contexte Multithread

Les tampons réseau sont les plus petites entités utilisées pour communiquer des données entre tâche MPI. Ces tampons sont utilisés quel que soit le protocole (*eager*, *rendez-vous*, ...). Comme ces tampons sont très fréquemment accédés, il est très important d'optimiser leur utilisation. Ces tampons se présentent souvent sous la forme de listes de type *FIFO*² centralisées protégées par des verrous.

La première optimisation mise en œuvre est la répartition des tampons réseau d'émission sur les différents nœuds NUMA. Cette optimisation permet de limiter la contention sur l'accès aux listes. Il permet aussi d'éviter les accès NUMA lors de la prise des verrous relatifs aux listes. La seconde propriété de cette optimisation est de permettre une copie des données locales au nœud NUMA dans le cas du protocole *eager* par exemple.

La seconde optimisation est liée à la réception des messages. Contrairement à l'émission, il est plus intéressant de disposer d'une seule et unique liste par *vrail* localisée sur le nœud NUMA le plus proche de la carte réseau. Le principe de cette liste est le suivant : les éléments de la liste sont utilisés par la carte réseau et libérés par les couches hautes une fois la réception du message effectuée. Il se pose alors le problème de la réintroduction des éléments ainsi libérés. Pour limiter la contention lors de la réintroduction, un cache local à chaque nœud NUMA est utilisé. Ainsi les éléments libérés sont d'abord réintroduits localement un par un et ensuite réintroduits par paquet dans la liste globale. Cette méthode permet de limiter le nombre de prises de verrous globaux.

Dans le but de réduire l'empreinte mémoire de l'approche, les tampons en émission et réception sont réduits à leur minimum au démarrage de l'application.

4.1.4 Évaluation de la méthode

Dans cette section, nous allons évaluer les performances et l'impact sur la consommation mémoire de notre approche de *vrails*. Nous commencerons notre évaluation par des micro-benchmarks multitrails et monorails. Ensuite, nous illustrerons notre méthode sur une application scientifique réelle nommée Athena[96].

Évaluation sur micro-benchmarks

Dans le but d'évaluer les performances ainsi que la consommation mémoire de notre approche, nous avons choisi d'utiliser le benchmark AllToAll issu de la suite IMB[58]. Ce test demande à toutes les tâches MPI de réaliser des communications avec toutes les autres tâches MPI. L'architecture cible pour ces évaluations est la partition Curie nœuds larges disposant de la technologie BCS de Bull (voir 2.1). Ces nœuds disposent de 4 cartes réseau, 2 niveaux NUMA, 16 processeurs pour un total de 128 cœurs. La figure 4.4 détaille les différentes configurations de *vrails* possibles.

La première série d'expérimentation utilise les configurations (b) et (c) de la figure 4.4 en utilisant donc les quatre cartes réseau par nœud. La seconde série utilise une seule carte réseau mais un ou plusieurs *vrails* pour illustrer le phénomène de contention.

Évaluation multitrails

La figure 4.5 présente les temps d'exécution du benchmark IMB[58] AllToAll pour les implémentations IntelMPI[59], Adaptive MPI(AMPI)[53, 54] et MPC et des tailles de messages de 1Mo. Le nombre de cœurs est soit 256 (2 nœuds de calcul) ou 512 (4 nœuds de calcul). Pour IntelMPI et AMPI, nous avons choisi la configuration offrant le meilleur niveau de performances.

Comme on peut le voir sur la figure 4.5(a) l'utilisation de multiples *vrails* et de multiples cartes réseau permet d'augmenter sensiblement les performances du support exécutif multithread MPC. Cette amélioration des performances se décompose en deux parties. Tout d'abord, l'utilisation de plusieurs cartes permet d'augmenter le débit global en sortie du nœud de calcul et de diminuer la contention (configuration 4 *vrails* 4 HCA vs 1 *vrail* 1 HCA). Ensuite, l'augmentation du nombre de *vrails* seuls (configuration 4 *vrails* 4 HCA vs 16 *vrails* 4 HCA) permet de réduire encore d'un cran la contention.

La figure 4.5(b) illustre l'aspect consommation mémoire de notre approche. Comme on peut le voir, notre approche réduit de manière significative l'empreinte mémoire de notre benchmark par rapport aux autres implémentations MPI. Cette réduction n'est pas seulement due au fait que MPC est un thread-based MPI car AMPI est aussi un thread-based MPI. Comme nous pouvons le constater, cette réduction de l'empreinte mémoire a pu être réalisée sans impact significatif sur les performances.

2. Files de type First In First Out

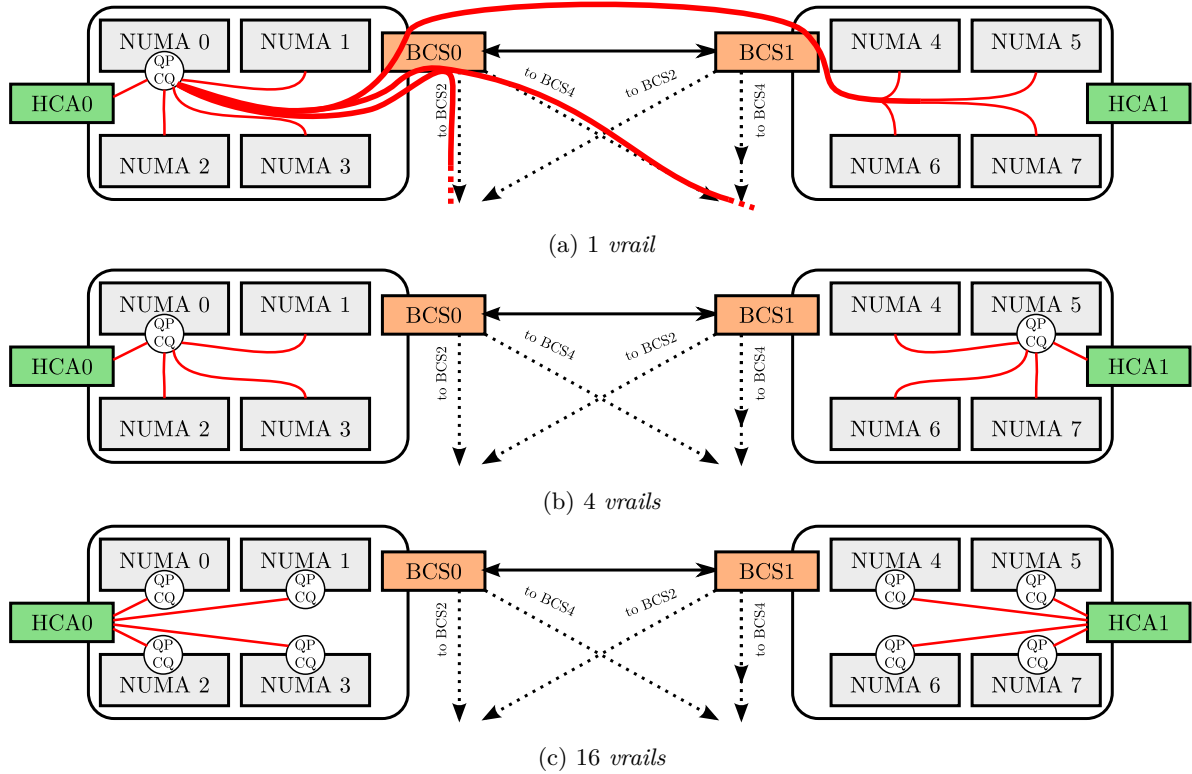


FIGURE 4.4 – Il y a trois configurations possibles pour une couche de communications multithread sur les nœuds 128 cœurs. La figure (a) utilise une carte réseau par nœud. Dans la figure (b), nous utilisons un *vrail* par nœud NUMA de niveau 1. La figure utilise un *vrail* par nœud NUMA de niveau 2 (un *vrail* par processeur).

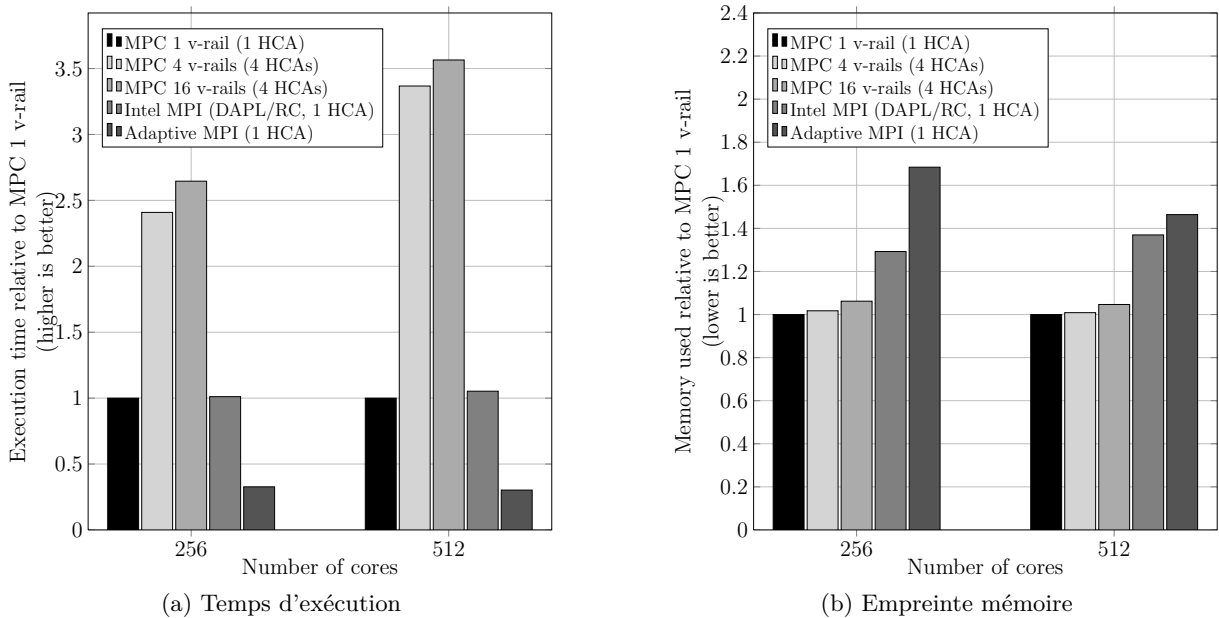


FIGURE 4.5 – Temps d'exécution et consommation mémoire sur le benchmark IMB AllToAll. Toutes les valeurs sont normalisées par rapport à MPC 1*vrail* 1 HCA.

Évaluation monorail

L'évaluation monorail vient compléter l'évaluation multirails et se focalise sur la contention d'accès aux listes de tampon réseau. La figure 4.6 illustre sur le même cas test AllToAll IMB et pour des tailles de messages de 1 octet (figure 4.6(a)) et 1Mo (figure 4.6(b)). Comme on peut le voir, l'utilisation de

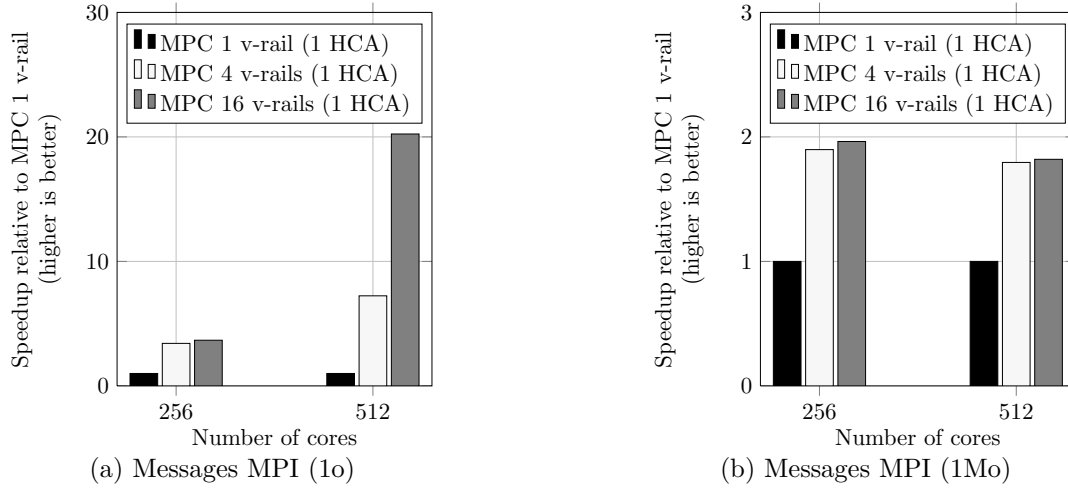


FIGURE 4.6 – Temps d'exécution sur le benchmark IMB AllToAll monorail

multiples *v-rails* permet d'obtenir un speedup d'un facteur 20 pour les messages de petite taille.

Les évaluations menées sur des micro-benchmark montrent que les performances et la consommation mémoire peuvent être optimisées en contexte multithread grâce au mécanisme de *v-rails* que nous avons introduit. Nous allons maintenant confirmer cela sur une application réelle.

Évaluation de l'application Athena

Athena est un code d'astro-physique et de magnéto-hydrodynamique (MHD)[96]. Ce code est écrit en C et dispose d'une parallélisation MPI. Nous avons choisi cette application car elle est extensible jusqu'à 25 000 cœurs³ et compatible MPI 1.3. Nous avons choisi de résoudre l'instabilité 3D Rayleigh-Taylor sur les nœuds larges de Curie et de mener une étude d'extensibilité faible de 256 à 6144 cœurs. La taille du maillage est de 154^3 mailles par cœur. La quantité mémoire par cœur a été fixée à 3Go pour éviter les phénomènes de swap.

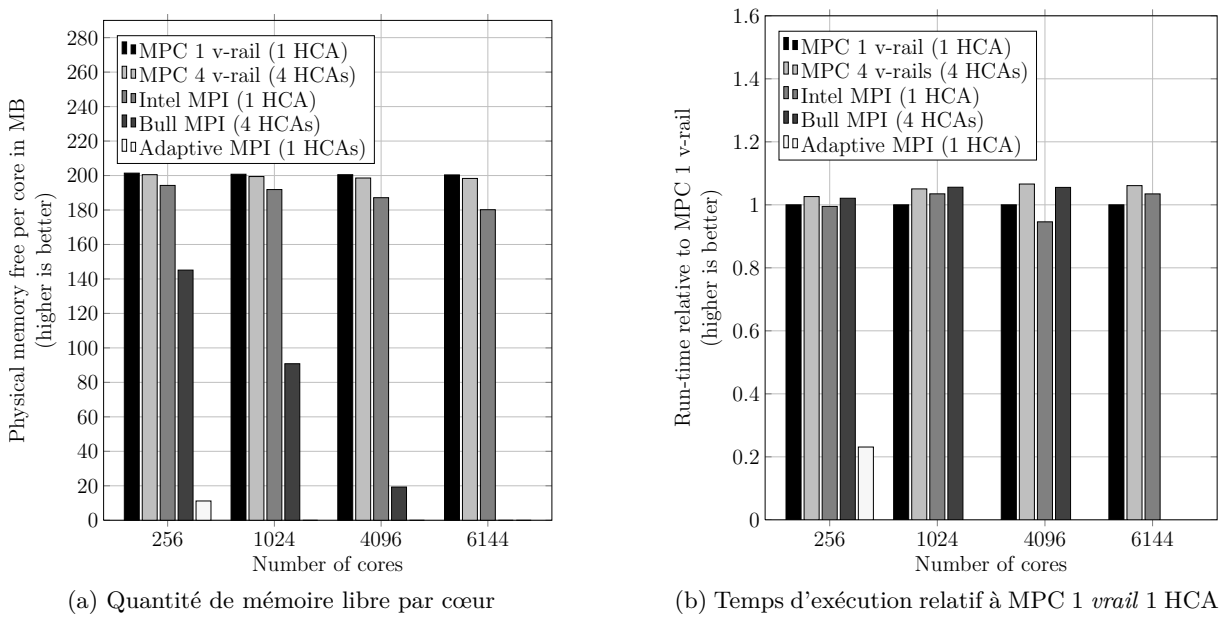


FIGURE 4.7 – Extensibilité faible du code Athena sur les supports exécutifs MPC, Intel MPI, Bull MPI et AMPI

3. Issue du site web d'Athena : <https://trac.princeton.edu/Athena/>

La figure 4.7 détaille les résultats de ces expériences. Comme on peut le voir, nous évaluons cette fois-ci Bull MPI[20] en plus de IntelMPI, AMPI et MPC. On constate aussi que notre approche permet d’avoir un bon compromis consommation mémoire/performances en contexte multithread. De plus, ces expérimentations illustrent bien l’intérêt à porter à l’allocation mémoire car AMPI et Bull MPI n’ont pas été en mesure de réaliser tous les tests du fait de leurs approches trop gourmandes en mémoire.

4.1.5 Conclusion

Dans cette section, nous avons montré que la gestion des accès réseau en contexte multithread doit tenir compte des effets NUMA ainsi que des problèmes de contention pour l’accès aux ressources partagées comme les listes de tampon. Notre contribution : les *vraills* sont une approche flexible qui permet à la fois de contrôler la contention tout en offrant une faible consommation mémoire. Ces *vraills* ont aussi été optimisés pour maintenir au mieux la localité NUMA des accès aux tampons comme aux cartes réseau.

Les travaux présentés ont été mis en œuvre dans le thread-based MPI MPC. Ils montrent aussi leur intérêt dans les approches plus classiques process-based MPI avec le support du MPI_THREAD_MULTIPLE. Avec ce mode, un processus MPI va devoir gérer un grand nombre de threads communiquant via le réseau. Pour obtenir de bonnes performances, il est donc important d’optimiser les couches basses des supports exécutifs. L’approche *vraills* que nous avons présentée pourrait donc être appliquée à n’importe quelle implémentation MPI et apporter un gain significatif en termes de performances et consommation mémoire.

4.2 Gestion de la progression des messages en contexte multithread

Lors de la réception d’un message ou encore pour faire progresser les protocoles de communication (par exemple le protocole de rendez-vous), les cœurs des nœuds de calcul sont mis à contribution. En effet, il n’est actuellement pas possible de reposer totalement sur les cartes réseau pour faire ce travail. Cette utilisation des cœurs de calcul pour des activités liées à MPI est problématique dans le cas des communications non-bloquantes ou l’on cherche à recouvrir les communications par du calcul.

Cette section présente un mécanisme de progression des messages nommé Collaborative-Polling [41] qui permet d’optimiser le recouvrement des communications par du calcul en contexte multithread.

4.2.1 La progression des messages non-bloquants

Une des principales difficultés de la progression des messages non-bloquants est de permettre la progression tout en ayant un impact minimum sur les cœurs de calcul. La figure 4.8(b) illustre bien le but recherché. Néanmoins, pour un grand nombre d’implémentations, c’est le cas de figure 4.8(a) qui se produit.

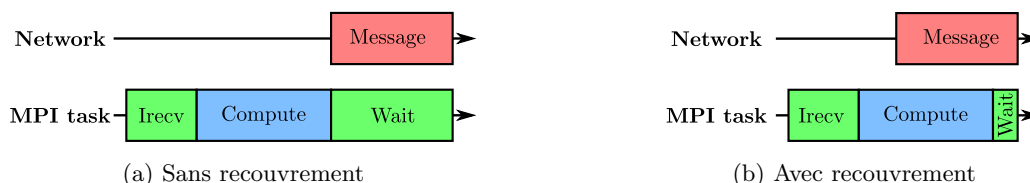


FIGURE 4.8 – Recouvrement calcul/communications en contexte MPI

Pour permettre un recouvrement des communications par du calcul, il est nécessaire d’interrompre les calculs en cours sur les cœurs de calcul. Plusieurs méthodes sont disponibles :

Géré par l’utilisateur : La progression des messages n’a lieu que lors des appels MPI. C’est donc à l’utilisateur de placer des appels MPI (par exemple MPI_Test) à l’intérieur de son code[57, 17].

Thread de progression : La progression est assurée par un ou plusieurs threads s’exécutant en tâche de fond[100, 52, 82].

Basé sur les interruptions : La progression des messages est ici assurée par des threads réveillés par des interruptions matérielles générées par exemple par la carte réseau[97, 70].

4.2.2 Notre contribution : le Collaborative Polling

Durant l'exécution d'un code de calcul parallèle, le temps passé à attendre des messages est du temps perdu. Il y a plusieurs causes à ces temps d'attente :

1. La distance, en termes de latence, entre les tâches MPI est variable (intra vs inter-nœud, nombre de niveaux de switch, ...)
2. Le nombre de voisins de chaque tâche n'est pas rigoureusement identique.
3. Déséquilibre de charge du réseau (contention sur les liens, ...).
4. Bruit système.

L'idée principale du Collaborative-Polling est d'utiliser les temps d'attente liés au déséquilibre de charge pour faire progresser les messages. En effet, dans un contexte multithread comme un thread-based MPI ou une implémentation `MPI_THREAD_MULTIPLE`, il est possible d'avoir une vision complète des différentes files de message et de leurs états. Il est donc possible, lorsque l'ordonnanceur de threads passe dans l'état *idle* sur un cœur, d'utiliser ce cœur pour faire progresser les messages en attente pour les autres cœurs.

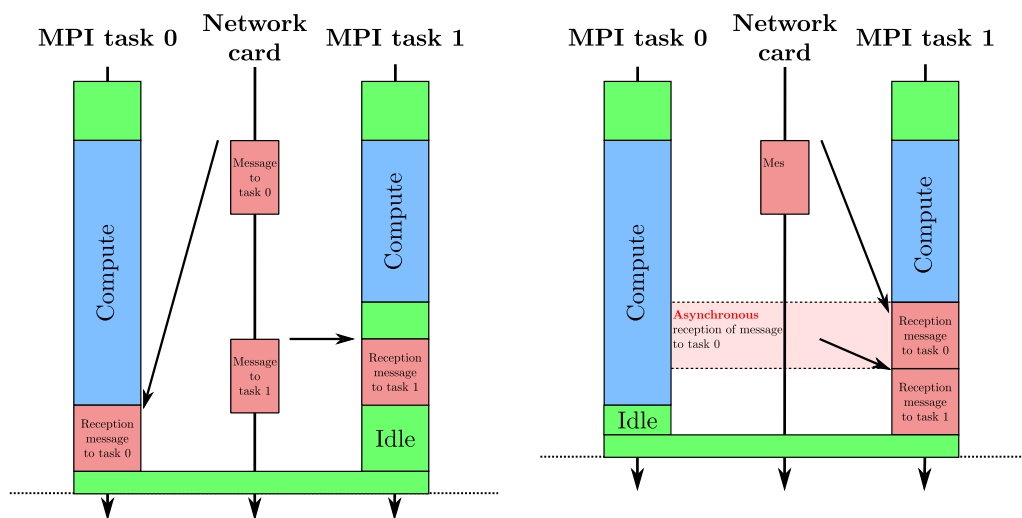


FIGURE 4.9 – Implémentation MPI sans Collaborative-Polling (gauche) et implémentation MPI avec Collaborative-Polling (droite)

La figure 4.9(droite) illustre le mécanisme de Collaborative-Polling dans le cadre d'un thread based-MPI. La tâche MPI 1 a une charge légèrement inférieure à la tâche 0. Lorsque la tâche 1 est bloquée sur la barrière, les cycles *idle* sont alors utilisés pour faire progresser les messages de la tâche 0. Ainsi, la tâche 0 n'a plus de progression de message à effectuer et le temps total d'exécution a été réduit.

L'implémentation du Collaborative-Polling a été réalisée au sein du support exécutif MPC. Cette implémentation tire parti des travaux présentés précédemment pour maintenir la localité des données. En effet, la scrutation des différentes files de message (*vrailes*) doit se faire en lien avec la topologie sous-jacente. La figure 4.10 illustre le mécanisme dans le contexte MPC pour le réseau Infiniband.

Comme on peut le voir, nous avons une stratégie de files locales avec un mécanisme de vol de tâche. Cette approche maximise la localité des accès tout en permettant une progression asynchrone des messages.

Le Collaborative-Polling a été appliqué au protocole *eager* comme au protocole de *rendez-vous*. La figure 4.11 illustre la progression du protocole de *rendez-vous*. Dans ce cas, la totalité de la gestion du protocole a pu être déportée sur un cœur libre.

4.2.3 Évaluation de la méthode

Dans cette section, nous allons maintenant présenter une évaluation du Collaborative-Polling sur l'application MPI EulerMHD[107]. Ces études ont été menées sur les nœuds 32 cœurs de la machine Curie (variante avec un seul niveau NUMA et donc sans le BCS de l'architecture présentée en 2.1.1). Nous avons comparé notre approche implémentée dans MPC avec MVAPICH2 1.7 (MV2), Open MPI 1.6.1 (OMPI) et Intel MPI 4.0.3.088 (IMPI).

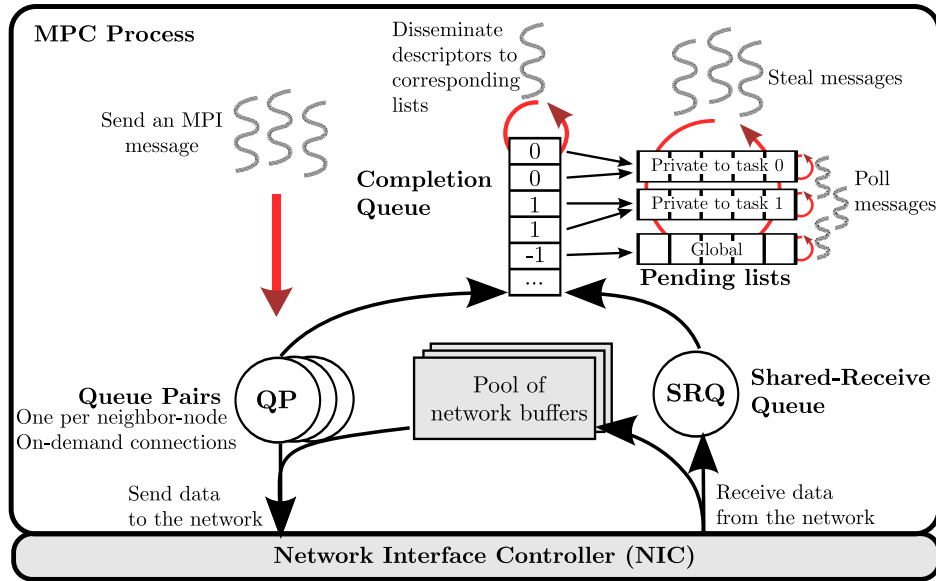


FIGURE 4.10 – Implémentation du Collaborative-Polling dans le contexte MPC pour Infiniband

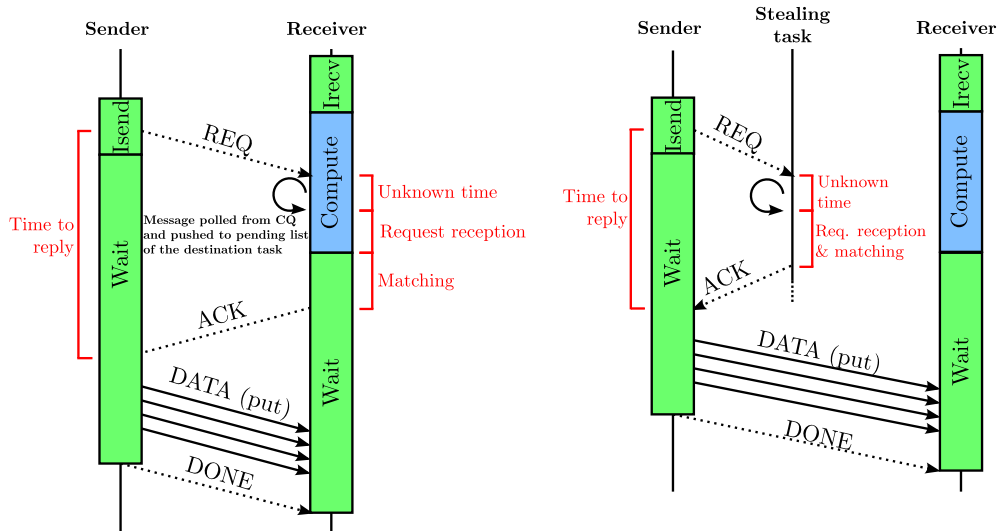


FIGURE 4.11 – Illustration du protocole de rendez-vous à gauche avec le Collaborative-Polling et à droite sans. Le Collaborative-Polling permet à une tâche MPI inactive de faire progresser la totalité des étapes du protocole.

EulerMHD est une application MPI résolvant à la fois les équations d'Euler et de la Magnétohydrodynamique d'ordre élevé sur un maillage cartésien 2D. A chaque itération, il y a des échanges de mailles fantômes avec les voisins ainsi qu'une réduction sur un entier. Cette évaluation a été réalisée sur 1024 tâches MPI et un maillage de 4096×4096 .

La figure 4.12 et le tableau 4.1, détaillent les résultats de ces évaluations. Comme on peut le voir, la méthode de Collaborative-Polling appliquée à MPC permet de réduire significativement le temps passé dans le MPI (temps assimilé à du temps perdu car non utile pour la simulation). Si on se compare avec les autres implémentations MPI ne disposant pas du Collaborative-Polling, on observe une nette différence sur le temps MPI. Si on regarde un peu plus en détail les temps d'exécution pour MPC on observe une nette amélioration des performances des opérations qui requièrent une progression des messages comme MPI.Wait ou MPI.Irecv. On observe de plus une amélioration de la fonction MPI.Allreduce du fait que le déséquilibre entre les tâches MPI ait été en partie gommé par le Collaborative-Polling. Enfin, on remarque que le temps de calcul n'a pas été impacté par la méthode de Collaborative-Polling.

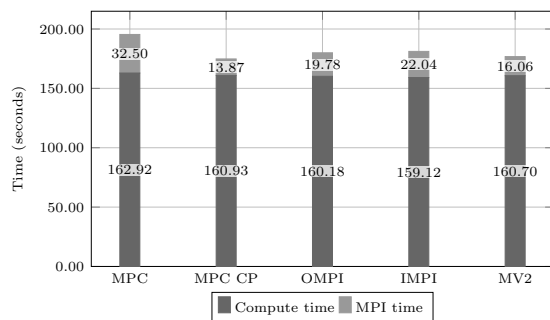


FIGURE 4.12 – Évaluation du temps d’exécution de l’application EulerMHD

Function	MPC	MPC CP	Speedup
Execution time	195.43	174.80	1.12
MPI time	32.50	13.87	2.34
Compute time	162.92	160.93	1.01
MPI.Wait	26.27	10.36	2.53
MPI.Allreduce	4.17	2.63	1.58
MPI.Irecv	1.24	0.18	6.84
MPI.Isend	0.83	0.69	1.19

TABLE 4.1 – Évaluation des temps MPI pour l’application EulerMHD

4.2.4 Conclusion

La méthode de Collaborative-Polling que nous venons de présenter permet une meilleure utilisation des ressources processeurs grâce à une vue globale au niveau du nœud de calcul. Cette vue globale est possible en contexte thread-based MPI mais aussi dans le cas d’un contexte MPI.THREAD.MULTIPLE. Nous avons pu observer une amélioration significative des performances sur une application réelle. D’autres évaluations de cette méthode sont présentées dans [41].

Nous n’avons pas encore évalué l’impact du Collaborative-Polling sur les communications collectives non-bloquantes introduites dans la norme MPI 3. Ces travaux vont être menés dans le cadre du projet DGCIS ELCI.

4.3 Gestion de l’empreinte mémoire

Comme nous l’avons vu précédemment, la consommation mémoire est une de nos priorités. En effet, avec l’augmentation du nombre de cœurs et donc de tâches MPI, la ressource mémoire critique. La taille des tampons réseau peut être une variable d’ajustement du ratio performances/consommation mémoire. Il est difficilement concevable de demander à l’utilisateur du support exécutif de choisir lui-même sa politique de gestion du réseau. C’est pourquoi, comme dans 3.4, nous avons pris le parti de laisser à l’application la quantité mémoire qu’elle demande et au support exécutif de s’adapter et de fournir les meilleures performances dans l’enveloppe mémoire disponible sur le nœud.

Il existe plusieurs méthodes pour contrôler la consommation mémoire. On peut citer par exemple les protocoles de connexion et de déconnexion à la demande. Dans cette section, nous allons présenter une nouvelle méthode qui cible l’optimisation de la consommation mémoire du protocole *eager* sur des cartes réseau rapide disposant du support RDMA⁴. Ces travaux ont été réalisés dans le cadre d’un support exécutif multithread disposant déjà de la connexion/déconnexion à la demande.

4.3.1 Le protocole *eager* en contexte RDMA

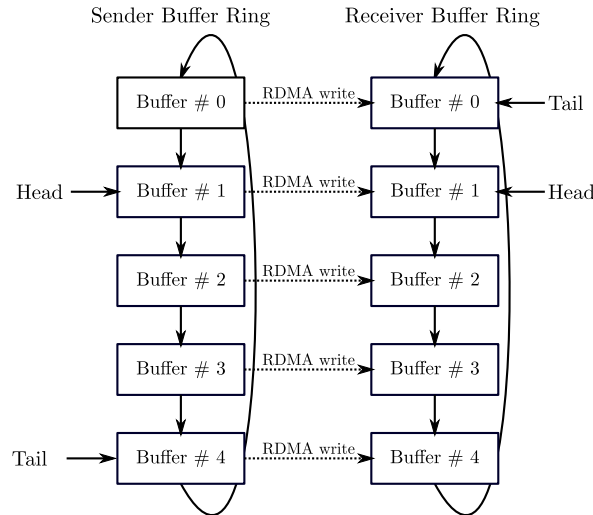
Le protocole *eager* en contexte RDMA a été décrit dans [73]. Dans ce protocole, deux ensembles de tampons (un en émission et un en réception) sont utilisés. La figure 4.13 illustre le fonctionnement de ce protocole. Les messages en émission sont d’abord copiés dans les tampons d’envoi. Ensuite, un *RDMA write* est généré pour écrire les données du tampon d’envoi dans le tampon de réception correspondant.

La performance du protocole *eager* en contexte RDMA est donc directement liée à la quantité et à la taille des tampons mis en œuvre. Comme nous pouvons le voir dans le tableau 4.2, les performances maximales sont obtenues avec au plus de 500Mo de tampon par processus MPI. Ces tests ont été réalisés sur la partition nœuds fins de la machine Curie et sur le benchmark NAS[6] FT classe D.

4.3.2 Contribution : Auto-ajustement des tampons Eager RDMA

L’idée maîtresse de cette contribution est de proposer un algorithme d’auto-ajustement dynamique du ratio performances/consommation mémoire. Nous allons dans un premier temps présenter l’algorithme

4. Remote Direct Memory Access : protocole qui permet à la carte réseau d’écrire et lire directement dans la mémoire d’un nœud distant.

FIGURE 4.13 – Le protocole *eager* RDMA

		RDMA		
Semantics	Execution Time	# Slots	% Miss	Memory (MB)
RDMA	36.11	1,024	0.00	1,009
RDMA	36.10	512	6.05	505
RDMA	36.61	256	17.97	252
RDMA	36.67	128	31.83	126
RDMA	36.96	64	45.90	63
RDMA	37.10	32	58.81	32
RDMA	37.27	16	71.30	16
RDMA	37.61	8	82.66	8
SR	37.63	0	100.00	0

TABLE 4.2 – NAS Fourier Transform (FT) classe D sur 512 tâches MPI, 32 nœuds. La taille des tampons est définie à 16Ko. Les résultats sont par processus MPI.

lui-même. Ensuite nous détaillerons le critère de réajustement pour augmenter les performances. Enfin, nous détaillerons le critère de réduction de l’empreinte mémoire.

Protocole d’ajustement

Ajuster dynamiquement les tampons *eager* RDMA est une collaboration émetteur/récepteur. La synchronisation émetteur/récepteur s’opère via un protocole d’échange d’informations en trois étapes. La figure 4.14 présente le protocole lorsqu’il est initié par l’émetteur.

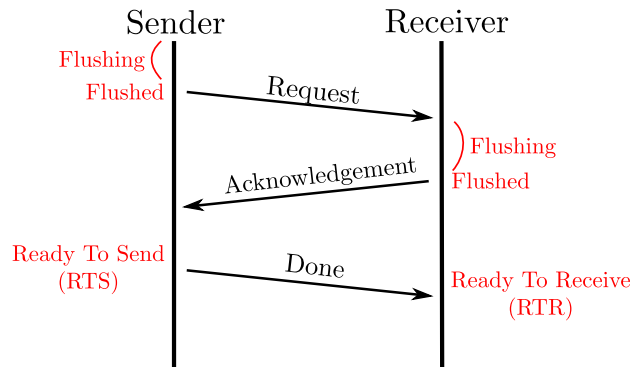


FIGURE 4.14 – Protocole d’ajustement des tampons RDMA. L’émetteur initie la requête.

Lorsqu’une opération d’ajustement est initiée, une requête est transmise avec la nouvelle configuration des tampons *eager* RDMA à utiliser. Une fois la requête arrivée au destinataire, ce dernier peut accepter

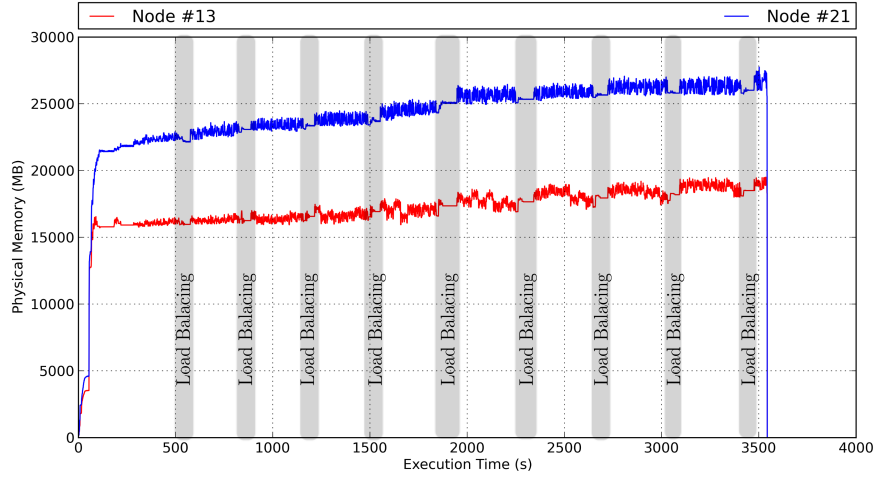


FIGURE 4.15 – Exécution HERA[64] sur 64 nœuds avec 1 024 tâche MPI en contexte MPC. Le maillage est composé de 256^3 mailles et 300 itérations sont réalisées. La figure reporte la mémoire physique allouée sur les nœuds 13 et 21.

ou refuser la nouvelle configuration. Les raisons d’un refus sont les suivantes : le récepteur ne peut pas allouer plus de mémoire pour les tampons *eager* RDMA ou alors une requête de déconnexion est en cours. De plus, avant le réajustement des tampons, il est nécessaire de s’assurer que ces tampons ne sont plus utilisés.

Durant la phase de réajustement des tampons, nous pouvons continuer à autoriser les communications entre tâches MPI via d’autres protocoles que *eager* RDMA ou via d’autre *vrrails*. Ce routage alternatif des messages peut avoir un impact sur les latences de communication, mais il permet de recouvrir le coût de l’algorithme de réajustement.

Nous avons identifié plusieurs cas de figure où le réajustement est nécessaire. Le premier cas est l’augmentation du nombre de tampons pour satisfaire les besoins de l’application. Le second cas de figure est de réduire la consommation mémoire pour libérer de la ressource pour l’application avant par exemple de procéder à de la déconnexion à la demande. Il faut noter que ces deux cas de figure peuvent s’enchaîner de manière itérative au cours de l’exécution d’une application. On peut citer, par exemple, le contexte d’une application utilisant la technique d’adaptation dynamique de maillage. Dans ce type d’application, la quantité mémoire requise par l’application varie au cours du temps comme on peut le voir sur la figure 4.15.

Ajuster pour augmenter les performances

Comme nous l’avons vu, le protocole *eager* RDMA est très rapide à condition d’avoir suffisamment de tampons pour couvrir les besoins applicatifs.

La variation des besoins de l’application peut être rapide et par phase pour gérer ce cas de figure, nous avons choisi une approche par échantillonnage sur les derniers messages envoyés plutôt qu’une vision globale de type profilage. Considérons les deux variables suivantes $size_{buffer}$ et $number_{buffer}$. La première représente la taille d’un tampon et la seconde le nombre de tampons. Pour chaque voisin, nous allons stocker dans *messages_size* la somme cumulée des tailles des derniers messages MPI et dans *messages_number* le nombre de messages échangés. La tâche MPI demandant l’ajustement calcule la nouvelle taille des tampons de la manière suivante :

$$size_{buffer} = \frac{messages_size}{messages_number} \quad (4.1)$$

Pour approximer le nombre de tampons nécessaires, nous allons utiliser l’algorithme suivant. Soit $max_pending_data$ le nombre maximal de tampons utilisés simultanément sur le réseau depuis le début de l’exécution de l’application. Soit $current_pending_data$ le nombre de tampons pour un canal RDMA. Avant chaque envoi, la variable $current_pending_data$ est incrémentée de la taille du message envoyé. $current_pending_data$ est décrémenté dès lors que le message a été reçu. A ce moment, si la variable $current_pending_data$ est supérieure à $max_pending_data$; cette dernière reçoit la valeur contenue dans

	Times (s)		RDMA (average per process)			
Mode	Init	Work	Mem. (MB)	# conn.	# reshaping	miss ratio
SR	432.22	554.97	0	0	0	0
RDMA (best config)	420.39	541.63	130.38	32.19	0	0.00
RDMA (1 reshaping)	428.38	545.31	2.38	32.19	32.19	0.33
RDMA (∞ reshaping)	429.22	538.40	76.70	32.19	42.13	0.06

TABLE 4.3 – Exécution HERA sur 32 nœuds et 512 tâche MPI. Le maillage est composé de 256^3 et l'exécution composée de 40 itérations. Le tableau présente les temps d'exécution en fonctions du nombre de d'ajustements de la taille des tampons *eager* RDMA.

current_pending_data. Avec ces informations, la tâche MPI demandant l'ajustement calcule le nouveau nombre de tampons de la manière suivante :

$$number_{buffer} = \frac{max_pending_data}{size_{buffer}} \quad (4.2)$$

Dans le cas d'une connexion à la demande, $size_{buffer}$ et $number_{buffer}$ sont calculés par l'émetteur et transférés au récepteur via le protocole décrit figure 4.14.

Ajuster pour réduire la consommation mémoire

Comme nous l'avons dit précédemment, la principale contrepartie du protocole *eager* RDMA est sa consommation mémoire. Pour ajuster dynamiquement la consommation mémoire, nous avons choisi une approche avec trois politiques :

Emergency : déconnexion des canaux RDMA et libération de la mémoire. Cette approche est très agressive et adaptée à une croissance rapide et soudaine de la consommation mémoire.

Normalization : équilibrage des consommations. Tous les canaux RDMA consommant plus que x octets voient leurs tampons réduire.

Least Recently Used (LRU) : déconnexion des canaux RDMA en commençant par les canaux les moins utilisés. Ce protocole est utilisé tant qu'il n'y a pas suffisamment de mémoire libérée.

Le protocole de réajustement de la configuration RDMA étant unidirectionnel, l'émetteur comme le récepteur peuvent choisir d'initier le réajustement sous contrainte mémoire.

4.3.3 Évaluation de la méthode

Pour évaluer l'impact de notre méthode d'ajustement des tampons du protocole *eager* RDMA, nous avons choisi d'utiliser l'application HERA (voir 2.1.2) sur la machine Curie nœuds fins avec 512 tâches MPI et le support exécutif MPC. Nous avons choisi comme période pour l'historique les 2000 derniers messages.

Le tableau 4.3 récapitule l'impact de notre politique d'ajustement. La version SR est la version qui n'utilise pas le protocole *eager* RDMA. La version RDMA (best config) représente la meilleure configuration i.e. la configuration dans laquelle il y a toujours un tampon de libre lors de l'envoi d'un message. Cette configuration est fixée dès le début de l'application. La version RDMA (1 reshaping) est une configuration dans laquelle le nombre initial de tampons est positionné à 0 et on autorise un seul réajustement par canal RDMA durant l'exécution. Enfin, la version RDMA (∞ reshaping) commence elle aussi avec un nombre de tampons égal à 0 mais n'a pas de limite sur le nombre de réajustement des tampons. Pour chacune des exécutions, nous avons instrumenté le support exécutif pour fournir la consommation mémoire des tampons, le nombre de réajustements ainsi que le taux de requêtes de tampons infructueuses du fait d'une liste de tampons vide.

Le temps d'exécution de l'application a été décomposé en deux parties. Tout d'abord la phase d'initialisation représente le temps passé par l'application pour construire le maillage initial et pour le répartir sur les tâches MPI. Comme on peut le constater, ce temps est directement impacté par la méthode de réajustement dynamique. En effet, la configuration initiale étant très optimiste sur le nombre de tampons, il y a un nombre élevé de requêtes infructueuses jusqu'aux premiers réajustements. Comme la phase d'initialisation n'est présente qu'au début de l'exécution du code, il est acceptable d'avoir un léger surcoût en termes de performance.

La seconde partie est l'exécution du calcul. C'est cette partie qu'il faut particulièrement optimiser pour les performances car son temps d'exécution est en principe largement supérieur à la phase d'initialisation. Dans notre exemple, nous avons forcé le nombre maximum d'itérations à 40 et le calcul n'était donc pas fini. Néanmoins, on constate une accélération significative du temps d'exécution grâce à notre approche avec un coût mémoire entre 1.7 et 54.8 fois inférieur à la configuration optimale en performances.

4.3.4 Conclusion

L'approche d'auto-ajustement du ratio performances/consommation mémoire présentée ici a été mise en œuvre au sein du support exécutif MPC et a montré des résultats significatifs sur une application réelle.

La méthode présentée s'inscrit dans la démarche générale d'optimisation de la consommation mémoire présentée dans le chapitre 3. Cette approche bénéficie aussi du support du Collaborative-Polling présenté en 4.2 pour autoriser l'ajustement de la consommation mémoire par les tâches les moins chargées au bénéfice des tâches les plus chargées.

4.4 Conclusion/Travaux futurs

4.4.1 Conclusion

Durant les travaux de thèse de Sylvain Didelot, nous avons mis en place un ensemble de méthodes pour utiliser efficacement les réseaux hautes performances en contexte multithread. Nous avons en particulier insisté sur les performances et la gestion de la contention. En effet, dans le contexte du calcul hautes performances, nous pouvons être amenés à utiliser plusieurs cartes réseau sur un nœud de type NUMA. Il est donc important de bien gérer l'accès aux ressources pour limiter les effets NUMA tout en maximisant la bande passante réseau avec une latence réduite. Pour atteindre ces objectifs, nous avons introduit le concept de *vraïl* qui apporte à la fois une solution au thread-based MPI comme MPC mais aussi à toute implémentation MPI désirant disposer d'un support MPI.THREAD.MULTIPLE efficace.

Ensuite, nous nous sommes focalisés sur la progression des messages en contexte multithread. Nous avons introduit la méthode de Collaborative-Polling qui permet une meilleure utilisation des ressources processeurs au sein du nœud de calcul. Cette méthode permet de mettre en place un recouvrement calculs/communications au sein du nœud sans avoir besoin de recourir à des threads de progression supplémentaires ni à du matériel spécifique.

Enfin, nous nous sommes intéressés à la consommation mémoire des couches réseau dans les supports exécutifs. Nous avons mis en place une méthode de gestion "au plus juste" des tampons réseau nécessaires pour une application. Cette méthode nous permet de réduire la taille et le nombre de tampons réseau dans le cas où l'application consomme la quasi totalité de la mémoire du nœud ou change de comportement en cours de calcul (par exemple des schémas de communication différents entre l'initialisation et le calcul à proprement parler).

4.4.2 Travaux futurs

Les travaux présentés ici ont permis de proposer une approche réseau optimisée pour le contexte multithread. Il serait bien d'étendre ces travaux pour proposer une approche intégrée avec les autres composants que peuvent constituer un support exécutif. Parmi ces composants, on peut citer l'allocateur mémoire.

Il serait aussi nécessaire de faire évoluer la méthode de Collaborative-Polling pour tenir compte des nouvelles opérations collectives non-bloquantes introduites dans la norme MPI 3. Il serait de plus intéressant de voir si les opérations *one sided* peuvent aussi profiter des évolutions que nous avons proposées.

Chapitre 5

Un support exécutif en contexte virtualisé

L'utilisation de la virtualisation apporte une solution transparente à de nombreux problèmes rencontrés actuellement dans l'exploitation de grappes pour le calcul intensif. Elle permet notamment d'améliorer la tolérance aux pannes, l'équilibrage de charge, l'isolation des utilisateurs, la portabilité et la pérennité des applications et offre une flexibilité nouvelle dans l'utilisation et le débogage de codes de niveau système personnalisés. Cependant, elle est encore peu utilisée pour les applications parallèles. D'une part à cause des problèmes de performances liés aux communications entre les machines virtuelles qui doivent passer par un périphérique virtuel émulé pour conserver la flexibilité apportée par la virtualisation. D'autre part à cause de la difficulté de configurer, instancier et maintenir une grappe de machines virtuelles ainsi que des coûts en temps, en mémoire vive et en espace de stockage qui en résultent.

Dans ce chapitre, nous allons décrire les travaux de thèse de François Diakhaté[36] soutenue en 2010 et quelques résultats de celle d'Antoine Capra débutée en 2012.

5.1 Passage de messages efficace entre machines virtuelles

À l'heure actuelle, les inquiétudes concernant les pertes de performances constituent probablement le plus gros frein à l'adoption de la virtualisation dans le cadre du calcul intensif. Les techniques de virtualisation du processeur et de la mémoire évoquées font que la plupart des codes de calcul séquentiels peuvent être exécutés dans des machines virtuelles avec un surcoût négligeable. Cependant, pour un code parallèle de type MPI, l'efficacité des communications entre les tâches rentre en jeu.

Nous nous sommes placés dans le contexte d'exécution suivant où chaque tâche MPI est hébergée dans sa propre machine virtuelle, ce qui signifie que les échanges de messages entre les tâches induisent des communications entre les machines virtuelles. Pour que l'application parallèle s'exécute efficacement, ces communications doivent être aussi performantes que si elles étaient effectuées directement sur les machines hôtes. Lors d'un échange de messages entre deux tâches d'une application MPI virtualisée, deux cas de figure se présentent.

Dans le cas où les machines virtuelles sont hébergées sur deux machines physiques distinctes, elles doivent pouvoir tirer efficacement parti du réseau hôte sous-jacent à l'aide du périphérique virtuel qui leur est exposé. Les périphériques virtuels Ethernet émulés en standard par la plupart des hyperviseurs induisent un surcoût en performance important, notamment dans le cas des réseaux rapides. Des techniques d'accès direct permettent de s'affranchir de ce surcoût mais ne sont pas idéales car elles limitent la flexibilité offerte par la virtualisation en limitant ou interdisant la migration de machines.

Il est aussi possible que les machines virtuelles soient co-hébergées, c'est-à-dire hébergées sur une même machine physique. Deux tâches exécutées directement sur la machine hôte peuvent exploiter très efficacement l'architecture à mémoire partagée sous-jacente pour communiquer. En revanche, lorsqu'elles sont virtualisées, elles ne peuvent communiquer que par le périphérique de communication exposé par l'hyperviseur. De ce fait, même si l'hyperviseur fournit un accès direct à un périphérique réseau haute performance, les communications entre les tâches restent moins efficaces que les communications en mémoire partagée qui auraient eu lieu si les tâches étaient exécutées sur l'hôte.

Aussi, de nouveaux canaux de communication entre machines virtuelles doivent être développés afin de pouvoir exécuter efficacement les applications parallèles de type passage de messages dans des machines

virtuelles.

5.1.1 Un périphérique virtuel pour le passage de messages

Afin que les machines virtuelles aient accès à un moyen de communication efficace pour le passage de messages, nous proposons d'introduire un périphérique virtuel de communication disposant d'une interface bas-niveau bien adaptée à la sémantique des communications MPI.

En effet, l'interface de communication proposée en standard est celle d'un périphérique Ethernet virtuel. Or, les réseaux rapides utilisés dans les grappes proposent une interface bien plus riche que celle d'un périphérique Ethernet ce qui permet notamment de court-circuiter le système d'exploitation lors des communications, de limiter les copies intermédiaires et de décharger le processeur. Ainsi, indépendamment du surcoût lié à l'émulation du périphérique Ethernet, cette interface standard n'est pas suffisante pour maximiser les performances des communications sur les grappes.

Émuler l'interface d'un réseau rapide existant offrirait plus d'expressivité que l'interface d'un périphérique Ethernet mais serait complexe à réaliser. Tout d'abord, ces interfaces sont souvent propriétaires et peu documentées. En effet, même si Infiniband est un standard, il ne définit qu'un ensemble de fonctionnalités dont l'interface bas-niveau diffère selon les constructeurs.

En outre, il n'est pas garanti qu'une telle interface se prête à une virtualisation efficace, puisque cela dépend avant tout de la fréquence des aller-retour déclenchés entre hôte et invité.

Aussi, l'introduction d'une interface paravirtualisée nouvelle semble nécessaire pour permettre à des machines virtuelles de s'échanger des messages efficacement. Pour exposer cette interface aux machines virtuelles de manière simple et portable, il est naturel de définir un nouveau périphérique virtuel. En effet, puisque les systèmes d'exploitation existants sont prévus pour piloter des interfaces matérielles, ils disposent de mécanismes adaptés pour les exploiter efficacement et en exporter les fonctionnalités aux applications en espace utilisateur. Le support d'un périphérique nouveau peut être ajouté de manière modulaire par l'écriture d'un pilote de périphérique dédié. De même, la plupart des hyperviseurs émulent déjà un grand nombre de périphériques et il est en général assez simple d'en ajouter un nouveau.

5.1.2 Des bibliothèques de communication en contexte virtualisé

Bien que nous introduisions un périphérique virtuel nouveau, nous souhaitons tout de même pouvoir exécuter une application MPI quelconque dans notre environnement sans modification. Pour cela une pile logicielle dédiée doit être utilisée dans les machines virtuelles. Elle comprend trois éléments :

1. Le pilote de périphérique. Ce pilote est assez basique : il prend en charge l'interface matérielle du périphérique virtuel et en exporte les fonctionnalités en espace utilisateur via l'implémentation d'appels système standard du système d'exploitation.
2. La bibliothèque de gestion du périphérique. À la manière de la bibliothèque MX qui permet d'exploiter les périphériques Myrinet, l'interface de notre bibliothèque fournit les primitives de base du passage de messages, c'est-à-dire notamment les communications point-à-point. Notons que cette bibliothèque implémente des communications fiables indépendamment des migrations de machines virtuelles qui sont transparentes pour les couches supérieures.
3. Une bibliothèque MPI supportant le périphérique virtuel. L'interface de notre bibliothèque bas-niveau étant proche de celle de MX, qui est supportée par de nombreuses implémentations MPI, ajouter le support de notre périphérique virtuel à une bibliothèque MPI existante ne devrait pas demander trop de développements. Néanmoins, afin de disposer de plus de flexibilité pour nos différents tests, nous avons développé une bibliothèque MPI réduite capable de l'exploiter. Elle supporte les principales opérations point-à-point et collectives définies par la norme MPI-1 ainsi que quelques types dérivés.

5.2 Un périphérique virtuel pour le passage de messages

Cette section détaille la conception de notre périphérique virtuel dédié au passage de messages.

5.2.1 Contraintes supplémentaires en environnement virtualisé

En contexte virtualisé, des contraintes supplémentaires sont à prendre en compte puisque les machines virtuelles sont isolées les unes des autres, et n'ont généralement pas directement accès aux périphériques réseau hôtes.

Transferts en mémoire partagée

La problématique du passage de messages entre machines virtuelles sur architecture à mémoire partagée est finalement assez proche du cas de processus exécutés nativement. En effet, tout comme les processus d'un système d'exploitation, les machines virtuelles sont exécutées dans des espaces d'adressage distincts. De ce fait, deux possibilités s'offrent à nous :

- Mettre en place un ensemble de pages physiques partagées par les machines virtuelles devant communiquer, de manière à ce qu'elles puissent mettre en œuvre, sans coût additionnel, les techniques utilisant un tampon de communication intermédiaire. Cette approche a été étudiée sur l'hyperviseur Xen en tirant parti de ses capacités de partage de pages pour mettre en œuvre des communications par socket [110, 66] ou par MPI [55] efficaces.
- Effectuer les copies dans un espace d'adressage où à la fois les tampons d'émission et de réception sont accessibles. Deux techniques peuvent être mises en œuvre. On peut accorder à l'une des deux machines virtuelles un droit de lecture ou d'écriture sur les pages mémoire contenant le message ou laisser effectuer l'opération par l'hyperviseur, de la même manière que cela est fait par le noyau dans le cas natif. Dans tous les cas, cela permet de n'effectuer qu'une seule copie et de profiter d'une bande passante maximale, mais impose des coûts de démarrage de communication importants, notamment s'il faut modifier une table des pages pour avoir accès aux tampons d'émission et de réception dans le même espace d'adressage. Cette technique sera donc réservée aux gros messages.

Nous sommes en présence d'un compromis entre latence et bande passante. Pour obtenir des performances optimales, notre solution devra sélectionner dynamiquement la méthode la plus appropriée, en fonction de la taille du message à transmettre.

Accès aux périphériques de communication

Dans le cas où les machines virtuelles sont hébergées sur des hôtes différents, elles doivent pouvoir envoyer des données par le périphérique réseau hôte. Une machine virtuelle peut manipuler directement le périphérique hôte s'il supporte la virtualisation matérielle, ou utiliser un périphérique virtuel émulé, les données étant alors réellement envoyées par l'hôte.

Les périphériques physiques supportant la norme SR-IOV et protégés par une IOMMU peuvent être directement manipulés par les machines virtuelles. Sur le chemin critique, l'hyperviseur n'est impliqué que lors de la délivrance des interruptions. Cela permet donc d'utiliser les techniques de communication natives en ne perdant que peu de performance.

Cependant, les contraintes importantes imposées par cette solution font qu'il est souhaitable de disposer de techniques alternatives performantes. Tout d'abord, peu de périphériques supportent actuellement les différentes normes nécessaires à une virtualisation matérielle sécurisée. Même si ce support se généralise, il est intéressant de pouvoir exploiter efficacement un maximum de matériels, la virtualisation étant supposée apporter de la portabilité. En outre, la machine virtuelle doit être capable de piloter le périphérique matériel, ce qui empêche de pouvoir déployer une machine virtuelle sur n'importe quel hôte puisqu'elle doit disposer des pilotes de périphérique correspondant au matériel sous-jacent. On se prive aussi de la possibilité d'utiliser des systèmes d'exploitation minimaux de type library-OS. Par ailleurs, les contraintes liées au punaisage de pages ainsi que la complexité de migrer les machines virtuelles en tenant compte de l'état du périphérique physique limitent la flexibilité apportée par la migration. Pour finir, l'hyperviseur n'étant plus impliqué dans les transferts de données, il n'est plus possible d'appliquer de politique de qualité de service ou de filtrer les paquets échangés par les machines virtuelles. Pour toutes ces raisons, nous avons choisi de ne pas utiliser d'accès directs aux périphériques hôtes.

L'utilisation d'un périphérique émulé a l'avantage et l'inconvénient de laisser l'hôte effectuer lui-même les communications sur le périphérique physique. On conserve ainsi l'abstraction du matériel apportée par la virtualisation au prix d'une perte de performance puisque des changements de contexte entre invité et hôte sont nécessaires pour chaque accès au périphérique.

Si cette contrainte impacte nécessairement la latence des échanges de message entre machines virtuelles, il est tout de même possible de tirer profit de certaines caractéristiques des réseaux rapides en passant par un périphérique émulé.

Pour mettre en œuvre les techniques de communication vues précédemment, l'invité doit pouvoir envoyer et recevoir des données dans des tampons de communication situés en espace utilisateur sans avoir à effectuer de copie supplémentaire. Cela implique que le périphérique virtuel émulé par l'hyperviseur propose une interface de type envoi/réception de messages ou RDMA adaptée afin que la destination finale des données soit connue au moment de leur réception.

5.2.2 Spécification du périphérique virtuel

Nous venons de voir qu'une interface adaptée au passage de messages entre machines virtuelles pourrait permettre d'en améliorer les performances sans avoir à fournir un accès direct au périphérique réseau hôte. Comme nous l'avons précédemment indiqué, introduire un périphérique virtuel est une bonne manière d'exposer une interface nouvelle aux systèmes d'exploitation invités. Nous présentons donc maintenant l'interface d'un périphérique virtuel simple permettant de passer des messages entre machines virtuelles.

Discussion

L'étude des techniques de passage de messages en mémoire partagée et sur réseau rapide effectuée précédemment montre que les principes mis en œuvre dans ces deux cas sont similaires.

Pour les messages de petite taille, des tampons de communication intermédiaires prédéfinis sont utilisés. En effet, partager ces tampons de communication entre processus, ou les enregistrer auprès du périphérique réseau, sont des opérations coûteuses qui doivent être effectuées hors du chemin critique. En outre, ces tampons intermédiaires permettent d'éviter l'utilisation de rendez-vous qui synchronisent émetteur et récepteur. On minimise ainsi la latence des communications, et pour des messages suffisamment petits, l'impact des copies intermédiaires supplémentaires est faible.

Lorsque les messages sont suffisamment gros, on cherche à s'affranchir de cette copie intermédiaire en transférant directement les données de la zone d'émission à la zone de réception. Ce transfert peut prendre la forme d'une copie mémoire dans le cas où les tâches sont exécutées sur un même nœud, ou d'une communication réseau de type RDMA dans le cas contraire. Les coûts supplémentaires liés aux rendez-vous, punaisages et enregistrements mémoire sont alors rentabilisés par les gains en bande passante et en utilisation processeur.

Il est donc possible d'utiliser une interface bas-niveau commune pour le passage de messages entre tâches virtualisées, qu'elles soient exécutées dans des machines virtuelles co-hébergées ou non.

Caractéristiques générales

Le périphérique implémente des communications point-à-point entre des extrémités de communication ouvertes dans différentes machines virtuelles formant une grappe virtuelle. Chaque extrémité de communication est identifiée par une adresse unique sur toute la grappe virtuelle.

Les différents mécanismes de communication proposés sont tous fiables : une donnée envoyée est garantie d'arriver à destination. En effet, la couche de communication utilisée dans l'hôte pour émuler le périphérique virtuel sera souvent elle-même fiable. Il serait donc superflu que la gestion de la fiabilité doive être réimplémentée dans la pile logicielle de l'invité. Il est plus efficace que l'hôte se charge d'assurer la fiabilité dans les cas où le réseau sous-jacent ne la gère pas directement.

Enfin, puisque notre objectif est de connecter des machines faisant partie d'une même grappe virtuelle créée spécifiquement pour l'exécution d'une application parallèle, nous considérons que les machines interconnectées se font confiance. Aussi, aucune garantie n'est offerte contre la possibilité pour une machine virtuelle d'intercepter, de corrompre, ou de falsifier un flux de communication.

Interface bas-niveau

L'interface bas-niveau du périphérique propose deux mécanismes de communication qui peuvent être utilisés en fonction de la taille du message à envoyer.

Tampons de communication : Notre périphérique virtuel embarque de la mémoire et permet à l'invité d'y allouer des tampons de communication pré-définis, de taille fixe. Ces tampons peuvent être projetés dans la mémoire d'un processus invité afin que des données puissent y être écrites et lues depuis une bibliothèque de communication en espace utilisateur (voir Figure 5.1).

Pour chaque envoi de données, la bibliothèque de communication va donc allouer autant de tampons de communication que nécessaire et y placer le message à transférer. Elle insère ensuite les tampons en question dans une file d'envoi après avoir indiqué les extrémités de communication source et destination dans l'en-tête du tampon.

Tous les tampons de communication reçus par une extrémité de communication sont placés dans une même file de réception par le périphérique virtuel. Cette file pouvant elle aussi être projetée en mémoire, il suffit, pour un processus invité, de scruter une unique adresse mémoire pour détecter l'arrivée de données. L'en-tête du tampon peut alors être lue pour déterminer l'identifiant de l'extrémité de communication source.

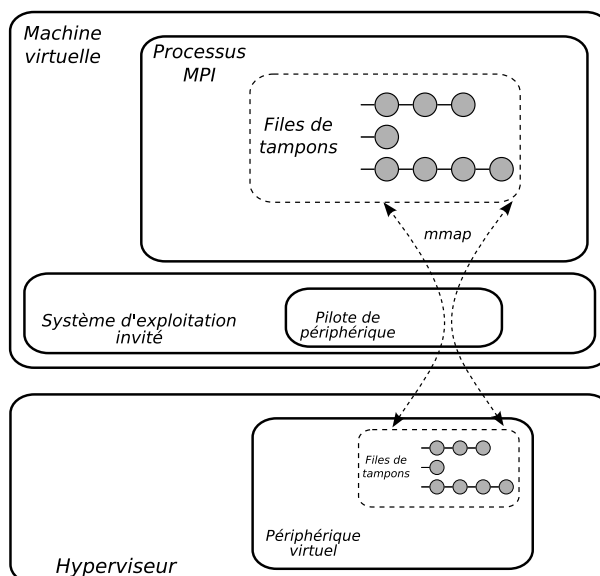


FIGURE 5.1 – Projection des tampons de communication en espace utilisateur invité

Ce type de communication est donc destiné à l’envoi des petits messages avec copie intermédiaire. Cette copie intermédiaire est effectuée par la bibliothèque de communication dans un tampon de communication fourni par le périphérique virtuel. Puisque ces tampons sont définis par le périphérique virtuel, et donc par l’hôte, ils peuvent être alloués dans une zone mémoire qui permet un transfert efficace vers des machines virtuelles co-hébergées ou non.

Accès mémoire distants : Le périphérique virtuel permet aussi d’effectuer des accès mémoire distants afin d’éviter les copies lors de la transmission de messages de taille plus importante.

Comme dans le cas des RDMA pratiqués par les périphériques réseaux rapides, les accès mémoire distants ne peuvent s’effectuer qu’entre zones mémoire qui ont été préalablement enregistrées. L’enregistrement d’une zone mémoire consiste à fournir au périphérique virtuel une suite de couples pointeur/taille décrivant une zone potentiellement non contiguë de la mémoire physique de l’invité. Cela implique, une fois de plus, de punaiser au préalable les pages concernées afin que les projections mémoire ne changent pas en cours de transfert. Un identifiant de zone mémoire est alors associé à la zone enregistrée.

Il est ensuite possible de déclencher une copie entre une zone mémoire distante et une zone mémoire locale, en spécifiant les identifiants des zones mémoire concernées, ainsi que l’extrémité de communication distante. La quantité de données à lire, ainsi que des décalages dans les zones mémoire locales et distantes peuvent aussi être indiqués afin de ne transférer qu’une partie d’une zone enregistrée.

Nous ne proposons que des accès mémoire distants en lecture car ils nous semblent mieux adaptés à la sémantique des communications par passage de messages que les écritures. En effet, la mise en correspondance entre les requêtes d’émission de messages et celles de réception de message ne peut être effectuée que du côté récepteur. Il est donc intéressant que le récepteur puisse initier le transfert de données de lui-même dès que la mise en correspondance est effectuée (voir Figure 5.2). On économise ainsi l’envoi d’un message et on augmente les possibilités de recouvrement entre calcul et communications.

Enfin, pour les deux canaux de communication, l’invité doit envoyer un signal au périphérique virtuel afin de lui indiquer que de nouvelles requêtes de communications lui ont été soumises et que les communications puissent progresser. Ce signal est implémenté de manière à déclencher une interruption logicielle afin que l’hôte puisse récupérer la main et effectuer les communications nécessaires.

Bibliothèque de passage de messages

L’interface bas-niveau décrite précédemment est complexe et fastidieuse à utiliser. Aussi, elle n’est pas censée être utilisée directement et ne sert qu’à implémenter une bibliothèque de passage de messages qui abstrait les détails d’utilisation du périphérique.

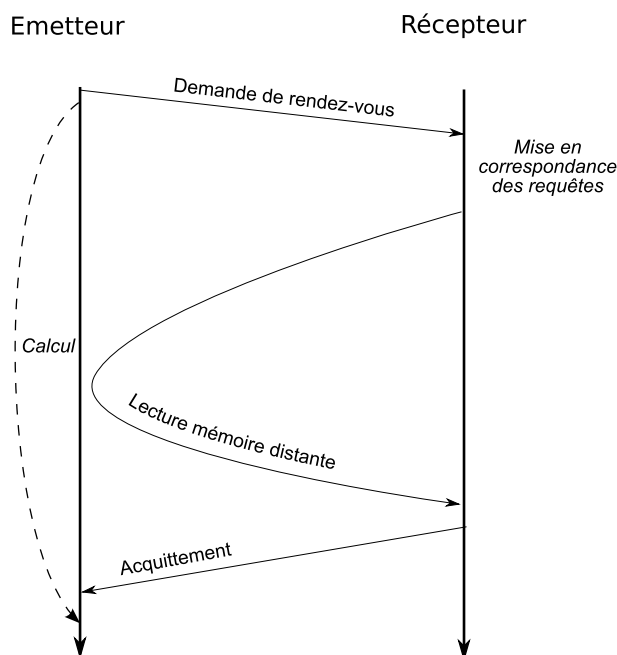


FIGURE 5.2 – Utilisation d'accès distants en lecture pour recouvrir les communications par du calcul

Cette bibliothèque implémente des communications par passage de messages de type point-à-point. Pour chaque message envoyé, la bibliothèque détermine le mode de communication le plus adapté en fonction de la taille du message.

Les petits messages sont envoyés en utilisant les tampons de communication fournis par le périphérique réseau. Les messages sont découpés en autant de tampons que nécessaire et un en-tête décrivant le message est ajouté. Côté récepteur, si une requête de réception correspondante a déjà été postée, les données sont copiées directement depuis le tampon de communication vers la zone de réception du message. À l'inverse, si le message est inattendu, il est copié dans un autre tampon en attendant qu'une requête de réception correspondante soit postée.

Pour les messages plus gros, les accès mémoire distants sont utilisés. Côté émetteur, la bibliothèque de communication commence par enregistrer la zone mémoire correspondant au message à envoyer puis effectue une demande de rendez-vous. Pour cela, elle envoie un tampon de communication contenant l'en-tête du message ainsi que l'identifiant de la zone mémoire à la machine virtuelle réceptrice. Lorsqu'une requête de réception correspondant à cet en-tête est postée, la zone mémoire de réception est à son tour enregistrée et un accès mémoire distant est utilisé pour rapatrier les données. Un acquittement est ensuite envoyé à l'émetteur pour lui signaler qu'il peut libérer la zone mémoire enregistrée.

Cette bibliothèque offre donc une interface à la fois simple d'utilisation et très bien adaptée à l'implémentation d'une bibliothèque de passage de messages standard telle qu'une bibliothèque MPI.

Mise en œuvre des communications

Nous étudions maintenant comment les requêtes de communication soumises au périphérique virtuel peuvent être traitées. Nous montrons en particulier en quoi l'interface choisie permet des transferts de données efficaces en mémoire partagée comme sur réseau rapide et discutons de l'impact des migrations de machines virtuelles sur le traitement des communications.

Transferts de données

Chacun des deux canaux de communication proposés permettent de mettre en œuvre les techniques natives de passage de messages étudiées précédemment.

Tampons de communication : Une fois qu'un tampon de communication a été soumis au périphérique virtuel pour envoi, si la machine virtuelle de destination se situe sur la même machine physique, un simple échange de pointeur permet d'insérer le tampon de communication dans la file de réception. En effet, puisque l'on considère que les machines virtuelles qui communiquent se font confiance, tous les tampons de communication peuvent être partagés entre les machines virtuelles. L'échange de

pointeur peut même être effectué directement dans le contexte de l'invité émetteur, sans impliquer l'hôte dans la communication.

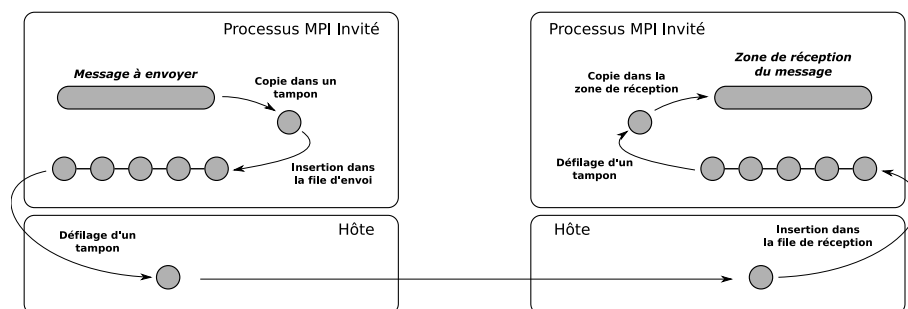


FIGURE 5.3 – Transfert de messages à l'aide des tampons de communication

Si les machines virtuelles sont hébergées sur des hôtes différents, l'hôte hébergeant la machine virtuelle émettrice doit transférer les données sur le réseau (voir Figure 5.3). Puisque les transferts se font entre des tampons de communication prédéfinis ils peuvent être enregistrés auprès du réseau hôte pour un transfert rapide sans copie intermédiaire. Lorsqu'un hôte détecte l'arrivée de données dans un tampon de communication, il l'insère dans la bonne file de réception.

Les copies supplémentaires sont donc évitées, mais le surcoût en latence induit par le changement de contexte entre invité et hôte ne peut être facilement contourné dans le cas général.

Une solution consiste néanmoins à utiliser un éventuel cœur libre de l'hôte pour l'émulation du périphérique virtuel.

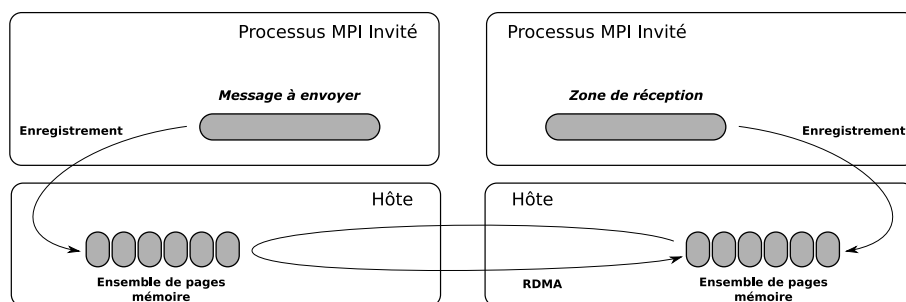


FIGURE 5.4 – Transfert de messages par accès mémoire distant

Accès mémoire distants : Les transferts par accès direct se font entre zones mémoire préalablement enregistrées et définies par un ensemble d'adresses physiques virtuelles. Cela permet à l'hôte d'y accéder simplement, indépendamment des translations d'adresses mises en place par l'invité. Il lui suffit de déterminer les adresses de l'hôte correspondant à ces zones mémoire et de transférer les données entre elles. Lorsque les machines virtuelles sont co-hébergées, cela revient à une simple copie mémoire. Dans le cas contraire, l'hôte peut profiter de l'étape d'enregistrement virtuel pour enregistrer les zones mémoire utilisées auprès du réseau physique et transférer ensuite les données sans copies intermédiaires (voir Figure 5.4). Puisque ce canal de communication sera uniquement utilisé pour des transferts de taille importante la latence induite par les changements de contexte est moins problématique.

Cas des migrations

Migrer une machine virtuelle utilisant un périphérique Ethernet virtuel pour communiquer est relativement simple. La machine virtuelle peut simplement être détruite sur l'hôte source puis recréée dans le même état sur l'hôte de destination. Puisque le réseau Ethernet n'assure pas la fiabilité des données, les paquets qui sont envoyés à la machine virtuelle pendant la durée de la migration peuvent simplement être ignorés. Le protocole de niveau supérieur utilisé par les machines virtuelles, généralement TCP, se charge alors de retransmettre les paquets perdus.

Nos choix de conception complexifient la gestion de la migration de machines virtuelles utilisant notre périphérique, et ce à deux niveaux. Tout d'abord, puisque nous assurons la fiabilité des communications, il

est de la responsabilité de l'hôte de faire en sorte que les données transmises pendant les migrations soient correctement délivrées. Par ailleurs, dans le cas des communications en mémoire partagée, les machines virtuelles peuvent être amenées à directement manipuler les files de réception de machines virtuelles co-hébergées, pour éviter les changements de contexte. Il faut donc s'assurer que les machines virtuelles ne sont pas migrées lorsqu'elles effectuent des opérations critiques afin de ne pas compromettre l'intégrité des données.

Avant chaque migration, l'hôte notifie toutes les machines virtuelles co-hébergées qu'une migration va avoir lieu, par l'intermédiaire d'une interruption déclenchée par le périphérique virtuel. Elles doivent alors interrompre leurs communications directes en mémoire partagée et en informer l'hôte. Lorsque toutes les communications en mémoire partagée sont stoppées, la migration peut avoir lieu et l'hôte notifie les machines virtuelles restantes que les communications en mémoire partagée peuvent reprendre.

Pour assurer la fiabilité des transferts, nous utilisons une technique similaire, mais appliquée au niveau des communications entre les hôtes. Avant chaque migration, l'hôte source diffuse un message à tous les hôtes impliqués dans la grappe virtuelle afin de leur demander d'arrêter d'envoyer des données concernant la machine virtuelle qui va migrer. Une fois que tous les hôtes ont accusé réception de cette requête, la machine virtuelle est transférée vers son hôte de destination. Celui-ci diffuse alors la nouvelle adresse de la machine virtuelle ce qui permet aux communications la concernant de reprendre.

Cette technique a l'intérêt d'être indépendante du nombre de machines virtuelles mises en jeu, son coût dépendant uniquement du nombre d'hôtes utilisés par la grappe virtuelle. En outre, une fois qu'une machine virtuelle est prête à être migrée, on est assuré qu'il n'y a plus de données la concernant en transit. Cela signifie que l'on peut utiliser le même algorithme pour effectuer des protections reprises de grappes virtuelles. En effet, en demandant simultanément l'arrêt des communications pour toutes les machines virtuelles on s'assure qu'il n'y a plus aucune donnée en cours d'envoi sur le réseau et que l'ensemble des machines virtuelles peuvent être sauvegardées dans un état cohérent.

5.3 Évaluation

5.3.1 Éléments d'implémentation

Afin de pouvoir virtualiser des applications existantes le plus simplement possible, nous avons choisi d'implémenter une bibliothèque MPI (nommée VMPI) basée sur notre périphérique virtuel. L'idée est de pouvoir lancer une application parallèle MPI dans une grappe de machines virtuelles aussi simplement que sur une grappe physique.

Avec VMPI, l'instanciation des machines virtuelles exécutant l'application parallèle est transparente pour l'utilisateur. Le lanceur associé à notre bibliothèque exécute la première instance de l'application dans une machine virtuelle légère contenant un noyau Linux minimal. Les fichiers de l'application sont lus et écrits directement sur le système de fichiers hôte par le périphérique virtuel 9P, et le périphérique virtuel de console permet de récupérer la sortie standard de l'application et de la rediriger sur le terminal de l'hôte.

5.3.2 Évaluation sur des logiciels de calcul

Nous souhaitons déterminer ici l'impact des différences de performances sur le temps d'exécution de codes de calcul scientifique typiques. Nous présentons donc, dans cette section, le résultat de deux tests de performances : le LINPACK et la plateforme de calcul HERA développée au CEA.

Machines de test

Les tests présentés ici ont été réalisés sur la machine Frotoy. Fortoy est une grappe composée de nœuds bi-processeurs quad-core E5462 cadencés à 2.8GHz et basés sur l'architecture Core. Chaque nœud est par ailleurs équipé de 8Go de mémoire vive. Nous disposons de 8 nœuds de la grappe sur lesquels KVM était installé ce qui nous a permis d'exécuter des applications parallèles virtualisées sur 64 cœurs.

High Performance LINPACK

Le High Performance LINPACK (HPL) est un test qui mesure le temps nécessaire à résolution d'un système linéaire dense en utilisant une factorisation LU. Ce test est particulièrement important dans la communauté du calcul hautes performances, puisque c'est celui qui est utilisé pour classer les machines au sein du Top500. Les algorithmes de résolution de systèmes linéaires denses permettent en

effet de très bien exploiter les capacités de calcul flottant des processeurs modernes. En particulier, ils peuvent être implémentés à l'aide des BLAS (pour Basic Linear Algebra Subprograms) qui sont un ensemble d'opérations couramment utilisées en algèbre linéaire, telles que les multiplications de matrices. Des bibliothèques de BLAS, telles que ATLAS, Goto BLAS, ou encore la MKL d'Intel fournissent des implémentations très optimisées de ces opérations pour différents processeurs. On s'approche ainsi de la puissance de calcul théorique des machines en terme de nombre d'opérations flottantes effectuées par seconde.

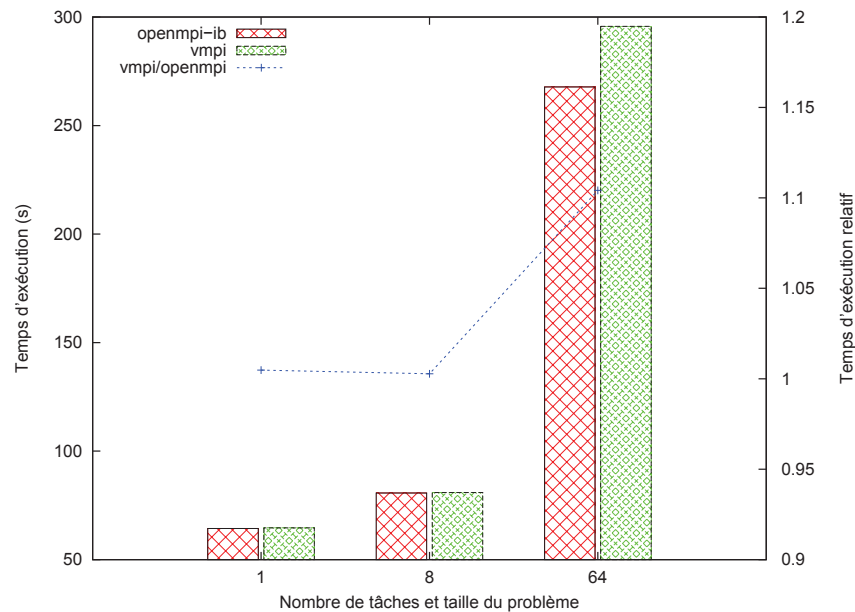


FIGURE 5.5 – Temps d'exécution du LINPACK

La figure 5.5 présente les temps d'exécution comparés en mode natif avec OpenMPI et virtualisé avec VMPI, sur Fortoy. On fait varier le nombre de tâches MPI utilisées de 1 à 64 et on utilise toujours autant de cœurs physiques que de tâches. Les exécutions avec 1 et 8 tâches sont donc effectuées sur un seul nœud, tandis que l'exécution à 64 tâches est effectuée sur 8 nœuds.

Le résultat de l'exécution séquentielle nous permet de vérifier que le surcoût lié à la virtualisation du processeur reste faible, même pour des codes très optimisés tels que le LINPACK.

Cependant, pour arriver à ce résultat, il nous a fallu prendre soin d'exposer aux machines virtuelles des processeurs virtuels ayant les mêmes identifiants CPUID que le processeur hôte. En effet la bibliothèque de BLAS MKL, que nous avons utilisée dans ce test, se base sur cet identifiant pour choisir des routines spécialement optimisées pour chaque version de processeur. Ce type de comportement nécessite une prise en charge particulière dans le cadre de la virtualisation, puisque cela peut empêcher de migrer les machines virtuelles entre des machines ayant des processeurs de différents types. Il faut dans ce cas déterminer un identifiant de processeur correspondant à un "plus petit dénominateur commun" des fonctionnalités supportées par les processeurs hôtes potentiels.

Le résultat des exécutions parallèles montre qu'il n'y a aucun surcoût pour l'exécution de tâches parallèles en mémoire partagée. Pour une exécution sur 8 nœuds, la perte de performance induite par la virtualisation est de l'ordre de 10%, ce qui reste raisonnable.

Plateforme AMR d'hydrodynamique : HERA

Pour ce test applicatif, nous avons utilisé l'application HERA décrite en 2.1.2.

La figure 5.6 présente les temps d'exécution de HERA pour un cas-test en trois dimensions simulant l'évolution hydrodynamique d'une coquille sphérique décrite par une équation d'état *stiffened gas* et d'un gaz décrit par l'équation d'état gaz parfait. Nous utilisons deux maillages, composés respectivement d'un million de mailles (1 octant) et de huit millions de mailles (8 octants), chaque maillage disposant de 3 niveaux de raffinement possibles. Comme pour le test précédent, on compare le mode d'exécution natif avec OpenMPI au mode d'exécution virtualisé avec VMPI sur la machine Fortoy. On étudie le cas où le code est exécuté de manière séquentielle ainsi que celui où il est parallélisé sur tous les cœurs d'un nœud

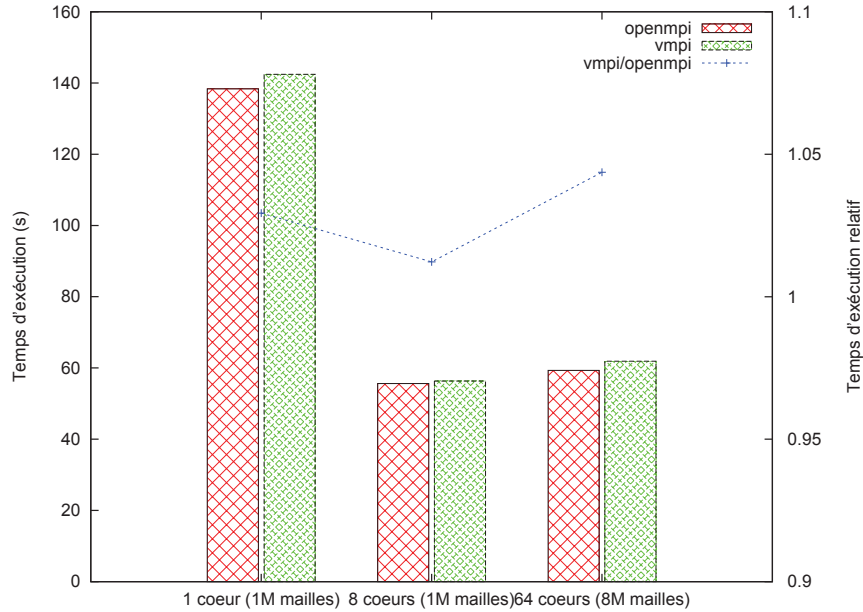


FIGURE 5.6 – Temps d'exécution de HERA

puis de 8 nœuds. Ce test permet de valider l'utilisation de la virtualisation, y compris pour l'exécution de véritables simulations numériques puisque le surcoût en performance reste dans tous les cas inférieur à 5%.

5.3.3 Migrations de machines virtuelles

Notre support exécutif permet de faire profiter simplement n'importe quelle application parallèle MPI des bénéfices apportés par la migration. En particulier, il devient possible, de façon transparente, de migrer d'un hôte à l'autre les différentes tâches d'une application MPI au cours de son exécution. Dans cette section, nous présentons les résultats de différents tests évaluant les performances de cette opération.

Performances des communications pendant les migrations

Notre premier test élémentaire met en évidence l'influence des migrations sur les performances des communications entre machines virtuelles. Pour cela, nous effectuons en boucle mille échanges de messages de taille nulle entre deux tâches MPI et mesurons la latence moyenne constatée à chaque itération de la boucle. Ces deux tâches sont exécutées dans deux machines virtuelles exécutées initialement sur deux nœuds de Fortoy. Toutes les dix secondes une migration est déclenchée afin de faire passer la deuxième tâche d'un hôte à l'autre. Les résultats sont présentés sur la figure 5.7.

À l'issue de la première migration, les machines virtuelles sont hébergées sur le même hôte ce qui permet aux tâches MPI de communiquer directement en mémoire partagée. On constate donc une diminution de la latence dès la première itération de la boucle suivant la migration, ce qui montre que le flot de messages échangés n'est pas interrompu longtemps. À l'inverse, à la vingtième seconde, la deuxième machine virtuelle retourne sur son hôte d'origine. Le coût de la migration est reflété sur la première itération de la boucle, puis on retrouve la latence initiale des communications inter-nœud à partir des itérations suivantes.

Impact sur le temps d'exécution de HERA

Nous nous intéressons maintenant à l'impact des migrations sur les performances d'une application parallèle. Nous utilisons pour cela la plateforme HERA et nous mesurons le temps d'exécution du cas-test décrit section 5.3.2, lorsque des migrations ont lieu. Nous étudions différentes configurations et présentons, pour chacune d'elles, le différentiel de temps par rapport à une exécution sans migration. Nous mesurons en outre le temps écoulé entre le début et la fin de l'ensemble des migrations, ainsi que le volume total de données transférées. Tous les tests sont exécutés sur la grappe Fortoy et les résultats sont reportés sur la figure 5.8.

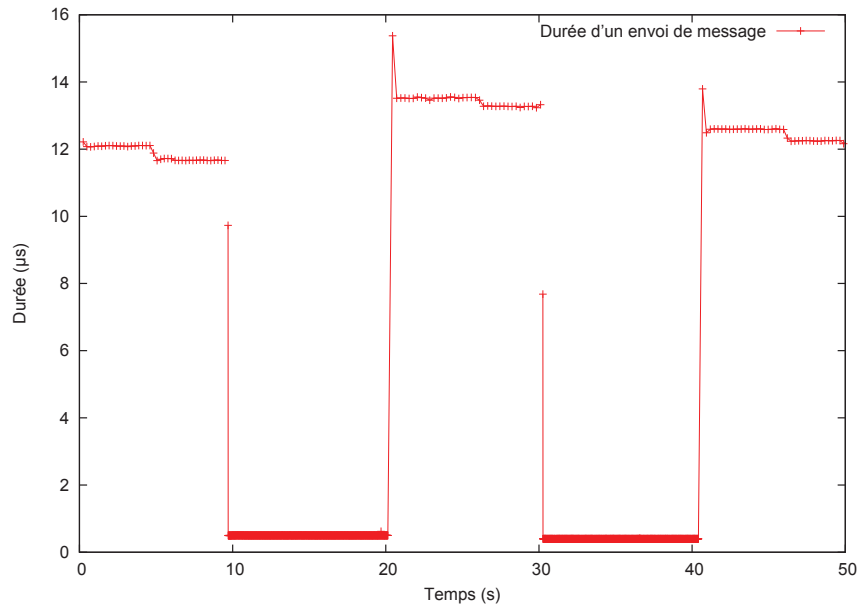


FIGURE 5.7 – Performances des communications pendant les migrations

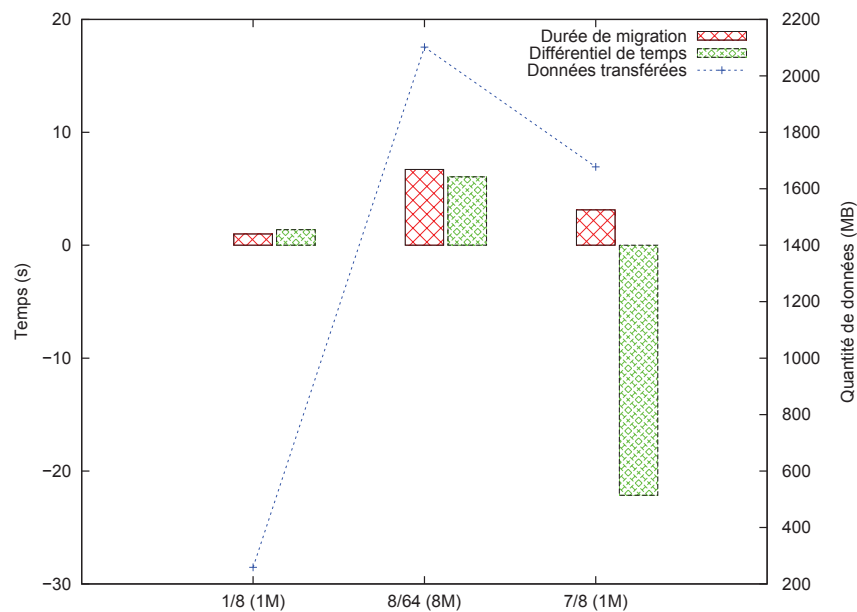


FIGURE 5.8 – Impact des migrations sur le temps d'exécution de HERA

Détail des configurations testées :

- 1/8 : Pour ce premier test, nous utilisons 8 tâches MPI virtualisées pour exécuter HERA avec le cas-test à un million de mailles. Initialement, les 8 tâches MPI sont exécutées sur 8 nœuds de Fortoy. Après dix secondes d'exécution, nous migrons une des machines virtuelles sur un neuvième nœud. Les conditions d'exécution sont donc les mêmes avant et après la migration et le différentiel de temps d'exécution relevé - 1,4 seconde - est intégralement lié à la migration. L'impact d'une migration sur le temps d'exécution d'une application parallèle est donc faible.
 - 8/64 : Pour le test suivant, HERA est exécuté dans 64 tâches MPI virtualisées et traite le cas-test à huit millions de mailles. Les machines virtuelles sont toujours exécutées sur huit nœuds de Fortoy. On teste cette fois-ci le coût de migrations simultanées, puisque l'on fait migrer simultanément les huit machines virtuelles d'un des nœuds vers un neuvième nœud.
- Une fois encore, les conditions d'exécution sont les mêmes avant et après la migration ce qui permet

de relever uniquement l'impact de la migration sur le temps d'exécution. Celui-ci est de 6,1 secondes, la quantité de données à transférer étant de 2,1Go, soit près de 10 fois plus que dans le cas précédent. Ce temps reste faible devant le temps d'exécution typique d'applications de calcul intensif.

- 7/8 : Ce dernier test a pour but de mettre en évidence l'intérêt des migrations de machines virtuelles pour exploiter des ressources de calcul inutilisées. On exécute HERA dans 8 tâches MPI virtualisées pour traiter le cas-test à un million de mailles. Initialement les 8 machines virtuelles sont co-hébergées sur un nœud de Fortoy. On a donc une machine virtuelle par cœur. Au bout de dix secondes, sept machines virtuelles sont migrées, afin que chaque machine virtuelle soit hébergée sur son propre nœud. Bien que la durée de migration soit de 3,1 secondes, le temps total d'exécution est réduit de 22 secondes par rapport au cas où les machines virtuelles restent sur le nœud initial, soit une réduction d'environ un tiers. En effet, chaque tâche MPI dispose alors d'une bande passante mémoire bien supérieure, ainsi que de la totalité de la mémoire cache de son processeur.

5.3.4 Allocation mémoire en contexte virtualisé

Comme nous l'avons mentionné au chapitre 3.2.2, nous avons mis en place une modification du noyau Linux pour optimiser la remise à zéro des pages mémoire. Comme cette modification nécessite un noyau non standard, nous avons cherché à évaluer si les bénéfices, en termes de performances, que nous avons obtenus sur l'hôte sont aussi transposables dans un contexte virtualisé.

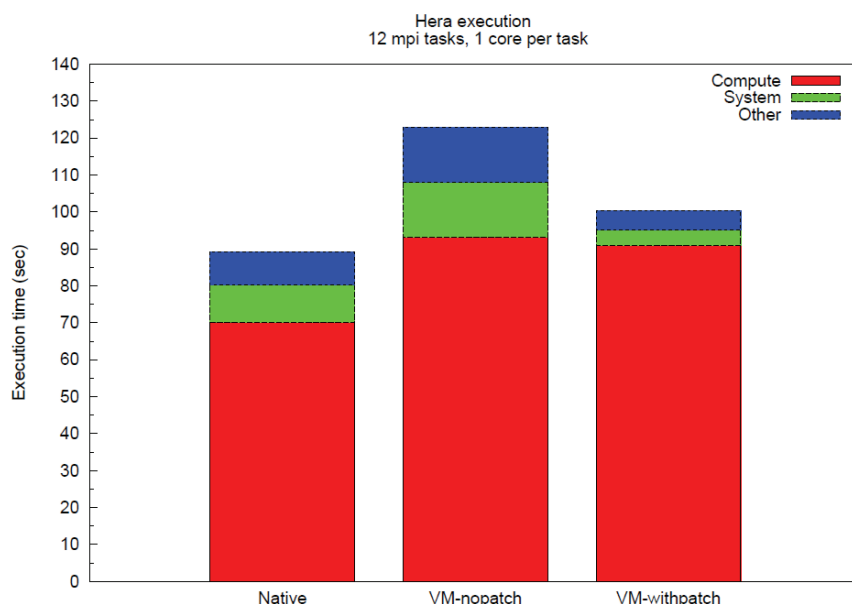


FIGURE 5.9 – Patch noyau en contexte virtualisé

La figure 5.9 illustre l'intérêt de disposer d'un noyau modifié. En effet, on constate une nette amélioration des performances de l'application HERA en contexte virtualisé.

5.3.5 Bilan des évaluations

Dans ce chapitre, nous avons tout d'abord mesuré les performances brutes de notre environnement d'exécution à l'aide de différents tests élémentaires. Cela nous a permis de vérifier qu'il est possible d'instancier une grappe de machines virtuelles pour exécuter une application parallèle, et ce sans incidence significative sur le temps de lancement de l'application. Nous avons par ailleurs constaté les très bonnes performances des opérations MPI en mémoire partagée grâce aux accès mémoire distants virtuels implémentés par notre périphérique. En revanche, les communications inter-nœud affichent une perte d'efficacité par rapport aux communications natives. En sus de la latence liée aux basculements entre hôte et invité, cette perte d'efficacité vient du fait que nous utilisons MPI pour implémenter les communications entre les nœuds.

Nous avons ensuite étudié l'impact de ces variations de performances sur l'exécution de logiciels représentatifs des applications parallèles de calcul scientifique. Les tests NAS nous ont confirmé que, pour

certaines applications très sensibles aux performances des communications, il nous faudra améliorer notre implémentation en nous affranchissant de MPI pour nous rapprocher des performances natives. Cependant nos autres tests applicatifs montrent que notre solution permet déjà d'exécuter des calculs parallèles dans des machines virtuelles avec un surcoût en temps d'exécution inférieur à 10%. En particulier, nos tests sur la plateforme HERA ont permis de confirmer la robustesse de notre solution ainsi que sa capacité à exécuter efficacement de véritables simulations numériques tout en bénéficiant des avantages apportés par la virtualisation tels que la possibilité de migrer les machines virtuelles.

5.4 Conclusion/Travaux futurs

5.4.1 Conclusion

Durant les travaux de thèse de François Diakhaté, nous avons mis en place une méthodologie permettant d'utiliser de manière efficace les couches réseau rapide. L'interface proposée nous a permis de montrer que la virtualisation peut être utilisée en contexte calcul hautes performances avec une pénalité réduite.

Nous avons aussi montré que la virtualisation peut permettre de mettre en place des optimisations spécifiques en espace noyau. En effet, s'il est très compliqué de modifier le noyau hôte, une modification du noyau invité est tout à fait possible. Grâce à cette technique, nous avons montré que les optimisations noyau mises en place dans le chapitre 3 ont aussi des gains significatifs en contexte virtualisé.

5.4.2 Travaux futurs

Avec l'arrivée de la virtualisation dans les centres de calcul, il devient possible de concevoir/optimiser des systèmes d'exploitation optimisés pour le calcul hautes performances. Nous avons déjà mis en œuvre des optimisations au niveau de l'allocation mémoire, mais il reste encore des évolutions des algorithmes d'ordonnancement et d'entrées/sorties qu'il serait très intéressant de mettre en place.

Chapitre 6

Profilage à grande échelle

Dans le contexte du calcul hautes performances, la problématique du débogage et du profilage est à grande échelle (i.e. sur un très grand nombre de cœurs) est cruciale. En effet, il existe de nombreux outils pour traiter ce problème sur un nombre restreint de cœurs mais le passage à l'échelle requiert la mise en place de nouvelles approches.

Le profilage d'application à l'échelle requiert des outils et méthodes générant le moins possible de perturbations sur l'application. Ces outils se doivent aussi d'être particulièrement extensibles. En effet, il n'est pas toujours possible de réduire le cas test à optimiser car la mesure alors réalisée n'est pas représentative du cas réel. L'autre objectif de ces travaux est de proposer une approche adaptée aux codes multithreads.

Ce chapitre détaille les travaux réalisés dans le cadre de la thèse de Jean-Baptiste Besnard[12].

6.1 Problématique

La première difficulté rencontrée dans le profilage à grande échelle est la gestion des accès disques dans l'approche traditionnelle par prise de trace. L'approche par prise de trace montre très rapidement ces limites car le volume de données à stocker augmente avec le nombre de cœurs. On se heurte donc à plusieurs problèmes. Le premier problème est la perturbation induite sur l'application à profiler par la gestion de ces données. En effet bien que les systèmes de fichiers parallèles offrent de très bonnes performances, la mesure d'événement de type MPI, qui peut survenir à une fréquence très élevée (le rythme maximum étant fonction de la latence du réseau rapide), l'impact sur l'application est donc très élevé. C'est pour cela que les outils de profiling demandent souvent de pouvoir stocker toutes les données de la trace en mémoire et attendent la fin de l'application pour écrire réellement les données. Cette approche engendre un deuxième problème qui est celui de la consommation mémoire. En effet, les applications ayant très rarement une extensibilité parfaite, les exécutions vont avoir tendance à utiliser la quasi totalité de la mémoire disponible sur les nœuds de calcul. Cette tendance est amplifiée par la diminution de la quantité mémoire par cœur promise par les architectures exascales.

Pour répondre à cette problématique, nous avons choisi d'élaborer une approche de profiling in situ. L'approche d'analyse in situ permet mécaniquement de réduire le volume de données écrites sur les systèmes de fichier par une valorisation/réduction de données directement à la source. L'idée principale est donc d'appliquer directement les filtres, habituellement réalisés en post-traitement, au moment de la prise de trace. Toujours dans le but de limiter la consommation mémoire ainsi que la perturbation sur l'application, nous avons choisi de déporter ce traitement sur des nœuds dédiés. De plus, ces nœuds pourront être positionnés judicieusement pour être au plus près du système de fichier pour l'écriture des données réduites/valorisées.

Notre méthode de profilage a été mise en œuvre dans l'outil MALP que nous avons développé.

6.2 L'outil MALP

Dans cette section, nous décrirons l'outil Multi-Application Online Profiling (MALP)[11, 12]. Cet outil a été conçu pour s'exécuter en limitant au maximum l'impact sur le système de fichiers parallèle mais aussi pour perturber le moins possible l'application à profiler. Il a de plus été optimisé pour être adapté au contexte multithread sur un très grand nombre de cœurs.

6.2.1 De la trace à l'analyse in situ

Les approches de profilage par prise de trace reposent généralement sur les bibliothèques d'entrées/sorties parallèles. On peut citer le cas de la bibliothèque ScoreP[1] qui repose sur SionLIB[48]. SocreP est une bibliothèque utilisée par de nombreux outils de prise de trace comme Scalasca[49], Vampir[81, 69], Periscope[8, 50] ou TAU[80, 95]. L'approche à base de trace est illustrée sur la figure 6.1.

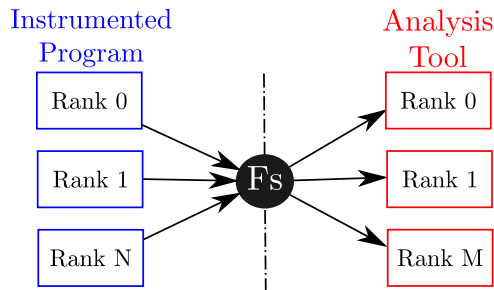


FIGURE 6.1 – Modèle de profilage à base de prise de traces.

Comme on peut le voir, cette approche nécessite un va-et-vient avec le système de fichiers qui devient l'élément central du mécanisme de prise de profilage. De plus, le nombre de cœurs augmentant, la taille des données à stocker sur le système de fichier croît aussi. On se heurte donc au problème de l'espace de stockage pour le profilage des plus grosses simulations. La réponse souvent donnée pour résoudre ce problème est de profiler un cas test plus petit. Néanmoins, comme nous avons pu le voir dans les chapitres précédents cela n'est pas sans impact car les bibliothèques MPI, par exemple, voient leur comportement changer au cours du temps et en fonction des données à échanger. Réduire le cas test pour le profiler et l'optimiser c'est courir le risque de ne pas optimiser les bonnes parties du code.

Une autre approche possible est l'analyse in situ avec un couplage en ligne. La figure 6.2 décrit l'approche de couplage en ligne. Dans cette approche, les données transitent directement sur le réseau sans passer par le système de fichier. Les données issues de l'instrumentation sont directement transférées des nœuds de calcul vers les nœuds d'analyse sur lesquels s'exécutent le ou les outils d'analyse.

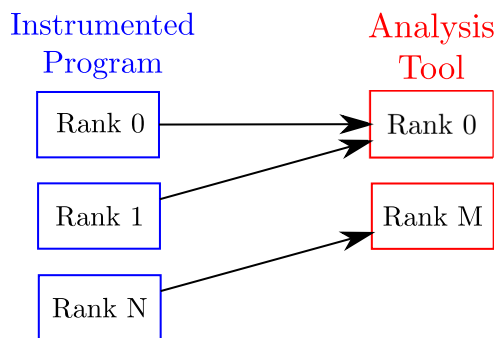


FIGURE 6.2 – Modèle de profilage à base de couplage réseau.

Pour la conception de MALP, nous avons choisi de prendre cette dernière approche.

6.2.2 Des nœuds de calcul vers les nœuds d'analyse

Comme nous l'avons vu précédemment, nous avons choisi une approche avec des analyses déportées sur des nœuds dédiés différents des nœuds de calcul. Nous devons donc disposer d'un mécanisme efficace de transfert des événements issus des sondes sur les nœuds de calcul vers les nœuds d'analyse. Nous avons choisi une approche de type flux. Comme on peut le voir sur la figure 6.3, ces flux sont des canaux entre les nœuds de calcul (writer) et les nœuds d'analyse (reader). Les premiers vont écrire les événements dans les flux et les seconds vont traiter les données issues des flux.

Les flux que nous avons mis en place sont de types persistants et asynchrones. Ils sont multi-directionnels et ont une sémantique proche des *pipes* UNIX. L'écriture dans un flux est non-bloquante tant qu'il reste des tampons de libres en local au niveau de l'écrivain. Ces tampons sont transférés de

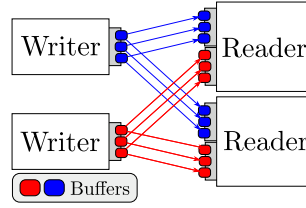


FIGURE 6.3 – Architecture des flux dans MALP.

manière asynchrone sur le réseau. Au niveau du récepteur, le mécanisme d'analyse va scruter de manière active l'arrivée de ces événements.

Nous avons de plus mis en place une topologie virtuelle qui va répartir les flux entre les processus MPI de l'application à profiler et les processus s'exécutant sur les nœuds d'analyse.

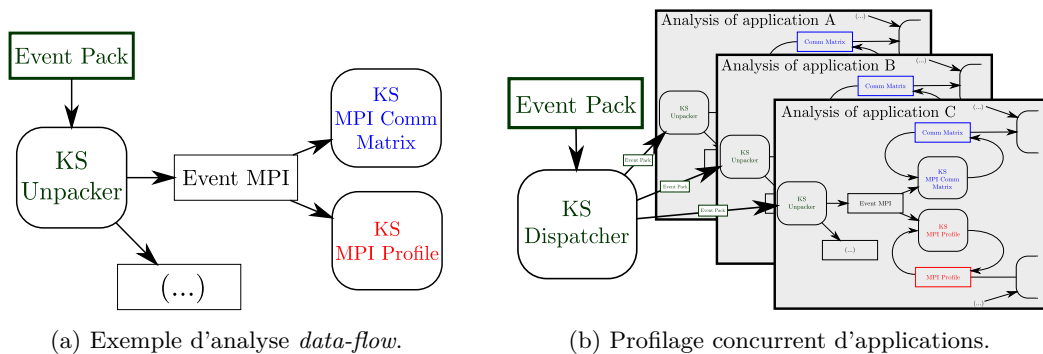
Cette méthode a été mise en œuvre au-dessus de MPI en mode MPMD avec création de sous-communicateurs. Nous avons choisi MPI car il est toujours présent dans les centres de calcul. De plus, MPI a de bonnes performances sur les réseaux rapides. Pour plus de détails se reporter à [12, 11].

6.2.3 Profilage multi-applications

Comme nous venons de le montrer, notre système de flux d'événements est totalement indépendant de la notion d'application. Il nous est donc tout à fait possible de profiler plusieurs applications en parallèle avec un partage des processus et nœuds d'analyse. Grâce à notre mécanisme, il est possible d'avoir un système composé d'une grappe de nœuds dédiés au profilage d'applications avec un mécanisme de connections à la demande des applications via `MPI.Comm.accept` par exemple. Dans le cadre de ces travaux, nous nous sommes limités au profilage multiple d'applications exécutées simultanément grâce au mode MPMD.

6.2.4 Blackboard

Un des buts principaux de MALP est l'extensibilité. Nous avons donc cherché à mettre en place un système supportant les analyses parallèles. Nous nous sommes orientés vers les *blackboards* [44, 34, 32, 35, 33]. Les *blackboards* sont de bons candidats car ils permettent un haut niveau de performances. Ils permettent aussi l'enchaînement et la parallélisation des actions. Il est donc possible pour un événement provenant d'une sonde dans le code d'être traité en parallèle par plusieurs analyses. Il est aussi possible à une analyse de gérer un événement. Ainsi, plusieurs analyses peuvent s'enchaîner.

FIGURE 6.4 – Implémentation du profilage à base de *blackboards*.

Les *blackboards* fournissent une structure de stockage anonyme. Elle permet la cohabitation pour des analyses en chaînes qui sont dédiées à un ensemble de types. Le traitement est enclenché chaque fois qu'une donnée est "déposée" sur le *blackboard*. Notre implémentation repose sur plusieurs threads qui consomment les données de manière parallèle. La figure 6.4(a) présente un exemple de flot de contrôle sur un *blackboard*. Les paquets d'événements provenant des flux issus de l'application profilée sont déposés sur le *blackboard*. Ce dernier déclenche le composant *KS Unpacker* qui décompose le paquet en événements élémentaires. Les événements sont ensuite traités par les analyses (par exemple le profilage MPI). Cette

approche peut être étendue à la gestion du profilage multi-applications. Comme on peut le voir sur la figure 6.4(b), il suffit de disposer de plusieurs *blackboards* et d'un mécanisme de répartition pour analyser plusieurs applications de manière concurrente.

Implémentation du Blackboard

Notre *blackboard* repose sur deux composants principaux : les *Data Entries (DE)* et *Knowledge Sources (KS)*. Une DE peut être définie comme un triplet $\{Type, Size, Payload\}$ où *Type* est un identifiant entier, *Size* est la taille de l'élément *Payload* qui est, quant à lui, une valeur quelconque qui a du sens pour l'analyse. La KS est un couple $\{\{Sensivities\}, Operation\}$ avec $\{Sensivities\}$ définie comme un ensemble de *Types* et *Operation* une fonction à appeler sur la donnée d'entrée.

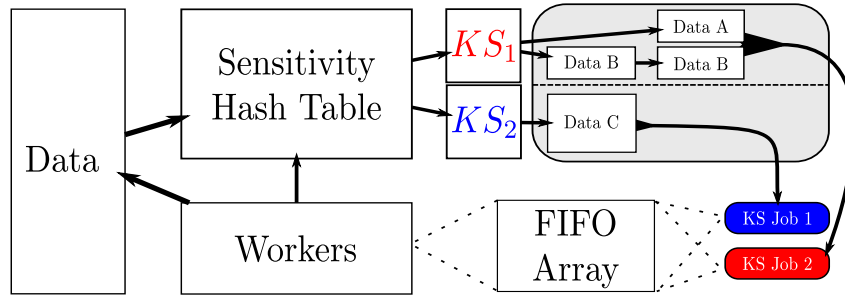


FIGURE 6.5 – Implémentation de notre architecture de *blackboard*.

La figure 6.5 présente l'implémentation de notre *blackboard*. Lorsqu'une DE est soumise au *blackboard*, le système de contrôle recherche toutes les KS qui disposent du type *Type* dans leur ensemble de $\{Sensivities\}$. Chaque fois qu'une KS est trouvée, une entrée du type $\{@\{Data\ entries\}, Operation\}$ est introduite dans un ensemble de listes de type FIFO (nous utilisons ici plusieurs listes pour réduire la contention). Plusieurs thread *workers* sont en charge de scruter ces listes pour exécuter les opérations sur les DE.

Limitations de l'implémentation actuelle

Comme nous l'avons dit précédemment, notre approche permet de profiler plusieurs applications à la fois. Néanmoins, comme nous nous sommes basés sur MPI et le mode MPMD, toutes les applications doivent être lancées via la même soumission à l'ordonnanceur de ressources. Si les applications ont des durées d'exécution différentes, la totalité des cœurs (toutes les applications et celles dédiées aux analyses) seront réservés pendant la durée de la plus longue des applications. Cette limitation est liée à notre utilisation de MPI. Elle pourrait être levée par l'utilisation des fonctions de connexion de processus MPI comme `MPI_Comm_Accept`, par l'utilisation des processus dynamiques de la norme MPI 2 ou encore via un accès direct au réseau rapide.

6.3 Évaluation

Dans cette section, nous allons présenter dans un premier temps différentes analyses que nous avons pu mettre en place au dessus de notre moteur MALP. Ensuite, nous évaluerons les performances de notre approche en termes d'extensibilité et d'impact sur les applications à profiler.

6.3.1 Analyses mises en œuvre

Dans cette section, nous allons présenter différentes analyses que nous avons pu mettre en place au dessus de MALP. Il faut bien avoir à l'esprit que toutes ces analyses peuvent s'exécuter en parallèle les unes des autres et que chaque analyse est elle-même parallélisée avec MPI.

Profils

La première analyse que nous avons mis en place au dessus de MALP est un mécanisme d'extraction de profil des applications.

- **Profils MPI** : toutes les fonctions MPI utilisées avec la taille transmise et la durée de l'appel.

- **Profils POSIX** : toutes les fonctions POSIX avec la durée de l'appel et la taille quand cela est possible.
- **Profils d'appel des fonctions** : nombre d'appels et durée des appels.

Ce type d'analyse très classique ne nécessite pas un grand nombre de ressources en termes de nœuds d'analyse si l'on se contente de profils MPI et/ou POSIX. C'est typiquement ce type d'analyse qui pourrait être systématiquement activé pour, d'une part, aider à l'optimisation des applications et d'autre part monitorer l'état de la machine. Si subitement les profils MPI s'envolent, c'est probablement qu'il y a un problème sur la machine si le code utilisateur n'a pas été changé.

Graphes de communication MPI

La deuxième analyse mise en œuvre est la détection du graphe de communication MPI. Ce graphe permet d'identifier quels sont les processus communiquant le plus entre eux.

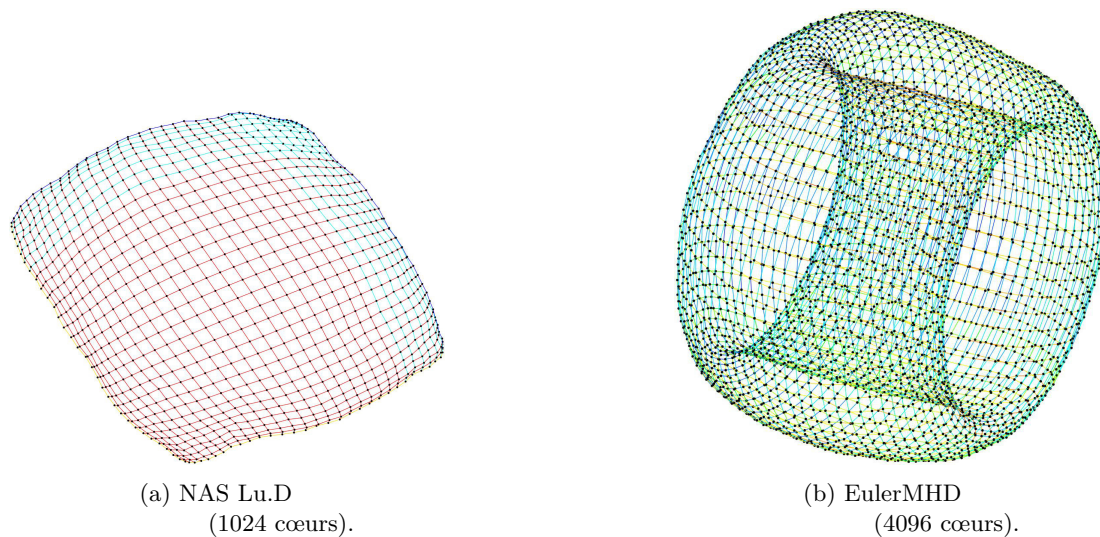


FIGURE 6.6 – Exemple de topologie de communications pondérée.

Comme on peut le voir sur la figure 6.6, MALP a pu générer la topologie des communications réseau sur un nombre de cœurs pouvant aller jusqu'à 4096.

Analyse des capacités d'asynchronisme du code

Une analyse importante pour les codes dans un objectif exascale est l'analyse des capacités d'asynchronisme. Cette analyse permet de déterminer l'intervalle de temps entre l'initiation d'une communication (MPI.Send, MPI.Recv) et l'attente de cette dernière (MPI.Wait).

La figure 6.7 décrit le temps moyen d'asynchronisme pour deux applications : EulerMHD et LBM. Comme on peut le voir sur la figure 6.7(a), bien que le code utilise des appels MPI non-bloquants les capacités de recouvrement de ce code sont très faibles (de l'ordre de $30\mu s$). Sur la figure 6.7(b), le code LBM exhibe beaucoup plus d'asynchronisme avec près de $50ms$. Le code LBM sera donc beaucoup moins sensible au bruit système.

Sortie OTF2

En plus des analyses présentées, nous avons mis en place une méta-analyse OTF2[68]. Cette analyse va en fait permettre de réaliser des sorties OTF2 directement à partir de MALP. Le principal intérêt de cette approche est de permettre l'utilisation d'outils standards d'analyse comme Scalasca, Vampir, ... tout en bénéficiant du mécanisme de déport d'analyse proposé par MALP. Cette sortie OTF2 peut être générée en parallèle des autres analyses grâce à notre système de *blackboard*. On peut donc envisager des approches où les analyses in situ de MALP font une première analyse et la trace OTF2 plus complète n'est conservée que si les analyses in situ montrent qu'il est nécessaire de faire une analyse plus poussée avec les outils standards.

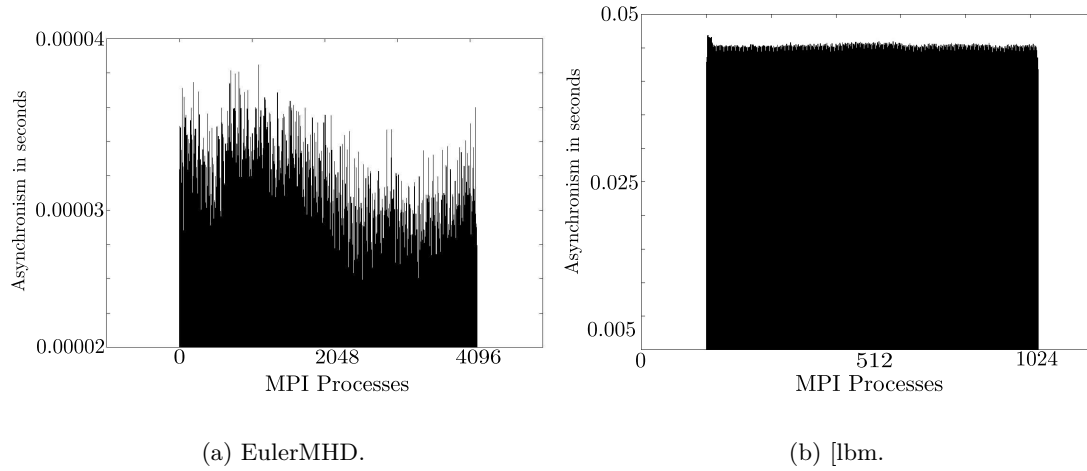


FIGURE 6.7 – Comparaison du niveau d’asynchronisme des applications EulerMHD et lbm.

De nombreuses autres analyses ont été mises en place au dessus du moteur de MALP. Elles sont décrites dans [12]

6.3.2 Surcoût de la méthode d’analyse in situ

Notre chaîne d’instrumentation a été évaluée sur les nœuds 32 cœurs de la machine TERA100. Nous avons réalisé cette expérimentation sur les applications suivantes : NAS-MPI Benchmarks (class C and D) et EulerMHD.

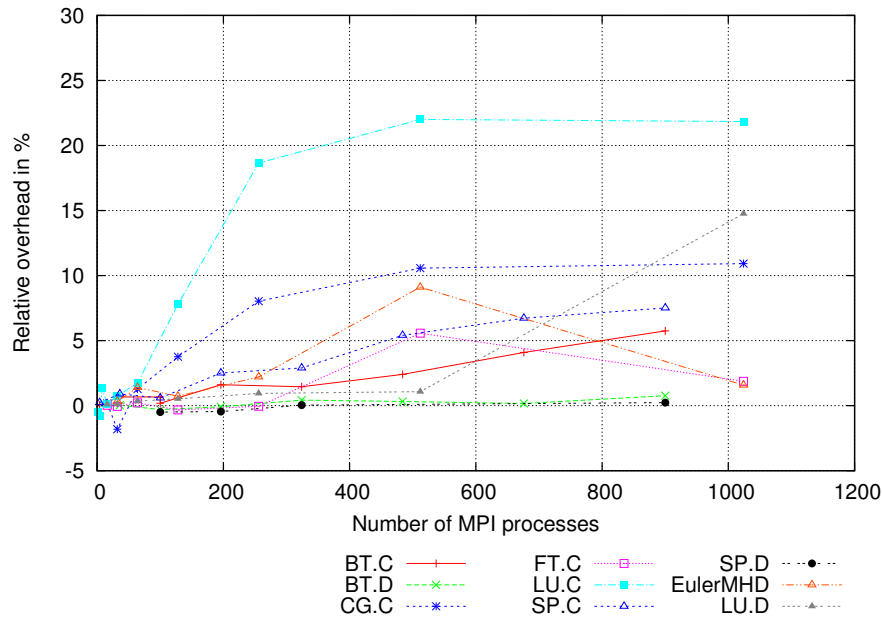


FIGURE 6.8 – Surcoût relatif de notre approche sur les NAS benchmarks et l’application EulerMHD (exécutions sur la machine TERA 100).

La figure 6.8 présente le surcoût de la méthode de profilage sur le temps d’exécution mesuré entre (MPI.Init and MPI.Finalize). Dans ces tests, nous avons réalisé une instrumentation MPI uniquement. Pour ces évaluations, nous avons mis un ratio de $\frac{1}{1}$: un nœud d’analyse pour un nœud de calcul. Nous constatons des surcoûts inférieurs à 25% mais très variables en fonction de l’application. En particulier, les cas NAS de classe C montre un surcoût supérieur à ceux de classe D. Ce phénomène peut s’expliquer en calculant la bande passante nécessaire pour l’extraction des données : $\overline{B}_i = \frac{\text{Total event size}}{\text{Execution time}}$. Avec des applications ayant une plus grosse charge de travail, le débit nécessaire pour le traitement de l’information est réduit. Si on compare le benchmark SP classe C par rapport à SP classe D pour 900 cœurs. SP.C

nécessite un débit de $\overline{B}_i(\text{SP.C}) = 2.37 \text{ Go/s}$ alors que SP.D nécessite un débit de seulement $\overline{B}_i(\text{SP.D}) = 334.99 \text{ Mo/s}$. Il est donc nécessaire de disposer de suffisamment de nœuds d'analyse pour atteindre la bande passante $\overline{B}_i$ nécessaire pour limiter l'impact de l'analyse sur le code à analyser. Une des forces de MALP est de disposer de cette variable d'ajustement.

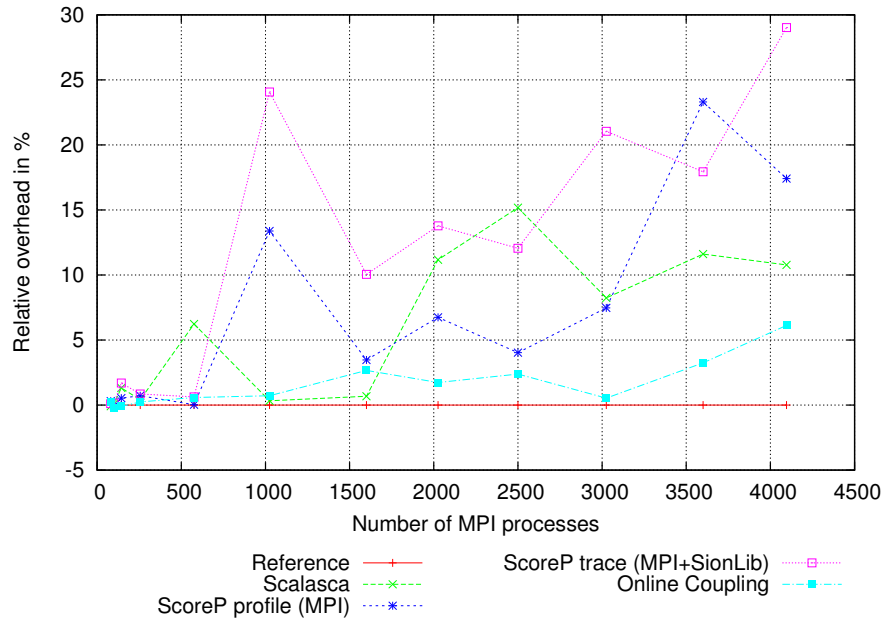


FIGURE 6.9 – Surcoût relatif de différents outils de profilage sur le benchmark NAS SP.D (exécutions réalisées sur la machine Curie).

La figure 6.9 montre le surcoût relatif d'une activité de profilage pour le NAS benchmark SP.D sur la machine Curie. Nous avons comparé MALP avec deux outils de profiling Scalasca 1.4.3[98, 49] et ScoreP 1.1.1[1]. Le dernier génère des traces OTF2 compatibles avec Vampir[81], Scalasca ou TAU. Les mesures ont été faites avec uniquement le profil MPI activé en utilisant SionLib[48] dans le cas des traces ScoreP. Comme on peut le voir sur ce benchmark, notre analyse en ligne est beaucoup moins impactante bien qu'elle manipule un volume de données plus important. Les traces ScoreP varient de 313 Mo à 116 Go alors que les volumes échangés au sein de MALP varient de 923.93 Mo à 333.22 Go. Ceci suggère que notre approche en ligne est plus extensible que l'approche basée sur le système de fichiers parallèle.

La figure 6.10 présente les isovaleurs du surcoût de l'instrumentation pour le benchmark NAS LU.D dans le cas d'une instrumentation MPI. Bien que le plus faible impact soit obtenu avec un ratio $\frac{1}{1}$, il est possible d'avoir un surcoût acceptable avec moins de cœurs d'analyse. Sur notre exemple, un ratio de 1 cœur d'analyse pour 10 cœurs de calcul permet d'obtenir un surcoût inférieur à 10%.

6.4 Conclusion/Travaux futurs

6.4.1 Conclusion

Dans ce chapitre, nous avons décrit une approche de profilage d'applications in situ utilisant des nœuds dédiés au profilage en plus des nœuds dédiés à l'application. Cette approche permet de limiter l'impact sur l'application à profiler et ce même à très grande échelle. De plus, l'analyse en ligne permet de réduire l'impact sur le système de fichier car seules les données "valorisées" sont stockées.

Ces travaux s'inscrivent dans une démarche d'optimisation d'une application, mais aussi de meilleure utilisation des ressources des centres de calcul. En effet, la grande modularité du moteur MALP permet de construire des analyses spécifiquement dédiées aux supports exécutifs. Une de ces analyses est la détection du taux d'asynchronisme d'une application. Avec cette métrique il est possible de piloter l'agressivité d'une méthode comme le Collaborative-Polling. En effet, il est inutile de chercher à traiter les données d'une autre tâche en générant du trafic NUMA si les capacités d'asynchronisme sont limitées.

Un autre avantage de l'approche MALP est la gestion des architectures Many-core ou GPGPU. En effet, ces architectures ne sont pas bien adaptées pour réaliser des activités de profilage, leurs cœurs

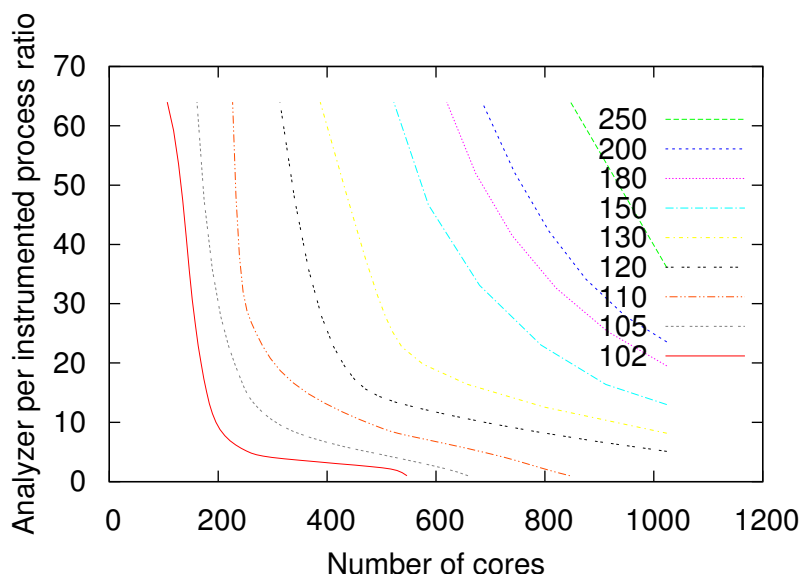


FIGURE 6.10 – Surcoût de MALP en extensibilité forte sur le benchmark NAS LU.D.

n'étant unitairement pas assez puissants. Les cœurs des hôtes sont quant à eux tout à fait capables de réaliser efficacement ces opérations. MALP, via le déport des tâches de profiling, peut facilement offrir un mécanisme de profilage efficace dans ce contexte.

Les travaux réalisés sur MALP ont donné lieu à une collaboration avec l'équipe du projet MAQAO qui a intégré une partie des travaux réalisés pour MALP dans MAQAO. MALP a aussi fait l'objet d'un dépôt à l'APP.

6.4.2 Travaux futurs

Les travaux réalisés autour de MALP peuvent être poursuivis de deux manières. Tout d'abord les travaux autour du moteur même. Comme nous l'avons précédemment décrit, MALP est basé sur MPI pour tous les aspects communication. Il pourrait être intéressant de s'affranchir de cette contrainte en ajoutant une couche basse réseau rapide. On pourra par exemple utiliser celle de MPC. Cela permettrait de mettre en place des grappes de profilage. Les applications pourraient alors se connecter dynamiquement à cette grappe. Pour pouvoir faire cela, le modèle MPI n'est pas adapté.

Le second axe d'évolution est situé autour des analyses. Nous avons déjà fait un premier pas vers les autres outils de profilage grâce au module OTF2 de MALP.

Chapitre 7

Conclusion

Les travaux de recherche présentés dans ce document reflètent mes activités de recherche durant ces sept dernières années. Ils représentent principalement les travaux de quatre doctorants que j'ai eu la chance de co-encadrer ainsi que celui de nombreux étudiants qui, par l'intermédiaire de stages de recherche ont contribué à cet effort. Le travail des ingénieurs, apprentis de l'équipe a contribué à la robustesse et la stabilité de ces travaux. Tous ces travaux s'inscrivent dans une démarche de recherche et développement cohérente et ont été conduits suivant une même démarche scientifique fédératrice. La thématique concerne l'étude et la conception de supports exécutifs efficaces multithread dans le contexte d'un centre de calcul et leur capacité à exploiter les architectures des super-calculateurs de manière performante tout en offrant une base solide à la conception d'applications hautes performances hybrides.

7.1 Travaux réalisés

Tous les travaux réalisés ont eu pour objectifs la performance mais ils devaient aussi pouvoir être intégrés dans un environnement de production. J'ai donc mené mes activités de recherche sur deux axes en fortes interactions. Le premier est tout d'abord le support exécutif lui-même. Le second est l'étude des environnements d'exécution comme la virtualisation qui ont à la fois un impact certain sur le support exécutif mais qui permettent aussi d'utiliser en contexte de production des optimisations de type noyau, impossible sans la virtualisation. Enfin, pour permettre l'utilisation de supports exécutifs comme MPC, il était nécessaire d'étudier, en concert avec les développeurs d'applications, des méthodes et outils les aidant à utiliser nos contributions.

7.1.1 Supports exécutifs multithread ...

Notre étude des supports exécutifs s'est placée dans le contexte multithread. En effet, je pense qu'avec l'évolution des architectures, l'utilisation du multithreading va croître. Avec MPC, nous offrons deux approches d'utilisation des threads. La première passe par le *thread-based* MPI. Cette approche est l'idée historique que j'avais mise en place pendant ma thèse. Cette approche a été fiabilisée et enrichie pour fournir à l'heure actuelle une implémentation MPI conforme au standard MPI 1.3. Je pense que pour l'avenir le MPI *pur* peut poser de grosses difficultés à l'extensibilité des codes. Cette difficulté n'est pas directement liée à MPI mais plus à la manière dont ce standard est classiquement utilisé. La réplication de données pose en effet de gros problèmes de consommation mémoire aux développeurs de codes. C'est pourquoi certains d'entre eux s'orientent vers un modèle à base de threads comme OpenMP en plus du MPI. Pour ces applications, il est important de disposer d'un support exécutif MPI offrant de bonnes performances MPI_THREAD_MULTIPLE.

Dans le contexte multithread de l'exécution des codes sur les architectures à mémoire hiérarchique, l'allocateur mémoire joue un rôle clé. En effet, cet allocateur doit à la fois être performant pour offrir aux applications un faible coût d'allocation. Il doit aussi maintenir la localité des données de manière transparente à l'utilisateur qui ne désire pas vraiment se soucier de ce problème tout en offrant tout de même des mécanismes permettant aux applications ou même en interne du support exécutif de contrôler le placement des données. On constate donc qu'un allocateur doit avant tout être très flexible.

La conception de notre allocateur offre cette flexibilité. Il permet par construction de garantir la localité des données. Cette localité des données est très importante pour les applications s'exécutant

en contexte *thread-based* MPI. En effet, avec ce modèle de programmation, les données de l'application sont par nature réparties de manière exclusive entre les threads. Notre allocateur avec ces tas locaux et ces *memory sources* NUMA permet d'allouer de manière locale et de conserver cette localité des données même dans le cas où l'application alloue/désalloue très fréquemment. Traditionnellement, ce type de comportement nécessite un grand nombre d'allers/retours avec le système d'exploitation pour garantir la localité avec les techniques de *first-touch*. Notre approche permet de plus, de ne pas avoir de recours à de fréquents appels au système d'exploitation. Ceci nous permet d'avoir de très bonnes performances en contexte NUMA multithread. Dans le cas où l'application est du type MPI + Thread, il est possible d'utiliser des primitives pour forcer la localité ou simplement utiliser le *first-touch*. Dans ces cas, la réutilisation des segments de mémoire libérés est gérée de manière astucieuse afin que les futures allocations obtiennent bien la localité désirée. Les travaux sur la localité des données ont aussi donné lieu à un dépôt de brevet en collaboration avec la société Bull.

Comme nous venons de le rappeler, notre allocateur va pour des raisons de performances limiter les interactions avec le système d'exploitation. Ceci se traduit par une augmentation de la consommation mémoire. Cette surconsommation mémoire est liée aux tampons qui vont conserver les segments mémoire libérés pour une possible réutilisation future. Néanmoins, il existe certains cas où une application peut être très consommatrice en mémoire ou avoir des pics de consommation mémoire. Dans ce cas de figure, il est important que l'allocateur puisse à la volée changer son comportement pour fournir à l'application les segments de mémoire demandés. De manière générale, le problème de la consommation mémoire a été étudié pour proposer des méthodes de factorisation de données ou de réduction de la taille des tampons avec comme objectif principal de permettre aux applications de repousser les limites des nœuds de calcul quitte à dégrader ponctuellement les performances.

Enfin, dans ce domaine la frontière est fine entre l'espace utilisateur et l'espace noyau. Nous nous sommes donc autorisés à franchir la barrière de l'espace utilisateur en proposant une approche en espace noyau pour limiter le coût de l'allocation mémoire dans le cadre du calcul hautes performances. Bien que cette modification ne puisse pas facilement être intégrée en standard dans le noyau Linux, nos études sur la virtualisation nous ont permis de mettre en œuvre cette technique dans un contexte de production.

Dans le contexte multithread, l'accès au réseau en contexte *thread-based* MPI comme en contexte `MPI_THREAD_MULTIPLE` est critique. Nous avons donc mené des travaux pour obtenir de bonnes performances en réduisant la contention sur l'accès aux structures réseau en contexte NUMA multi-rail. Ayant obtenu de bonnes performances en contexte multithread, nous nous sommes posés la question de ce que pouvait nous apporter le multithreading comme opportunités d'optimisation de supports exécutifs. Nous avons donc proposé la méthode de Collaborative-Polling qui permet de bénéficier du multithreading pour recouvrir les communications par du calcul. Enfin, toujours dans notre politique de réduction de l'empreinte mémoire, nous avons étudié comment adapter dynamiquement la consommation mémoire des couches réseau.

Dans le contexte multithread, l'accès aux ressources réseau doit se faire sans générer de contention ni de trafic NUMA important qui perturberait les cœurs de calculs non-impliqués dans la communication. Pour adresser ce problème, nous avons introduit le concept de *vraill*. Un *vraill* représente un ensemble de tampons et un accès à une ressource réseau. Comme nous l'avons vu, les *vraills* peuvent se partager une même ressource réseau physique. Les *vraills* sont donc des abstractions de la ressource réseau. Ils sont liés à l'allocateur mémoire pour positionner les tampons au plus près des cartes réseau. Nous pouvons faire varier le nombre de ces *vraills* pour contrôler la contention. L'étude que nous avons menée a abouti sur une implémentation de cette abstraction qui nous a permis de gérer efficacement la contention multithread dans le contexte de nœuds de calcul de grande taille avec des effets NUMA marqués et plusieurs cartes réseau.

Comme nous venons de le rappeler, le multithreading complexifie l'accès à la ressource réseau. Néanmoins, il peut aussi faciliter le recouvrement des communications par du calcul en offrant une vision globale partagée des ressources réseau. Pour bénéficier de cet avantage, nous avons introduit le concept de Collaborative-Polling. Le Collaborative-Polling nous permet au sein d'un nœud de calcul de faire de manière transparente de la délégation de progression de messages. L'idée derrière le Collaborative-Polling est d'utiliser les cycles "perdus", i.e. les cycles passés à attendre une communication pour faire progresser l'ensemble des messages à traiter sur le nœud de calcul. Cette méthode est particulièrement adaptée aux codes faiblement déséquilibrés (bruit système) et avec peu ou pas de recouvrement exprimé au niveau de l'application. Cette méthode est aussi une alternative efficace aux threads de progression pour une implémentation MPI. Cette méthode a été mise en œuvre au sein de MPC et a montré des gains en performance significatifs.

Enfin, tous ces travaux sur les couches réseau ont été menés avec le souci de la consommation mémoire. Nous avons donc proposé et mis en place dans MPC une architecture réseau capable de contrôler dynamiquement son empreinte mémoire. Cette architecture ne fait pas que limiter son empreinte, elle est capable dynamiquement d'ajuster sa consommation aux besoins de l'application en jouant sur le ratio consommation mémoire/performances mais aussi en évitant de conserver des structures/connexions qui ne sont plus ou peu utilisées.

Tous ces travaux ne peuvent être utilisés par les développeurs d'applications que si premièrement elles sont valables dans un environnement de production d'un centre de calcul et deuxièmement, si elles disposent d'outils adaptés pour le profilage et le débogage.

7.1.2 ... dans l'environnement d'un centre de calcul

En parallèle de nos études sur les supports exécutifs, nous nous sommes posé la question du contexte d'exécution de ces travaux. Nous nous sommes tout d'abord intéressés à l'exécution des supports exécutifs en contexte machines virtuelles. En effet, ces dernières sont de plus en plus utilisées dans le contexte d'un centre de calcul. Actuellement, elles sont encore limitées aux nœuds d'administration, mais leur flexibilité en fait une solution qui devrait apparaître aussi sur les nœuds de calcul.

Nous avons donc mené une étude sur l'utilisation des machines virtuelles en contexte calcul hautes performances. Après avoir constaté un impact limité des machines virtuelles sur les performances séquentielles des applications, nous nous sommes attelé au parallélisme. Tant que l'on reste au sein d'une seule machine virtuelle, l'impact reste toujours limité. Néanmoins, dès qu'il s'agit d'utiliser les réseaux rapides, la problématique est beaucoup plus complexe. En effet, si l'on veut pouvoir conserver les capacités de migration et une abstraction du matériel sous-jacent, il faut disposer d'une interface réseau rapide et non de type Ethernet comme le proposent traditionnellement les machines virtuelles.

Nous avons donc proposé une étude et une implémentation d'un support réseau hautes performances en contexte virtualisé nommé VMPI. VMPI est une interface conçue pour pouvoir construire une implémentation MPI performante entre machines virtuelles. VMPI nous permet de disposer de tous les avantages des machines virtuelles (migration, protections/reprises, portabilité, ...) sans pour autant sacrifier les performances. Nous avons montré que l'impact de la virtualisation pouvait être réduite à 10% de surcoût sur le temps d'exécution total en contexte multi-nœuds MPI.

Nous avons aussi profité de cette étude pour évaluer le comportement de notre support exécutif et en particulier les modifications que nous avons apportées au noyau Linux. Cette évaluation a montré qu'il était possible et bénéfique de spécialiser un système d'exploitation au contexte calcul hautes performances tout en continuant de pouvoir être exécuté dans l'environnement de production classique d'un centre de calcul.

Nous nous sommes ensuite posé la question du profilage à l'échelle d'applications multithread. En effet, une des contraintes du profilage est d'être le moins possible intrusif dans le comportement de l'application. C'est pourquoi nous nous sommes orientés vers l'utilisation de nœuds dédiés. Le profilage doit être aussi extensible que l'application la plus extensible s'exécutant sur le centre de calcul. On peut donc dire que l'outil de profilage doit pouvoir profiler la machine. On constate de plus que les applications vont s'influencer les unes les autres. Il est donc intéressant de pouvoir profiler plusieurs applications au sein du même outil.

Ces études ont été concrétisées dans l'outil MALP. MALP a été conçu pour limiter au maximum la pression sur le système de fichiers parallèle. En effet, le système de fichiers est une ressource critique sur les grands centres de calcul. Trop solliciter ce système de fichiers a pour effet d'impacter les applications s'exécutant lors du profilage y compris l'application à profiler. MALP est donc conçu autour d'un moteur d'analyse in situ. Ce moteur va permettre de limiter le volume écrit sur le système de fichiers au strict minimum utile. En effet, seules les données "valorisées", i.e. les données qui ont été traitées/réduites/consolidées, seront écrites.

Toujours dans l'objectif de réduire l'impact sur l'application profilée, nous avons opté pour l'utilisation de nœuds de calcul dédiés plutôt que de faire l'analyse sur les nœuds utilisés par l'application. Cette approche a trois avantages. Le premier avantage est de ne pas perturber les phases de calcul de l'application. Seules les phases de communication peuvent être impactées lors du transfert des informations issues des sondes insérées dans l'application vers les nœuds d'analyse. Pour limiter ce phénomène, nous avons mis en place des mécanismes asynchrones. Le deuxième avantage de la méthode est l'adaptation aux machines dont l'accès au système de stockage n'est pas uniforme. Il est alors possible de positionner les

nœuds d'analyse sur les nœuds de calcul qui auront les performances entrée/sortie les plus élevées. Dans le cadre de l'utilisation de machines virtuelles, il peut être utile de placer les processus d'analyse sur des nœuds non virtualisés qui auront un accès direct aux ressources de stockage. Le troisième avantage de cette méthode est ne pas réduire l'espace mémoire disponible pour l'application. En effet, seuls quelques tampons sont utilisés et donc l'impact sur la mémoire disponible est très faible.

Enfin, MALP a été optimisé pour être capable de profiler plusieurs applications qui s'exécutent de manières concurrentes. En effet, les applications s'exécutant de manière concurrente sur un super calculateur partagent des ressources comme certains switchs réseau, elles peuvent se perturber mutuellement. Il est donc intéressant de fournir un outil qui donnera un diagnostic pertinent dans ce contexte d'exécution réel au lieu d'une analyse *in vitro*. Il est ainsi possible d'optimiser une application pour qu'elle ait le meilleur niveau de performances quelles que soient les applications s'exécutant sur la machine.

7.2 Projets et perspectives

L'importance des supports exécutifs dans le calcul hautes performances n'a cessé d'augmenter ces dernières années avec l'apparition de "l'hyperparallélisme" requis par l'évolution des architectures. Les projections Exascale confirment cette évolution. La principale difficulté, outre l'appréhension de la complexité matérielle, est de permettre au développeur d'applications scientifiques d'exprimer toutes les informations nécessaires au support exécutif afin que ce dernier puisse adapter son comportement et obtenir de bonnes performances. Une autre difficulté à laquelle les supports exécutifs doivent faire face est l'adaptation au contexte d'exécution de production d'un centre de calcul. Dans chacun des différents chapitres que nous avons vus, nous avons présenté des perspectives spécifiques à chacune des thématiques. Ici, nous allons donner plus une vision globale des perspectives pour les supports exécutifs ainsi que l'environnement des centres de calcul.

7.2.1 Supports exécutifs, localité des données et équilibrage de charge

Contexte

Le premier point critique dans l'évolution des architectures est la gestion des données. En effet, les architectures sont de plus en plus hiérarchiques[60, 19]. Il est donc nécessaire, pour obtenir de bonnes performances, de placer les données au plus proche du cœur qui va les exécuter. Dans un contexte parfaitement équilibré entre les tâches, il est aisé de garantir la localité par exemple par du *punaisage* (courant en contexte MPI). En revanche dans les cas déséquilibrés, qui représentent la majorité des exécutions, il sera nécessaire d'adapter dynamiquement l'ordonnancement et le placement des tâches en fonction des données. Il sera donc nécessaire de faire évoluer les supports exécutifs et en particulier les ordonnanceurs en créant des liens entre ces derniers et les mécanismes d'allocation mémoire. A l'heure actuelle, les ordonnanceurs cherchent uniquement à maintenir la localité, des threads par exemple, pour éviter les accès NUMA. Néanmoins, rien ne garantit à l'ordonnanceur que les données soient vraiment présentes sur tel ou tel nœud NUMA.

Cette problématique de localité des données est aussi présente dans les accès réseaux, en particulier, en contexte multirail. Le multirail est de plus en plus répandu car il permet la conception de nœuds de calcul de grande taille. De plus, il permet, comme nous l'avons vu, de diminuer la contention d'accès aux cartes. Le multirail peut être physique i.e. plusieurs cartes réseau ou canaux virtuels (Queue Pair dans le contexte InfiniBand). La localité des données est particulièrement critique sur les réseaux rapides car ces réseaux utilisent les mécanismes de RDMA. Les performances de ce mécanisme sont significativement liées aux placements des données au même titre que les cœurs de calcul. Actuellement, la seule précaution prise pour optimiser le placement des données est le placement des tampons mémoire au plus près des cartes réseau. Néanmoins, ceci n'a d'effet que sur les messages de petites tailles (protocole *eager*) ainsi que sur les données de contrôle transférées dans le cadre du protocole réseau de rendez-vous. La gestion des données utilisateur est uniquement faite, a priori, en considérant le respect par l'utilisateur du modèle MPI pur monothread où toutes les données sont locales par construction.

Les supports exécutifs (ordonnanceur, couches réseau, allocateur mémoire) peuvent tirer parti de la localité des données. Néanmoins, ils ne peuvent que faire des conjectures car les modèles de programmations utilisés par les développeurs de codes ne permettent pas de définir de manière précise la localité des données. Il faut donc définir à la fois une manière d'exprimer, de manière précise, le placement des données et fournir le support exécutif approprié pour utiliser au mieux cette information. Pour répondre à cette problématique, nous avons choisi de concentrer notre étude sur un algorithme déséquilibré en par-

ticulier : le transport de particule Monté-Carlo[43]. Ce type d'application a pour caractéristique d'avoir une charge déséquilibrée car non prédictible du fait du tirage aléatoire de la direction des particules. De plus, le schéma d'accès aux données est lui aussi difficile à prévoir. En effet, pour une particule donnée, on ne peut savoir à l'avance quelles seront les mailles, et donc les matériaux, qu'elle va devoir traverser (voir figure 7.1). De plus, une particule peut en générer de nouvelles au cours du trajet de cette dernière. Le nombre d'événements que va subir une particule va donc être aléatoire et donc le temps nécessaire pour traiter une particule n'est pas connu à l'avance.

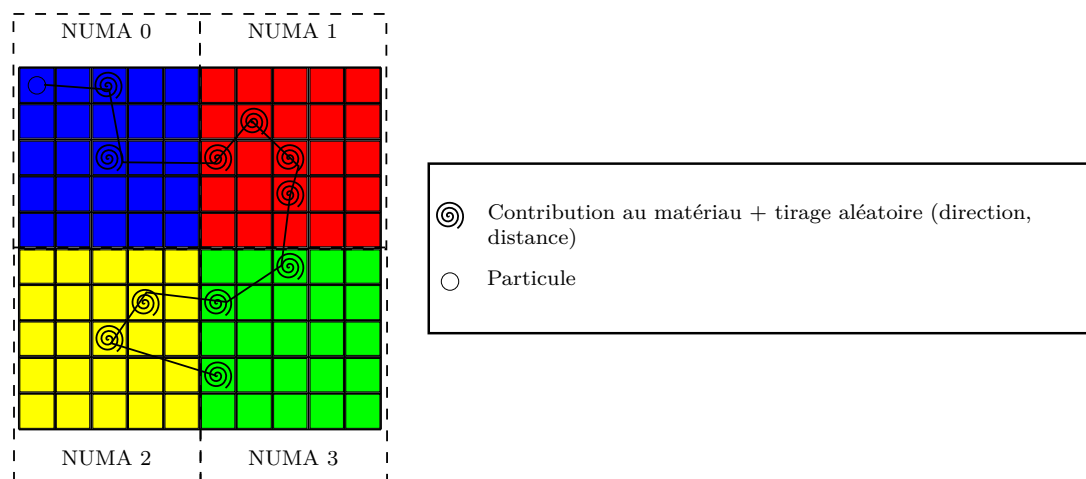


FIGURE 7.1 – Principe de l'algorithme de transport de particules Monté-Carlo

Pour paralléliser une telle méthode, nous nous heurtons donc rapidement au problème d'équilibrage de charge. En effet, si l'on répartit les particules par exemple sur des threads, la répartition a priori ne peut pas garantir l'équilibrage de la charge entre les threads. Ces problèmes d'équilibrage de charge vont aussi induire des problèmes de localité des données. En effet, pour équilibrer la charge, il faut dynamiquement choisir quel thread va exécuter quelle particule sans connaître le temps de traitement de la particule ; temps de traitement qui va être allongé si la particule est traitée par un thread qui devra réaliser des accès distants aux structures de données (matériau par exemple). Si de plus, on doit exécuter cet algorithme de transport de particules sur un maillage décomposé, par exemple via MPI, il se pose alors la problématique de "sortie" des particules du sous-domaine. En effet, une particule située dans une maille du bord du sous-domaine peut prendre une direction qui va lui imposer de changer de sous-domaine. Cette sortie de sous-domaine va engendrer une migration des particules d'un sous-domaine à l'autre. Avec une approche MPI classique, la décomposition du maillage en sous-domaine va définir la répartition des particules (les particules devant être traitées par le sous-domaine qui contient les mailles sur lesquelles les particules contribuent). Or, rien ne garantit que cette décomposition équilibre le travail sur les particules entre les sous-domaines.

Modèles de programmation pour applications déséquilibrées

Pour répondre à la problématique du transport de particules Monté-Carlo sur supercalculateur, il est nécessaire de faire une étude approfondie des différents modèles de programmation permettant de lever les contraintes de l'approche classique MPI + thread soulevées précédemment. Nous avons réalisé une étude préliminaire sur la base d'un code réel[43] et de petites maquettes utilisant le modèle classique MPI. Ces études nous ont donné les résultats préliminaires suivants :

Approche OpenMP pur L'approche OpenMP pur permet de bien utiliser les nœuds de calcul mais ne permet pas de tirer partie de l'aspect multi-nœuds. En intra-nœud, nous constatons que le modèle mémoire partagée nous permet d'équilibrer efficacement la charge de calcul. Néanmoins, cet équilibrage est fortement impacté par les aspects NUMA. Pour obtenir de bonnes performances en terme d'équilibrage de charge, il est nécessaire de pouvoir réaliser du vol de tâche. En effet, il est préférable de voler du travail de manière transparente sur les cœurs les plus chargés plutôt que de devoir interrompre un cœur déjà chargé pour qu'il donne du travail. L'approche mémoire partagée offre donc un avantage certain pour la mise en place d'une stratégie d'équilibrage de charge efficace.

Approche MPI pur L’approche MPI pur “classique” (hors MPI one-sided) permet de garantir une bonne localité des données. En effet, une particule ne peut évoluer que dans le sous-domaine géré par le processus qui la contient. Les données relatives aux particules sont elles aussi bien localisées. La contre-partie de cette approche est sa faible capacité d’équilibrage de charge. En effet, comme chaque maille n’est présente que dans un seul sous-domaine, les particules devant interagir sur cette maille devront nécessairement être traitées par le processus qui détient cette maille. L’équilibrage de charge doit donc être réalisé au niveau du partitionnement du maillage et est donc par nature relativement statique. Enfin, en pur MPI, la sortie d’une maille d’un sous-domaine implique une communication de type point à point coordonnée entre le processus qui voit la particule sortir de son sous-domaine et celui qui la voit entrer.

Approche MPI + OpenMP réplication totale de domaine L’approche couplée MPI + OpenMP avec réplication totale de domaine consiste à répliquer l’approche OpenMP pur. Dans chaque processus MPI, nous allons trouver le domaine de calcul complet ainsi que les threads OpenMP en charge de traiter les particules. Cette approche permet d’augmenter le nombre de particules traitées lors de la simulation sans pour autant perdre les capacités d’équilibrage de charge de l’approche OpenMP. En effet, avec cette technique on constate statistiquement que la charge dans chaque processus MPI est équivalente dès lors que le nombre de particules dans chaque processus est équivalent. Cette approche souffre néanmoins d’un problème majeur : la synchronisation des sous-domaines. En effet, il est nécessaire en fin de phase Monté-Carlo de fusionner les contributions des particules sur chacun des réplicats. Cette fusion génère classiquement une opération de type MPI_Allreduce dimensionnée à la taille du domaine de calcul. Cette réduction peut représenter, d’après nos évaluations, jusqu’à 50% du temps de calcul. Enfin, l’approche réplication de domaines est limitée par la quantité mémoire disponible sur les nœuds de calcul. Il faut suffisamment de mémoire pour pouvoir stocker le domaine complet plus un nombre significatif de particules pour que cette approche soit utilisable. Plus le nombre de particules par réplicat est élevé, plus le coût de la synchronisation est faible.

Approche MPI réplication partielle de domaine La réplication partielle de domaine permet de lever la limitation liée au MPI pur sans pour autant engendrer une consommation élevée comme la réplication totale de domaine. En effet, cette approche suit la même démarche que le MPI pur en décomposant le domaine de calcul, mais nous allons en plus répliquer les mailles qui pourraient être utilisées dans la phase de transport Monté-Carlo. Cette approche permet donc de ne répliquer que les mailles effectivement utilisées par les particules de chaque processus MPI. L’autre avantage de cette approche est qu’elle ne nécessite plus de réduction sur la totalité du domaine, mais seulement les mailles répliquées. Néanmoins, la réplication partielle ne résout pas la problématique des particules sortant du domaine de calcul (si la réplication n’a pas été suffisante). En effet, il est très difficile de prévoir à l’avance les mailles qui seront utilisées lors de la phase transport de particules. Cette approche de réplication n’apporte donc une solution que dans les cas très simples 1D voir 2D.

Ces résultats préliminaires nous ont montré que, sur le problème de transport de particules Monté-Carlo, l’approche classique MPI + thread couplée avec une répartition statique des données ne permet pas “a priori” d’équilibrer la charge et ne permet pas non plus d’avoir une localité des données. Il est donc nécessaire de trouver une nouvelle approche au niveau des modèles de programmations pour avoir une approche performante pour ce type de problème. Il faudra aussi s’assurer que le modèle de programmation choisi dispose d’un support exécutif suffisamment efficace pour tenir les performances. En effet, pour résoudre notre problème, nous avons besoin de pouvoir exprimer du parallélisme à grain fin (pour permettre l’équilibrage de charge) et ce sur architecture mémoire distribuée (car les problèmes que nous avons à résoudre ne peuvent tenir sur la mémoire d’un seul nœud de calcul). Nous allons donc nous orienter vers des modèles de type PGAS (Partitioned Global Address Space)[30, 46, 15, 29] ou “MPI One-sided”[83].

Approche PGAS Les PGAS permettent de facilement distribuer des données sur différents processus ou nœuds de calcul. La différence par rapport aux approches de type MPI se fait sur le mécanisme d’accès aux données distantes. En effet, dans les PGAS, l’accès se fait de manière quasi transparente du point de vue de la cible. Le modèle permet de réaliser les accès distants par des mécanismes réseau comme les RDMA qui tirent bien parti des réseaux rapides présents sur les calculateurs. Dans le contexte des applications de type Monté Carlo, nous allons étudier l’efficacité de ces mécanismes d’accès distant qui vont nous permettre, par exemple, d’éviter le transfert de particules qui requièrent d’accéder à des mailles non présentes localement et/ou réaliser du vol de particules sur les processus les plus chargés. Dans le premier cas, nous allons voir tout d’abord s’il est possible d’exprimer un accès à des mailles distantes et quelles contraintes sur les structures de données

sont à mettre en place. Nous évaluerons aussi différents algorithmes de réplication paresseuse de mailles pour permettre une meilleure localité des données. Enfin, nous travaillerons sur les supports exécutifs eux-mêmes pour déterminer quels sont les points-clés pour obtenir de bonnes performances. Dans le second cas, nous allons étudier l'efficacité et la programmabilité des mécanismes de synchronisation fine (opérations arithmétiques atomiques, compare and swap, ...) présents dans les PGAS pour concevoir des algorithmes de vol de tâches appliqués au transport de particules Monté Carlo distribuées sur une grappe de calcul. Nous porterons aussi une attention particulière à la cohabitation des modèles PGAS avec le modèle MPI. En effet, si l'approche PGAS peut être très adaptée au problème du transport, il n'est pas du tout clair que ce soit aussi le cas pour les autres modèles physiques qui pourraient être plus facilement exprimés avec un modèle MPI.

Approche “MPI One-sided” Une alternative aux modèles PGAS est l'approche MPI One-sided. L'interface MPI One-sided est apparue dans MPI-2 et a été enrichie dans la norme MPI-3. Elle permet à l'image des PGAS de réaliser des accès distants et des synchronisations fines. Cette interface offre par construction une bonne cohabitation avec les approches MPI traditionnelles. Il est donc très important de l'évaluer pour voir quelles sont ses forces et faiblesses. Nous étudierons aussi comment un support exécutif optimisé/étendu pourrait nous aider à obtenir de bonnes performances sur le problème du transport de particules Monté Carlo.

Les PGAS ou l'approche “MPI One-sided” permettent de réaliser des mécanismes d'équilibrage de charge via vol de tâches ou d'accès distant aux données qui ne sollicitent pas les cœurs les plus chargés même durant la phase de vol. Les travaux que nous envisageons ne sont pas liés aux mécanismes de vol de tâches eux-mêmes mais cibleront plutôt les supports exécutifs. Le but de cette étude est de définir une approche et un support exécutif adéquats pour répondre au problème d'équilibrage de charge en contexte transport de particules Monté-Carlo.

Supports exécutifs

La performance d'un mécanisme d'équilibrage de charge distribué adapté au transport de particules Monté-Carlo va fortement dépendre du support exécutif sous-jacent qui devra à la fois être très réactif (en particulier lors des accès sur un nœud distant), mais devra aussi fournir toutes les informations de localité des données nécessaires pour que l'algorithme d'équilibrage de charge fasse les bons choix. Pour mener à bien notre étude sur les algorithmes, nous étudierons en détail trois mécanismes clés des supports exécutifs : l'ordonnanceur de tâches, l'allocateur mémoire et le support du réseau rapide.

Ordonnanceur L'ordonnanceur de tâches est le mécanisme central responsable du maintien de la localité des données. En effet, comme nous l'avons mentionné dans le chapitre allocation mémoire (voir 3), l'allocation mémoire est souvent paramétrée en first-touch. C'est ensuite à l'ordonnanceur de tâches de garantir que les tâches ne vont pas être ordonnancées “loin” de leurs données (sur un nœud NUMA distant). Dans le cadre de notre étude de l'équilibrage de charge en transport de particules Monté-Carlo, nous allons porter une attention particulière aux algorithmes d'ordonnancement topologiques [88, 87].

Allocateur mémoire L'allocateur mémoire est le point de passage obligé pour toutes les structures de données. C'est aussi lui qui a toute la connaissance de la distribution des données sur les nœuds NUMA. Nous allons faire évoluer ces allocateurs pour qu'ils coopèrent avec l'ordonnanceur de tâches pour maintenir la localité des données tout en permettant l'équilibrage de charge.

Réseau La composante réseau va tenir aussi un rôle important dans les performances des PGAS ou MPI One-sided. En effet, l'accès aux données distantes va se faire via les interfaces réseau rapide. Les optimisations nécessaires seront doubles. Tout d'abord, il faudra mettre en place un support réseau efficace en contexte multithread pour les opérations unidirectionnelles. En effet, les études préliminaires nous ont montré qu'une approche multithread au niveau du nœud de calcul est très appropriée pour l'équilibrage de charge. Ces travaux sont dans la suite logique des travaux menés au chapitre 4, mais cette fois-ci appliqués aux opérations unidirectionnelles. Nous envisageons aussi un lien fort avec l'allocateur mémoire. En effet, nous avons remarqué dans notre étude préliminaire que la méthode de réplication de domaine par partie peut être une solution à notre problème. Il faudra donc étudier la possibilité de mécanismes de cache et de prefetching via RDMA pour maintenir la localité et rapatrier des données via le réseau en avance de phase.

Conclusion

Le problème du transport de particules Monté-Carlo est un problème complexe qui a attiré notre attention car il requiert d'étudier la pile complète allant de l'algorithme haut niveau (modèle de programmation) jusque dans les supports exécutifs. Nous allons utiliser ce fil directeur pour mener une étude approfondie des modèles de programmation pour applications déséquilibrées ainsi que l'interaction de ces nouveaux modèles avec les approches existantes. En effet, on peut reprocher aux nouveaux modèles de programmation de nécessiter une réécriture complète des applications. Nous avons donc choisi de porter une attention toute particulière à la cohabitation des supports exécutifs. Ici on ne parle pas de "runtime stacking" car ce n'est pas un modèle qui en appelle un autre mais bien plusieurs modèles qui s'exécutent en alternance suivant le problème physique à résoudre. Nous nous attarderons aussi à proposer des évolutions des supports exécutifs pour permettre d'atteindre le niveau de performances désiré. Il existe de nombreux travaux sur les transferts de données, que se soit la littérature liée à l'utilisation des caches ou du prefetching, ou encore les transferts de données sur architecture hétérogène (par exemple avec des GPUs). Ici, nous nous focaliserons sur le support exécutif lui-même et l'interaction entre les composants (ordonnanceur, allocateur et réseau) pour permettre de recouvrir efficacement ces transferts.

7.2.2 Consommation mémoire

Contexte

Avec l'évolution des architectures des supercalculateurs et l'évolution des modèles physiques et des modèles numériques, la mémoire consommée par les codes de calculs devient une ressource critique. En effet, avec l'arrivée des processeurs multi/many cœurs, la quantité mémoire par cœur a tendance à stagner voir diminuer ; bien que la quantité de mémoire totale sur le calculateur soit en augmentation. Dans ce contexte, la notion de quantité mémoire par cœur est très importante. En effet, comme nous l'avons vu précédemment, la localité des données est une notion très critique pour les performances. Il faut donc s'attacher à conserver une vision locale de la gestion de l'allocation mémoire (localité par nœud NUMA, localité dans les caches, ...). Ceci implique une vision de la mémoire par cœur ou petit groupe de cœurs pour éviter les dégradations de performances. Dans le même temps, les applications voient leurs besoins en mémoire augmenter. Par exemple, la modélisation des phénomènes physiques devient de plus en plus évoluée et requiert plus de variables et donc plus de mémoire. Il est donc important de gérer au plus juste la mémoire utilisée et veiller à ne pas gaspiller de ressources sans pour autant sacrifier les performances.

Les applications que nous avons été amené à étudier présentent un autre phénomène : des schémas d'allocation et une consommation mémoire par phases. On peut par exemple citer les différents modèles physico-numériques qui s'enchaînent, les phases de calculs et celles de communications, les phases de calculs et celles d'entrées/sorties, ... Chacune de ces phases a son propre schéma d'allocation, ses variables temporaires et donc sa propre consommation mémoire. On constate souvent sur les codes le phénomène suivant : la quantité mémoire durant les phases de calculs est raisonnable (70-80% de la mémoire du nœud) et un pic de consommation mémoire survient lors des phases d'entrées/sorties. Ce pic de consommation peut aller jusqu'à provoquer l'arrêt du code du fait de cette surconsommation mémoire. L'arrêt du code survient car sur de nombreux calculateurs, le mécanisme de *swap* est désactivé pour garantir une meilleure stabilité des nœuds de calculs. Les utilisateurs doivent donc prêter une attention particulière à l'empreinte mémoire de leurs cas tests car il n'y a aucune possibilité de dépassement, même temporaire, de la mémoire disponible sur le nœud. Or, sur un nœud de calculs, il ne s'exécute pas seulement les applications des utilisateurs. Il y a aussi tous les démons systèmes (batch manager, système de fichier, démons de log, ...) ainsi que la bibliothèque MPI. L'utilisateur doit donc dimensionner son cas en fonction de la mémoire réellement disponible (environ 90% de la mémoire physique sur les machines du CEA) et doit connaître précisément la consommation mémoire maximale de chacune des phases qu'il va utiliser.

L'évaluation précise, a priori, de la consommation d'une application est très difficile à évaluer. C'est pourquoi, nous nous proposons d'étudier des méthodes permettant d'offrir plus de flexibilité à l'utilisateur en utilisant des algorithmes d'allocation mémoire tenant compte de ce phénomène de phases et permettant aux codes de calcul de "passer" les pics de consommation mémoire.

Approche statique ou manuelle

Comme nous l'avons vu au chapitre 3, une approche statique permet, via un ensemble de paramètres, de faire varier le ratio consommation mémoire par rapport au coût de l'allocation/désallocation (appels à `malloc`, `realloc`, `free`, ...). Cette approche nous permet donc de contrôler la consommation mémoire au plus juste ou bien d'obtenir un allocateur très peu coûteux. Néanmoins, cette approche ne permet

pas de répondre directement au problème des phases de calcul. En effet, il faudrait pouvoir changer les paramètres d'allocation en cours de calcul. Ce changement de paramètres requiert de pouvoir modifier les différents seuils de l'allocateur au cours de l'exécution. Cette fonctionnalité a été implémentée dans notre allocateur. Néanmoins, pouvoir changer les seuils ne suffit pas. En effet, même après avoir changé les seuils, nous héritons de la distribution de la mémoire avant le changement de seuils. Cette mémoire est potentiellement fragmentée et il existe des tampons mémoire réservés préalablement par l'allocateur. Dans l'approche que nous avons mis en place, il est possible de purger certains de ces tampons. Pour résoudre le problème de la fragmentation, il est nécessaire de réaliser des désallocation/réallocation des blocs mémoire pour permettre la défragmentation de la mémoire.

Une méthode statique ou manuelle de la gestion mémoire peut donc permettre de contrôler la consommation mémoire d'un code ou des phases d'un code. Néanmoins, cette approche va impacter les performances. Comme nous l'avons vu, un allocateur mémoire économe en consommation mémoire va fréquemment solliciter le système d'exploitation et avoir des politiques de recherches de blocs libres coûteuses pour éviter la fragmentation. Cet allocateur va aussi très probablement sacrifier la localité mémoire au sens NUMA au bénéfice de la consommation mémoire. L'approche statique va donc réduire les performances du code au cas où il y ait un dépassement de la consommation mémoire. Cette approche est donc difficilement recevable en contexte HPC. Il serait préférable d'avoir une approche dynamique qui choisirait parmi les différentes méthodes que nous avons vues au chapitre 3 celle la plus appropriée en fonction de la mémoire libre sur le nœud de calcul. Il sera aussi nécessaire de mettre en place des méthodes pour basculer d'un mode à l'autre sans pour autant hériter des "mauvaises" décisions du mode précédent.

Approche dynamique

L'approche dynamique de réduction de l'empreinte mémoire des applications soulève le problème du modèle de coût ainsi que des mécanismes de détection de la consommation mémoire. Regardons dans un premier temps la détection de la consommation mémoire. Il existe deux façons de connaître la consommation mémoire. La première consiste à interroger le système d'exploitation. Cette méthode est précise car le système a la vision exacte du nombre de pages mémoire affectées à chaque processus. Néanmoins, cette interrogation a un coût très élevé car elle requiert des appels systèmes. Sous Linux, cette interrogation demande l'ouverture d'un fichier, la lecture de ce fichier et le traitement des données. Il est donc difficilement envisageable d'effectuer cette interrogation avant chaque allocation mémoire. De plus, cette vision ne donne qu'un instantané. En effet, les mécanismes d'allocation paresseuse (first touch) ne permettent pas de connaître, au moment de l'allocation, l'incrément exact de mémoire consommée (il se peut que certaines zones mémoire ne soient jamais utilisées et donc jamais allouées). C'est la différence que l'on constate entre la mémoire virtuellement consommée et celle résidente. A chaque interrogation du système nous disposons donc de la mémoire maximale qui peut être allouée et de la consommation instantanée. Ces données peuvent varier entre deux interrogations même s'il n'y a pas eu d'appels à l'allocateur mémoire.

La consommation mémoire maximale peut être évaluée directement en espace utilisateur (sans appels systèmes). En effet, lors de l'allocation mémoire sur un système UNIX, l'allocateur fait soit un appel *mmap* ou un appel *sbrk*. L'allocateur a donc une connaissance précise de la quantité de mémoire virtuelle consommée. Cette valeur est une approximation de la quantité mémoire réellement consommée. Néanmoins, la mise à jour de cette mesure à un coût sur le parallélisme. En effet, le maintien à jour de cette information nécessite une synchronisation globale à l'échelle du nœud de calcul (incrément atomique). Pour des raisons de performances, nous devrions sans doute nous contenter d'une valeur approchée (par exemple, seulement précise par nœud NUMA) et sommée de manière relâchée (sans prise de verrous) entre les différents nœuds NUMA. La précision d'une telle mesure sera plus faible lors des phases d'allocation massive car il se peut que la sommation ne tienne pas compte des allocations en cours. Néanmoins, le temps de l'opération de réduction des consommations des nœuds NUMA est largement inférieure au temps d'une allocation faisant augmenter la mémoire virtuelle consommée ($\text{temps}(\sum_{i=0}^{nb \text{ nœuds NUMA}} \text{consommation mémoire virtuelle}_i) \ll \text{temps}(mmap/sbrk)$). La précision de la mesure non synchronisée devrait donc être suffisante pour la mise en place des modèles de coût.

Les modèles de coût à étudier vont donc devoir être adaptatifs et utiliser des mesures approchées dans les phases de calcul où la consommation mémoire est faible et devront utiliser des méthodes plus précises lorsque la consommation mémoire devient critique (consommation proche de toute la mémoire du nœud de calcul). Ces modèles de coût devront en outre faire évoluer les différents seuils en cours de calcul pour réguler l'empreinte mémoire. Les modèles de coût devront être couplés à des mesures échantillonnées pour voir l'impact des modifications de politiques. Par exemple, une détection de forte consommation

mémoire par la méthode d'échantillonnage, et ce, dans une phase sans appel à l'allocateur, va nécessiter de remettre en cause les décisions passées. Il est donc nécessaire de pouvoir modifier les allocations déjà effectuées.

Pour pouvoir modifier les allocations effectuées, nous avons à notre disposition deux approches. La première approche se situe au niveau du système d'exploitation lui-même. En effet, le mécanisme de pagination mémoire nous offre la flexibilité de changer une adresse physique (et donc la mémoire associée) sans avoir besoin de modifier au niveau mémoire virtuelle les allocations déjà réalisées (pas d'impact sur le code applicatif). Il est possible par exemple de fusionner les pages mémoire ayant des données communes via des mécanismes comme KSM (voir chapitre 3.4.2). L'étude que nous avons faite nous a montré les limites de KSM en contexte calcul hautes performances. Néanmoins, il est possible de faire évoluer cette technique pour la rendre plus extensible avec plusieurs démons en charge de la fusion des données communes. De plus, il faudrait introduire la notion de localité des données dans KSM pour maintenir l'aspect NUMA de l'allocateur. L'approche de fusion des pages communes permet de compresser et donc de réduire la consommation mémoire au niveau système. En revanche, elle ne permet pas de défragmenter la mémoire.

En ce qui concerne la défragmentation, nous envisageons plutôt une approche basée sur les *Garbage Collectors*. En effet, pour pouvoir efficacement défragmenter la mémoire, il faut pouvoir déplacer les données déjà allouées. Déplacer les données revient à changer les adresses mémoires virtuelles des blocs concernés. Il faut donc un suivi exact de l'utilisation des adresses mémoire et des liens entre les structures de données. Les technologies *Garbage Collectors* comme [13] utilisées principalement dans les machines virtuelles telles que JAVA ou C# disposent du suivi des blocs mémoire et de mécanismes de défragmentation. Dans notre étude, en contexte calcul hautes performances, la fonctionnalité de base des *Garbage Collectors* qu'est la libération automatique de blocs mémoire inutilisés ne sera pas utilisée pour réguler l'empreinte mémoire des applications. Nous nous concentrerons sur les mécanismes de défragmentation mémoire qui vont nous permettre de repositionner tous les blocs mémoire d'une application en changeant toutes les références à ces blocs. Cette défragmentation va nous permettre de supprimer les petits blocs mémoire inutilisés qui sont d'une taille inférieure à la taille d'une page et qui par conséquent ne peuvent pas être rendus au système d'exploitation. L'utilisation de ces mécanismes combinée avec un allocateur mémoire dédié HPC doit permettre de mieux contrôler la consommation mémoire et de l'adapter en fonction de la mémoire disponible sur le nœud de calcul.

Enfin, notre approche va disposer d'un mécanisme dit de dernier recours : de type *swap*. Nous avons dit auparavant que le *swap* était désactivé sur les nœuds de calcul. Cette désactivation est due au fait que certains démons systèmes peuvent voir leurs données mises sur disque par le mécanisme de *swap* et donc réduire leur réactivité. Cette diminution de réactivité peut engendrer de mauvaises interprétations des systèmes de monitoring des nœuds et donc remonter des erreurs aux administrateurs systèmes. Dans notre cas, nous envisageons une approche de *swap* purement applicative. L'application va d'elle-même utiliser un mécanisme de *swap* pour réduire son empreinte mémoire. Dans ce contexte, seule l'application sera fortement ralentie sans impacter significativement les autres applications et démons s'exécutant sur le nœud. Cette méthode de dernier recours doit permettre de supporter des pics de consommation mémoire très courts et permettre à l'application de poursuivre son exécution.

Conclusion

Nous allons proposer à travers cette étude une méthode de contrôle de la consommation mémoire des applications en fonction de la mémoire disponible sur les nœuds de calcul. La figure 7.2 résume notre approche (les pourcentages sont donnés à titre d'exemple). Avec cette approche, il sera plus aisé pour l'utilisateur d'applications d'utiliser efficacement les grands clusters des centres de calculs.

7.2.3 Virtualisation en contexte calcul hautes performances

Contexte

L'évolution de la pile logicielle des calculateurs met en avant la virtualisation. L'utilisation de machines virtuelles est déjà courante dans les domaines de l'administration des machines ou encore dans le *cloud*. Dans le domaine du calcul hautes performances, cette technologie n'est pas encore très utilisée car elle souffre de limitations de performances ou de fonctionnalités en particulier sur réseaux rapides. En effet nous avons vu, grâce aux travaux présentés dans les chapitres 4 et 5 que l'utilisation des réseaux rapides en contexte HPC et virtualisé requiert de nombreuses optimisations. Dans les travaux présentés au chapitre

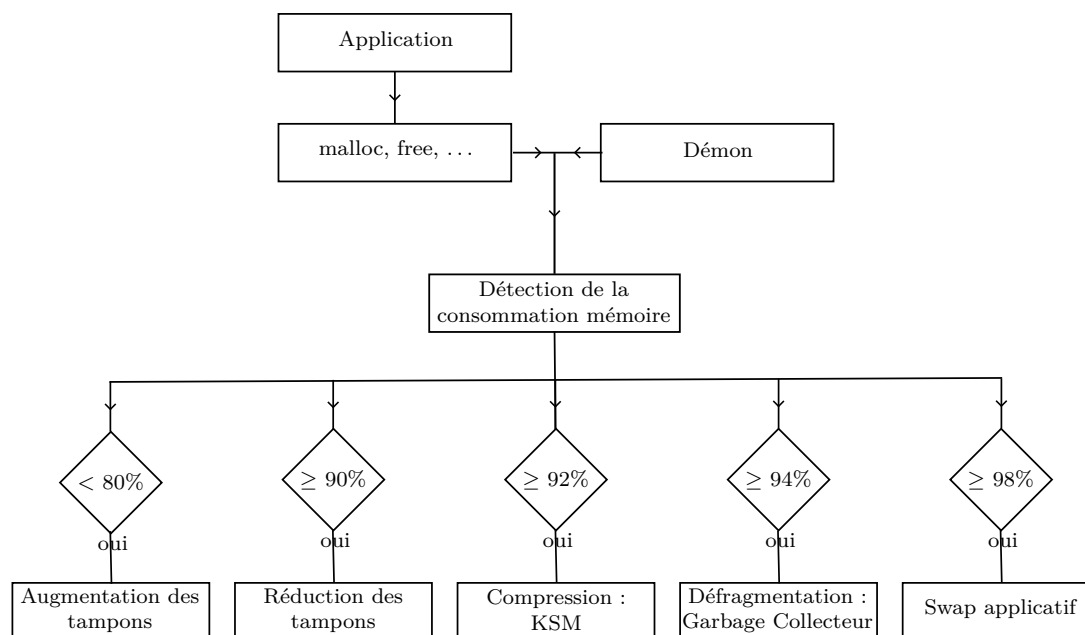


FIGURE 7.2 – Mécanisme d’ajustement dynamique de la consommation mémoire

5 nous avons réalisé une analyse des besoins en contexte virtualisé ainsi qu’un premier prototype de machines virtuelles compatible avec les besoins du HPC.

Les travaux que nous comptons mener, pour étendre les résultats déjà obtenus, devront déterminer quels sont les pré-requis nécessaires au niveau des réseaux rapides en contexte calcul hautes performances. Ces travaux devront de plus montrer qu’il est possible d’utiliser les réseaux rapides sans pour autant sacrifier la portabilité des machines virtuelles que sont la migration de machines virtuelles et les mécanismes de tolérance aux pannes via la création de points de reprises. Nous allons détailler ici les pistes que nous comptons explorer.

Méthode de projection du matériel dans les machines virtuelles

L’utilisation des réseaux rapides en contexte virtualisé repose actuellement sur une méthode dite de projection des cartes réseau dans la machine virtuelle. Cette projection consiste à fournir l’accès direct à la carte réseau depuis la machine virtuelle à la manière de l’utilisation du matériel en contexte natif. Cette méthode a l’avantage de ne pas nécessiter de modification de la pile logicielle invitée : du pilote de la carte à l’application, tout est identique au mode natif. Cette technique de projection peut être réalisée soit en reposant sur un support matériel au niveau de la carte[42, 86] soit en retirant l’accès de l’hôte à la carte pour le donner à l’invité.

Cette méthode offre une grande simplicité d’utilisation et permet d’atteindre des performances identiques ou très proches du mode natif. Néanmoins, avec cette méthode, nous perdons la flexibilité offerte par les machines virtuelles. En effet, la migration de machines virtuelles demande alors d’être réalisée pour chaque matériel projeté, il n’y a plus de possibilité de migrer d’un matériel à un autre, il faut maintenir les pilotes à jour dans les machines invitées, ... Ceci est dû au fait que nous avons choisi d’avoir un comportement dans la machine virtuelle identique à celui de l’hôte. Nous héritons donc des contraintes de l’hôte. Réaliser de la migration avec des matériels projetés dans les machines virtuelles requiert donc de modifier toute la pile logicielle comme il aurait été nécessaire de le faire en natif + MPI pour faire de la tolérance aux pannes par exemple. Or, de nombreux travaux ont montré la difficulté de réaliser ce type de mécanismes [56, 16, 23].

Une autre contrainte de la méthode de projection est l’adhérence de la machine virtuelle au matériel sous-jacent. En effet, la projection requiert d’avoir un pilote de périphérique réseau rapide compatible avec le périphérique voire le firmware du périphérique en question. Dans ce contexte, la machine virtuelle devient liée au matériel et on peut se poser la question de l’utilisation d’une “couche d’abstraction” qui reste liée aussi fortement à l’hôte. C’est sans doute ce dernier point qui ralentit le plus l’adoption de la technologie machine virtuelle en HPC.

Nous nous proposons donc d'étudier une méthode qui permettrait de lever les deux contraintes citées précédemment sans pour autant impacter significativement les performances.

Méthode d'abstraction des périphériques réseaux

La méthode d'abstraction des périphériques réseaux consiste à définir un périphérique réseau qui émule un réseau rapide. Dans les travaux présentés au chapitre 5, nous avons montré qu'il était possible de concevoir une interface virtuelle adaptée au modèle de programmation MPI supportant la migration. Néanmoins, cette approche ne permettait pas de mettre en place de la tolérance aux pannes car il était impossible d'arrêter réellement le démon du périphérique virtuel s'exécutant sur les hôtes des nœuds de calcul.

La méthode d'abstraction des périphériques doit permettre à la fois d'avoir de bonnes performances mais aussi d'autoriser la tolérance aux pannes. Nous allons maintenant esquisser une approche permettant de résoudre ce problème. La première fonctionnalité des réseaux rapides qu'il faut garantir est la performance. Cette dernière se décompose en deux parties : latence et bande passante. La latence des réseaux rapides est assurée par le fait que l'accès à la carte réseau est réalisé en *OS bypass* qui permet d'accéder à la carte réseau sans avoir à impliquer le système d'exploitation. Pour garantir ce type de méthode en contexte virtualisé, il est nécessaire de ne pas avoir à quitter le mode invité mais d'accéder directement à l'hôte (sans *hypercall*). L'interface que nous allons mettre en œuvre devra donc permettre l'*OS-bypass* mais aussi l'*hyperviseur-bypass*. Nous allons donc nous orienter vers une interface descriptive de la structure de message qui va permettre à l'hôte de pouvoir interpréter les messages sans avoir à faire des aller-retour avec l'invité. Comme nous l'avons vu au chapitre 5 maintenir une latence faible en contexte *hyperviseur-bypass* nécessite d'avoir des threads chargés de faire avancer les communications. Nous avons donc trois solutions qui s'offrent à nous. La première consiste sur architectures manycœurs ou architectures disposant de threads matériels de dédier un cœur logique. Cette approche peut s'avérer très intéressante si l'application n'a pas une extensibilité suffisante pour occuper pleinement tous les cœurs logiques. Dans le cas d'applications plus extensibles, nous pourrions baser la progression des messages sur des techniques comme le *collaborative polling* qui permet d'utiliser les périodes d'attente pour faire progresser l'ensemble des messages du nœud. Enfin, dans le cas d'applications parfaitement parallélisées (sans phases d'attente), l'utilisation des *hypercall* pourra être envisagée. En effet, ces applications vont être fortement asynchrones pour pouvoir tolérer les variations de latence sur un réseau en charge et pour avoir un recouvrement des communications par le calcul. Ces applications seront donc par construction moins sensibles à la latence des communications. On peut donc se permettre de dégrader légèrement la latence dans ce cas.

En ce qui concerne le débit, les performances de bande passante des réseaux rapides reposent principalement sur le mécanisme de RDMA. Comme nous l'avons vu, ce mécanisme permet de donner à la carte réseau une adresse mémoire et une taille ; la carte se charge du transfert des données à la vitesse maximale. L'interface que nous allons concevoir devra donc permettre ce type d'accès à la carte sans pour autant interdire la migration de machines virtuelles. Au sein des bibliothèques MPI, les RDMA sont utilisés par exemple dans les algorithmes de types rendez-vous. Cet algorithme nécessite un échange de messages entre l'émetteur et le récepteur comme nous l'avons vu au chapitre 4. Pour que cet algorithme soit efficace dans le contexte des machines virtuelles, il est nécessaire d'éviter les aller-retour entre l'hôte et l'invité. En effet, comme nous l'avons vu, le coût d'un *hypercall* est élevé. Il sera donc nécessaire de proposer la progression de l'algorithme de rendez-vous directement dans le driver de réseau virtuel.

La méthode d'abstraction des périphériques réseau permet une plus grande flexibilité que la méthode de projection. En effet, l'abstraction permet de ne plus avoir d'adhérence entre la pile logicielle dans la machine virtuelle et les pilotes réseau. Ceci nous permet donc d'envisager une tolérance aux pannes capable de "survivre" à la mise à jour de la machine hôte. Cette approche nous permet de plus de déporter une grande partie des algorithmes de l'invité vers l'hôte et donc de libérer le MPI invité de toutes fonctionnalités liées à la tolérance aux pannes. En particulier, le MPI invité peut rester une souche MPI quelconque, il sera juste nécessaire de lui adjoindre un support compatible avec notre abstraction.

Un support exécutif supportant migration et tolérance aux pannes

Nous avons vu précédemment et dans le chapitre 5, que la méthode d'abstraction de périphérique réseau est viable pour faire de la tolérance aux pannes et de la migration. Dans les travaux que nous comptons faire, nous ne traiterons pas explicitement les algorithmes haut niveau de gestion des pannes, mais adresserons uniquement les aspects supports exécutifs nécessaires pour permettre la mise en place de la tolérance aux pannes. Dans cette section, nous nous plaçons dans l'espace hôte. Nous allons esquisser un

type d'interface qui sera fourni à l'invité ainsi que la manière dont nous comptons utiliser le périphérique réseau rapide.

Nous pensons qu'il est nécessaire de laisser le moins d'activités liées au protocole réseau dans le MPI invité. Nous envisageons de nous inspirer des caractéristiques d'interfaces comme Portals[92] ou Elan[109] qui permettent de déporter tout ou une partie des algorithmes MPI. En effet, avec les réseaux supportant ces interfaces, il est possible de réaliser les "matching" des messages point à point directement dans la couche réseau. La phase de matching MPI consiste à associer un MPI.Send avec un MPI.Recv en fonction des communicateurs, tag et rang fournis par l'utilisateur lors des appels MPI. Ces interfaces réseau permettent aussi de déporter des algorithmes plus complexes comme des collectives asynchrones. Avec ce type d'interface, il est possible d'obtenir un niveau de performances élevé mais aussi d'assurer un recouvrement maximum des communications par le calcul.

Cette approche de gestion de la complexité des protocoles MPI dans la couche réseau est aussi très adaptée à la mise en place de mécanismes de tolérance aux pannes. Sur réseau rapide, les principales difficultés rencontrées lors de la mise en place de mécanismes de tolérance aux pannes sont la perte de connexion entre les différentes cartes réseau et la perte des messages soumis à la carte et non encore traités. Pour résoudre le premier problème, une approche non connectée comme Portals permet de lever cette contrainte de reconnexion au niveau de la couche MPI. Avec ce mode, le processus de destination est défini par un entier (rang du processus dans le MPI.COMM_WORLD). C'est alors dans la couche réseau qu'est réalisée dynamiquement la traduction rang vers identifiant réel de réseau rapide. Cette recherche dans la "table de routage" (de type table de hashage) de la couche d'abstraction a été évaluée à moins d' $1\mu s$ sur nos prototypes d'implémentation réalisés au sein de MPC en contexte natif. Cette méthode de routage s'apparente aux tables de routage mises à jour dynamiquement dans les réseaux TCP/IP et qui ont fait leurs preuves depuis des années pour les serveurs haute disponibilité. Nous avons mené une étude préliminaire sur l'impact d'un mécanisme de routage dynamique couplé à un support du multirail en contexte InfiniBand sans virtualisation. Cette étude a montré qu'il y avait un surcoût d'environ $1\mu s$ par rapport à une approche statique. Cette approche couplée au mécanisme de déconnexion présenté au chapitre 4.3.2 vont nous permettre dans le cas de création de points de reprise coordonnés et sans messages "en vol", de déconnecter tous les supports réseaux rapides pour n'avoir plus qu'un rail TCP/IP (facile à rétablir en cas de chute de nœud). Le mécanisme de connexion à la demande sera en charge de rétablir les connexions sur le réseau rapide. L'impact sur les performances sera visible au redémarrage de l'application sur erreur. Néanmoins, ce dernier ne sera pas plus élevé que lors du démarrage standard qui est lui aussi souvent basé sur un mécanisme de connexion à la demande. Lors de la création du point de reprise, il est envisageable de réaliser une déconnexion factice qui consisterait uniquement à s'assurer que plus aucun message n'est en transit sur le réseau et ne pas réellement réaliser les déconnexions.

L'approche présentée nous permet de gérer la connexion/déconnexion coordonnée au réseau rapide indépendamment de l'implémentation MPI (cette dernière doit néanmoins disposer d'un driver pour notre interface réseau), de l'application et du réseau sous-jacent. Il nous reste à traiter le statut des messages "en vol". La principale difficulté est la perte de ces messages. Pour résoudre ce problème, une approche simple est l'utilisation du mode synchrone de MPI (MPI.Ssend/MPI.Issend par exemple). Avec ce mode, un envoi n'est terminé que lorsque le message a été effectivement reçu. Néanmoins, cette approche impacte fortement les performances. Si on regarde plus en détail les implémentations MPI, on se rend compte que, bien que l'utilisateur ait une vision asynchrone du transfert des messages, les couches basses MPI ont quand à elles besoin de savoir quand le message est reçu pour libérer les tampons mémoire dans le processus émetteur. Il est donc tout à fait possible, sans perte de performances, d'avoir un mode synchrone entre le MPI de l'invité et la couche d'abstraction réseau de l'hôte. Pour éviter la perte de messages, il est donc nécessaire de traiter la partie asynchrone du transfert de messages dans l'invité et ainsi avoir une sauvegarde transparente des messages en cours de traitement lors de la création des points de reprise via sauvegarde de la machine virtuelle. Si un message n'a pas pu être émis réellement suite au redémarrage de l'application, il suffit alors dans les vérifications de libération des tampons de retransférer l'entête du message à la couche réseau hôte. Tous les autres protocoles doivent être gérés dans la couche hôte. Ainsi, ils seront automatiquement redémarrés lors de la reprise de l'application.

Conclusion

L'approche que nous avons présentée nous permet de réaliser de manière transparente pour l'application et sans prérequis sur le support réseau sous-jacent un mécanisme de création de points de reprise coordonnés en contexte MPI avec messages en vol. L'implémentation MPI ne doit pas être tolérante aux pannes mais seulement disposer d'un driver réseau compatible avec notre interface. Cette dernière étant proche des interfaces Elan et Portals déjà présentes dans les implémentations MPI, l'ajout de ce

nouveau réseau ne devrait pas poser de problème majeur. La méthode de sauvegarde des données utilisateur utilisera les mécanismes des sauvegardes sur disque des machines virtuelles. Une extension de ces travaux pourrait être la gestion des points de reprises non coordonnées. Dans ce cas, il faudrait réaliser la sauvegarde de messages dans la couche hôte pour être indépendant du MPI.

7.2.4 Conclusion

Comme nous venons de le voir, il est particulièrement important d’appréhender le problème de l’optimisation des supports exécutifs dans sa globalité et en particulier son interaction avec les architectures des centres de calcul. Par architecture, il ne faut pas entendre uniquement l’architecture matérielle, qui est bien sûr très importante, mais aussi le contexte d’exécution et les contraintes sur les ressources critiques. Les recherches que je compte mener vont présenter une approche d’équilibrage de charge sur le problème du transport de particules Monté-Carlo via les modèles de programmation émergents. Je m’attacherai aussi à tenter d’apporter des solutions aux problèmes fréquemment rencontrés sur les centres de calcul que sont la gestion de la ressource mémoire et la tolérance aux pannes. Toutes ces activités auront pour cible les architectures présentes et les futures architectures de type exaflopiques.

Bibliographie

- [1] Dieter an MEY, Scott BIERSDORFF, Christian BISCHOF, Kai DIETHELM, Dominic ESCHWEILER, Michael GERNDT, Andreas KNÜPFER, Daniel LORENZ, Allen D. MALONY, Wolfgang E. NAGEL, Yury OLEJNIK, Christian RÖSSEL, Pavel SAVIANKOU, Dirk SCHMIDL, Sameer S. SHENDE, Michael WAGNER, Bert WESARG et Felix WOLF : Score-P : A Unified Performance Measurement System for Petascale Applications. *Dans Proc. of the CiHPC : Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, 2012.
- [2] Victor ARSLAN, Patrick CARRIBAULT, Cédric ENAUX, Hervé JOURDREN et Marc PÉRACHE : Calcul hautes performances en transfert radiatif. *Chocs avancées 2009*, pages 38–39, 2010.
- [3] Cédric AUGONNET : *Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime System's Perspective*. Thèse de doctorat, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2011.
- [4] Cédric AUGONNET, Samuel THIBAUT, Raymond NAMYST et Pierre-André WACRENIER : StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23:187–198, février 2011.
- [5] Olivier AUMAGE, Elisabeth BRUNET, Nathalie FURMENTO et Raymond NAMYST : New madeleine : a fast communication scheduling engine for high performance networks. *Dans Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.
- [6] David BAILEY, Tim HARRIS, William SAPHIR, Rob van der WIJNGAART, Alex WOO et Maurice YARROW : *The NAS Parallel Benchmarks 2.0*. 1995.
- [7] Philippe BALLEREAU, Patrick CARRIBAULT, Frédéric DUBOC, David DUREAU, Cédric ENAUX, Hervé JOURDREN et Marc PÉRACHE : Méthodes de raffinement adaptatif de maillage et modèles avancés de programmation pour le calcul haute performance. *Chocs*, (41):81–87, 2012.
- [8] Shajulin BENEDICT, Ventsislav PETKOV et Michael GERNDT : PERISCOPE : An Online-Based Distributed Performance Analysis Tool. *Dans Matthias S. MÜLLER, Michael M. RESCH, Alexander SCHULZ et Wolfgang E. NAGEL, éditeurs : Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010.
- [9] Emery D. BERGER, Kathryn S. MCKINLEY, Robert D. BLUMOFÉ et Paul R. WILSON : Hoard : a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35:117–128, November 2000.
- [10] Emery David BERGER : *Memory management for high-performance applications*. Thèse de doctorat, 2002. AAI3108460.
- [11] J.-B. BESNARD, Marc PÉRACHE et William JALBY : Event streaming for online performance measurements reduction. *Dans Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 985–994, Oct 2013.
- [12] Jean-Baptiste BESNARD : *Profiling and debugging by efficient tracing of hybrid multi-threaded HPC applications*. Thèse de doctorat, Université Versailles Saint-Quentin en Yvelines, 45 Avenue des États Unis — 78000 Versailles, juillet 2014.
- [13] Hans-Juergen BOEHM et Mark WEISER : Garbage collection in an uncooperative environment. *Software : Practice and Experience*, 18(9):807–820, 1988.
- [14] Dan BONACHEA et Jaein JEONG : Gasnet : A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring*, 2002.
- [15] Dan BONACHEA et Jaein JEONG : GASNet : A Portable High-Performance Communication Layer for Global Address-Space Languages. *Dans CS258 Parallel Computer Architecture Project, Spring 2002*, 2002.

- [16] G. BOSILCA, A. BOUTEILLER, F. CAPPELLO, S. DJILALI, G. FEDAK, C. GERMAIN, T. HERAULT, P. LEMARINIER, O. LODYGINSKY, F. MAGNIETTE, V. NERI et A. SELIKHOV : Mpich-v : Toward a scalable fault tolerant mpi for volatile nodes. *Dans Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov 2002.
- [17] Ron BRIGHTWELL, Rolf RIESEN et Keith D. UNDERWOOD : Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *IJHPCA*, 2005.
- [18] Francois BROQUEDIS, Jérôme CLET-ORTEGA, Stéphanie MOREAUD, Nathalie FURMENTO, Brice GOGLIN, Guillaume MERCIER, Samuel THIBAULT et Raymond NAMYST : hwloc : A generic framework for managing hardware affinities in hpc applications. *Dans 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.
- [19] BULL : Bull coherent switch. <http://support.bull.com/ols/product/platforms/hw-extremcomp/hw-bullx-sup-node/hw-extremcomp/hw-bullx-sup-node/BCS/index.htm>.
- [20] BULL : Bullx mpi. <http://bull.com>.
- [21] BULL : Bullx super-node 6010 support. <http://support.bull.com/ols/product/platforms/hw-extremcomp/hw-bullx-sup-node/bullx-s6010>, 2010.
- [22] Jie CAI, Alistair P. RENDELL et Peter E. STRAZDINS : Non-threaded and threaded approaches to multirail communication with uDAPL. *Dans NPC*, pages 233–239. IEEE Computer Society, 2009.
- [23] Franck CAPPELLO : Fault tolerance in petascale/ exascale systems : Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.
- [24] Patrick CARRIBAULT, François DIAKHATÉ, Hervé JOURDREN et Marc PÉRACHE : MPC : une plateforme parallèle unifiée pour le HPC. *Chocs*, (41):22–28, 2012.
- [25] Patrick CARRIBAULT, Marc PÉRACHE et Hervé JOURDREN : Thread-local storage extension to support thread-based MPI/OpenMP applications. *Dans Proceedings of the 7th International Conference on OpenMP in the Petascale Era, IWOMP'11*, pages 80–93, Berlin, Heidelberg, 2011. Springer-Verlag.
- [26] Patrick CARRIBAULT, Marc PÉRACHE, Sylvain DIDELOT, Bettina KRAMMER et Marc TCHIBOUKDJIAN : Exploring Parallel Programming Models and Runtime Environments : The MPC Framework. *Intel European Exascale Labs Annual Report*, pages 34–37, 2011.
- [27] Patrick CARRIBAULT, Marc PÉRACHE et Hervé JOURDREN : Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. *Dans Mitsuhisa SATO, Toshihiro HANAWA, MatthiasS. MÜLLER, BarbaraM. CHAPMAN et BronisR. SUPINSKI, éditeurs : Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More*, volume 6132 de *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2010.
- [28] Patrick CARRIBAULT, Marc PÉRACHE et Hervé JOURDREN : Hiérarchie des données en parallélisme hybride. *Chocs avancées 2011*, pages 42–43, 2012.
- [29] Barbara CHAPMAN, Tony CURTIS, Swaroop POPHALE, Stephen POOLE, Jeff KUEHN, Chuck KOELBEL et Lauren SMITH : Introducing openshmem : Shmem for the pgas community. *Dans Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS '10*, pages 2 :1–2 :3, New York, NY, USA, 2010. ACM.
- [30] Daniel CHAVARRIA-MIRANDA, Sriram KRISHNAMOORTHY et Abhinav VISHNU : Global futures : A multithreaded execution model for global arrays-based applications. *Dans Proceedings CCGRID*, pages 393–401, 2012.
- [31] Jérôme CLET-ORTEGA, Patrick CARRIBAULT et Marc PÉRACHE : Evaluation of openmp task scheduling algorithms for large NUMA architectures. *Dans Euro-Par 2014 – Parallel Processing*, 2014. Rang A.
- [32] Daniel D CORKILL : *Design alternatives for parallel and distributed blackboard systems*. Computer and Information Science, University of Massachusetts, 1988.
- [33] Daniel D. CORKILL : *Design Alternatives for Parallel and Distributed Blackboard Systems*. Rapport technique, Amherst, MA, USA, 1988.
- [34] Daniel D CORKILL : Collaborating software : Blackboard and multi-agent systems & the future. *Dans Proceedings of the International Lisp Conference*, volume 10, 2003.

- [35] K. J. DANHOF, J. QUISENBERRY et M. ZARGHAM : Concurrency in blackboard systems. *Dans Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems - Volume 1*, IEA/AIE '90, pages 109–113, New York, NY, USA, 1990. ACM.
- [36] François DIAKHATÉ : *Contribution à l'élaboration de supports exécutifs exploitant la virtualisation pour le calcul hautes performances*. Thèse de doctorat, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2010.
- [37] François DIAKHATÉ, Marc PÉRACHE, Raymond NAMYST et Hervé JOURDREN : Efficient shared memory message passing for inter-vm communications. *Dans* Eduardo CÉSAR, Michael ALEXANDER, Achim STREIT, Jesper Larsson TRÄFF, Christophe CÉRIN, Andreas KNÜPFER, Dieter KRANZLMÜLLER et Shantenu JHA, éditeurs : *Euro-Par 2008 Workshops - Parallel Processing*, pages 53–62. Springer-Verlag, Berlin, Heidelberg, 2009.
- [38] Dave DICE et Alex GARTHWAITE : Mostly lock-free malloc. *Dans Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 163–174, New York, NY, USA, 2002. ACM.
- [39] Sylvain DIDELOT : *Improving memory consumption and performance scalability of HPC applications with multi-threaded network communications*. Thèse de doctorat, Université Versailles Saint-Quentin en Yvelines, 45 Avenue des États Unis — 78000 Versailles, juin 2014.
- [40] Sylvain DIDELOT, Patrick CARRIBAULT, Marc PÉRACHE et William JALBY : Improving MPI communication overlap with collaborative polling. *Dans* JesperLarsson TRÄFF, Siegfried BENKNER et JackJ. DONGARRA, éditeurs : *Recent Advances in the Message Passing Interface*, volume 7490 de *Lecture Notes in Computer Science*, pages 37–46. Springer Berlin Heidelberg, 2012. Rang C.
- [41] Sylvain DIDELOT, Patrick CARRIBAULT, Marc PÉRACHE et William JALBY : Improving MPI communication overlap with collaborative polling. *Computing*, 96(4):263–278, 2014. Rang A.
- [42] Yaozu DONG, Xiaowei YANG, Xiaoyong LI, Jianhui LI, Kun TIAN et Haibing GUAN : High performance network virtualization with sr-ioV. *Dans High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10, Jan 2010.
- [43] David DUREAU et Gaël POËTTE : Hybrid parallelism models for neutron monte-carlo solver in an AMR framework. *Dans Proceedings of the Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, 2013.
- [44] R. ENGELMORE et T. MORGAN : *Blackboard systems*. Insight series in artificial intelligence. Addison-Wesley, 1988.
- [45] Jason EVANS : A scalable concurrent malloc(3) implementation for freebsd, 2006.
- [46] FRAUNHOFER : site web de gpi. [http://http://www.gpi-site.com/gpi2/](http://www.gpi-site.com/gpi2/).
- [47] Andrew FRIEDLEY, Torsten HOEFLER, Matthew L LEININGER et Andrew LUMSDAINE : Scalable high performance message passing over infiniband for open mpi. *Dans Proceedings of 2007 KiCC Workshop, RWTH Aachen*, 2007.
- [48] Wolfgang FRINGS, Felix WOLF et Ventsislav PETKOV : Scalable massively parallel I/O to task-local files. *Dans Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.
- [49] Markus GEIMER, Felix WOLF, Brian J. N. WYLIE, Erika ÁBRAHÁM, Daniel BECKER et Bernd MOHR : The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 2010.
- [50] Michael GERNDT, K FÜRLINGER et E KEREKU : Periscope : Advanced techniques for performance analysis. *Dans Proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005)*, pages 15–26. Citeseer, 2005.
- [51] Wolfram GLOGER : Ptmalloc : <http://www.malloc.de/en/>.
- [52] Georg HAGER, Gabriele JOST et Rolf RABENSEIFNER : Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. *Dans Proceedings of Cray User Group*, 2009.
- [53] Chao HUANG, Orion LAWLOR et L. V. KALÉ : Adaptive MPI. *Dans LCPC*, 2004.
- [54] Chao HUANG, Gengbin ZHENG, Laxmikant KALÉ et Sameer KUMAR : Performance evaluation of adaptive MPI. *Dans Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

- [55] Wei HUANG, M.J. KOOP, Qi GAO et D.K. PANDA : Virtual machine aware communication libraries for high performance computing. *Dans Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.
- [56] J. HURSEY, J.M. SQUYRES, T.I. MATTOX et A. LUMSDAINE : The design and implementation of checkpoint/restart process fault tolerance for open mpi. *Dans Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.
- [57] J. B. White III et S. W. BOVA : Where's the overlap ? - an analysis of popular MPI implementations. Rapport technique, août 1999.
- [58] INTEL : Intel mpi benchmarks (imb). <http://software.intel.com/en-us/articles/intel-mpi-benchmarks>.
- [59] INTEL : Intel MPI library. <http://software.intel.com/en-us/intel-mpi-library>.
- [60] INTEL : Many integrated core architecture. <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>.
- [61] Julien JAEGER, Patrick CARRIBAULT et Marc PÉRACHE : Data-management directory for OpenMP 4.0 and OpenACC. *Dans Dieter MEY, Michael ALEXANDER, Paolo BIENTINESI, Mario CANNATARO, Carsten CLAUSS, Alexandru COSTAN, Gabor KECSKEMETI, Christine MORIN, Laura RICCI, Julio SAHUQUILLO, Martin SCHULZ, Vittorio SCARANO, Stephen L. SCOTT et Josef WEIDENDORFER, éditeurs : Euro-Par 2013 : Parallel Processing Workshops*, volume 8374 de *Lecture Notes in Computer Science*, pages 168–177. Springer Berlin Heidelberg, 2014.
- [62] Julien JAEGER, Patrick CARRIBAULT et Marc PÉRACHE : Fine-grain data management directory for OpenMP 4.0 and OpenACC. *Concurrency and Computation : Practice and Experience*, 2014. Rang A.
- [63] Chapman & Hall/CRC computational Science Series JEFFREY S. VETTER, éditeur. *Contemporary High Performance Computing : From Petascale toward Exascale*. 2013.
- [64] Hervé JOURDREN : Hera : A hydrodynamic amr platform for multi-physics simulations. *Dans Tomasz PLEWA, Timur LINDE et V. GREGORY WEIRS, éditeurs : Adaptive Mesh Refinement - Theory and Applications*, volume 41 de *Lecture Notes in Computational Science and Engineering*, pages 283–294. Springer Berlin Heidelberg, 2005.
- [65] Simon KAHAN et Petr KONECNY : "mama!" : a memory allocator for multithreaded architectures. *Dans 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.
- [66] Kangho KIM, Cheiyol KIM, Sung-In JUNG, Hyun-Sup SHIN et Jin-Soo KIM : Inter-domain socket communications supporting high performance and full binary compatibility on xen. *Dans Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 11–20, New York, NY, USA, 2008. ACM.
- [67] Andi KLEEN : A NUMA API for LINUX. Rapport technique, avril 2005.
- [68] Andreas KNÜPFER, Ronny BRENDDEL, Holger BRUNST, Hartmut MIX et Wolfgang E NAGEL : Introducing the open trace format (otf). *Dans Computational Science-ICCS 2006*, pages 526–533. Springer, 2006.
- [69] Andreas KNÜPFER, Holger BRUNST, Jens DOLESCHAL, Matthias JURENZ, Matthias LIEBER, Holger MICKLER, Matthias S MÜLLER et Wolfgang E NAGEL : The vampir performance analysis tool-set. *Dans Tools for High Performance Computing*, pages 139–155. Springer, 2008.
- [70] Rahul KUMAR, Amith R. MAMIDALA, Matthew J. KOOP, Gopalakrishnan SANTHANARAMAN et Dhabaleswar K. PANDA : Lock-free asynchronous rendezvous design for MPI point-to-point communication. *Dans Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*, 2008.
- [71] Doug LEA : A memory allocator.
- [72] Jiuxing LIU, Abhinav VISHNU et Dhabaleswar K. PANDA : Building multirail infiniband clusters : MPI-level design and performance evaluation. *Dans SC*, novembre 2004.
- [73] Jiuxing LIU, Jiesheng WU et Dhabaleswar K. PANDA : High performance RDMA-based MPI implementation over infiniband. *International Journal of Parallel Programming (IJPP)*, 32(3):167–198, juin 2004.

- [74] Miao LUO, Jithin JOSE, Sayantan SUR et Dhabaleswar K. PANDA : Multi-threaded UPC runtime with network endpoints : Design alternatives and evaluation on multi-core architectures. *Dans HiPC*, pages 1–10. IEEE, 2011.
- [75] Aurèle MAHÉO, Patrick CARRIBAUT, Marc PÉRACHE et William JALBY : Optimizing collective operations in hybrid applications. *Dans Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 121 :121–121 :122, New York, NY, USA, 2014. ACM. Rang C.
- [76] Aurèle MAHÉO, Souad KOLIAÏ, Patrick CARRIBAUT, Marc PÉRACHE et William JALBY : Adaptive openmp for large numa nodes. *Dans* BarbaraM. CHAPMAN, Federico MASSAIOLI, MatthiasS. MÜLLER et Marco RORRO, éditeurs : *OpenMP in a Heterogeneous World*, volume 7312 de *Lecture Notes in Computer Science*, pages 254–257. Springer Berlin Heidelberg, 2012.
- [77] Zoltan MENYHART et Marc PÉRACHE : Method, computer program and device for managing memory access in a multiprocessor architecture of NUMA type, octobre 3 2013. US Patent App. 13/993,665.
- [78] Stéphanie MOREAUD, Brice GOGLIN *et al.* : Impact of numa effects on high-speed networking with multi-opteron machines. *Dans PDCS*, 2007.
- [79] Stéphanie MOREAUD, Brice GOGLIN et Raymond NAMYST : Adaptive MPI multirail tuning for non-uniform input/output access. 2010.
- [80] Alan MORRIS, Allen D. MALONY, Sameer SHENDE et Kevin HUCK : Design and Implementation of a Hybrid Parallel Performance Measurement System. *Dans Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010.
- [81] W. E. NAGEL, A. ARNOLD, M. WEBER, H.-Ch. HOPPE et K. SOLCHENBACH : VAMPIR : Visualization and Analysis of MPI Resources. *Supercomputer*, 1996.
- [82] Tan NGUYEN, Pietro CICOTTI, Eric BYLASKA, Dan QUINLAN et Scott BADEN : Bamboo – translating MPI applications to a latency-tolerant, data-driven form. *Dans SC'12 CD-ROM : Conference on High Performance Computing Networking, Storage and Analysis*, Salt Lake City, UT, USA, novembre 2012. ACM SIGARCH/IEEE Computer Society.
- [83] R. NISHTALA, P. HARGROVE, D. BONACHEA et K. YELICK : Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. *Dans Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1–12, may 2012.
- [84] Marc PÉRACHE : Nouveaux mécanisme au sein des ordonnanceurs de threads pour une implantation efficace des communications collectives sur machines multiprocesseurs. *Dans Actes des Rencontres Francophones du Parallélisme (RenPar'16)*, pages 231–236, Le Croisic (France), mars 2005.
- [85] Marc PÉRACHE : *Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances*. Thèse de doctorat, spécialité informatique, CEA/DAM Île de France, Université de Bordeaux 1, Domaine Universitaire, 351 Cours de la libération, 33405 Talence Cedex, octobre 2006. 141 pages.
- [86] B. PFAFF, J. PETTIT, T. KOPONEN, K. AMIDON, M. CASADO et S. SHENKER : Extending networking into the virtualization layer. *Dans Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.
- [87] L. PILLA, C. Pousa RIBEIRO, D. CORDEIRO, C. MEI, A. BHATELE, P. NAVAUX, F. BROQUEDIS, J.-F. MEHAUT et L. KALE : A hierarchical approach for load balancing on parallel multi-core systems. *Dans In Proceedings of the 41st International Conference on Parallel Processing, ICPP 2012*, 2012.
- [88] L. PILLA, C. Pousa RIBEIRO, P. COUCHENEY, F. BROQUEDIS, B. GAUJAL, P. NAVAUX et J.-F. MEHAUT : A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. *Future Generation of Computing Systems (FGCS)*, 2014.
- [89] Kévin POUGET, Marc PÉRACHE, Patrick CARRIBAUT et Hervé JOURDREN : User level db : a debugging api for user-level thread libraries. *Dans Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, April 2010.
- [90] Marc PÉRACHE, Patrick CARRIBAUT et Hervé JOURDREN : MPC-MPI : An MPI implementation reducing the overall memory consumption. *Dans* Matti ROPO, Jan WESTERHOLM et Jack DON-GARRA, éditeurs : *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 de *Lecture Notes in Computer Science*, pages 94–103. Springer Berlin Heidelberg, 2009. Rang C.

- [91] Marc PÉRACHE, Hervé JOURDREN et Raymond NAMYST : MPC : A unified parallel runtime for clusters of NUMA machines. *Dans* Emilio LUQUE, Tomàs MARGALEF et Domingo BENÍTEZ, éditeurs : *Euro-Par 2008 – Parallel Processing*, volume 5168 de *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin Heidelberg, 2008. Rang A.
- [92] Rolf RIESEN, Ron BRIGHTWELL, Kevin PEDRETTI, Brian BARRETT, Keith UNDERWOOD, Trammell HUDSON et Arthur B. MACCABE : The portals 4.0 message passing interface, 2008.
- [93] Mark RUSSINOVICH et David A. SOLOMON : *Windows Internals : Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th édition, 2009.
- [94] Paul Menage SANJAY GHEMAWAT : Tcmalloc : Thread-caching malloc, <http://goog-perftools.sourceforge.net/>.
- [95] Sameer S. SHENDE et Allen D. MALONY : The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 2006.
- [96] James M STONE, Thomas A GARDINER, Peter TEUBEN, John F HAWLEY et Jacob B SIMON : Athena : a new code for astrophysical mhd. *The Astrophysical Journal Supplement Series*, 178(1): 137, 2008.
- [97] Sayantan SUR, Hyun-wook JIN, Lei CHAI et Dhabaleswar K PANDA : RDMA Read Based Rendezvous Protocol for MPI over InfiniBand : Design Alternatives and Benefits. *Alternatives*, 2006.
- [98] Zoltán SZEKENYI, Todd GAMBLIN, Martin SCHULZ, Bronis R. de SUPINSKI, Felix WOLF et Brian J. N. WYLIE : Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. *Dans IPDPS, Anchorage, AK, USA*. IEEE Computer Society, 2011.
- [99] Marc TCHIBOUKDJIAN, Patrick CARRIBAULT et Marc PÉRACHE : Hierarchical local storage : Exploiting flexible user-data sharing between MPI tasks. *Dans Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 366–377, May 2012. Rang A.
- [100] Rajeev THAKUR et William GROPP : Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. *Dans PVM/MPI*, pages 46–55, 2007.
- [101] Sébastien VALAT : *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance*. Thèse de doctorat, Université Versailles Saint-Quentin en Yvelines, 45 Avenue des États Unis — 78000 Versailles, juillet 2014.
- [102] Sébastien VALAT, Marc PÉRACHE et William JALBY : Introducing kernel-level page reuse for high performance computing. *Dans Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC '13*, pages 3 :1–3 :9, New York, NY, USA, 2013. ACM.
- [103] Jean-Yves VET, Patrick CARRIBAULT, Albert COHEN *et al.* : Multigrain affinity for heterogeneous work stealing. *Dans Programmability Issues for Heterogeneous Multicores*, 2012.
- [104] Marc WOLFF : Analyse mathématique et numérique du système de la magnétohydrodynamique résistive avec termes de champ magnétique auto-généré.
- [105] Marc WOLFF, Stéphane JAOUEN et Lise-Marie IMBERT-GÉRARD : Conservative numerical methods for a two-temperature resistive MHD model with self-generated magnetic field term. *Dans CEMRACS'10 research achievements : Numerical modeling of fusion*. 2011.
- [106] Marc WOLFF, Stéphane JAOUEN et Lise-Marie IMBERT-GÉRARD : High-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. 2011.
- [107] Marc WOLFF, Stéphane JAOUEN, Hervé JOURDREN et Eric SONNENDRÜCKER : High-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. *Discrete and Continuous Dynamical Systems - Series S*, 2012.
- [108] Xi YANG, Stephen M. BLACKBURN, Daniel FRAMPTON, Jennifer B. SARTOR et Kathryn S. MCKINLEY : Why nothing matters : the impact of zeroing. *Dans 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.
- [109] Weikuan YU, Tim S. WOODALL, Richard L. GRAHAM et Dhabaleswar K. PANDA : Design and implementation of open MPI over quadrics/elan4. *Dans 19th International Parallel and Distributed Processing Symposium (IPDPS)*, avril 2005.
- [110] Xiaolan ZHANG, Suzanne MCINTOSH, Pankaj ROHATGI et JohnLinwood GRIFFIN : Xensocket : A high-throughput interdomain transport for virtual machines. *Dans Renato CERQUEIRA et RoyH. CAMPBELL, éditeurs : Middleware 2007*, volume 4834 de *Lecture Notes in Computer Science*, pages 184–203. Springer Berlin Heidelberg, 2007.

-
- [111] Stéphane ZUCKERMAN, Marc PÉRACHE et William JALBY : Fine tuning matrix multiplications on multicore. *Dans* Ponnuswamy SADAYAPPAN, Manish PARASHAR, Ramamurthy BADRINATH et ViktorK. PRASANNA, éditeurs : *High Performance Computing - HiPC 2008*, volume 5374 de *Lecture Notes in Computer Science*, pages 30–41. Springer Berlin Heidelberg, 2008. Rang A.