Vers un solveur de systèmes linéaires creux adapté aux machines NUMA

Mathieu Faverge

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Vers un solveur de systèmes linéaires creux adapté aux machines NUMA

Mathieu Faverge

INRIA Bordeaux - Sud-Ouest,
351, cours de la libération,
33405 Talence Cedex - France
faverge@labri.fr

Résumé
Les solveurs de systèmes linéaires ont fait d'énormes progrès au cours des dernières années et commencent désormais à exploiter les architectures multi-cœurs et leur mémoire partagée. Le solveur PASTIX est développé dans une version hybride MPI/thread pour gagner en consommation mémoire sur les buffers de communications. On souhaite désormais l'adapter aux architectures NUMA et disposer d'un ordonnancement dynamique pour ces architectures car les modèles de coût ne peuvent intégrer complètement les caractéristiques de ces architectures. On étudiera dans une première partie l'importance d'une nouvelle gestion mémoire pour les architectures NUMA, puis comment nous avons implémenté cet ordonnancement dynamique. Des résultats illustrerons nos travaux et nous finirons par l'étude d'un cas challenge correspondant à un problème à 10 millions d'inconnues.

Mots-clés : Systèmes linéaires creux, multi-cœurs, ordonnancement dynamique, architectures NUMA

---

1. Introduction

Au cours des dernières années, les solveurs directs de systèmes linéaires creux ont fait d'énormes progrès [1, 5, 7, 10]. Il est désormais possible de résoudre efficacement et dans un laps de temps raisonnable des problèmes 3D de plusieurs millions d'inconnues. La multiplication des super-calculateurs basés sur des architectures multi-cœurs SMP (Symmetric Multi-Processor) a conduit les solveurs à proposer des implémentations mieux adaptées à ces nouvelles architectures. Les solveurs PASTIX ou WSMP proposent des implémentations hybrides MPI+threads [8]. Cette technique permet de résoudre de grands problèmes 3D où le surcoût mémoire dû aux buffers de communications est un frein à l'utilisation de méthodes directes. Aujourd'hui, l'augmentation du nombre de cœurs dans les nœuds de calculs a fait apparaître des effets dissymétriques sur les coûts d'accès mémoires, ce sont les effets NUMA (Non Uniform Memory Access). Nous présentons ici une étude des effets NUMA sur différentes architectures et les modifications que nous avons apportées au solveur PASTIX pour prendre en compte ces effets. Nous présenterons ensuite un ordonnancement dynamique de ce solveur adapté à ces architectures et les résultats obtenus sur différents problèmes. Pour terminer, nous étudierons un problème de taille importante pour valider nos solutions.

2. Effets NUMA et allocation mémoire

Les architectures multiprocesseurs modernes sont principalement basées sur des systèmes à mémoire partagée avec un comportement NUMA. Ces ordinateurs sont composés de plusieurs processeurs, eux-mêmes composés de plusieurs cœurs. Chacun d'eux est associé à une unité mémoire et sont interconnectés par un système de cohérence de cache leur donnant accès à l'intégralité de la mémoire. Ce type d'architecture implique une structure très hiérarchisée dont les coûts d'accès à la mémoire varient grandement d'une unité mémoire à une autre [9]. De plus, la bande passante est également variable à cause de l'entrelacement des accès qui peuvent se perturber entre eux. Ces architectures imposent de prendre en compte le placement des données lors de leur allocation et de leur utilisation pour réduire

au maximum la distance unité de calcul / mémoire ainsi que les échanges de données entre les chipsets. Pour cela, les systèmes d'exploitation courants fournissent différentes API plus ou moins dédiées aux architectures NUMA qui permettent au programmeur de contrôler le placement de ses données et processus de calculs. Dans un premier temps, nous avons mis en avant les effets NUMA grâce à ces API sur différentes architectures avec différentes combinaisons de placement sur différentes fonctions BLAS. Pour cela nous avons fixé les threads de calculs sur les différentes unités de calcul et nous avons étudié le comportement par défaut de la plupart des systèmes qui consistent à allouer les données au plus près de l'unité de calcul qui est la première à y accéder.

		Placement du thread de calcul							
		0	1	2	3	4	5	6	7
Données	0	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.31	*1.57*
	1	1.33	**1.00**	*1.57*	1.31	1.33	**1.00**	*1.57*	1.31
	2	1.28	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	3	*1.56*	1.32	1.34	**1.00**	*1.56*	1.31	1.33	**0.99**
	4	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.30	*1.58*
	5	1.33	**1.00**	*1.58*	1.30	1.33	**1.00**	*1.57*	1.30
	6	1.29	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	7	*1.57*	1.32	1.33	**1.00**	*1.57*	1.32	1.35	**1.00**

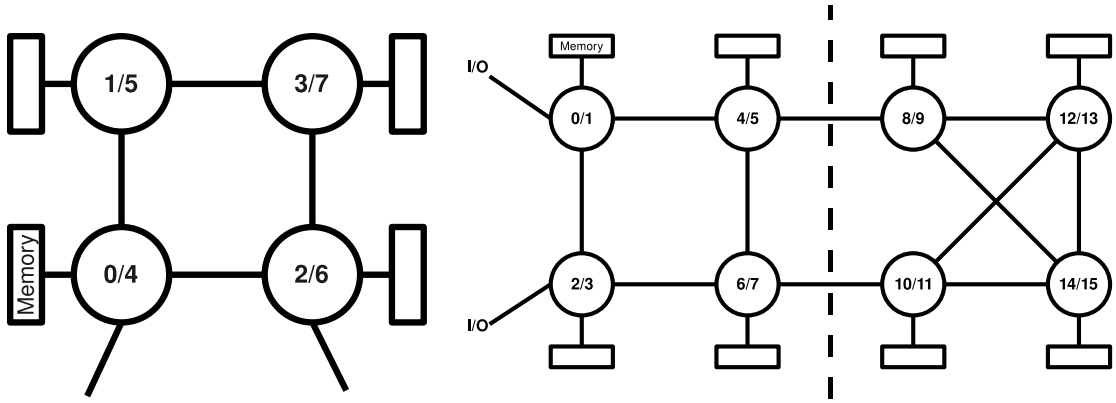
(a) Fonction dAXPY

		Placement du thread de calcul							
		0	1	2	3	4	5	6	7
Données	0	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	1	1.04	**1.00**	*1.07*	1.04	1.04	**1.00**	*1.08*	1.04
	2	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04
	3	*1.08*	1.04	1.04	**1.00**	*1.07*	1.04	1.05	**1.00**
	4	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	5	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04
	6	1.04	*1.07*	**1.00**	1.05	1.04	*1.07*	**1.00**	1.04
	7	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**

(b) Fonction dGEMM

FIG. 1 – Influence du placement des données sur NUMA8 en temps normalisé par rapport au meilleur placement

Les tableaux 1(a) et 1(b) montrent le facteur NUMA d'un nœud du cluster qu'on appellera par la suite NUMA8[1] avec des matrices double précision de taille 128×128. Les résultats confirment la présence d'une zone mémoire par chipset de deux cœurs et la structure en carré de l'architecture qui impose un ou deux liens HyperTransport pour l'accès aux données. Cette architecture est présentée sur la figure 2(a) ainsi que la numérotation de cœurs que l'on retrouve dans les résultats. On observe également que plus le nombre d'opérations par donnée est faible plus les facteurs NUMA augmentent ($1,58$ sur les cœurs les plus éloignés). Inversement, les tests avec des routines BLAS matrices$\times$matrices ont des facteurs plus faibles étant donné que le temps de calcul devient prépondérant sur le temps d'accès.



(a) NUMA8 : Quatre opterons bi-cœurs

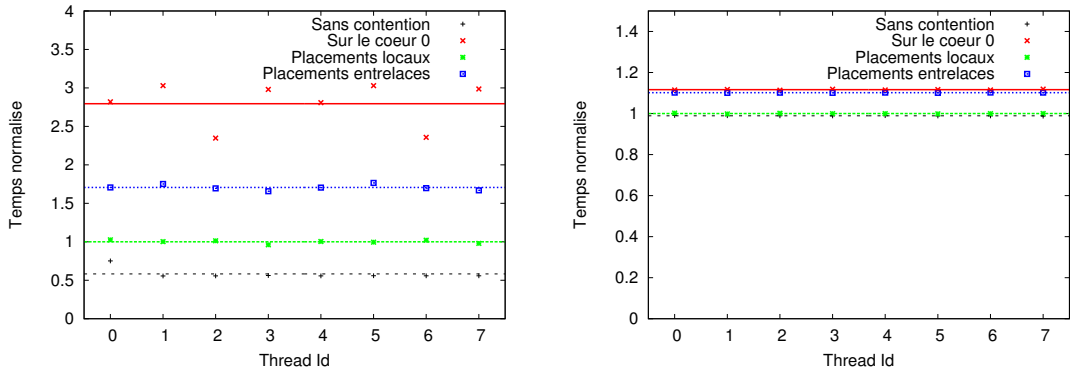
(b) NUMA16 : Huit opterons bi-cœurs

FIG. 2 – Architectures NUMA utilisées

Au vu de ces tableaux, il est nécessaire de prendre en compte les effets NUMA lors de l'allocation mé-

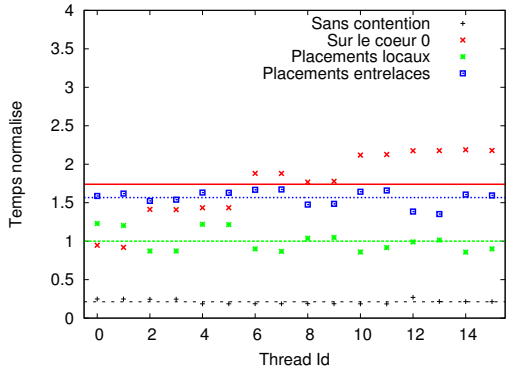
[1] NUMA8 est un cluster de 10 nœuds de 4 opterons bi-cœurs avec 32G de mémoire par nœud.

moire des applications multi-threadées sur les architectures hiérarchiques. Cet exemple nous a montré l'influence du placement mémoire sur des tests n'exploitant pas complètement la machine et n'est pas représentatif des applications réelles. Nous avons souhaité évaluer ce facteur NUMA avec les problèmes de contention qui peuvent apparaître sur une machine chargée. L'expérience est basée sur un ensemble de calcul BLAS sur des matrices/vecteurs. On compare trois différentes manières de placer les données : *Sur le cœur 0* (les données sont allouées sur la zone mémoire du premier cœur), *Placements locaux* (chaque thread de calcul alloue les données au plus proche) et *Placements entrelacés* (les données sont placées au pire endroit possible selon les facteurs NUMA obtenus dans la première expérience). La dernière courbe *(Sans contention)* représente le temps de calcul de chaque cœur dans la première expérience pour le même lot de données, i.e. sans contention. Les résultats mettent en avant le facteur NUMA important de cette architecture et surtout l'augmentation de celui-ci avec les phénomènes de contention. Le surcoût peut atteindre un facteur $2,7$ dans le pire des cas 3(a) et si les données sont mal placées, on obtient un facteur moyen de $1,7$. Les tests passés sur l'architecture NUMA16[2] montrent que les effets diffèrent d'une architecture à l'autre, mais augmentent légèrement sur les opérations matrices$\times$matrices avec le nombre de cœurs.

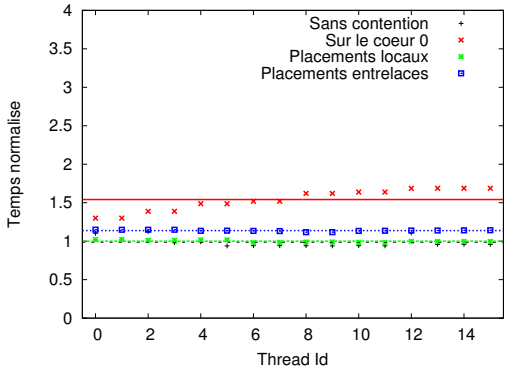


(a) dAXPY sur l'architecture NUMA8

(b) dGEMM sur l'architecture NUMA8



(c) dAXPY sur l'architecture NUMA16



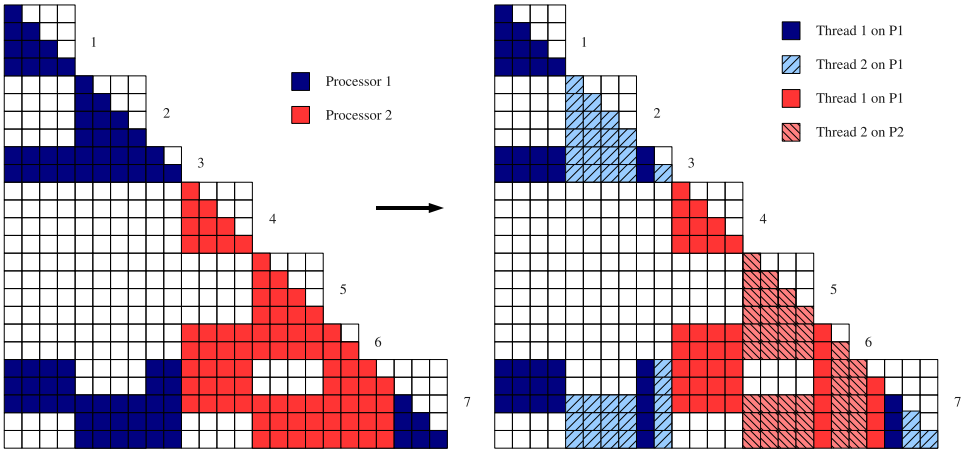
(d) dGEMM sur l'architecture NUMA16

FIG. 3 – Influence de la contention sur deux architectures NUMA

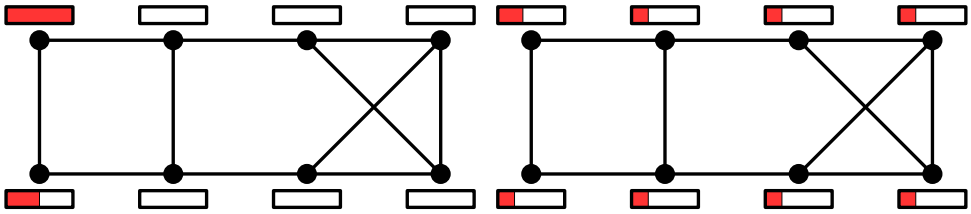
Le problème souvent rencontré dans les applications multi-threadées est que l'on sépare la phase d'initialisation des données de celle des calculs, ce qui donne une allocation massive en séquentiel au plus proche du premier cœur et un ensemble de threads qui viennent par la suite prendre les données dont ils ont besoin. La solution la plus simple est en général de retarder cette étape pour la faire exécuter par les threads qui vont exécuter les calculs. Ainsi, le système d'exploitation qui alloue généralement les

[2] Il s'agit d'un nœud de 8 opterons bi-cœurs avec 64Go de mémoire, dont la structure est représentée sur la figure 2(b).

données au plus proche du cœur qui les modifie pour la première fois, répartit les données là où elles seront utilisées.



(a) Répartition des allocations de la matrice pour mieux prendre en compte les effets NUMA



(b) Allocation initiale (c) Nouvelle allocation << NUMA-aware >>

FIG. 4 – Allocation pour architecture NUMA

Nous avons implémenté cette solution dans notre version hybride MPI/thread du solveur PASTIX. Dans sa version initiale, une étape séquentielle réalise l'initialisation avec l'allocation d'un unique tableau pour la matrice par nœud de calcul comme représenté sur le schéma gauche de la figure 4(a). On se retrouve dans le pire cas de l'expérience précédente où les données se retrouvent concentrées à proximité d'un seul chipset (Fig. 4(b)). Nous avons donc changé nos structures de données afin de répartir l'allocation des blocs colonnes de la matrice et leur initialisation sur les threads de calculs qui vont devoir les factoriser. Ainsi, la mémoire utilisée est mieux répartie sur le nœud (Fig. 4(c)) et les données sont plus proches des threads qui vont les utiliser comme le préconisait la dernière expérience. Le tableau 1 dans le paragraphe suivant met en évidence l'influence de la prise en compte des effets NUMA dans l'allocation des données sur le temps de factorisation de différents problèmes sur différentes architectures. Les gains ainsi obtenus vont de 5% à 35% entre la version initiale (colonne V0) et la nouvelle version (colonne V1).

Les résultats sur ce solveur hautes performances confirment les premières expériences et montrent qu'il est important de bien prendre en compte les facteurs NUMA et la localité des données. Ceci améliore grandement les temps d'exécution des algorithmes qui ont potentiellement de gros besoins en mémoires. De plus, on constate que ce phénomène devient de plus en plus important avec l'augmentation de la taille des machines qui impliquent de nouveaux niveaux hiérarchiques.

3. Ordonnancement dynamique pour architecture NUMA

On présente dans cette partie nos travaux sur la conception d'un ordonnancement dynamique pour les applications avec un graphe de dépendance sous forme d'un arbre. Ces applications comprennent entre autres les solveurs directs de systèmes linéaires creux qui sont basés sur un arbre d'élimination.

Le principal problème est de trouver un moyen de préserver la proximité des données lors du vol de travail. L'avantage d'un tel arbre d'élimination est que les contributions se font principalement le long du chemin vers la racine et peu ou pas vers le reste de l'arbre. Les sous-arbres sont ainsi plus facilement répartis sur la machines contrairement aux solutions basées sur des graphes directs acycliques présentées dans [4]. Dans un premier temps, on assigne donc à chaque thread une partie contiguë de l'arbre puis, pour conserver la proximité entre thread et mémoire, on cherche à faire un vol de travail qui va conserver le plus possible cette affinité entre un thread et les données qui lui sont proches.

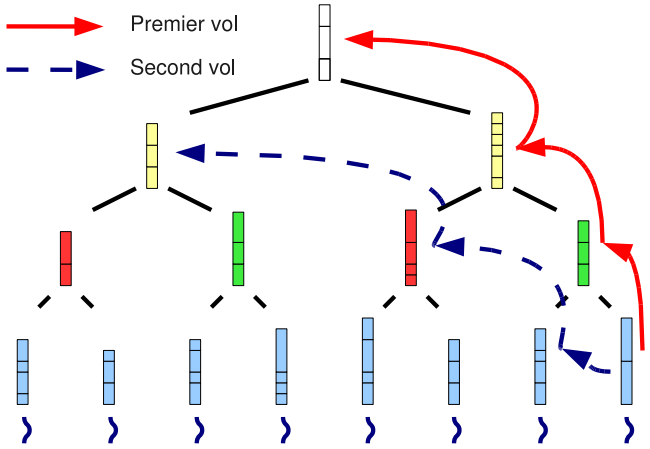


FIG. 5 – Algorithme de vol de travail

On se concentre désormais uniquement sur le solveur PASTIX. Dans sa version initiale, PASTIX ordonnance statiquement les tâches de calculs grâce à une simulation basée sur des modèles de coûts des communications et des fonctions BLAS.
La solution proposée est composée de deux étapes de prétraitement. La première distribue efficacement les données sur les nœuds de calculs du cluster grâce aux modèles de coûts utilisés dans l'ordonnancement statique. La seconde étape construit un arbre de files de tâches à l'aide d'un algorithme de répartition proportionnelle sur l'arbre d'élimination. On forme ainsi des listes de tâches associées à différents ensembles de threads (cœurs) sur la machine qui seront les candidats privilégiés pour l'exécution de ces tâches car ils seront plus proches des données. Les listes des feuilles de ce nouvel arbre ne possèdent qu'un seul candidat, et celles des noeuds de l'arbre ont pour candidats l'union des ensembles associés à leurs fils. La plus grande partie des blocs colonnes à factoriser se retrouvent dans les feuilles de l'arbre, ce qui préserve l'affinité mémoire pour tous ces calculs puisque chaque thread alloue ses données à proximité. Pour les noeuds de l'arbre, nous utilisons une répartition cyclique de l'allocation des blocs colonnes sur l'ensemble des candidats. L'allocation n'est donc pas optimale puisqu'on ne garantit pas la localité des données mais, permet une répartition équilibrée sur une zone de la machine restreinte si la numérotation des threads de calcul suit la structure de la machine. Si un processus n'a plus de travail dans sa propre liste lors de la factorisation, il va tenter d'en trouver sur le chemin critique de l'application comme le montrent les flèches continues sur la figure 5. S'il ne trouve rien, on étend alors les recherches aux tâches plus éloignées sur la machine dans les branches voisines au chemin critique comme le montrent les flèches pointillées.
Les deux diagrammes de Gantt (Fig. 6) montrent la réduction des temps d'inactivités (blocs noirs) grâce à l'ordonnancement dynamique sur la factorisation de la matrice `10Millions` avec 8 processus de 32 threads par rapport à l'ordonnancement statique. Chaque couleur représente un niveau dans l'arbre d'élimination. Les blocs noirs représentent les temps d'inactivité et les flèches représentent les communications. On retrouve la réduction de ces temps d'inactivité sur les temps de factorisation présentés dans le tableau 1 qui montre le résultat des améliorations apportées au solveur PASTIX pour les architectures multi-cœurs. La colonne V0 montre les temps de référence de la version d'origine. Les colonnes V1 et

Matrix	N	NNZ_A	NNZ_L	NUMA8			NUMA16			SMP16		
				V0	V1	V2	V0	V1	V2	V0	V1	V2
MATR5	485 597	24 233 141	1 361 345 320	437	410	**389**	527	341	**321**	162	161	**150**
AUDI	943 695	39 297 771	1 144 414 764	256	217	**210**	243	185	**176**	101	100	**100**
NICE20	715 923	28 066 527	1 050 576 453	227	**204**	227	204	168	**162**	91.40	91	**90.30**
INLINE	503 712	18 660 027	158 830 261	9.70	**7.31**	7.32	20.90	15.80	**14.20**	5.80	**5.63**	5.87
NICE25	140 662	2 914 634	51 133 109	3.28	**2.62**	2.82	6.28	**4.99**	5.25	2.07	1.97	**1.90**
MCHLNF	49 800	4 136 484	45 708 190	3.13	**2.41**	2.42	5.31	3.27	**2.90**	1.96	1.88	**1.75**
THREAD	29 736	2 249 892	25 370 568	2.48	2.16	**2.05**	4.38	2.17	**2.03**	1.18	1.15	**1.06**
HALTERE	1 288 825	10 476 775	405 822 545	134	136	**129**	103	**93**	94.80	48.40	47.90	**47.40**

TAB. 1 – Comparaison du temps de factorisation en secondes sur trois versions du solveur PASTIX. V0 est la version initiale avec l'ordonnancement statique et allocation globale. V1 est la version avec allocation locale (NUMA) et ordonnancement statique. V2 est la version avec allocation locale et ordonnancement dynamique.

V2 montrent les améliorations apportées : V1 l'allocation pour machine NUMA et V2 possède en plus l'ordonnancement dynamique. Tous les temps de ce tableau sont obtenus avec un nombre de threads égal au nombre de cœurs disponibles sur l'architecture : 8 threads sur l'architecture NUMA8 et 16 sur les architectures NUMA16 et SMP16[3]. On constate que la plupart des résultats sont améliorés par l'ordonnancement dynamique quelque soit l'architecture, ou en tout cas ne dégrade pas les performances du solveur.

4. Etude sur un problème à 10 millions d'inconnues

Nous allons nous concentrer maintenant sur l'étude de la résolution d'un problème à 10 millions d'inconnues dont les caractéristiques sont détaillées dans le tableau 2. Cette matrice correspond à un problème d'électromagnétisme (en arithmétique complexe) qui nous a été soumis par le CEA CESTA et constitue un chalenge difficile pour les solveurs basés sur une méthode directe. Les résultats ont été obtenus sur le cluster IBM Power6 de l'IDRIS qui est composé de nœuds à 32 cœurs et 128Go de mémoire reliés par un réseau fédération IBM. Nous souhaitons ici seulement valider notre ordonnancement dynamique sur cette machine sans tenir compte des effets NUMA car elle est la seule machine à laquelle nous ayons accès à proposer 32 cœurs par nœuds avec suffisamment de mémoire pour résoudre un problème de cette taille.

Caractéristiques	
N	10 423 737
NNZ_A	89 072 871
NNZ_L	6 724 303 039
OPC	4.41834e+13

TAB. 2 – Caractéristiques de la matrice.

	4x32	8x32
Version Statique	289	185
Version Dynamique	240	184

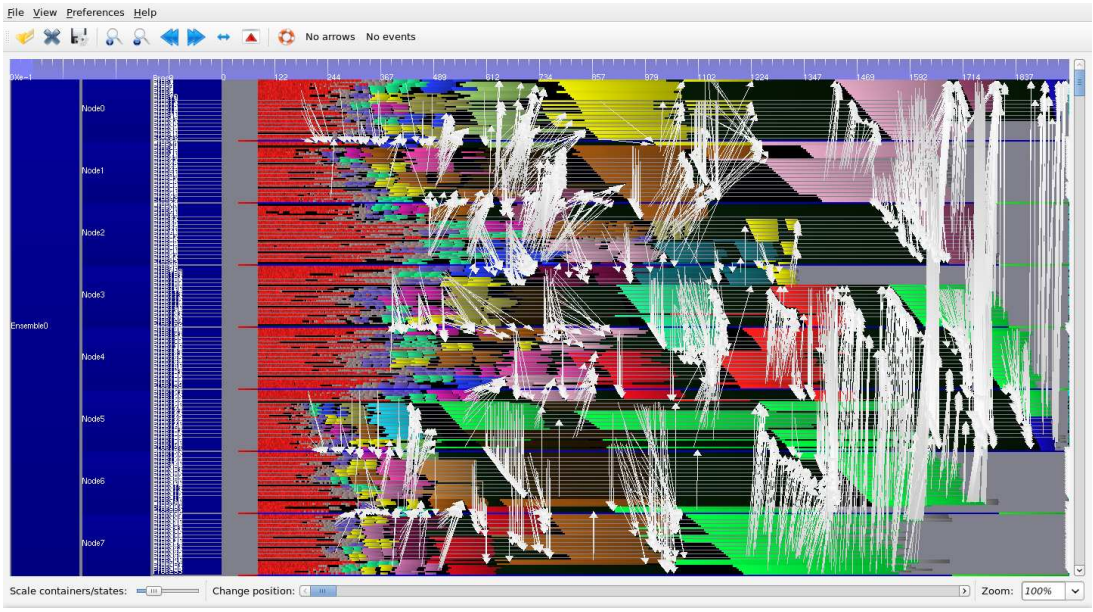
TAB. 3 – Temps de factorisation de la matrice 10Millions sur 4 et 8 processus de 32 threads sur la machine Vargas de l'IDRIS

Le tableau 3 montre un gain important sur 4 nœuds, mais en revanche les résultats sur 8 nœuds montrent peu d'améliorations. En observant le détail de l'exécution sur 8 nœuds grâce aux traces obtenues avec le logiciel VITE[4] comme celles présentées sur les figures 6(a) et 6(b), on constate que le choix d'une distribution par blocs colonnes (1D) montre ses limites.

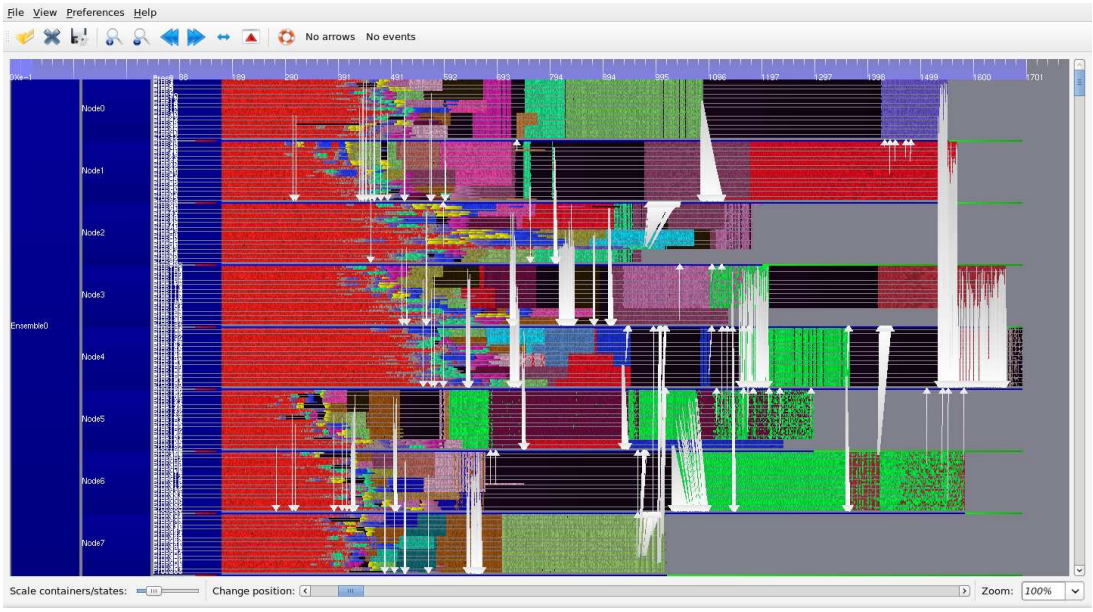
Le solveur PASTIX autorise également l'utilisation d'une distribution par blocs (2D) pour les supernœuds de l'arbre d'élimination. Cela donne un grain plus fin qui est mieux réparti sur les différents nœuds de calculs [6]. Plusieurs articles [3, 2] ont montré qu'il est nécessaire de passer à une distribution des données 2D pour avoir une bonne scalabilité sur les solveurs de systèmes linéaires. Or ici, les

[3] SMP16 est un cluster de 10 nœuds de 16 power5 avec 28Go de mémoire par nœud interconnectés par un réseau IBM fédération.

[4] VITE est un logiciel de visualition de traces d'exécution écrites au format Pajé réalisé par un groupe de 7 étudiants de l'ENSEIRB (`http://vite.gforge.inria.fr` et `http://www-id.imag.fr/Logiciels/paje/index.html`).



(a) Ordonnancement statique



(b) Ordonnancement dynamique

FIG. 6 – Diagramme de Gantt pour la matrice *10Millions* sur la machine vargas de l'IDRIS avec 8 processus MPI de 32 threads. Les blocs noirs représentent les temps d'inactivités et les flèches blanches les communications.

derniers nœuds de l'arbre d'élimination sont denses et de grande taille, ce qui nécessite encore plus une distribution par blocs des données. Avec l'aide de la simulation réalisée pour l'ordonnancement statique, nous avons constaté une amélioration de l'ordonnancement lorsque l'on distribue au moins les cinq derniers niveaux de l'arbre d'élimination par blocs. Cependant la construction de l'ordonnancement statique (i.e. le calcul de la distribution pour l'ordonnancement dynamique) pose problème. Nous passons de 40 secondes de pré-traitement à 850 secondes en raison de la complexité quadratique de l'algorithme qui calcule la distribution. Nous avons implémenté une nouvelle solution à l'aide de l'or-

donnancement dynamique qui nous permet de faire des calculs avec une distribution 2D au sein d'un même nœud. Dans un premier temps, la distribution des données se fait toujours selon un découpage par blocs colonnes avec une simulation basée sur les modèles de coûts qui distribue les données parmi les nœuds disponibles. Puis au sein d'un nœud en mémoire partagée, lors de la factorisation d'un bloc colonne, si la taille du bloc colonne est suffisamment importante et que plusieurs threads sont candidats à sa factorisation, on crée dynamiquement de multiples sous tâches associées à un découpage par blocs. Ce phénomène est mis en évidence sur la figure 6(b) où le temps de calculs de chaque tâches 2D est beaucoup plus faible que sur la première version, et où la répartition des calculs au sein d'un nœud est beaucoup plus uniforme.
Cette solution nous a permis de gagner un peu plus de 15% sur le temps de factorisation sur 8 nœuds. Le temps de factorisation obtenu est finalement de 153 secondes. Cependant, la distribution initiale actuelle reste toujours problématique et empêche une diminution plus importante du temps de factorisation par cette méthode. Une possibilité serait de combiner une répartition initiale en 2D sur un nombre de niveaux de l'arbre d'élimination moins important pour ne pas trop augmenter le temps de la phase d'analyse avec une gestion 2D dynamique sur le reste de l'arbre.

5. Conclusion

Les modifications apportées au solveur PASTIX pour l'allocation des données donnent de très bon résultats et c'est un principe facilement applicable à la plupart des applications. Cela met aussi en évidence l'importance de bien prendre en compte les facteurs NUMA des nouvelles architecures dans son application pour obtenir facilement des gains sur le temps de calcul. Les résultats sur l'ordonnancement dynamique sont également très concluants, puisque les temps de factorisation suivant les cas restent équivalent à la prédiction statique et/ou améliorent les temps de la version d'origine. Il n'y a pas non plus de surcoût à l'utilisation sur les architectures SMP. On ne détériore pas non plus les résultats sur machines NUMA malgré le fait que la localité mémoire ne soit pas optimale sur les nœuds du haut de l'arbre d'élimination. Enfin, l'étude d'un cas challenge a permis de valider nos solutions actuelles et d'identifier certaines limites du solveur PASTIX. La possibilité de faire du calcul à grain plus fin au sein des nœuds de calcul apporte déjà un gain important sur le temps de factorisation de grands problèmes. Mais les traces d'exécution obtenues sur ce cas test montrent que la distribution initiale des données peut-être améliorée. Nos futurs travaux vont s'orienter sur la possibilité de combiner une distribution statique 2D-blocs avec un complément dynamique à grain plus fin au sein de chaque nœud.

Bibliographie

1. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIMAX*, 23(1) :15–41, 2001.
2. J. J. Dongarra, R. A. Vandegeijn, and D. W. Walker. Scalability issues affecting the design of a dense linear algebra library. *Journal of Parallel and Distributed Computing*, 22(3) :523 – 537, 1994.
3. I. S. Duff. Sparse numerical linear algebra : direct methods and preconditioning. Technical Report TR/PA/96/22, CERFACS, 1996.
4. F.Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. Technical Report UT-CS-09-638, Univ. of Tennessee Computer Science, April 2009.
5. A. Gupta. Recent progress in general sparse direct solvers. In *LNCS*, volume 2073, pages 823–840, 2001.
6. P. Hénon, P. Ramet, and J. Roman. PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Irregular'2000*, volume 1800 of *LNCS*, pages 519–525, Cancun, Mexique, May 2000.
7. P. Hénon, P. Ramet, and J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, January 2002.
8. P. Hénon, P. Ramet, and J. Roman. On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. In *PPAM'05*, volume 3911 of *LNCS*, pages 1050–1057, Poznan, Pologne, September 2005.
9. P. P. Janes J. Antony and A. P. Rendell. Exploring thread and memory placement on NUMA architectures : Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *HiPC*, pages 338–352, 2006.
10. X. S. Li and J. W. Demmel. SuperLU_DIST : A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2) :110–140, June 2003.