Page de garde

Remerciements

Table des matières

Annexes

INTRODUCTION

PARTIE I : Contexte scientifique et revue de l'état de l'art

Chapitre 1 : Cadre de travail et contexte scientifique

1.1 Classe des algorithmes itératifs parallèles à large échelle dans une grille de calcul

Dans le cadre de ces travaux, nous nous sommes intéressés particulièrement sur la performance d'une classe d'algorithmes parallèles dits itératifs. De plus en plus, cette méthode itérative est utilisée pour résoudre des problèmes dans différents domaines scientifiques tels que la mécanique, la prévision du temps, le traitement d'images ou encore l'économie financière. Elle consiste à appliquer, contrairement à la méthode de résolution « directe », à partir d'une valeur initiale X_0 une transformation à un vecteur inconnu de rang n par des itérations successives afin de s'approcher par approximation à la solution recherchée X* avec une valeur résiduelle la plus réduite possible.

$$X^{k+1} = \text{f}\,(\,X^k\,),\ \text{k} = 0,1,\ldots \tag{1}$$

où chaque x_k est un vecteur à n dimension et f une fonction de R^n vers R^n.

La solution du problème sera donc le vecteur X* tel que X* = f (X*), c'est-à-dire X* est un point fixe de f.

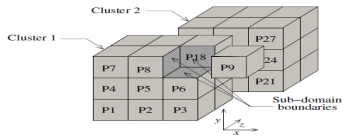
L'exécution en parallèle d'un tel algorithme consiste au découpage (partitionnement) du problème en plus petits morceaux (ou blocs) et d'assigner chaque bloc à une unité de calcul. Chaque processeur tourne le même algorithme de façon concourante jusqu'à la détection de la convergence locale qui peut être obtenue soit par l'atteinte d'un nombre maximum fixé d'itérations soit que la différence entre les valeurs du vecteur inconnu entre deux itérations successives est devenue inférieure à la valeur résiduelle convenue. Cette condition de convergence locale peut être écrite comme suit :

$$(k \leq MaxIter) \text{ or } (\|X_l^k - X_l^{k+1}\|_\infty \leq \epsilon)$$

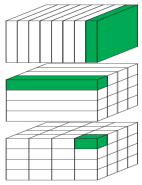
La convergence globale sera déclarée lorsque tous les processeurs ont atteint leur convergence locale. De façon générale, plusieurs travaux ont démontré la convergence de ces méthodes itératives pour la résolution de systèmes linéaires ou non linéaires avec un taux de convergence élevé [7, 8]. Lors de l'exécution dans chaque bloc de calcul, l'algorithme peut demander l'échange de données comme des résultats intermédiaires par exemple entre des processeurs voisins avant d'entamer une nouvelle itération. Les sections suivantes vont détailler les notions liées à la résolution de cet algorithme.

0.0.1 Partitionnement du problème

Comme expliqué plus haut et appliquant le principe du "diviser pour regner", le problème de résolution d'un algorithme itératif parallèle commence par un découpage de la matrice $n \times n$ en entrée en plus petits blocs dont le nombre dépend du nombre de processeurs disponibles. On parle de « décomposition de domaine » en considérant les données en priorité en opposition à la « décomposition fonctionnelle » où le partitionnement se base sur le calcul : diviser le calcul en des tâches indépendantes assignées aux processeurs. La figure Figure 1a présente un exemple de découpage en domaines de la matrice initiale entre deux clusters constitués chacun de 18 processeurs, soit un total de 36 processeurs.



(a) Découpage d'une matrice tridimensionnelle entre deux clusters formés de 18 processeurs chacun



(b) Décomposition en domaines 1D, 2D et 3D

FIGURE 1 – Partitionnement du problème

Chaque cluster va prendre en charge un bloc de 18 "sous-domaines". Chaque processeur P_i tournera l'algorithme sur le cube qui lui est assigné. Les sous domaines s'échangent des données par leurs points périphériques [9] au niveau du cluster mais aussi entre les clusters en suivant une organisation logique d'un anneau virtuel dont les nœuds sont les processeurs P_i.

Une fois partitionnée en m blocs, la relation reccurente de l'équation (1) peut s'écrire :

$$x_{k+1} = (x_1^k, x_2^k, \ldots, x_n^k), k = 1, \ldots n \tag{2}$$

ou en termes de blocs :

$$X_{k+1} = (X_1^k, X_2^k, \ldots, X_n^k), k = 1, \ldots m \tag{3}$$

Donc, on peut écrire :

$$X_{k+1} = F(X_k)$$

$$(X_1^{k+1}, X_2^{k+1}, \ldots, X_m^{k+1}) = (F_1(X_k), F_2(X_k), \ldots, F_m(X_k)) \tag{4}$$

Où :

$$X_i^{k+1} = F_i(X^k) = Fi(X_1^k, X_2^k, \ldots, X_m^k)\ pour\ i = 1, \ldots, k$$

L'exemple donné montre un partitionnement « naturel » du problème initial par un découpage uniforme avec des blocs de même taille. Il met en exergue deux facteurs importants à tenir en compte lors de cette opération :

- essayer de répartir uniformément la charge assignée à chaque processeur : effectivement, un déséquilibre de charge entre les unités de calcul peut impacter négativement la performance globale du système ;
- réduire au maximum les communications entre les processeurs : ces temps d'échange coûtent aussi chers au niveau de la performance globale.

Selon le type de l'algorithme, on peut faire un classement en trois catégories [21] selon le partitionnement ou la décomposition de domaine choisie (Figure 1b) :

- 1D où la matrice est découpée suivant des briques dont deux dimensions de longueur n et la dernière plus courte que n.
- 2D avec des briques dont une dimension est de longueur n et les deux autres plus courtes que n ;
- et enfin, 3D avec des briques dont les 3 dimensions sont plus courtes que n.

0.0.2 Modes d'exécution synchrone et asynchrone

Lors de l'exécution des algorithmes itératifs parallèles sur un environnement de type grille de calcul, le temps de communication résultant des échanges de données entre les unités de calcul est aussi important que le temps de calcul lui-même. En effet, un ratio montrant un équilibre entre ces deux temps constitue un des objectifs dès le partitionnement du problème. Le temps de communication est impacté sur la façon dont les échanges sont effectués.

(a) Modèle de communication synchrone (b) Modèle de communication asynchrone

FIGURE 2 – Modèles de communication

D'une part, ces paquets de données peuvent être transférés de façon « synchrone » : dans ce cas, une coordination de l'échange est assurée par les deux parties. A la fin de chaque itération, l'émetteur, une fois la poignée de main établie, envoie les données et attend jusqu'à la réception d'un accusé de réception par le récepteur. L'algorithme même est en mode synchrone parce qu'une étape de synchronisation de tous les processeurs est nécessaire avant d'entamer une nouvelle itération. La figure Figure 2a montre les actions dans le temps lors d'un échange en mode synchrone entre deux processeurs. Les flèches montrent la date d'envoi par P_1 et la date de réception du paquet par P_2. On parle ici de mode de communication « bloquante » : la nouvelle itération ne peut commencer tant que tous les processus n'ont pas fini leurs communications.

D'autre part, l'échange de données peut s'effectuer en mode « asynchrone ». Dans ce cas, l'émetteur peut envoyer de l'information au destinataire à tout moment et aucune synchronisation n'est nécessaire. Chaque processeur travaille avec les données qu'il reçoit au fil du temps. La communication est ici non bloquante. La conséquence immédiate de ce mode de communication est l'absence des périodes où le traitement est arrêté (CPU stalled ou idle) parce qu'il doit

attendre l'accusé de réception du récepteur (Figure 2b). En mode asynchrone, le temps entre chaque itération peut varier notablement dû à la différence éventuelle de la puissance de chaque processeur ou encore de la performance des différents réseaux de communication utilisés. [7] montre à travers des algorithmes itératifs classiques les intérêts de la mise en œuvre de communication asynchrone lors de la résolution mais aussi les éventuels inconvénients. Parmi les avantages de ce mode de communication, la réduction du temps de synchronisation entre processeurs peut impacter positivement le temps global d'exécution surtout en environnement hétérogène. De même, le chevauchement du calcul avec la communication des données peut aussi améliorer la performance de l'application. Enfin, un partitionnement lors de de la décomposition du domaine tenant compte de l'absence de synchronisation en mode asynchrone peut aussi contribuer à la performance en répartissant efficacement le calcul. Les inconvénients de l'asynchronisme peuvent venir de la détection de la convergence globale étant donné qu'il n'y a pas de synchronisation des opérations. L'arrêt doit être décidé après une forme de communication globale à un certain point de l'algorithme ; il peut se faire lors de la communication inévitable entre processus pour annoncer la convergence locale. Un autre problème est aussi la tolérance aux pannes quoique cette défaillance peut aussi concerner le mode synchrone : si un des processus contribuant dans la résolution du problème se plante, tout le processus itératif peut s'écrouler si un mécanisme de reprise sur panne est mis en place.

1.2 Méthodes de résolution parallèles du problème de Poisson et de l'algorithme two-stage multisplitting de Krylov

0.0.3 Algorithme de Jacobi

0.0.4 Méthode de résolution GMRES

Native

Version « two-stage »

0.0.5 Solveur multisplitting

Version simple

Version améliorée

1.3 SIMGRID/SMPI : Simulateur d'exécution d'algorithmes parallèles MPI dans une grille de calcul

0.0.6 MPI - Message Passing Interface

0.0.7 Simulateur SIMGRID

1.4 Motivations

1.5 Conclusion partielle

Chapitre 2 : Etat de l'art et travaux de recherche associés

2.1 Concepts et définitions

Dans cette section, des concepts et des définitions relatifs à nos travaux sont passés en revue.

0.0.8 Performance de l'application parallèle et scalabilité

La performance d'une application dans un environnement distribué peut être définie comme « la capacité de réduire le temps pour résoudre le problème quand les ressources de calcul augmentent » [20]. L'objectif est de minimiser le temps d'exécution globale de l'application en ajoutant des ressources supplémentaires (processeurs, mémoire, ...). D'où la notion de « scalabilité » ou "montée en charge" ou encore "passage à l'echelle" dont l'objectif principal est d'accroitre la performance quand la complexité ou la taille du problème augmentent. Comme nous allons voir tout au long de ce chapitre, deux catégories de facteurs concourent à la difficulté de la prédiction des applications parallèles en considérant leur performance après la montée en charge des ressources : d'une part, on peut énumérer les facteurs liés à l'écosystème d'exécution tels que le nombre de processeurs, la taille de la mémoire et de sous-système de stockage, la latence et la bande passante des réseaux de communication ; d'autre part, les facteurs liés au code lui-même impactent aussi la performance de l'application affectant ainsi la prédiction : il s'agit par exemple de la fréquence de la communication et de la synchronisation, la faible parallélisation mais aussi le mauvais ordonnancement des tâches (équilibrage de charge) [20].

Afin de quantifier la performance d'un code, plusieurs métriques ont été définies mais le temps d'exécution global nécessaire pour atteindre la fin du programme reste le plus simple. On peut écrire :

$$T_{exec} = T_{calc} + T_{comm} + T_{surcharge} \tag{5}$$

où : T_{exec} : Temps d'exécution global

T_{calc} : Temps de calcul

T_{comm} : Temps de communication

$T_{surcharge}$: Temps de surcharge.

Le temps de calcul représente le temps pris par le code pour effectuer des calculs tandis que le temps de communication enregistre le temps des échanges de données ou d'instructions entre les processeurs. Le temps de surcharge comprend le temps pris lors des initialisations telles que la création des threads au début du programme mais aussi le temps de fermeture de l'application à la fin. En général, le temps de surcharge est négligeable par rapport aux temps de calcul et de communication.

Des métriques liées directement à la performance du processeur sont bien connues telles que le MIPS (Millions d'instructions par seconde), FLOPS (Floating Point Operations per second), SPECint ou encore SPECfp qui sont des benchmarks pour évaluer la performance du processeur sur des opérations arithmétiques respectivement sur des entiers ou des nombres réels. Par ailleurs, plusieurs métriques rapportées à la performance de l'application parallèle ont été définies mais nous allons retenir les trois les plus utilisées, à savoir le « speedup », « l'efficacité » du code et la loi d'Amdahl.

Le speedup est le rapport entre le temps utilisé pour l'exécution séquentielle du code et le temps pour son exécution en parallèle. Ce rapport peut être obtenu aussi comme le ratio entre le temps d'exécution du code sur un processeur et le temps d'exécution avec n processeurs. Ainsi, il mesure le gain escompté en résolvant le problème en parallèle au lieu d'un lancement en séquentiel.

$$S(n) = T_{Exec\_Seq}/T_{Exec\_Par}(n) \tag{6}$$

où : S(n) : speedup pour n processeurs

n : nombre de processeurs

T_{Exec_Seq} le temps d'exécution en mode séquentiel

T_{Exec_Par} le temps d'exécution en en parallèle.

L'efficacité E(n) représente la performance de chaque unité de calcul. Elle s'obtient en divisant le speedup par le nombre de processeurs n. On peut aussi l'écrire comme le rapport entre le temps d'exécution séquentielle et le temps d'exécution parallèle multiplié par le nombre de processeurs n.

$$E(n) = S(n)/n = T_{Exec\_Seq}/(n \times T_{Exec\_Par}(n)) \tag{7}$$

La loi de Amdahl donne une limite du speedup maximum qu'on peut obtenir avec un nombre de processeurs n donné. Elle stipule que si f compris entre 0 et 1 est la fraction du temps de la partie séquentielle du code, on a :

$$S(n) \leqslant \frac{1}{f + \frac{1-f}{n}} \tag{8}$$

Pour un système parallèle « idéal », le speedup est égal à n et l'efficacité à 1. Dans la pratique, le speedup est toujours inférieur à n avec une limite haute dûe à la loi de Amdahl et l'efficacité a une valeur entre 0 et 1. On peut démontrer que l'efficacité est une fnction décroissante du nombre de processeurs n tandis qu'elle est une fonction croissante de la taille du problème.

Dans le cadre de nos travaux, nous avions introduit une métrique utilisée lors de la comparaison de différentes variantes d'algorithmes résolvant le même problème exécutés en différents mode de communication (synchrone ou asynchrone). Ainsi, le « gain relatif » entre l'exécution de deux variantes de code résolvant un problème donné est le ratio entre le

temps d'exécution global du premier algorithme et le temps d'exécution global du deuxième algorithme selon le mode retenu pour chaque code.

$$G_{relatif} = T_{Exec\_Algo\_1}/T_{Exec\_Algo\_2} \times 100 \tag{9}$$

0.0.9 Taux d'erreur lors de la prédiction

Lors de l'exercice de prédiction sur la performance d'une application parallèle, un modèle est construit à partir des observations passées des variables considérées (données empiriques observées)afin de pouvoir prédire les résultats (données calculées) pour des nouvelles valeurs de ces variables. L'objectif lors de cette modélisation est de minimiser l'écart entre les valeurs calculées théoriques et les valeurs réelles observées.

Dans le cadre de la classe des algorithmes numériques itératifs consacrée à ces travaux, un autre taux d'erreur ϵ est déterminé d'avance et qui sert à détecter la convergence locale de l'algorithme [9]. A chaque itération, la différence entre la valeur approchée calculée, solution du problème, et celle obtenue à l'itération précédente est calculeé : si elle est inférieure au taux d'erreur accepté, l'algorithme s'arrête en ayant atteint la convergence sinon, on repart pour une nouvelle itération.

A l'itération k, la convergence est atteinte quand :

$$(\|X_l^k - X_l^{k+1}\|_\infty \leq \epsilon)$$

0.0.10 Weak contre strong scaling

Un des objectifs de nos travaux consistent à exécuter les algorithmes choisis en simulant leur exécution sur des plateformes de plus en plus larges avec un nombre de processeurs et de cores de plus en plus grand. Deux modes existent pour cette montée en charge donnant des résultats différents : le « weak » et le « strong » scaling.

La différence entre ces deux modes repose sur la variation de la taille du problème lors de la montée en charge (scaling). Pour le « weak » scaling, on essaie d'observer le comportement du programme en gardant le même nombre d'éléments à traiter par processeur ou core. Dans ce cas, les ressources de calcul additionnelles va augmenter proportionnellement à la taille du problème en entrée. Ainsi, la problématique ici est de résoudre un problème de plus grande taille. Par ailleurs, le « strong » scaling essaie de résoudre un problème donné plus vite. Ainsi, dans ce cas, la taille du problème en entrée reste constante même si on adjoint une capacité plus grande aux unités de calcul.

La figure Figure 3 montre que le temps d'exécution décroit (resp. reste constant) quand le nombre de processeurs augmente en strong mode (resp. en weak mode). De même, le speedup croit avec le nombre de processeur en strong mode tandis qu'il reste constant en weak mode.

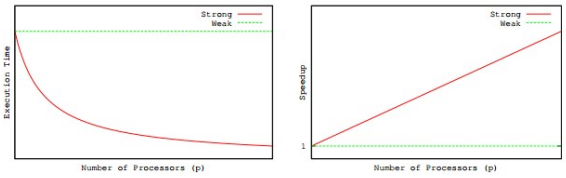


FIGURE 3 – Weak vs Strong scaling : Temps d'exécution et Speedup

2.2 Problématique sur la prédiction à large échelle de la performance des applications

La prédiction de la performance des applications parallèles à large échelle constitue ces dernières années une des préoccupations majeures des scientifiques et des utilisateurs des systèmes de calcul à haute performance. En effet, en considérant le coût de lancement nécessaire mais aussi le temps d'exécution imparti pour une telle application, il est toujours d'intérêt de disposer d'un outil ou d'un moyen afin de connaître le comportement de l'application en montant en charge. Pour cela, il s'agit d'estimer le temps total d'exécution T_{exec} dans ces conditions. De plus, dans le cadre d'un calcul sur la grille,l'objectif est de déterminer la configuration idéale, en termes de blocs et de nombre de noeuds (processeurs, coeurs) par bloc, pour obtenir le meilleur coût mais aussi le temps optimal d'exécution de l'application.

Dans ce chapitre, dans un premier temps, les problématiques et difficultés inhérentes à cet exercice de prédiction de la performance des applications parallèles sont abordées. Ensuite, nous allons passer en revue les solutions possibles apportées à ces problèmes.

De prime abord, on peut diviser en deux grands groupes, selon leurs objectifs, les travaux relatifs à la prédiction de la performance en environnement parallèle et de calcul à haute performance.

D'une part, la prédiction peut viser l'objectif de la conception, le développement et la mise au point de systèmes

qui n'existent pas encore physiquement. Cette catégorie regroupe entre autres la conception de nouvelles architectures de matériels (CPU, Mémoire, Stockage) [...] mais aussi par exemple, la mise en oeuvre d'une nouvelle infrastructure de réseaux de communication [...]. Plusieurs utilisations peuvent être exploitées pour ce type de prédiction. En effet, outre le calibrage de systèmes pour une exécution optimale, il permet le débogage et la mise au point des applications avec un ensemble de contraintes, que ce soit matérielles ou logicielles [..]. Notons tout de suite que cette dernière application sur le réseau a fait l'objet de nombreux travaux ces dernières années, permettant de déterminer ou d'estimer d'avance la performance et l'efficacité de la solution future projetée et éventuellement de corriger et d'améliorer les imperfections.

D'autre part, la prédiction de la performance d'une application parallèle se porte sur la détermination du temps d'exécution de la dite application en montant en charge sur une large échelle. Encore une fois, dans ce cas aussi, on ne dispose pas de l'environnement d'exécution cible mais on essaie de déterminer quel serait le temps total, donc le coût imputé au lancement de l'application sous diverses conditions. Ces dernières sont déterminées par plusieurs facteurs dont les principaux sont les paramètres d'entrée de l'application tels que la taille du problème à résoudre mais aussi les caractéristiques et la puissance globale intrinsèque de la grille de calcul de lancement : nombre de blocs, de processeurs / coeurs, les paramètres de la capacité du réseau de communication inter et intra-noeuds de la grille, ... Ainsi, une telle prédiction permet de conduire une analyse « what-if » du comportement de l'application si par exemple, on va multiplier par 10 ou 100 la taille du problème en entrée, mais aussi si on double la capacité de l'environnement cible en ajoutant d'autres blocs à la grille ou en apportant plus de processeurs dans chaque bloc. Les travaux rapportés dans cette thèse se focalisent plutôt sur cette seconde catégorie de prédiction de la performance d'applications spécifiquement écrites en MPI dans un environnement de grille de calcul.

Facteurs liés à l'écosystème

La prédiction de la performance des applications parallèles approchant le plus possible de la réalité avec un taux d'erreur minimal dépend de plusieurs facteurs pouvant avoir des impacts décisifs sur les résultats. En effet, à titre d'exemple, la modification de la topologie ou des paramètres de l'infrastructure du réseau de communication tels que la latence ou la taille de la bande passante aura inévitablement des conséquences sur la performance globale de l'application parallèle. En donnant un autre exemple, il est clair que la montée en charge en augmentant la taille du problème avec une plus grande capacité de calcul proposant un plus grand nombre de processeurs ou de coeurs modifiera la performance de l'application. Ainsi, de façon générale, plusieurs problématiques se posent quant au lancement d'une application parallèle dans une grille de calcul mais aussi, plusieurs facteurs influencent directement le comportement et la performance du système. Nombreux travaux ont déjà proposé des modèles de prédiction à large échelle sur la performance du code parallèle avec un taux d'efficacité plus ou moins acceptable. Certains de ces modèles seront détaillés dans le paragraphe 2.4.

Les scientifiques et les utilisateurs désirant lancer l'exécution d'un programme en environnement parallèle ont tous été confrontés à la même problématique de mise à disponibilité de l'environnement d'exécution. En effet, la réservation des ressources nécessaires pour lancer le système n'est pas toujours immédiate mais en plus, le coût peut ne pas être négligeable dans un contexte de rareté des machines super puissantes pourtant très sollicitées par différents acteurs [...]. Cette problématique peut être parfois accentuée par la non disponibilité de l'infrastructure cible parce que justement, les résultats obtenus par le lancement de l'application qui pourra déterminer les caractéristiques techniques de l'environnement cible. Ainsi, cette contrainte majeure doit être levée durant tout le cycle de vie de développement de l'application.

En effet, les coûteux développements et écritures du code de l'application, les opérations répétitives lors de sa mise au point ainsi que les tests itératifs de lancement requièrent un environnement réel disposant de la capacité nécessaire à ces opérations, ce qui n'est pas évident. Un autre facteur lié à cette problématique a toujours été aussi l'estimation à l'avance de cette capacité de calcul nécessaire afin d'avoir un environnement le plus adéquat afin d'éviter le gaspillage en cas de surestimation ou l'échec d'exécution en cas de sous-estimation. Cette estimation concerne les ressources primaires requises telles que le processeur, la taille mémoire DRAM et cache ainsi que le sous-système de stockage pour la capacité de calcul d'une part mais aussi les paramètres du réseau de communication (local ou distant) pour le temps de communication et d'échange de messages d'autre part. L'architecture inhérente à la grille de calcul composée d'entités reliées par des réseaux distants ajoute une autre considération pour la communication entre les processus parallèles sur le caractère hétérogène de l'infrastructure que ce soit la puissance de calcul des serveurs (différents types de processeurs) que le type des liaisons existants entre les blocs de la grille (réseaux hétérogènes). En effet, les environnements complexes de type grille de calcul actuels sont composés généralement de machines physiques dotées de processeurs multi-coeurs de différentes architectures (niveau de cache, latence entre processeurs, ...). De plus, en analysant la structure du réseau de communication dans la grille, on peut distinguer (1) d'abord, les échanges internes au niveau d'un élément d'un bloc (entre les coeurs d'un processeur et entre les processeurs d'un même serveur physique), (2) ensuite, les échanges « intra-blocs » caractérisant le trafic entre les différents éléments d'un bloc et (3) enfin, les échanges « inter-blocs » définissant la communication entre les blocs de la grille. Tant au niveau de leur topologie qu'en termes d'efficacité, ces trois niveaux de communication peuvent présenter des caractéristiques complètement différentes et hétérogènes. Ainsi, les deux premiers réseaux sont implémentés généralement dans un contexte de réseau local avec un temps de latence très court et une bande passante large. Tandis que le réseau de liaison entre les blocs de la grille peuvent être de type distant (lignes spécialisées distantes, canaux satellites de communication, réseau de type Internet, ...) donc d'une efficacité moindre en termes de latence et de bande passante mais aussi sujet à des perturbations diverses (Figure 4). Ces aspects liés à l'architecture de grille de calcul rendent la prédiction de la performance des applications parallèles plus difficiles. En effet, une surcharge élevée due à des perturbations sur le réseau inter-blocs de la grille peut fausser complètement les résultats de la prédiction du temps de communication global de l'application.

0.0.10.1 Facteur architecture des processeurs

Un autre facteur ayant un impact sur le temps d'exécution global est d'une part, le modèle d'architecture des processeurs de calcul et d'autre part, la puissance intrinsèque de ces derniers.

La course à la puissance nécessaire aux applications de calcul de haute performance ne cesse de s'accélérer de plus en plus vite exigeant une capacité de calcul de plus en plus grande. C. Willard [12] résume ce phénomène en disant que lorsqu'un problème - la conception d'un pont par exemple - est résolu, la solution trouvée n'est plus utile parce qu'on ne va pas refaire la conception. On passe généralement à un problème plus complexe - la conception d'un autre ouvrage plus complexe par exemple. La conséquence de cette course (actuellement du pentascale vers l'exascale) a suscité le développement des architectures de processeurs multi-coeurs dont l'accroissement de la puissance a dépassé la traditionnelle loi de Moore (renvoi). De plus, des co-processeurs spécialisés et autres accélérateurs (GPU : Graphic Processing Units []) ont été adjoints aux processeurs multi-coeurs pour améliorer le temps de calcul. Une autre architecture variante du multi-coeurs est le MIC (Many Integrated Core) [Intel Xeon Phi]. Ce type d'unité de calcul joue au départ le rôle de co-processeur pour les applications à haute intensité de calcul. Ainsi, plusieurs cœurs ont été pressés au niveau du pro-

cesseur (« socket ») emmenant un parallélisme au niveau de la puce. La Figure 4 donne un aperçu de l'architecture d'un processeur multi-coeurs. La performance d'une telle entité de calcul repose sur la vitesse d'accès des cœurs aux données

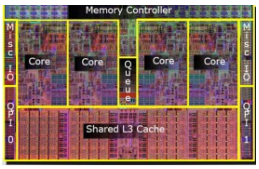
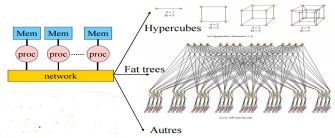


FIGURE 4 – Architecture des CPU multicoeurs

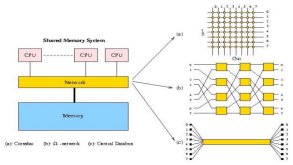
en mémoire. En effet, elle est dotée d'un bus rapide et une hiérarchie de cache mémoire beaucoup plus rapide d'accès que la RAM. En termes d'architecture, la classification de Flynn (1972) [] a créé quatre catégories de machines parallèles selon les flots de données et les flots d'instructions : SISD (Single instruction, single data), SIMD (Single instruction, multiple data), MISD et MIMD (Multiple instruction, multiple data). Cette dernière classe regroupant les machines parallèles généralistes actuelles se décline en trois sous-catégories :

- - Machine MIMD à mémoire partagée (Figure 5b) : Les unités de calcul accède à la mémoire partagée via un réseau d'interconnection (généralement, de type GigabitEthernet (renvoi) ou Infiniband (renvoi)). Il existe trois types d'implémentation : le crossbar, le Omega-Network et le Central Databus.
- Machine MIMD à mémoire distribuée (Figure 5a) : Chaque unité de calcul est doté de son espace mémoire propre. Un réseau d'interconnexion intègre l'ensemble assurant la communication entre ces unités. Il existe trois types de machines MIMD à mémoire distribuée : les hypercubes, les fat trees et les autres.
- Machine MIMD hybride (Figure 5c) : Dans ce cas, le système est la combinaison des deux modèles précédents : un ensemble de processeurs partage un espace mémoire et ces groupes sont interconnectés par un réseau.

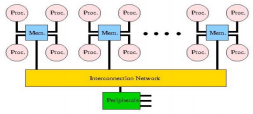
A titre d'exemple de machines parallèles, le site Top500.org [14] classe suivant différents critères les plus performantes. Ainsi, la fig. .. montre l'évolution de la puissance de calcul mondiale dont le top actuel développe un pic de



(a) Modèle MIMD Distribué



(b) Modèle MIMD partagé



(c) Modèle MIMD hybride

FIGURE 5 – Modèles de mémoire MIMD

performance théorique proche de 50 PetaFlops (33 Linpack PetaFlops (renvoi)) avec 3.120.000 cores (16 noeuds avec des processeurs de 2x12 cores par nœud) et plus de 1.240.000 Gb de mémoire (64 Gb par noeud) avec des accélérateurs 3 $\times$ Intel Xeon Phi par noeud. Il s'agit de la machine Tianhe-2 (MilkyWay-2) de la National Super Computer Center à Guangzhou en Chine [15]. A la tendance actuelle, l'atteinte de l'exaflops n'est pas loin.

Pour arriver à de telles puissances, diverses architectures de processeurs ont vu le jour ces dernières années. Outre l'Intel Xeon Phi cité plus haut, les processeurs basés sur les circuits intégrés FPGA (Field Programmable Gate Array) montrent une flexibilité efficace pour s'adapter par configuration au type d'applications à traiter [14]. En effet, cette architecture permet la programmation de la « matrice de blocs logiques » interconnectée par des liaisons toutes aussi programmables. Cette possibilité de programmation des circuits et des interconnexions entraine aussi la réduction de la consommation d'énergie. Par ailleurs, les unités GPU (Graphics Processing Unit) sont initialement des co-processeurs produits par AMD et NVIDIA pour des applications à fort rendu graphique, libérant ainsi la charge au processeur. Par la suite, elles ont été complètement programmables et se sont montrées très efficaces pour les algorithmes vectoriels.

0.0.10.2 Facteur : Mémoire et stockage

Les différentes architectures de processeurs parallèles vues plus haut se trouvent toutes confrontées au problème de chargement de données à traiter en mémoire. Ainsi, elles se sont dotées de contrôleurs de mémoire incorporés mais aussi divers niveaux de caches pour faire face à cette différence de vitesse de traitement entre les processeurs et les mémoires dynamiques. Par exemple, les machines SIMD utilisent des registres de communication internes pour communiquer avec les autres CPUs. Pour les machines de type MIMD où différentes tâches sont exécutées par chaque processeur à un instant

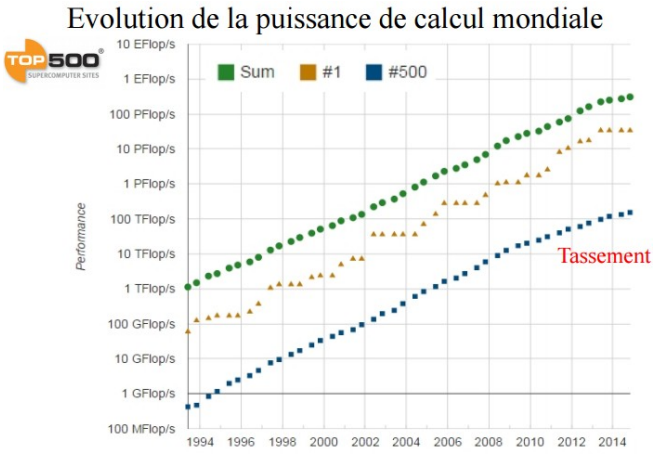
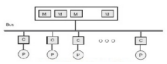


FIGURE 6 – Evolution de la puissance de calcul mondiale

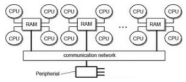
donné entraînant ainsi une synchronisation obligatoire pour des échanges de données entre processeurs, ces derniers peuvent exploiter la mémoire partagée pour effectuer ces transferts ou prévoir des bus dédiés à cette fin [16].

Par ailleurs, les mémoires, non intégrées au processeur, et les sous-systèmes de stockage constituent aussi un facteur important ayant un impact sur le temps d'exécution de l'application parallèle. En effet, les mémoires externes sont utilisées soit pour échanger des données entre les CPU, soit pour accéder à la zone mémoire pour lire, écrire ou mettre à jour des données. Dans ce domaine, en considérant les architectures parallèles MIMD, on peut classer en deux grandes catégories selon les modèles de mémoire [17] : (1) les multiprocesseurs et (2) les multicomputers (Fig …). La première catégorie regroupe les machines à mémoire partagée (« shared memory ») qui se subdivisent en trois classes selon le mode d'accès des CPU aux mémoires : (1) UMA ou « Uniform Memory Access » où tous les CPU accèdent une page mémoire physique de façon « uniforme », avec le même temps d'accès tolérant ainsi la mise à l'échelle. Dans ce cas, les CPU sont tous connectés aux mémoires via un bus ((Figure 7b)). Un système d'adressage global est appliqué à l'ensemble des mémoires physiques. (2) NUMA ou « Non Uniform Memory Access » où les groupes de CPU accèdent à des mémoires locales à travers des buses et les groupes sont interconnectés par un réseau de communication ((Figure 7a)). Dans ce cas, le temps d'accès des CPU aux pages mémoires varie selon que ces dernières sont locales ou distantes. L'espace d'adressage des mémoires se fait au niveau de chaque groupe de CPU. (3) L'architecture COMA (« Cache Only Memory Access ») est un hybride avec un modèle de programmation de mémoire partagée mais une implémentation physique de mémoire distribué ((Figure 7c)). Dans ce cas, chaque noeud détient une partie du système de l'espace d'adressage. Le partitionnement des

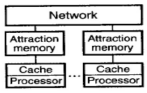
données étant dynamique, la structure COMA n'associe pas la même adresse à une page physique de la mémoire. Les mémoires locales dans ce cas de figure jouent finalement un rôle de cache au processeur.



(a) Architecture UMA



(b) Architecture NUMA



(c) Architecture COMA

FIGURE 7 – Modèles de mémoire MIMD

Malgré que dans le cadre de nos travaux, nous n'avions pas eu une contrainte particulière en termes de système de stockage, une brève revue des problématiques liées à ce sous-système en environnement de calcul parallèle est présentée parce qu'il peut influencer à large echelle sur la prédiction de la performance de l'application. Les systèmes traditionnels ont opté pour des architectures NFS (Network File System) ou de type NAS (Network Attached Storage) ou encore de type SAN (Storage Access Network). Malgré que les systèmes de stockage NFS et NAS sont relativement faciles à mettre en oeuvre, l'inconvénient majeur est qu'ils présentent un point de défaillance unique (SPOF) et ont des difficultés de monter en échelle. Pour le système SAN, les données sont stockées dans des baies de stockage accessibles par les unités de calcul à travers un réseau basé sur des canaux de fibres et des adapteurs de haut débit (HBA) ; ce qui rend le coût de l'implémentation rapidement excessif dès que le nombre de noeuds augmente. Dans un environnement d'applications parallèles, le réseau de communication doit avoir une très haute performance pour répondre aux besoins d'échange mais aussi d'accès aux données. En plus, il doit avoir la flexibilité et la capacité de monter en échelle suivant la demande du système. Ces caractéristiques requis sont accentués par la variabilité des besoins en entrées/sorties des applications HPC : dans le même lot d'applications exécutées, certaines accèdent à des données de manière séquentielle tandis que d'autres demandent des entrées/sorties aléatoires fortement sensibles. Les solutions apportées dénommées « système de fichiers parallèle » reposent sur la conception d'une architecture répondant à ces prérequis. Dans ce type de système de fichiers, les blocs de données sont répartis par morceaux dans différents serveurs et dans différentes locations du système de stockage. On peut ainsi accroitre le débit de stockage et d'extraction au fur et à mesure que le nombre de serveurs ou de baies de stockage augmentent.L'architecture sera réalisée par :

- l'introduction d'une couche de « noeuds de services de fichiers » entre les noeuds de calcul et les baies de stockage des données. Ces noeuds sont reliés en clusters via un réseau rapide de type Infiniband.
- L'ajout des «serveurs de metadata » (MDS : MetaData Server) qui gèrent les métadonnées accessibles à partir des « baies de stockage des métadonnées » (MDA) avant d'extraire les données proprement dites sur les baies de stockage en arrière-plan.

Les métriques utilisées pour caractériser une telle architecture sont le nombre nominal d'entrées/sorties par seconde (IOPS) d'une part et le débit de la bande passante du réseau reliant les différents composants (Gb/s) d'autre part. Plusieurs solutions globalement efficaces ont été avancées respectant cette architecture. On peut citer les « systèmes de fichiers ouverts » tels que pNFS (Parallel NFS), GFS, XFS, PVFS (Clemson University), MogileFS [...] mais Lustre [...] présenté dans la figure ... est largement utilisé en environnement de calcul parallèle : au moins, la moitié des clusters « top 10 » utilise ce modèle et plusieurs laboratoires l'ont aussi adopté (Pacific Northwest National Lab (PNNL), Lawrence Livermore National Lab (LLNL) mais aussi Los Alamos National Lab (LANL). Lustre utilise les OST («Object Storage Targets ») dans les serveurs de fichiers (en opposition au « Block Storage Device ») pour assurer la cohérence et la résilience du système de fichiers. A titre indicatif, le cluster de PNNL [19] avec 1800 processeurs Itanium délivrant jusqu'à 11 TFlops utilise Lustre avec une capacité de stockage de 53 Toctets avec une bande passante de 3.2 Gbits/s. Chaque nœud du cluster peut accéder au serveur parallèle Lustre avec un débit de 650 Mb/s.

La mise en œuvre des systèmes de fichiers parallèles pour les calculs à haute performance s'approche des technologies utilisées en entreprise pour exploiter les applications à données intensives traitant de très grandes masses de données. En effet, les « sciences de données », « big data », « analytics » (business intelligence, Datamart, Data Mining) demandent des accès très rapides à des grands volumes de données variées, structurées ou non structurées, pour en extraire une information utile. Pour cela, le principe « d'apporter le calcul auprès des données » (« Bring the compute to the data ») est appliqué en lieu et place du traditionnel « extraire et charger en mémoire les données du système de stockage pour traitement par l'unité de calcul ». Hadoop [...], une plateforme de traitement de « big data » la plus utilisée, combine dans la même machine physique les « nœuds de calcul » et les « nœuds de données ». Cet ensemble d'outils ayant une architecture fortement distribuée utilise le mécanisme de transfert des données du système de stockage « globalement partagé et persistent » ayant une large capacité vers le système de fichier local avant traitement.

0.0.10.3 Facteur : Réseaux de communication

Dans un contexte d'exécution parallèle et distribuée des applications, la communication entre les processus de calcul pour échange de données ou d'instructions est critique et peut constituer un goulot d'étranglement pour le temps d'exécution et la montée en charge de l'applicaiton. En effet, la performance globale quantifiée par le temps d'exécution de l'application dépend fortement de la nature et de la typologie des réseaux de communication. Il a été mis en exergue dans les paragraphes précédents l'importance du trafic de données entre chaque unité de calcul et les différentes couches de mémoire vive utilisées par le système. Dans un environnement de grilles de calcul, de clusters ou de P2P, d'autres types de réseaux de communication influencent cette performance.

0.0.11 Facteurs liés au code de l'application

Outre ces problématiques liées directement à l'environnement de lancement, plusieurs autres facteurs liés au code de l'application lors de son exécution peuvent influencer le comportement du système rendant aussi la prédiction de la performance complexe et difficile. Ces facteurs liés au comportement du code lors de son exécution en parallèle vont influencer la performance globale en impactant le temps de calcul et le temps de communication des données entre les unités de calcul.

0.0.11.1 Facteur : Taille du problème

Parmi les facteurs impactant le temps de calcul, la taille du problème peut avoir une grande influence sur le temps de calcul surtout en strong scaling. En effet, dans ce mode de scalabilité, la taille du problème étant fixe alors qu'on augmente la puissance de calcul par l'ajout de processeurs et coeurs supplémentaires, le temps de calcul va varier en fonction de ces changements. En mode weak scaling où la taille du problème augmente dans la même proportion que l'accroissement du nombre de processeurs / coeurs, le temps de calcul global attendu reste théoriquement plus ou moins constant. La taille du problème qui ne cesse d'augmenter pour le besoin des applications parallèles constitue un élément impactant le temps total d'exécution du code.

0.0.11.2 Performance de la parallélisation

Dans cette section, la notion de "performance de la parallélisation" est intrduite pour caractériser la performance d'un code une fois executé en mode parallèle. C. Rosas et Al. [...] définit cette mesure (ηParallel) comme étant le produit des trois facteurs fondamentaux "normalisés" suivants dont chaque facteur est quantifié par une valeur entre 0 et 1 :

$$\eta Parallel = LB \times Ser \times Trf \tag{10}$$

Où :

- L'efficacité de la « répartition des charges » LB ("Load Balancing") est définie comme étant « la perte d'efficacité potentielle» sur le temps de calcul de chaque processus. Elle est mesurée comme étant le rapport entre le temps de calcul moyen par processeur et le temps de calcul maximum enregistré sur l'ensemble des processeurs participants :

$$LB = [\sum_{k=1}^{p} eff_k)/p]/max(eff_k) \tag{11}$$

  où : p est le nombre de processeurs et eff_k ("Efficiency") le temps de calcul utilisé par le processeur k.
- L'efficacité de la « sérialisation » : Elle représente l'inefficacité causée par les « dépendances dans le code » qui se traduit par la nécessité d'échanger des données entre les processeurs. Ces dernières peuvent impacter de façon importante la performance du code parallèle. Ce facteur est mesuré comme étant le temps maximum enregistré pour tous les processeurs présents lors de l'exécution du code en faisant abstraction du temps des échanges : on considère comme si on est en présence d'une architecture à « communication instantanée » c'est-à-dire un réseau

avec une bande passante infinie et une latence égale à 0. Dans ce cas, ideal (eff_i) est l'efficacité du processeurs i sans le temps de communication.

$$Ser = max(ideal(eff_i)) \tag{12}$$

- L'efficacité du « transfert » de données : La montée en charge de la taille du problème impactera la taille des données à échanger entre les processus. Ce facteur est défini comme étant la perte de performance globale due aux transferts des données. En prenant en compte le temps de communication, il est mesuré comme le ratio entre le maximum entre les temps relatifs d'exécution des processus concurrents (rapport entre le temps d'exécution T_i d'un processus et le temps total réel d'exécution T du code) et l'efficacité de la sérialisation Ser :

$$Trf = max(T_i/T)/Ser \tag{13}$$

Les auteurs ont montré que cette mesure de la performance de la parallélisation est indépendante du temps absolu total d'exécution. Pour les algorithmes itératifs, cette métrique ne dépend pas du nombre d'itérations avant l'arrêt de l'algorithme : le temps d'exécution d'une itération reste constant.

Cette quantification de la performance de la parallèlisation du code repose sur les trois paramètres suivants appelés aussi « inhibiteurs de la performance » qui décrivent selon [12] la "sensibilité" du code : (1) la sensibilité à la fréquence CPU, (2) la sensibilité à la bande passante mémoire et enfin (3) le temps consacré aux communications et les entrées / sorties. Selon l'algorithme considéré ou l'aspect scientifique du code, l'application peut être influencée par ces paramètres. L'analyse du code par le profiling et l'optimisation pourront aider à cette sensibilité du code et à améliorer la performance de sa parallèlisation.

Dans le cadre de ces travaux, à plus large échelle, c'est-à-dire en augmentant la taille du problème en entrée comme la capacité de calcul disponible, les facteurs suivants vont influencer de plus en plus le temps d'exécution de l'application impactant ainsi la performance de la parallélisation du code. Selon [18], même si la surcharge engendrée par la parallélisation du code (« surcharge due à la parallélisation ») ainsi que celle naturellement subie par le système comme dans une exécution séquentielle (« surcharge système ») peuvent ne pas être négligeables, on constate comme précédemment que les facteurs liés à « l'oisivité » des processeurs ainsi que la communication entre les différentes couches mémoires (DRAM, cache, « mémoire d'attraction » (renvoi)) peuvent peser lourdement à grande échelle sur la performance globale de l'application. La surcharge due à la parallélisation provient de l'initialisation par processeur pour une exécution parallèle (qui n'existe pas lors d'une exécution séquentielle). Le partitionnement des tâches mais aussi les tâches de vérrouillage et de déverrouillage lors d'une entrée et de sortie d'une section critique du code contribue à l'importance de ce facteur. La surcharge système comme les défauts de pages, l'interruption horloge, le mécanisme de fork/join, ... peut être accentuée par rapport à une exécution séquentielle surtout pour les programmes à haut degré de parallélisme parce que ces actions sont inhérentes à un processeur et l'augmentation du nombre de processeurs lors d'une exécution parallèle peut engendrer une surcharge système non négligeable. Toutefois, comme avancé plus haut, ces surcharges peuvent ne pas être significatives comparées au temps perdu suite à l'oisivité (idle) des blocs de calcul. Cette dernière est surtout due à une parallélisation insuffisante ou encore par une répartition des charges non optimale. Enfin, le facteur communication nécessaire pour le thread courant de chercher des données qui ne sont pas localisées dans ses mémoires caches locales peut affecter dramatiquement la performance de la parallélisation du programme. En effet, pendant cette recherche, l'unité de calcul reste bloqué (stalled).

2.3 Techniques de profiling et instrumentation des applications parallèles

2.4 Méthodes de prédiction de la performance de l'application parallèle

2.5 Conclusion partielle

PARTIE II - Travaux de contributions, résultats et perspectives

Chapitre 3 : Comparaison par simulation à large échelle de la performance de deux algorithmes itératifs parallèles en mode asynchrone

3.1 Protocoles et expérimentations

3.2 Résultats

3.3 Conclusion partielle

Chapitre 4 : Simulation avec SIMGRID de l'exécution des solveurs linéaires en mode synchrone et asynchrone sur un environnement multi-coeurs simulés

4.1 Protocoles et expérimentations

4.2 Résultats

4.3 Conclusion partielle

Chapitre 5 : Modèle de prédiction de la performance à large échelle d'un algorithme itératif parallèle

5.1 Approche et méthodologie

5.2 Expérimentations et résultats

5.3 Conclusion partielle

Chapitre 6 : Conclusion générale et perspectives

6.1 Conclusion générale

6.2 Travaux futurs et perspectives

BIBLIOGRAPHIE ET REFERENCES

[6] J.M. BAHI, S. CONTASSOT-VIVIER, R. COUTURIER. Interest of the asynchronism in parallel iterative algorithms on meta-clusters. *LIFC - Université de Belford-Montbéliard*.

[7] T.P. COLLIGNON and M.B. van GIJZEN. Fast iterative solution of large sparse linear systems on geographically separated clusters. *The International Journal of High Performance Computing Applications* 25(4) 440–450.

[8] D. BERTSEKAS and J. TSITSIKLIS. Parallel and Distributed Computation, Numerical Methods. *Prentice Hall Englewood Cliffs N. J., 1989*.

[9] C. E. RAMAMONJISOA, L. Z. KHODJAV, D. LAIYMANI, A. Giersch and R. Couturier. Simulation of Asynchronous Iterative Algorithms Using SimGrid. *2014 Femto-ST Institute - DISC Department - Université de Franche-Comté, IUT de Belfort-Montbéliard*

[10] M. J. VOSS and R. EIGEMANN. Reducing Parallel Overheads Through Dynamic Serialization. *Purdue University School of Electrical and Computer Engineering*.

[11] K. J. BARKER, K. DAVIS, A. HOISIE, D. J. KERBYSON, M. LANG, S. PAKIN and J. C. SANCHO. Using performance modeling to design large-scale systems. *Los Alamos National Laboratory(LANL), New Mexico*.

[12] M. DUBOIS and X. VIGOUROUX. Unleash your HPC performance with Bull. *Maximizing computing performance while reducing power consumption*. http ://www.hpctoday.fr/published/regional/operations/docs/W-HPCperformance-en1.pdf

[14] Site du top500. http ://www.top500.org

[15] C. HARRIS et al. HPC Technology Update. *Pawset Supercomputing Center - Sept 2015*. http ://www.pawsey.org.au/wp-content/uploads/2015/09/Pawsey_HPC_Technology_Update_20150923.pdf

[16] A. J. van der STEEN, J. J. DONGARRA. Overview of Recent Supercomputers. *Academic Computing Centre Utrecht, the Netherlands, Department of Computer Science, University of Tennessee, Knoxville, Mathematical Sciences Section, Oak Ridge, National Laboratory, Oak Ridge*. http ://citeseerx.ist.psu.edu/viewdoc/download ?doi=10.1.1.49.3743&rep=rep1

[17] V. RAJPUT , S. KUMAR, V.K.PATLE. Performance Analysis of UMA and NUMA Models". *School of Studies in Computer Science Pt.Ravishankar Shukla University, Raipur,C.G.* http ://www.ijcset.net/docs/Volumes/volume2issue10/ijcset201202

[18] D. NGUYEN, Raj VASWANI and J. ZAHORIAN. Parallel Application Characterization for Multiprocessor Scheduling Policy Design. *Department of Computer Science and Engineering - University of Washington, Seattle, USA*.

[19] M. EWAN. Exploring Clustered Parallel File Systems and Object Storage. *2012*. https ://software.intel.com/en-us/articles/exploring-clustered-parallel-file-systems-and-object-storage

[20] F. SILVA, R. ROCHA : Parallel and Distributed Programming - Performance Metrics. *DCC-FCUP*.

[21] G. BALLARD et Al. Communication Optimal Parallel Multiplication of Sparse Random Matrices". *UC Berkeley, INRIA Paris Rocquencourt, Tel-Aviv University*. http ://www.eecs.berkeley.edu/~odedsc/papers/spaa13-sparse.pdf