



SPIM

Thèse de Doctorat



UFC

école doctorale **sciences pour l'ingénieur et microtechniques**
UNIVERSITÉ DE FRANCHE-COMTÉ

Titre

Sous-titre

■ PRÉNOM NOM

SPIM

Thèse de Doctorat



école doctorale **sciences pour l'ingénieur et microtechniques**
UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X |

THÈSE présentée par

PRÉNOM NOM

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Titre

Sous-titre

Soutenue publiquement le 17 septembre 2012 devant le Jury composé de :

INCROYABLE HULK	Rapporteur	Professeur à l'Université de Gotham City Commentaire secondaire
SUPER MAN	Examineur	Professeur à l'Université de Gotham City
BAT MAN	Directeur de thèse	Professeur à l'Université de Gotham City

REMERCIEMENTS

SOMMAIRE

I	PARTIE I : Contexte scientifique et revue de l'état de l'art	3
1	Cadre de travail et contexte scientifique	5
1.1	Classe des algorithmes itératifs parallèles à large échelle dans une grille de calcul	5
1.1.1	Partitionnement du problème	6
1.1.2	Modes d'exécution synchrone et asynchrone	7
1.2	Méthodes de résolution parallèles du problème de Poisson et de l'algorithme two-stage multisplitting de Krylov	9
1.2.1	Algorithme de Jacobi	9
1.2.2	Méthode de résolution GMRES	9
1.2.3	Solveur multisplitting	9
1.3	Simulateurs d'exécution d'algorithmes parallèles MPI dans une grille de calcul	9
1.3.1	Calcul sur grille	9
1.3.2	Généralités sur la simulation	11
1.3.3	MPI - Message Passing Interface	13
1.3.4	Simulateur SIMGRID - SMPI	15
1.4	Motivations	15
1.5	Conclusion partielle	15
2	Etat de l'art et travaux de recherche associés	17
2.1	Concepts et définitions	17
2.1.1	Performance de l'application parallèle et scalabilité	17
2.1.2	Taux d'erreur lors de la prédiction	19
2.1.3	Weak contre strong scaling	19
2.2	Problématique sur la prédiction à large échelle de la performance des applications	20
2.2.1	Facteurs liés à l'écosystème	21
2.2.1.1	Facteur architecture des processeurs	22
2.2.1.2	Facteur : Mémoire et stockage	24

2.2.1.3	Facteur : Réseaux de communication	28
2.2.2	Facteurs liés au code de l'application	28
2.2.2.1	Facteur : Taille du problème	29
2.2.2.2	Performance de la parallélisation	29
2.3	Techniques d'analyse de performance des applications parallèles	32
2.3.1	Généralités et objectifs	32
2.3.2	Approches et méthodologie	32
2.3.3	Quelques outils d'analyse de performance	34
2.4	Méthodes de prédiction de la performance des applications parallèles . . .	35
2.5	Conclusion partielle	35
II	PARTIE II - Travaux de contributions, résultats et perspectives	37
3	Comparaison par simulation à large échelle de la performance de deux algorithmes itératifs parallèles en mode asynchrone	39
3.1	Protocoles et expérimentations	39
3.2	Résultats	39
3.3	Conclusion partielle	39
4	Simulation avec SIMGRID de l'exécution des solveurs linéaires en mode synchrone et asynchrone sur un environnement multi-coeurs simulés	41
4.1	Protocoles et expérimentations	41
4.2	Résultats	41
4.3	Conclusion partielle	41
5	Modèle de prédiction de la performance à large échelle d'un algorithme itératif parallèle	43
5.1	Approche et méthodologie	43
5.2	Expérimentations et résultats	43
5.3	Conclusion partielle	43
6	Conclusion générale et perspectives	45
6.1	Conclusion générale	45
6.2	Travaux futurs et perspectives	45

III Annexes	57
A Premier chapitre des annexes	59
B Second chapitre des annexes	61

INTRODUCTION

I

PARTIE I : CONTEXTE SCIENTIFIQUE ET REVUE DE L'ÉTAT DE L'ART

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.

$$X^{k+1} = f (X^k), k = 0, 1, \dots \quad (1.1)$$

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 concurrente 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 :

$$(k \leq \text{MaxIter}) \text{ or } (\|X_j^k - X_j^{k+1}\|_\infty \leq \epsilon) \quad (1.2)$$

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.

1.1.1/ 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 ?? 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.

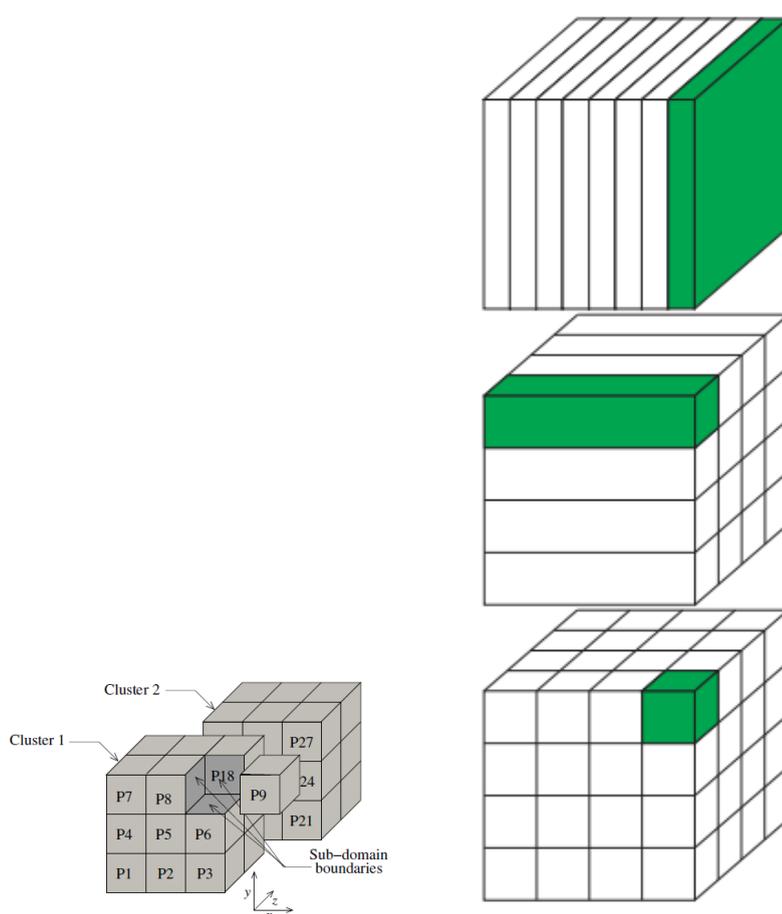


FIGURE 1.1 – Découpage d'une matrice tridimensionnelle entre deux clusters formés de 18 processeurs chacun

FIGURE 1.2 – Décomposition en domaines 1D, 2D et 3D

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 noeuds sont les processeurs P_i .

1.1. CLASSE DES ALGORITHMES ITÉRATIFS PARALLÈLES À LARGE ÉCHELLE DANS UNE GRILLE

Une fois partitionnée en m blocs, la relation récurrente de l'équation (1.1) peut s'écrire :

$$x_{k+1} = (x_1^k, x_2^k, \dots, x_n^k), k = 1, \dots, n \quad (1.3)$$

ou en termes de blocs :

$$X_{k+1} = (X_1^k, X_2^k, \dots, X_m^k), k = 1, \dots, m \quad (1.4)$$

Donc, on peut écrire :

$$X_{k+1} = F(X_k) \quad (1.5)$$

$$\iff (\exists F_k)(X_1^{k+1}, X_2^{k+1}, \dots, X_m^{k+1}) = (F_1(X_k), F_2(X_k), \dots, F_m(X_k)) \quad (1.6)$$

Où :

$$X_i^{k+1} = F_i(X^k) = F_i(X_1^k, X_2^k, \dots, X_m^k) \text{ pour } i = 1, \dots, k \quad (1.7)$$

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 ??) :

- 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 .

1.1.2/ 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.

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 1.3 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.

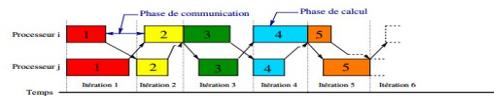


FIGURE 1.3 – Modèle de communication synchrone

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 1.4). 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 oeuvre 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 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 RÉOLUTION PARALLÈLES DU PROBLÈME DE POISSON ET DE L'ALGORITHME

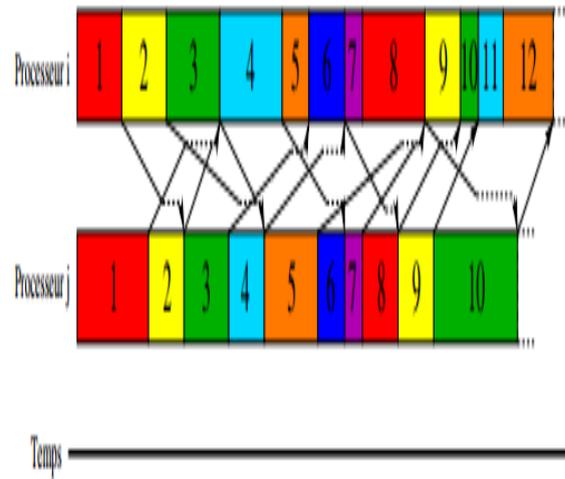


FIGURE 1.4 – Modèle de communication asynchrone

1.2/ MÉTHODES DE RÉOLUTION PARALLÈLES DU PROBLÈME DE POISSON ET DE L'ALGORITHME TWO-STAGE MULTISPLITTING DE KRYLOV

1.2.1/ ALGORITHME DE JACOBI

1.2.2/ MÉTHODE DE RÉOLUTION GMRES

Native

Version « two-stage »

1.2.3/ SOLVEUR MULTISPLITTING

Version simple

Version améliorée

1.3/ SIMULATEURS D'EXÉCUTION D'ALGORITHMES PARALLÈLES MPI DANS UNE GRILLE DE CALCUL

1.3.1/ CALCUL SUR GRILLE

Une grille de calcul est caractérisée par "un type de système parallèle et distribué qui permet le partage, la sélection et l'agrégation de ressources distribuées géographiquement

selon leurs capacités” [25] afin de résoudre un problème complexe donné. Ainsi, une grille est composée d’un ensemble de grappes de machines interconnectées entre elles à travers un réseau de communication qui peut s’étendre sur des zones géographiques éloignées (Figure 1.5). Les capacités de calcul, les mémoires, les applications et les systèmes de stockage sont partagées par les applications parallèles et distribuées. Le calcul sur une grille est caractérisé par un environnement ”hétérogène, dynamique et scalable”.

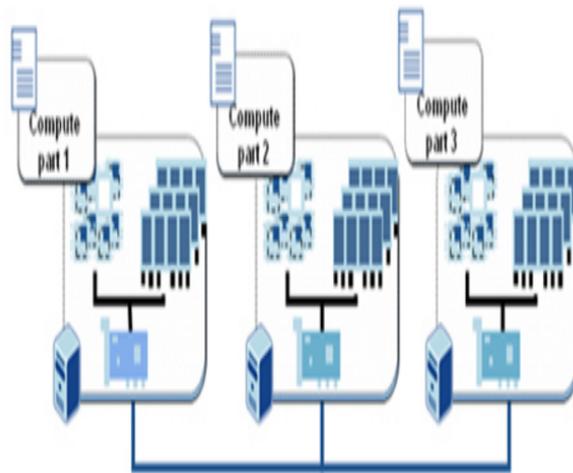


FIGURE 1.5 – Architecture d’une grille de calcul

L’hétérogénéité montre la variété des éléments composant la grille de calcul. On peut être en présence de différentes architectures de processeurs dans les machines d’une grappe ou entre les grappes. Les fréquences d’horloge de ces processeurs peuvent être aussi différentes. De même, l’architecture ou la méthode d’accès des mémoires (DRAM, stockage) utilisées dans la grille de calcul peut être aussi de types différents. Enfin, la topologie ainsi que la performance des réseaux de communications interconnectant les éléments de la grille peuvent être aussi avoir des débits complètement hétérogènes.

Le caractère dynamique de la grille résulte de la relative facilité de changer de configuration. On peut ainsi tailler dynamiquement l’allocation des ressources de la grille aux utilisateurs selon les besoins de leur demande respective. Cet aspect a été élargi à ”l’élasticité” de l’environnement dans le cadre du ”cloud computing”.

Enfin, la scalabilité de la grille de calcul découle de sa conception modulaire permettant d’ajouter d’autres composants selon les besoins. Pour augmenter par exemple la capacité de calcul de la grille, il suffit d’ajouter de nouveaux clusters pour une plus grande puissance globale de la grille.

Le milieu de la recherche dispose d’une grille de calcul dédié : le Grid’5000 [26, 27] est une grille répartie géographiquement dans différentes villes de France (Figure 1.6) mettant à disposition un ”banc d’essai polyvalent à grande échelle” pour les expérimentations

1.3. SIMULATEURS D'EXÉCUTION D'ALGORITHMES PARALLÈLES MPI DANS UNE GRILLE DE CAL

de la recherche en informatique particulièrement le calcul parallèle sur grille, sur le cloud, le calcul à haute performance mais aussi sur le Big Data. Grid'5000 permet aux utilisateurs l'accès à des ressources importantes de calcul dans un environnement complètement configurable et contrôlable. Il peut aussi fournir une trace détaillée ainsi que d'autres informations de mesure sur le comportement de l'application lors de l'exécution pour une étude ultérieure.

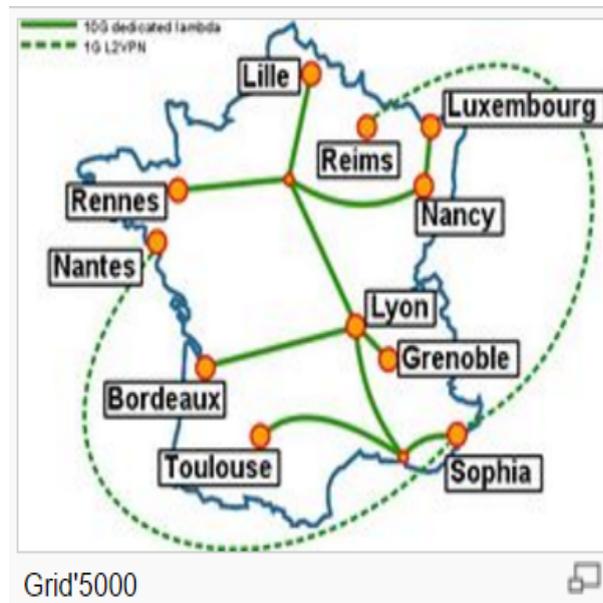


FIGURE 1.6 – Grid'5000 : Répartition géographique

Grid'5000 est construit autour de plus de 1000 noeuds physiques de différents constructeurs composés de plus de 2000 processeurs (Intel Xeon et AMD Opteron) avec un total de plus de 10.000 coeurs. Plus de 650 différentes cartes d'interface réseau Ethernet, Infiniband et Myrinet sont interconnectés avec plus de 40 accélérateurs de type NVIDIA GPU et Intel Xeon Phi. Dès sa conception, Grid'5000 a pris en compte la diversité des intérêts et des besoins des différents utilisateurs. En effet, dépendant de leur centre d'intérêt peuvent se focaliser sur les protocoles réseau ou les systèmes d'exploitation particuliers ou d'autres problématiques sur la tolérance aux pannes, ces derniers peuvent configurer leur propre environnement de lancement de leurs applications. La reproductibilité des résultats a été soigneusement étudiée pour permettre une analyse ultérieure de la performance. De plus, Grid'5000 assure la scalabilité, la qualité de service (QoS) mais aussi et surtout la sécurité de l'environnement par le verouillage de la connexion vers Internet par exemple.

1.3.2/ GÉNÉRALITÉS SUR LA SIMULATION

La simulation est largement utilisée dans divers domaines de la recherche scientifique. Elle consiste au processus de la mise en oeuvre et "de la conduite d'expérimentations sur un modèle (une représentation simplifiée du réel) dans le but de comprendre le comportement du système modélisé sous des conditions sélectionnées ou de l'évaluation de diverses stratégies pour le fonctionnement du système sous la limite imposée par

les critères de développement et d'exploitation" [29]. Particulièrement, la simulation de l'exécution d'une application parallèle distribuée étudie son comportement (résultats en sortie, temps de performance, scalabilité, ...) sur un environnement virtuel imitant au mieux le fonctionnement d'une plateforme physique réel ou d'un système en cours d'élaboration (banc d'essai) ou encore d'une hypothétique machine non encore réalisée. Ainsi, la simulation informatique se focalise sur le comportement dynamique du modèle à travers le temps. Plusieurs raisons motivent une telle simulation : à titre d'exemple, de réduire les coûts de la conception d'un système et d'éviter les erreurs, de produire dans un temps raisonnable des résultats en sortie d'un modèle ayant un temps d'exécution élevé, de répondre à des scénarios d'exécution avec des questions "what-if" (tests et évaluations), ou encore de créer des outils de simulation pour des formations ou des jeux.

Dans le cadre d'une grille de calcul, les simulateurs ou les outils de simulation permettent à l'inverse des plateformes réelles l'évaluation de la performance des expérimentations "répétables et contrôlables" [25] sur des configurations flexibles et scalables. En effet, les environnements réels montrent leurs limites sur leur rigidité de passage à l'échelle mais aussi sur la flexibilité de disposer un environnement de calcul particulier répondant aux besoins précis de l'application à un moment donné. Selon la classification dans [30], la simulation d'applications sur une grille de calcul rejoint la classe de simulation "virtuelle" par l'utilisation d'équipements de simulation par des personnes réelles. De façon générale, le simulateur utilise une échelle de temps "discret", c'est-à-dire le temps est découpé en intervalles qui peuvent être réguliers ou non. Dans le cas d'un système à temps discret régulier, le simulateur maintient et met à jour éventuellement un ensemble de "variables d'état" qui reflètent l'état du système physique à un instant t donné. Un "événement" est associé à chaque instant donné à une "transition d'état". Pour des comparaisons futures, on distingue le "temps physique" comme étant le temps considéré au niveau du système physique, du "temps de simulation" et "le temps de l'horloge murale" désigne le temps de simulation du modèle. Toutefois, le "temps de simulation" est une notion abstraite utilisée par le simulateur pour évaluer le temps de simulation. Il est défini [30] comme étant "un ensemble de valeurs totalement ordonné E où chaque valeur représente un temps du système physique à modéliser et qui vérifie les conditions suivantes :"

Soient E l'ensemble des temps discrets de simulation et P l'ensemble des temps du système physique.

$$\begin{aligned} \text{Si } (T_1 \in E, T_2 \in E) \text{ et } (P_1 \in P, P_2 \in P) \text{ et } (T_1 < T_2) \\ \Rightarrow ((P_1 < P_2) \text{ et } \exists K \in \mathbb{N}, T_2 - T_1 = K \times (P_2 - P_1)) \end{aligned} \quad (1.8)$$

La définition précédente montre le lien linéaire étroit entre les intervalles de temps de simulation et celles des temps physiques. Ce qui permet d'estimer entre autres le temps d'exécution probable d'une application à partir du temps de simulation observé. Outre ce temps global de l'outil de simulation et les variables d'état, une liste des événements à exécuter complète la composition du simulateur au temps discret.

Le changement des variables d'état peut s'effectuer soit à une fréquence régulière du temps de simulation (exécution rythmée par le temps) soit au début et à la fin d'un événement donné (exécution rythmée par les événements). Dans le cas d'une simulation d'une application parallèle et distribuée où plusieurs processeurs ou coeurs

1.3. SIMULATEURS D'EXÉCUTION D'ALGORITHMES PARALLÈLES MPI DANS UNE GRILLE DE CALCUL

interconnectés concourent à résoudre ensemble le problème posé, plusieurs autres aspects liés à l'environnement doivent être considérés :

- L'initialisation du système ;
- Les échanges de données entre les processus ;
- La synchronisation des processus ;
- La détection de deadlock et la reprise ;
- L'arrêt et la fermeture du système.

Le tableau 1.1 donne quelques exemples de simulateurs pour des applications parallèles et distribuées sur une grille de calcul [28, 25].

OUTIL	DESCRIPTION	DEVELOPPEUR	APPLICATIONS CIBLE
SimJava	SimJava fournit un processus de simulation avec une animation à travers d'entités communiquant entre elles http://www.dcs.ed.ac.uk/home/hase/simjava/	Université de Edinburgh (UK)	Simulation d'événements discrets
Bricks	Bricks est un outil d'évaluation de performance analysant divers schémas d'ordonnement dans un environnement de grille de calcul http://matsu-www.is.titech.ac.jp/takefusa/bricks/	Tokyo Institute of Technology (Japan)	Simulation de grille
Microgrid	Microgrid permet la simulation d'une montée en charge des applications sur grille de calcul en utilisant des ressources clusterisées http://www-csag.ucsd.edu/projects/grid/microgrid.html	University of California at San Diego (USA)	Simulation de grille
Simgrid	Simgrid simule les applications distribuées dans un environnement distribué hétérogène http://grail.sdsc.edu/projects/simgrid/	University of California at San Diego (USA)	Simulation de grille
Gridsim	Gridsim permet la modélisation et la simulation d'entités impliquées dans le calcul parallèle et distribué par la création et le pilotage de différentes ressources http://www.buyya.com/gridsim/	Monash University Australie	Simulation de grille

TABLE 1.1 – Quelques outils de simulation pour une grille de calcul

Simgrid est l'outil choisi dans le cadre de ces travaux pour étudier le comportement et évaluer la performance d'applications parallèles distribuées à grande échelle. Une section de ce chapitre sera dédiée à la description plus détaillée de cette plateforme.

1.3.3/ MPI - MESSAGE PASSING INTERFACE

MPI ou "Message Passing Interface" est les spécifications d'une librairie d'interface pour le transfert de message entre les processus d'une application parallèle. A sa version MPI-3 (2015), elle est largement utilisée dans la recherche dans le domaine du calcul à haute performance avec des compilateurs C/C++ et Fortran généralement. La facilité de l'utilisation et la portabilité à travers différents systèmes hétérogènes ont guidé le développement de ces spécifications MPI standards. Ces derniers peuvent être matérialisés sur différentes plateformes cibles telles qu'une grille de calcul, des machines multiprocesseurs et multicores à mémoires partagées ou distribuées, un réseau de stations de travail interconnectés ou encore des environnements hybrides obtenus par la combinaison de ces architectures. Principalement, les standards MPI sont implémentés sur différents systèmes d'exploitation soit avec MPICH [32] ou OpenMPI [33] tous les deux des logiciels libres à haute performance et portable développés par des consortiums de chercheurs et des partenaires et collaborateurs industriels. Plusieurs domaines sont couverts par les spécifications de MPI dont les plus importants sont cités ci-dessous [31,32,33].

- Groupes, contexte et communicateur : Définit l'initialisation de l'environnement d'exécution du programme parallèle MPI. Un groupe de processeurs est formé et un unique contexte de communication est créé et les deux sont intégrés ensemble dans un communicateur.
- La gestion de l'environnement MPI : Permet à l'utilisateur d'interagir avec l'environnement MPI créé lors du lancement du programme parallèle. Elle assure par abstraction la portabilité de l'application entre des plateformes matérielles et logicielles différentes.
- La gestion des processus : Définit la création des processus participant à l'exécution de l'application mais aussi détermine la topologie et la gestion des groupes de processus en accord par exemple avec des architectures complexes comme les grilles de calcul.
- Les types de données : Permettent de créer des structures de données complexes en mémoire à partir des types de données de base comme l'entier, le float, etc...
- Les communications : Rassemblent les spécifications des protocoles d'échanges de messages entre les processus. On distingue les communications point à point, les communications collectives mais aussi les entrées / sorties parallèles.

Le programme MPI s'exécute sur chaque processeur une fois que l'environnement logique est créé par la routine MPI_Init. Ce dernier est constitué d'un groupe de processus, d'un contexte et d'un communicateur (par défaut MPI_COMM_WORLD), voir la figure 1.7-a. Chaque processus est identifié par son rang dans le groupe associé au communicateur (MPI_Comm_rank). Le nombre total de processus en jeu est donné par MPI_Comm_size. A la fin du code, MPI_Finalize termine l'exécution en environnement MPI. De façon générale, une erreur arrête tous les processus en jeu. Toutefois, le programmeur peut gérer et personnaliser les erreurs au niveau de chaque processus ou globalement. Une routine MPI qui se termine avec succès retourne le code MPI_SUCCESS.

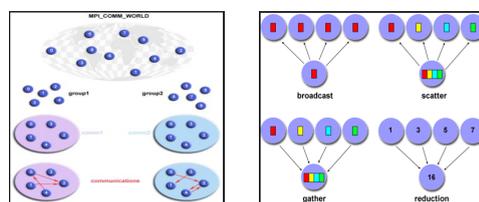


FIGURE 1.7 – Groupes et communicateur (a) - MPI - Opérations collectives (b)

Au niveau de la communication, le transfert de message peut se faire d'un processus vers un autre (point à point). Pour cela, les routines `MPI_SEND` et `MPI_RECV` et leurs variantes permettent respectivement d'envoyer et de recevoir un message. L'adresse du tampon contenant le message à traiter sont passées à ces fonctions avec le type de données ainsi que le nombre d'objets. La destination dans le cas d'un envoi est spécifiée par le rang du processus d'arrivée du message dans le communicateur considéré. Une variable de statut de l'opération permet de connaître si l'opération a réussi ou a échoué. Cet échange peut se faire de manière synchrone ou asynchrone (resp. bloquant ou non bloquant).

Contrairement à une communication point à point, une communication dite collective transfère un message à partir d'un processeur vers un ensemble de processeurs. L'exemple le plus courant est le "broadcast" ou diffusion où un processeur envoie le même message à destination d'un ensemble de processeurs. La figure 1.7-b montre les échanges entre les processus après l'appel à cette opération mais aussi d'autres types de communications collectives. Un processus avec `MPI_Scatter` distribue une structure de données à d'autres processus participants tandis que `MPI_Gather` rassemble des données de plusieurs processus participant en une seule structure. Enfin, les opérations de réduction appliquent une opération (somme, produit, maximum, minimum, etc ...) à un ensemble de processus et retourne le résultat vers le processus appelant. La synchronisation des processus peut être obtenue avec la routine `MPI_Barrier` qui, une fois lancée par un processus, bloque ce dernier jusqu'à ce que tous les processus de son groupe atteigne cette barrière comme un point de rendez-vous.

1.3.4/ SIMULATEUR SIMGRID - SMPI

SimGrid est utilisé pour la simulation et l'étude du comportement d'applications parallèles dans un contexte d'un environnement complexe, hétérogène, distribué et dynamique. Il est conçu sur une simulation basée sur les événements ("event driven") à un niveau d'abstraction et de fonctionnalités répondant aux applications et aux infrastructures [26].

1.4/ MOTIVATIONS

1.5/ CONCLUSION PARTIELLE

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.

2.1.1/ 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'échelle" dont l'objectif principal est d'accroître 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 (équilibre 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 :

$$T_{exec} = T_{calc} + T_{comm} + T_{surcharge} \quad (2.1)$$

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.

$$S(n) = T_{Exec_Seq} / T_{Exec_Par}(n) \quad (2.2)$$

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 .

$$E(n) = S(n)/n = T_{Exec_Seq} / (n \times T_{Exec_Par}(n)) \quad (2.3)$$

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 :

$$S(n) \leq \frac{1}{f + \frac{1-f}{n}} \quad (2.4)$$

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 fonction 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 avons 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.

$$G_{relatif} = T_{Exec_Algo.1} / T_{Exec_Algo.2} \times 100 \quad (2.5)$$

2.1.2/ 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 calculée : 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 :

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

2.1.3/ 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 cœurs 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 cœur. 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 2.1 montre que le temps d'exécution décroît (resp. reste constant) quand le nombre de processeurs augmente en strong mode (resp. en weak mode). De même, le speedup croît avec le nombre de processeur en strong mode tandis qu'il reste constant en weak mode.

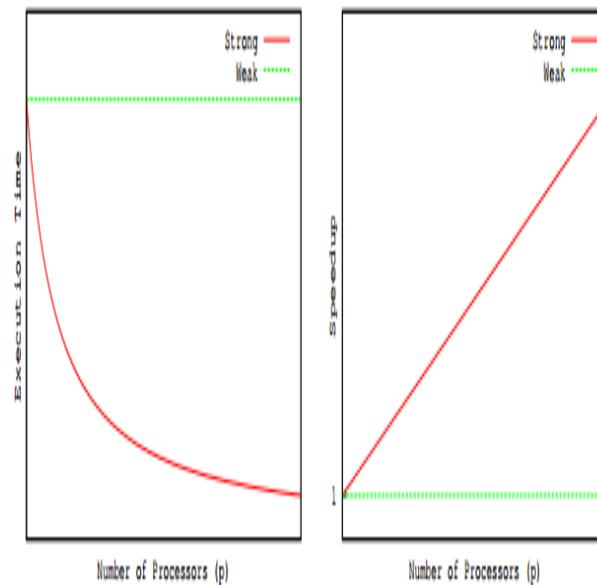


FIGURE 2.1 – Weak vs Strong scaling : Temps d'exécution et Speedup

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,

2.2. PROBLÉMATIQUE SUR LA PRÉDICTION À LARGE ÉCHELLE DE LA PERFORMANCE DES APP

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.

2.2.1/ 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 2.2). 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.

2.2.1.1/ 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

2.2. PROBLÉMATIQUE SUR LA PRÉDICTION À LARGE ÉCHELLE DE LA PERFORMANCE DES APP

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 coeurs ont été pressés au niveau du processeur (« socket ») emmenant un parallélisme au niveau de la puce. La Figure ?? donne un aperçu de l'architecture d'un processeur multi-coeurs.

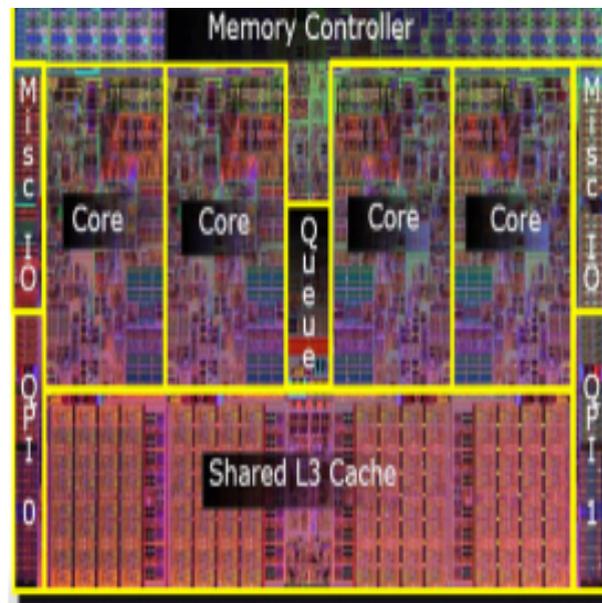


FIGURE 2.2 – Architecture des CPU multicoeurs

La performance d'une telle entité de calcul repose sur la vitesse d'accès des coeurs aux données 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 2.4) : Les unités de calcul accède à la mémoire partagée via un réseau d'interconnexion (généralement, de type Giga-bitEthernet (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 2.3) : 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 2.5) : 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.

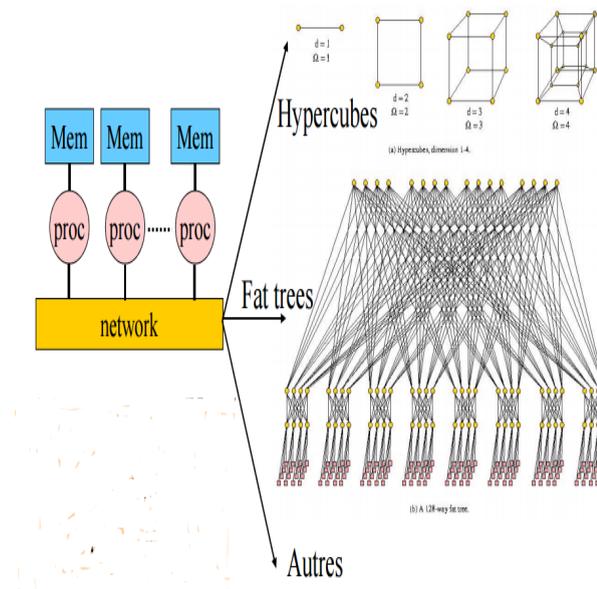


FIGURE 2.3 – Modèle MIMD Distribu 

A titre d'exemple de machines parall les, le site Top500.org [14] classe suivant diff rents crit res les plus performantes. Ainsi, la figure 2.6 montre l' volution de la puissance de calcul mondiale dont le top actuel d veloppe un pic de performance th orique proche de 50 PetaFlops (33 Linpack PetaFlops (renvoi)) avec 3.120.000 coeurs (16 noeuds avec des processeurs de 2x12 coeurs par noeud) et plus de 1.240.000 Gb de m moire (64 Gb par noeud) avec des acc l rateurs 3 × Intel Xeon Phi par noeud. Il s'agit de la machine Tianhe-2 (MilkyWay-2) de la National Super Computer Center   Guangzhou en Chine [15]. A la tendance actuelle, l'atteinte de l'exaflops n'est pas loin.

Pour arriver   de telles puissances, diverses architectures de processeurs ont vu le jour ces derni res ann es. Outre l'Intel Xeon Phi cit  plus haut, les processeurs bas s sur les circuits int gr s FPGA (Field Programmable Gate Array) montrent une flexibilit  efficace pour s'adapter par configuration au type d'applications   traiter [14]. En effet, cette architecture permet la programmation de la « matrice de blocs logiques » interconnect e par des liaisons toutes aussi programmables. Cette possibilit  de programmation des circuits et des interconnexions entraine aussi la r duction de la consommation d' nergie. Par ailleurs, les unit s GPU (Graphics Processing Unit) sont initialement des co-processeurs produits par AMD et NVIDIA pour des applications   fort rendu graphique, lib rant ainsi la charge au processeur. Par la suite, elles ont  t  compl tement programmables et se sont montr es tr s efficaces pour les algorithmes vectoriels.

2.2.1.2/ FACTEUR : M MOIRE ET STOCKAGE

Les diff rentes architectures de processeurs parall les vues plus haut se trouvent toutes confront es au probl me de chargement de donn es   traiter en m moire. Ainsi, elles se sont dot es de contr leurs de m moire incorpor s mais aussi divers niveaux de caches pour faire face   cette diff rence de vitesse de traitement entre les processeurs et les m moires dynamiques. Par exemple, les machines SIMD utilisent des registres de com-

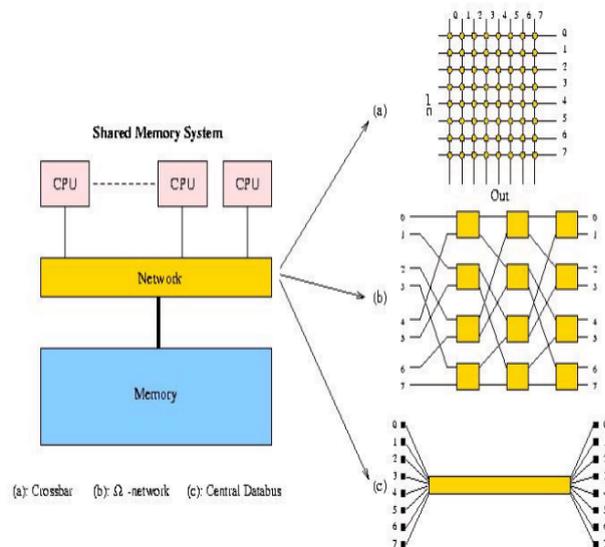


FIGURE 2.4 – Modèle MIMD partagé

munication internes pour communiquer avec les autres CPUs. Pour les machines de type MIMD où différentes tâches sont exécutées par chaque processeur à un instant 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 2.7). 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 2.8). 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 2.9). 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

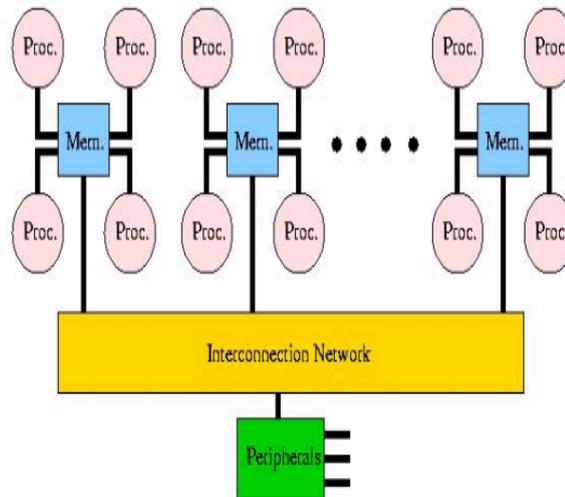


FIGURE 2.5 – Modèle MIMD hybride

au processeur.

Malgré que dans le cadre de nos travaux, nous n'avons 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 échelle 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 adaptateurs 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 requises 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 accroître 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

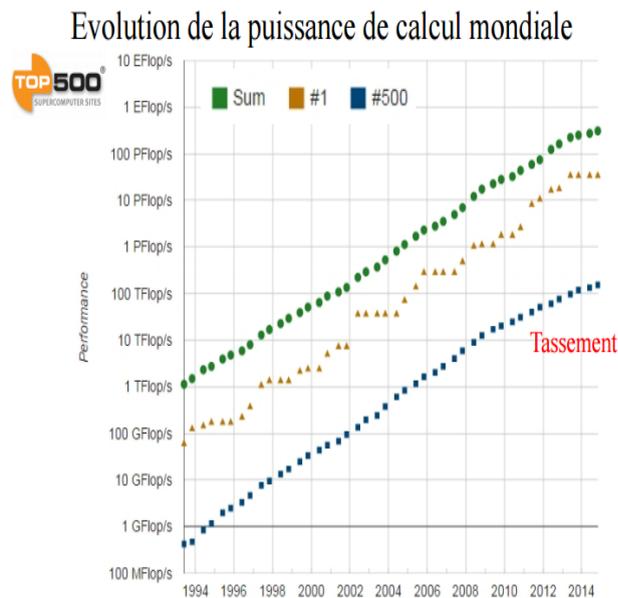


FIGURE 2.6 – Evolution de la puissance de calcul mondiale

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 noeud du cluster peut accéder au serveur parallèle Lustre avec un débit de 650 Mb/s.

La mise en oeuvre 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

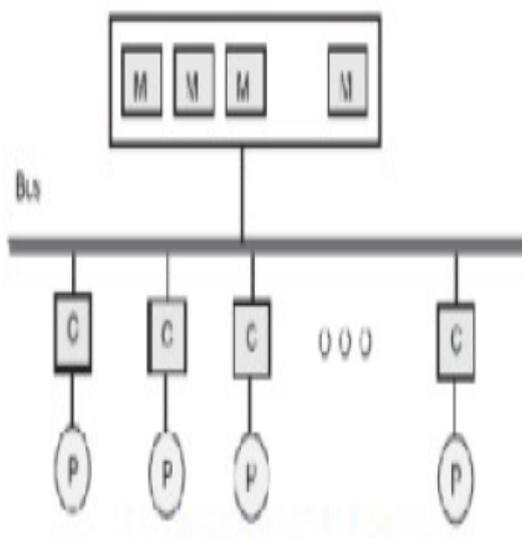


FIGURE 2.7 – Mémoire MIMD : Architecture UMA

« 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 « noeuds de calcul » et les « noeuds 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 persistant » ayant une large capacité vers le système de fichier local avant traitement.

2.2.1.3/ 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'application. 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.

2.2.2/ 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

2.2. PROBLÉMATIQUE SUR LA PRÉDICTION À LARGE ÉCHELLE DE LA PERFORMANCE DES APP

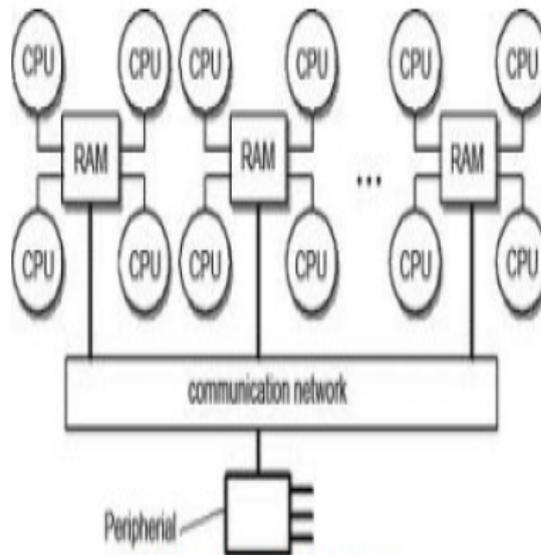


FIGURE 2.8 – Mémoire MIMD : Architecture NUMA

vont influencer la performance globale en impactant le temps de calcul et le temps de communication des données entre les unités de calcul.

2.2.2.1/ 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.

2.2.2.2/ 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 ($\eta_{Parallel}$) comme étant le produit des trois facteurs fondamentaux "normalisés" suivants dont chaque facteur est quantifié par une valeur entre 0 et 1 :

$$\eta_{Parallel} = LB \times Ser \times Trf \quad (2.6)$$

Où :

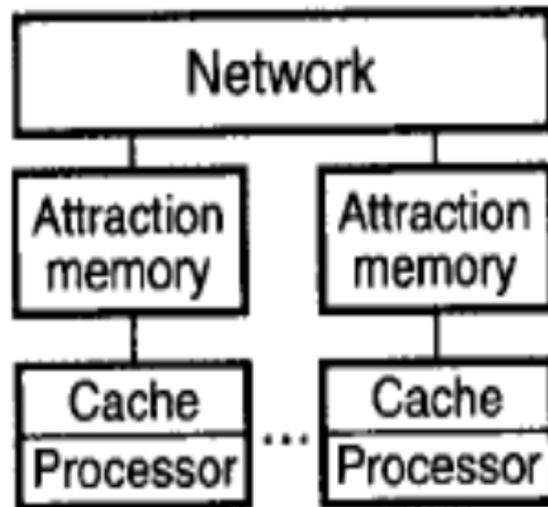


FIGURE 2.9 – Mémoire MIMD : Architecture COMA

- 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 :

$$LB = \left[\sum_{k=1}^p \text{eff}_k / p \right] / \max(\text{eff}_k) \quad (2.7)$$

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, $\text{ideal}(\text{eff}_i)$ est l'efficacité du processeur i sans le temps de communication.

$$Ser = \max(\text{ideal}(\text{eff}_i)) \quad (2.8)$$

- L'efficacité du « transfert » de données : La montée en charge de la taille du problème impacterait 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 :

$$Trf = \max(T_i/T)/Ser \quad (2.9)$$

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 parallélisation 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 parallélisation.

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 verrouillage 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 D'ANALYSE DE PERFORMANCE DES APPLICATIONS PARALLÈLES

2.3.1/ GÉNÉRALITÉS ET OBJECTIFS

L'analyse de la performance des applications parallèles est largement utilisée et même recommandée lors de l'écriture et la mise au point du programme. En effet, pour déterminer et estimer le coût de l'exécution du code, il est d'usage de procéder à l'analyse de la performance dans le but d'optimiser le programme parallèle afin de trouver la meilleure performance en termes de coûts (réduction du temps d'exécution, efficacité de l'utilisation des ressources, ...).

Cette opération consiste surtout à détecter les "régions" et "hotspots" qui correspondent aux parties du code les plus consommatrices de ressources (CPU, mémoire) en particulier celles qui consomment le plus de temps de calcul ou de communication. Elle permet aussi de localiser les éventuels goulots d'étranglement lors de l'exécution du code. Les résultats de cette analyse permet de guider le développeur sur ses actions pour améliorer le code par la réécriture de certaines parties du code par exemple ou de procéder à un meilleur découpage du problème pour une meilleure répartition des charges et l'utilisation des mémoires ou encore par la modification de l'algorithme pour permettre une parallélisation plus poussée. Plusieurs outils existent avec différentes approches pour effectuer cette analyse. La section suivante montre que le modèle de performance établi lors de cette analyse permet aussi d'anticiper sur la prédiction de la performance de l'application parallèle avec la montée en charge [21]. En effet, l'analyse de la performance d'un code peut être utilisée pour prédire le comportement du programme soit d'une part sur un environnement de machines déterminé (benchmarking) soit d'autre part, avec une taille de problème plus importante.

2.3.2/ APPROCHES ET MÉTHODOLOGIE

Dans le domaine du calcul parallèle, l'analyse du code d'une application suit les trois étapes suivantes [21,22] :

- L'acquisition et la collecte des données
- L'enregistrement des données collectées
- La représentation des résultats de l'analyse

Les deux derniers points sont regroupés sous le nom générique de "profiling" ou de "tracing" selon le modèle adopté de l'acquisition des données. La figure 2.10 montre ces trois couches de l'analyse de performance et décrit les différentes techniques utilisées pour cette analyse. Les flèches tracées sur la figure montrent les combinaisons possibles entre les techniques présentées. D'ailleurs, dans la pratique, d'autres combinaisons peuvent être expérimentées pour atteindre les objectifs fixés.

Cette approche à trois étapes commence par la collecte des données sur la performance du code qui consiste à deux techniques les plus utilisées à savoir le "sampling" (ou "l'échantillonnage") et "l'instrumentation basée sur les événements".

- Le "sampling" ou "l'échantillonnage" capture les données décrivant l'état du code lors de l'exécution du programme à chaque instant défini par la fréquence de l'échantillonnage. IL est réalisé généralement avec la mise en place d'un timer qui déclenche la collecte des données selon une période définie. Ces dernières

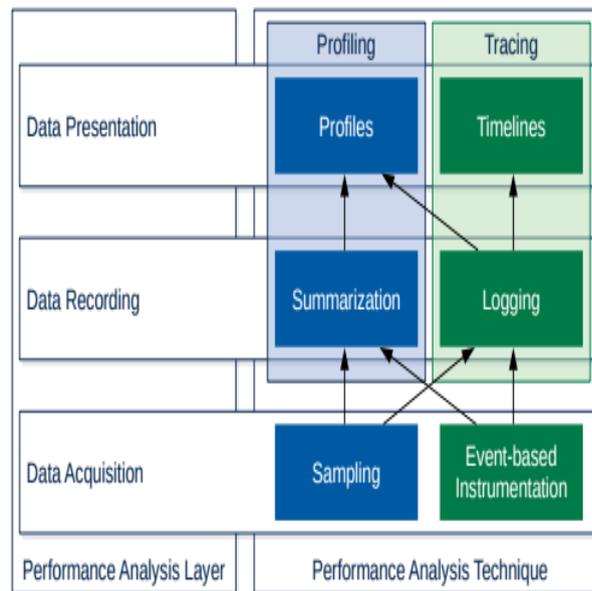


FIGURE 2.10 – Classification des techniques d'analyse de la performance

se rapportent sur les statistiques relatives aux appels de fonctions ("call-path" des fonctions) mais aussi sur les compteurs matériels [22]. Ainsi, il est d'usage de collecter le temps d'exécution d'une fonction ou combien de fois la fonction a été appelée ou encore de façon plus détaillée, combien de fois une ligne de code est exécutée. Evidemment, l'efficacité de la méthode dépend du taux d'échantillonnage : les informations entre deux points de collecte ne sont pas disponibles pour l'analyse ultérieure. Par contre, la surcharge engendrée par la technique peut être contrôlée par l'utilisateur par un choix adéquat de la fréquence de l'échantillonnage.

L'alternative pour collecter les données de la performance d'une application parallèle se porte sur l'instrumentation basée sur les événements. D'abord, de façon générale, l'instrumentation du code consiste à ajouter manuellement ou automatiquement des instructions supplémentaires à des endroits choisis afin de rapporter à chaque passage des informations spécifiques. A titre d'exemple, on peut positionner un timer au début d'une portion du code et d'arrêter ce timer à la sortie de cette région. On peut ainsi collecter le temps total d'exécution consommé par l'application pour exécuter cette partie du programme. Cette technique est largement utilisée par exemple pour déterminer le temps de communication nécessaire lors d'un appel d'une instruction MPI de transfert ou collective (MPI_send, MPI_receive ou autre MPI_Barrier). Cette modification directe qui nécessite une récompilation du code est aussi appelée "instrumentation au niveau de la source". D'autres techniques utilisant des outils existent telles que les "libraries wrapping" ou la "réécriture du code binaire" [22]. Ces dernières n'ont pas besoin d'une recompilation du code.

- La deuxième étape du processus de la collecte des données en vue d'une future analyse consiste à enregistrer soit en mémoire soit sur un support de stockage externe les données obtenues lors de l'étape précédente. Deux techniques peuvent être exploitées à cette fin. D'abord, le "logging" ou le "tracing" permet d'ajouter

le facteur temps sur les données collectées. Ainsi, avant le stockage, chaque entrée de données est estampillée d'une date de l'événement (au format date - heure). Cette opération peut ajouter un temps de surcharge non négligeable lors de l'exécution.

Afin de réduire cette dernière mais aussi pour optimiser la taille du fichier de trace obtenu, la technique de "summarization" consiste à agréger les données après la collecte et de ne stocker que le minimum d'informations utiles. Ce dernier est généralement appelé le "profile" de l'application [21,22]. Certains détails peuvent être perdus avec cette méthode mais il s'agit ici de faire une balance entre la taille la granularité de l'information et la taille des données stockées.

- La troisième et dernière étape de l'analyse de la performance concerne la visualisation des données collectées en vue de l'analyse proprement dite et des décisions à prendre pour améliorer et optimiser l'exécution de l'application. Dans la même ligne de l'étape précédente, soient les données sont visualisées "au fil du temps" en suivant l'exécution du code sur les différentes machines de l'environnement parallèle, soient elles sont représentées par un groupement selon un facteur compréhensible par l'analyste (par fonction par exemple), on est en présence d'une technique générant un "timelines" ou un "profile" de l'application respectivement.

Noter que l'approche présentée dans cette section présente les techniques en vue d'optimiser le code de l'application pour un meilleur temps d'exécution en l'occurrence. Ainsi, elle ne prend pas en compte la performance lors de la scalabilité de l'application pour une prédiction du comportement du code lors du passage à l'échelle. Cette partie sera traitée au paragraphe ... Plusieurs outils d'analyse de la performance parallèle utilisant une ou des combinaisons de ces différentes techniques tels que Gprof, PerfExpert, IPM, TAU, PAPI, HPCToolkit, SCala [...] sont largement utilisés. La prochaine section donne plus de détails sur certains de ces produits.

2.3.3/ QUELQUES OUTILS D'ANALYSE DE PERFORMANCE

- IPM

TAU a été conçu à l'Université d'Oregon comme un outil open source d'évaluation de performance [24]. Il intègre le profiling et le tracing constituant une plateforme complète couvrant les trois étapes de l'analyse d'une application parallèle. L'instrumentation du code peut être effectuée d'une façon complètement automatique avec un package fourni ("PDT - Program Database Toolkit - for routines") collectant toutes les informations sur les régions et hotspots du code, l'utilisation mémoire, les boucles, les entrées/sorties,... Selon le paramétrage de lancement, TAU peut collecter des informations les plus fines telles que le temps passé à chaque instruction dans une boucle ou le temps passé dans les communications à une étape du programme particulièrement dans les instructions collectives MPI par exemple. Toutes ces données peuvent par la suite être visualisées sous forme graphique (Paraprof 3D browser) pour une analyse fine afin d'optimiser la performance.

- SCALASCA

2.4/ MÉTHODES DE PRÉDICTION DE LA PERFORMANCE DES AP-
PLICATIONS PARALLÈLES

2.5/ CONCLUSION PARTIELLE



PARTIE II - TRAVAUX DE CONTRIBUTIONS, RÉSULTATS ET PERSPECTIVES

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

4

SIMULATION AVEC SIMGRID DE L'EXÉCUTION DES SOLVEURS LINÉAIRES EN MODE SYNCHRONES ET ASYNCHRONES SUR UN ENVIRONNEMENT MULTI-COEURS SIMULÉS

4.1/ PROTOCOLES ET EXPÉRIMENTATIONS

4.2/ RÉSULTATS

4.3/ CONCLUSION PARTIELLE

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

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 Belford-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. *Pawsey 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://cite-seerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.3743&rep=rep1&type=pdf>
- [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/ijcset2012021006.pdf>
- [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>
- [22] T. Ilsche, J. Schuchart, R. Schöne, and Daniel Hackenberg. Combining Instrumentation and Sampling for Trace-based Application Performance Analysis. *Technische Univer-*

sität Dresden, Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany

[23] J.A. Smitha, S.D. Hammond, G.R. Mudalige - J.A. Davis, A.B. Mills, S.D.Jarvis. A New Profiling Tool for Large Scale Parallel Scientific Codes. *Department of Computer Science, University of Warwick, Coventry, UK*

[24] S. Shende - New Features in the TAU Performance System - *ParaTools, Inc and University of Oregon. 2014.*

[25] M. Mollamotalebi¹, R. Maghami¹, A. S. Ismail - "Grid and Cloud Computing Simulation Tools" - *International Journal of Networks and Communications 2013, 3(2) : 45-52 - DOI : 10.5923/j.ijnc.20130302.02*

[26] F. Cappello et al. - Grid'5000 : a large scale and highly reconfigurable Grid experimental testbed - *INRIA, LRI, LIP, IRISA, LORIA, LIFL, LABRI, IMAG*

[27] Grid'5000 - <http://www.grid5000.org>

[28] A. Sulistio, C. Shin Yeo et R. Buyya - Simulation of Parallel and Distributed Systems : A Taxonomy and Survey of Tools Grid Computing and Distributed Systems (GRIDS)- *Laboratory Dept of Computer Science and Software Engineering The University of Melbourne, Australia.*

[29] <http://www.dau.mil/> - Defense Acquisition University (DAU) - Ft Belvoir (VA) - USA.

[30] R. M. Fujimoto - Parallel and Distributed Simulation Systems - *Georgia Institute of Technology - John Wiley & Sons, Inc. - ISBN 0-471-18383-0 - 2000*

[31] MPI : A Message-Passing Interface Standard Version 3.- *University of Tennessee, Knoxville, Tennessee. - 2015*

[32] MPICH : www.mpich.org

[33] OpenMPI : www.openmpi.org

TABLE DES FIGURES

1.1	Découpage d'une matrice tridimensionnelle entre deux clusters formés de 18 processeurs chacun	6
1.2	Décomposition en domaines 1D, 2D et 3D	6
1.3	Modèle de communication synchrone	8
1.4	Modèle de communication asynchrone	9
1.5	Architecture d'une grille de calcul	10
1.6	Grid'5000 : Répartition géographique	11
1.7	Groupes et communicateur (a) - MPI - Opérations collectives (b)	14
2.1	Weak vs Strong scaling : Temps d'exécution et Speedup	20
2.2	Architecture des CPU multicœurs	23
2.3	Modèle MIMD Distribué	24
2.4	Modèle MIMD partagé	25
2.5	Modèle MIMD hybride	26
2.6	Evolution de la puissance de calcul mondiale	27
2.7	Mémoire MIMD : Architecture UMA	28
2.8	Mémoire MIMD : Architecture NUMA	29
2.9	Mémoire MIMD : Architecture COMA	30
2.10	Classification des techniques d'analyse de la performance	33

LISTE DES TABLES

1.1	Quelques outils de simulation pour une grille de calcul	13
-----	---	----

LISTE DES DÉFINITIONS



ANNEXES

A

PREMIER CHAPITRE DES ANNEXES

B

SECOND CHAPITRE DES ANNEXES

Résumé :

Ceci est le résumé en français

Mots-clés : Algorithmes itératifs, Performance, Simulation, Simgrid, Grid Computing

Abstract:

This is the abstract in English

Keywords: Keyword 1, Keyword 2

SPIM