Modelisation et simulation d'applications dynamique pour plateformes Exascale

Steven QUINITO MASNADA

Encadrants : Arnaud LEGRAND and Luka STANISIC

Juin 2015

Résumé Dans le domaine des supercalculateurs, la course à la performance est un point crucial. Actuellement, le calculateur le plus puissant (le TianHe-2) est capable d'effectuer environ 33.86 Peta d'opérations flotantes par secondes. Cependant cette course est freinée par un facteur qui prend désormais d'une importance capitale, le coût énergétique. En effet, reprennons l'exemple du supercalculateur chinois, la consommation du TianHe-2 atteint presque les 18MW et avec la génération exascale la consommation estimée sera entre 20MW et 40MW. Dans l'état des fait, ce n'est pas réalisable et pour pouvoir atteindre l'exaflops, il nécessaire d'optimiser d'autres points que la puissance des puces. Evidemment des optimisations peuvent être faites au niveau matériel afin de réaliser des composants à hautes efficacités énergétiques. On peut également optimiser le rendement en utilisant au mieux les capacités du matériel. Cette optimisation ce fait donc du côté logiciel et pour cela il nous faut envisager un changement de méthode programmation, c'est cette dernière que nous allons étudier. L'objectif de mon stage au sein de l'équipe MESCAL, sous la tutelle d'Arnaud Legrand, est donc de tenter de mesurer le gain d'une telle solution.

Dans cette optique, en nous basant sur les standards de programmation en HPC, nous verrons comment nous pourrions évaluer les performances d'un nouveau paradigme programmation.

F. Author
first address
Tel. : +123-45-678910
Fax : +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

1 Introduction

La majorité des supercalculateurs actuels, comme le montre le site top500 sont des clusters massivement parallèles et souvent de type hétérogènes(CPU-GPU). De ce fait certains standard ce sont imposés.

Il y a tout d'abord la norme MPI (Message Passing Interface), qui est une API de communication basée sur l'envoi et la récéption de message. Elle réputée pour être performante et portable. Elle est de plus haut niveau que les sockets.

Ensuite, il y a l'API OpenMP qui est une interface de multihreading de plus haut niveau de PThread. Elle permet de découper facilement des traitements mais cependant elle ne permet d'avoir de contrôle sur la priorité des threads, comme cela reste à la charge de l'ordonnanceur du noyau.

Enfin, l'API CUDA permet tirer partie de la puissance de calcul des GPU. Pour cela il est nécessaire de spécifier explicitement de ce que l'on veut envoyer aux GPUs et on doit également gérer la synchronisation.

Si l'on veut optimiser le rendement d'une application afin que celle-ci tire partie de toute la puissance disponible, il faut faire en sorte d'occuper au maximum le plus d'unités de calculs possible. Le problème est que l'on se retrouve à devoir utiliser plusieurs paradigme à la fois ce qui complique grandement la programmation.

Généralement, on procède soit en déléguant tous les calculs aux GPUs, et les CPUs sont en idle. Soit on réparti la charge entre les CPUs et les GPUs de manière complètement statique [4]. L'inconvénient est que la mise en pratique est très difficile car il est ardu de trouver un bon équilibrage.

Cependant même si l'on arrive à équilibrer les charges correctemment, on peut avoir des cas où certaines unités de calculs ne sont pas occupées alors qu'elles le pourraient. Cela se produit quand par exemple lorsque certaines unités de calculs attendent la terminaisons de certain traitements alors que d'autres auraient put être effectuer en attendant. Cela est dû au fait que l'exécution soit statique et ce qui induit un idle time artificiel. De plus cette solution n'est pas portable car le découpage des traitements ce fait en fonction de la plateforme cible.

La solution serait donc d'avoir une gestion dynamique des charges. Mais cela s'avère bien plus compliqué, voir impossible à réaliser directement avec ces méthodes de programmation. Alors essayons en une autre.

La librairie StarPU [1] est un système runtime qui permet une répartition des traitements de manière dynamique et opportuniste. Pour ce faire elle introduit un nouveau paradigme basé sur les tâches. StarPU génère un graphe de dépendance permettant d'optimiser l'ordonnancement de ces dernières.

La première version de StarPU a été conçu spécialement pour des architectures hybrides. Une version récente (StarPU MPI) [4] a été réalisée pour bénéficier d'un ordonnancement et d'une exécution qui soit à la fois dynamique et opportuniste dans un contexte distribuée, afin de répartir la charge entre les différents noeuds.

Nous allons donc voir comment évaluer les performances d'applications basés sur StarPU MPI.

Pour cela nous verrons, dans une première partie, les différentes approches pour l'évaluation de performances d'applications en HPC et pourquoi nous avons choisi le simulateur Simgrid. Dans une seconde partie, nous examinerons en détail Simgrid et StarPU ainsi que les différents problèmes que nous avons rencontrés. Après quoi, dans une troisième parte, nous verrons les méthodes employées pour répondre à ces problèmes. Dans une quatrième partie, nous aborderons les modifications apportées à Simgrid afin de pouvoir effectuer les mesures. Ensuite dans une cinquième partie, nous verrons le processus de validation de ces changement. Et pout finir, dans une sixième partie, nous conclurons sur les résultats que nous avons réussit à obtenir.

2 État de l'art

En HPC, il y a trois grandes approches possible pour évaluer les performances d'applications.

2.1 Test sur systèmes réels

Cette approche consiste à lancer la vrai application sur le système réel afin d'effectuer les mesures. Cependant cette méthode peut se révéler très coûteuse et il n'est pas toujours possible d'avoir accès à la plateforme. De plus comme les expérimentations ne peuvent être effectuées sur que sur un petit nombre de plateforme notamment à cause de coût, on ne peut pas vraiment extrapoler les résultats. Dernier point important, nous n'avons pas de contrôle sur les décisions d'ordonnancements, d'une exécution à l'autre on peut avoir des résultats différents ce qui fait que les expériences ne sont pas reproductibles.

2.2 L'approche par rejeu de trace

Cette méthode consiste à exécuter une première fois l'application sur un système réel pour ensuite pour ensuite rejouer la trace post-mortem. Elle est couramment employé dans le contexte d'application MPI mais est ici totalement inadaptés car nous avons à faire à des programmes qui sont non déterministes. En effet, on ne pourra pas connaître les autres actions qu'il était possible d'effectuer plutôt qu'une autre, ni leurs impacts.

2.3 La simulation/émulation

On a d'une part la simulation où l'on crée un faux environnement proche de la réalité et où les actions ne sont pas réellement effectués. Dans notre cas

on simulerait donc la plateforme de même que l'OS. Ainsi, les expérimentations peuvent être effectuées à partir de n'importe quel système, il n'est plus nécessaire d'avoir accès à la plateforme, ce qui rend cette approche peu coûteuse. Par ailleurs il est facile d'extrapoler les résultats car on peut simuler un nombre important de plateformes. Ensuite la simulation permet d'avoir un temps d'exécution plus court qu'avec des tests réels car on n'effectue que certains traitements ce qui nous permet pouvoir effectuer un grand nombre de mesures. Enfin comme la simulation nous permettrait d'avoir un contrôle sur l'ordonnancement, nous pourrions avoir un système déterministe qui nous permettrait d'avoir des expériences qui peuvent être reproduites.

Et on a d'autre part l'émulation où l'on exécuterait en vrai le programme sur le système simulé. Ainsi, seul le runtime de StarPU sera réellement exécuté [5], nous pourrons donc étudier son impact sur le performances dans un contexte MPI.

C'est pour ses divers avantages que nous avons opter pour la simulation / émulation. Le logiciel qui a été choisi est Simgrid [{**?**] Simgrid}, un simulateur de systèmes distribués, de grilles de calculs, de systèmes peer to peer et cloud. De plus StarPU a récemment été porté au-dessus de Simgrid et concilie l'approche simulation / évaluation.

3 Analyse du problème

3.1 Simgrid

La structure de SimGrid est composé de plusieurs APIs. Il y a tout d'abord l'API SIMIX qui permet de simuler la partie OS. C'est elle qui s'occupe notamment de la gestion et de l'ordonnancement des processus et également des mécanismes de synchronisation. Sous SimGrid, les processus sont modélisés par des threads, ce qui signifie que leur espace d'adressage est partagé ce qui nous permet de simuler un environnement à mémoire partagée.

Ensuite, au dessus SIMIX, il y a d'une part l'API MSG. Cette dernière permet à l'utilisateur créer et manipuler des processus de manière simple. C'est cette API qui est généralement utilisé pour la plupart des applications classiques et hybrides.

Et d'autre part, il y a l'API SMPI qui a été développée spécifiquement pour simuler des applications MPI. Actuellement la majeure partie des fonctionnalités de MPI ont été implémentées. La simulation de code MPI est assez compliquée et SimGrid est un des seul simulateur à le permettre. Pour ce faire, on compile l'application que l'on veut tester en remplaçant le mpi.h classique par le mpi.h de Simgrid. Ensuite, à l'édition de liens on remplace le main de l'application par le main de Simgrid. Ce dernier a pour rôle de préparer l'exécution du simulateur en créant la plateforme et en déployant les processus SMPI qui exécuterons chacun le main de l'application MPI. Comme dans le cadre d'applications MPI on est dans un environnement à mémoire distribuée et que sous SimGrid les processus sont modélisés par des threads, afin que ces

derniers aient leurs propre espace mémoire, l'approche suivi par SMPI consiste à privatiser les variables des processus en créant pour chaque processus une nouvelle zone mémoire dans le tas grâce à un mmpant, recopiant le segment données dans celui-ci le segment données et à chaque changement de contexte faire pointer vers la zone correspondant à celle du processus.

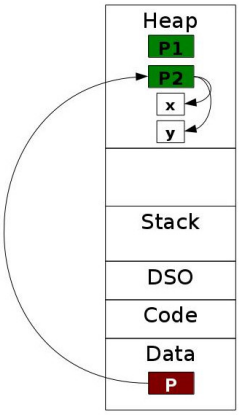

Figure 1 Privatisation du segment données

Enfin, il l'API SURF qui a pour objectif de décrire les caractéristique de la plateforme et de la simuler. On lui fournit donc une modèle de performance qui permettra d'estimer la durée des calculs et des transferts.

3.2 StarPU-MSG : Architecture générale

Comme à la base StarPU visait le modèle CPUs-GPUs, l'API la plus proche était MSG, notamment par rapport à la création de threads et pour la synchronisation. StarPU a donc été modifié pour pouvoir fonctionner au dessus du simulateur SimGrid en se basant sur MSG. Ainsi, l'application (le runtime de StarPU) est réellement exécutée, mais les allocations mémoires des tâches ne sont pas effectuées, les codes de calcul sont simulés et remplacés par un délais de même pour les transferts CUDA.

3.3 StarPU-SMPI :Ce qui coince

Avec StarPU MPI, la modélisation est différente. On est à la fois un environnement à mémoire partagée (entre les CPU et le GPU d'une même machine) et un environnement à mémoire distribuée (entre les différents noeuds). On doit donc permettre d'avoir des modèles différents selon qu'on est entre noeud où à l'intérieur d'un noeud. Il nous faut donc activer la privatisation de

variables entre les noeuds mais également le partage de variables à l'intérieur de chacun noeuds. Pour cela nous avons besoin de faire fonctionner MSG et SMPI ensemble. Or non seulement StarPU est essentiellement basé sur MSG et de plus MSG et SMPI n'ont pas été prévu pour fonctionner ensemble, il nous par ailleurs initialiser correctement la partie MSG et la partie SMPI.

4 Méthodologie

Comme nous travaillons avec Simgrid et StarPU à la fois, nous utilisons un dépôt complexe comprenant les deux et gérer avec l'outils submodule de git. Ce dernier nous permet de gérer des sous dépôt indépendemment, ainsi il est plus aisé de traiter les mises à jours de ces derniers.

Afin de pouvoir retracer le cheminement de mon travail, mais aussi de pouvoir garder le fil d'un jour à l'autre, un cahier de laboratoire est tenu en org-mode et est hébergé sur github. Cela permet également à mon tuteur de stage de savoir chaque jours l'avancement du projet et des difficultés rencontrées.

Comme on l'a vu précédemment il est nécessaire d'apporter quelques modifications au niveau du simulateur. Dans ce but, il a été dans un premier temps nécessaire de consulter la documentation afin de comprendre le fonctionnement et l'architecture de Simgrid. Ensuite il a fallut explorer le code afin de déterminer où et comment apporter les modifications. Pour cela les outils tels que GDB, Valgrind, les etags et CGVG ont été d'une aide précieuse.

5 Contribution

La toute première chose à réaliser afin de pouvoir effectuer des mesures, a été la gestion du partage du segment de données au niveau du simulateur dans un contexte SMPI. Comme la mémoire est partagée au sein d'un noeud, nous avons fait en sorte que les processus d'un même noeud aient leurs segment données en commun. Le principe est le suivant, il y a dans un premier temps, les processus SMPI qui sont créés au lancement de l'application avec leur propre espace de données. Puis ces dernier peuvent à leurs tours créer de nouveau processus. Ceux-ci héritent donc du segment de données du processus qui les a créés. Nous avons donc fait pointés le segment données des processus fils sur celui du père et un échange est effectué au changement de contexte.

Une fois la gestion du partage mise en place, nous avons constaté qu'il y avait un cas que nous n'avions pas pris en compte : celui des librairies dynamiques. Voici comment sont stockés les bibliothèques en mémoire :

En effet, nous n'avons privatisé que le segment données des processus or, les variables globales des librairies dynamiques (DSO sur le schéma ci-dessous) ne se trouvent pas dans le segment données du processus et se retrouvent donc accessible à tous les processus.

La solution qui nous avons employé est d'utiliser donc une version statique de la librairie. Ainsi, les variables globales se retrouvent dans le segment

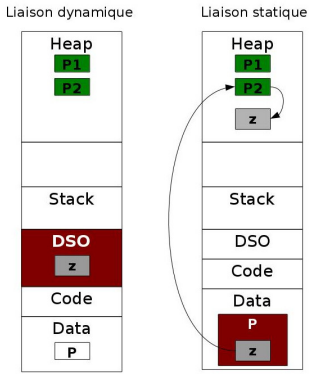

Figure 2 Emplacement en mémoire des bibliothèques

données du processus et ainsi la privatisation et le partage s'effectue grâce au mécanisme précédent. Cependant cette solution comporte une limitation car elle nécessite de changer la chaîne de compilation des applications utilisant StarPU, mais cela sera suffisante pour effectuer nos tests.

6 Validation

6.1 Test simple

Dans le but de tester le bon fonctionnement des modifications apportées, un test illustrant le fonctionnement de StarPU a été fourni et enrichi. Ce dernier permet ainsi d'isoler le problème afin de pouvoir nous concentrer dessus. Ce test, initialise Simgrid et la partie SMPI comme cela est fait du côté de StarPU et fait appel à une bibliothèque dynamique et manipule des variables globales. Ainsi lors de l'exécution de ce test, on doit pouvoir constater que pour des processus appartenant à un même noeuds, les valeurs des variables globales du programme et des bibliothèques dynamiques sont bien identiques. Ce qui après plusieurs correction a été le cas.

6.2 Test de StarPU - SMPI

Comme les résultats du test simples étaient ceux attendu, nous sommes passé à un test utilisant cette fois la vrai bibliothèque StarPU. Cette dernière est fourni avec des exemples de programme MPI notamment d'algèbre linéaire tel que l'algorithme de Cholesky. Nous nous sommes servi de ces dernier afin de valider les modifications. Cependant, malgré les ajouts apportés au test, ce dernier était incomplet et il semble qu'il y a avoir des soucis au niveau de l'initialisation de Simgrid côté StarPU.

7 Conclusion

Pour conclure, nous avons voulu voir s'il était possible de mesurer l'influence d'un runtime dynamique sur les performances d'applications MPI. Parmi les différentes techniques de mesures de performances, nous avons fait le choix de la simulation / émulation car elle nous semble la plus avantageuse, en raison de son coût, mais aussi en terme de scalabilité.

Pour vérifier si cette approche est effectivement possible, nous avons modifié Simgrid afin de pouvoir faire fonctionner StarPU MPI dessus. Nous avons donc mis en place le partage du segment données entre les processus de même noeud et la privatisation entre les processus de noeuds différents.

Malheureusement par manque de temps il n'a pas encore été possible de corriger le problème d'initialisation et donc les mesures prévues n'ont pas encore pu être réalisées. Bien qu'aucune expérimentation n'est pu être faite, les problèmes rencontrés sont plutôt des problèmes d'ordre techniques et ne nous permettent pas d'invalider notre hypothèse.

Afin de pouvoir conclure sur la question, il faudra finir de corriger la phase d'initialisation côté StarPU et également apporter quelques correctifs à Simgrid. Ensuite nous pourrons effectuer les simulations et les mesures. Pour ce faire les mesures seront faites sur le logiciel Chameleon (un solveur d'algèbre linéaire basé sur StarPU). Enfin, dans le but de valider le résultat des expérimentations, un test grandeur nature sera fait sur Grid5000.

Acknowledgments

Je souhaite remercier...

Références

1. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation : Practice and Experience*, vol. 23, pp. 187–198, Feb. 2011.
2. P. Bedaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, and B. Videau, "Toward Better Simulation of MPI Applications on Ethernet/TCP Networks," *Benchmarking and Simulation of High Performance Computer Systems*, Nov. 2013.
3. H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, pp. 2899–2917, 2014.
4. C. Augonnet, O. Aumage, N. Furmento, S. Thibault, and R. Namyst, "StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators," 2014.
5. L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," *Concurrency and Computation : Practice and Experience*, 2015.