Vers un modèle de composants supportant l'ordonnancement de tâches pour le calcul de haute performance

Jérôme Richard

Compas'2015 : Parallélisme / Architecture / Système
Lille, France, du 30 juin au 3 juillet 2015

Vers un modèle de composants supportant l'ordonnancement de tâches pour le calcul de haute performance

Jérôme Richard

LIP, ENS Lyon, 46 Allée d'Italie - 69364 Lyon - France,
Équipe INRIA-CNRS-ENSL-LyonI Avalon
Mél : jerome.richard@inria.fr

Résumé
Les applications de haute performance ont une durée de vie souvent plus grande que celle des plate-formes sur lesquelles elles reposent. L'adaptation de ces applications à différentes plate-formes est un processus nécessaire, long et coûteux. Les composants logiciels offrent de nombreux avantages de génie logiciel simplifiant l'adaptation des applications. Parallèlement, on souhaiterait garder de bonnes performances à travers les adaptations. Les modèles d'ordonnancement de graphes de tâches permettent de tirer parti efficacement des architectures hétérogènes tout en apportant des performances portables. Cet article propose et évalue un modèle de composants avec ordonnancement de tâches visant à profiter des avantages des deux approches sur des SMP. Les résultats montrent que le modèle proposé dispose d'avantages provenant des approches à composants (séparation des préoccupations) et des approches à tâches (équilibrage de charge).

Mots-clés : Composants, tâches, ordonnancement, HPC, SMP

1. Introduction

La durée de vie des applications dédiées au calcul de haute performance (HPC) peut atteindre plusieurs décennies alors que celle d'un super-calculateur est de l'ordre de quelques années. Les applications doivent donc être adaptées à plusieurs architectures matérielles. Chaque adaptation à un nouveau matériel peut nécessiter des transformations profondes, voire même le changement d'algorithmes.

Les modèles de composants [13] proposent d'assembler des boîtes noires (composants) disposant de points d'interactions (interfaces) afin de former une application. Ils permettent d'exprimer, de manipuler et de réutiliser simplement la structure des applications rendant leur adaptation plus simple à travers différentes plate-formes. Ils facilitent la séparation des préoccupations et permettent la réutilisation de parties logicielles réduisant ainsi les coûts de développement.

De plus, pour augmenter la précision des calculs des applications scientifiques tout en limitant l'impact sur la mémoire et le temps de calcul, de plus en plus d'applications ont recours à des structures irrégulières. C'est par exemple le cas des applications effectuant un raffinement

adaptatif de maillages (AMR) ou encore des calculs sur des matrices creuses. Ces structures irrégulières introduisent des problèmes d'équilibrage de charge dynamique (puisque leur évolution est souvent difficilement prévisible). Cela conduit à une sous-utilisation des ressources impactant les performances.

Les modèles d'ordonnancement dynamique de tâches [1] proposent de décomposer les calculs des applications en tâches interdépendantes qui sont ensuite judicieusement ordonnancées sur un ensemble de ressources. Ils ont pour but de permettre une meilleure utilisation des ressources pour les applications complexes et/ou irrégulières, notamment en équilibrant la charge entre les ressources de calcul. Certains modèles permettent également de tirer parti d'architectures hétérogènes.

Les applications scientifiques auraient beaucoup à gagner en utilisant un modèle combinant les avantages des approches à base de composants et des approches à base de tâches, autant au niveau de l'aspect génie logiciel que de la portabilité des performances. Cependant, il n'existe actuellement pas de modèle conciliant tous ces aspects. Cet article propose un modèle préliminaire dont les objectifs sont de disposer des avantages des composants (séparation des préoccupations, réutilisation, etc.), des avantages des ordonnanceurs de graphes de tâches (expression simple du parallélisme asymétrique, équilibrage de charge, etc.) et de passer à l'échelle. Ce premier modèle cible uniquement les architectures constituées d'un nœud de multiples processeurs identiques accédant à une unique mémoire partagée (SMP). Il ne prend donc pas en considération l'impact des données et de leur transfert ce qui permet ainsi de simplifier grandement le modèle.

La section 2 analyse les approches existantes à base de composants et celles à base d'ordonnancement de graphes de tâches dans un contexte de haute performance et explicite les enjeux qui nous ont menés à l'élaboration d'un modèle voulant unifier les deux. La section 3 présente en détail le modèle proposé. La section 4 évalue le modèle en terme de performances, de réutilisation et de facilité d'adaptation. Enfin, la section 5 conclut cet article et fournit quelques perspectives.

2. Analyse et travaux connexes

Dans cette section, nous présentons les principes communs aux modèles de composants, ainsi que quelques modèles existants. Puis nous nous intéressons aux modèles d'ordonnancement de graphes de tâches. Enfin nous nous focaliserons sur les approches existantes se rapprochant le plus des objectifs du modèle visé afin de voir en quoi elles répondent à nos besoins.

Composants logiciels

Les modèles de composants facilitent le développement d'applications en les rendant plus modulaires. Ils aident à exprimer et à manipuler leur structure. Cette approche propose de construire une application en assemblant des unités logicielles (boîtes noires), nommées composants, déployables et dotées de points d'interactions bien définis, nommés interfaces (ou ports). Afin de former une application, les composants sont instanciés, puis leurs interfaces sont connectées. Un modèle de composants est un modèle de programmation qui définit spécifiquement ce que sont les composants et comment les composer.

Très peu de modèles de composants fournissent des performances suffisantes pour les applications scientifiques parallèles de haute performance. Parmi eux on peut citer CCA [4] et L^2C [5]. Dans ces modèles de composants HPC, les composants disposent d'interfaces utilisant (*uses*)

ou fournissant (*provides*) des services. Les interfaces *uses* des instances de composant sont ensuite connectées aux interfaces *provides* via des connexions lors de la phase de composition (assemblage).

CCA [4] (Common Component Architecture) propose des composants parallèles et des connexions locales (*uses/provides*) inter-langages. Cependant, on attend d'un modèle de composants qu'il expose toutes les interactions possibles entre les composants dans l'assemblage. Or ce n'est pas le cas des communications inter-processus, tel que les communications MPI.

L^2C [5] (Low Level Component) est un modèle minimaliste sans surcoût à l'exécution. Initialement développé dans l'équipe Avalon, ce modèle propose des composants, des connexions locales (*uses/provides*), mais aussi des connexions MPI (partage d'un communicateur MPI par les composants) et des connexions CORBA.

Ces deux modèles de composants dédiés au calcul haute performance permettent de décrire que partiellement l'utilisation des ressources d'une application au cours du temps. Chaque composant est libre d'utiliser les ressources à disposition comme il le souhaite. Ainsi, les composants risquent de monopoliser des ressources inutilement (cœurs CPU, cœurs GPU, etc.), ou à contrario, de partager inefficacement les ressources. La gestion des ressources au niveau de l'assemblage est déléguée au développeur qui doit alors mettre en place lui-même des mécanismes pour éviter une sous-utilisation des ressources. Cela peut être évité via des mécanismes d'ordonnancement globaux à l'assemblage.

Modèles d'ordonnancement de graphes de tâches

Les modèles d'ordonnancement de graphes de tâches permettent d'associer des ressources aux tâches. Il devient ainsi possible pour un ordonnanceur de fixer les tâches sur les ressources à l'exécution afin de maximiser leur utilisation à travers le temps en prenant en compte les dépendances du graphe.

Les modèles d'ordonnancement de graphes de tâche permettent de s'abstraire des détails de bas niveau concernant l'utilisation de threads et d'exprimer simplement les cas de parallélisme irrégulier tout en exploitant pleinement les architectures matérielles. Un sous ensemble restreint d'approches existantes prend en considération les problématiques liées au calcul haute performance. Parmi ces approches, on peut citer StarPU [1], PaRSEC [6], StarSs [12] et OmpSS [10]. Nous allons nous concentrer uniquement sur quelques-unes d'entre elles.

StarPU est un support exécutif fournissant des stratégies d'ordonnancement portables et efficaces sélectionnées de manière transparente à l'exécution. Il utilise un gestionnaire mémoire distribuée virtuellement partagée gérant les données des tâches. Il propose de créer des types de tâches appelées *codelets* avec plusieurs implémentations associées à des types de ressources et choisit ensuite à l'exécution la meilleure implémentation, l'ordre d'exécution et la ressource de calcul sur laquelle l'exécuter afin de maximiser les performances.

PaRSEC est un ordonnanceur de graphes de tâches qui contrairement à StarPU ne stocke pas une représentation complète du graphe de tâches à l'exécution permettant de gérer des millions de tâches simultanément sans impacter le temps de calcul et réduisant ainsi l'empreinte mémoire. La description du graphe de tâches se fait dans un langage spécifique. PaRSEC utilise un compilateur qui analyse les dépendances de façon statique, avant l'exécution. L'aspect distribué concerne uniquement l'équilibrage de charge à l'intérieur d'un nœud : un cœur d'un nœud peut voler du travail à un autre cœur d'un nœud.

Bien que les modèles de tâches fournissent un apport considérable en terme de performances

et portabilité des performances tout en limitant la complexité de programmation, ils ne permettent pas d'adapter simplement la structure des applications ou même de définir des dépendances spatiales (connexions *uses/provides*) entre différentes parties logicielles, que les modèles de composants sont capables d'offrir.

Les modèles de composants et les modèles d'ordonnancement de graphes de tâches ne répondent pas seuls aux objectifs fixés : chacun dispose d'avantages et de défauts que l'autre approche pourrait aider à résoudre. Nous allons donc nous concentrer sur les modèles qui se rapproche le plus de nos objectifs, à savoir ceux qui proposent des avantages issus simultanément des modèle de composants et des modèles de tâches.

Discussion sur les approches alternatives au modèle cible
Les objets actifs [11] [9] constituent une approche intéressante découplant l'exécution de méthodes de l'invocation de méthodes sur des objets résidant chacun dans un thread de contrôle. Ils simplifient la mise en œuvre de systèmes concurrents et proposent des mécanismes d'appels de méthodes asynchrones et d'ordonnancement de requêtes. Bien qu'ils offrent certains avantages similaires aux approches basées sur des tâches comme un équilibrage de charge dynamique, ces approches n'offrent pas un moyen de définir des dépendances entre les requêtes réduisant ainsi les possibilités de l'ordonnanceur. De plus, l'ordonnanceur dispose d'une vision locale des données empêchant leur réutilisation au fil du temps car il n'a pas connaissance des flots de données entre les requêtes.

Le projet Peppher [3] fait partie des travaux existants s'approchant le plus du modèle visé. Peppher est un framework visant à obtenir des performances portables efficacement sur des architectures hétérogènes. Basé sur StarPU, ce framework combine des méthodes d'ordonnancement statique et dynamique, des techniques de transformation et de compilation, un modèle de composants minimaliste (bibliothèque), un gestionnaire de ressources à l'exécution et met en place des mécanismes de retour sur les performances. Malheureusement, les composants du framework ne permettent pas de définir de dépendances spatiales (*e.g.* interfaces *uses/provides*) et servent principalement à associer un ensemble de fonctions à des métadonnées pour améliorer l'ordonnancement.

GCM [2] (Grid Component Model) est un modèle de composant basé sur Fractal [8] qui l'étend pour cibler les application sur grilles de calcul. GCM apporte de nombreuses fonctionnalités telles que la composition hiérarchique, la réflexivité, la reconfiguration, des ports asynchrones, des communications collectives... Le support des ports asynchrones repose sur des objets actifs qui ne permettent pas de décrire les dépendances temporelles entre les opérations. De plus GCM introduit un surcoût trop élevé à l'exécution pour les applications de haute performance [5].

STCM [7] (Spatio-Temporal Component Model) résulte de l'unification d'un modèle de composants avec un modèle de workflow. Ce modèle fournit aux applications des unités composant-tâches (issues de la fusion d'un composant et d'une tâche) pouvant être composées spatialement (via des connexions *uses/provides*) et temporellement (via des flots de données). Malheureusement, le modèle ne supporte pas l'hétérogénéité et explicite le parallélisme dans l'assemblage empêchant l'ordonnancement des composant-tâches et donc un équilibrage de charge dynamique. De plus, le modèle, n'étant pas spécifiquement dédié à la haute performance, ne propose pas d'interfaces permettant d'exposer les communications MPI.

2.1. Discussion

Les modèles de composants fournissent de nombreux avantages d'ingénierie logicielle tels que la réutilisation de parties applicatives et la séparation les préoccupations, mais ne proposent pas d'exprimer simplement les cas de parallélisme asymétrique tout en exploitant pleinement les architectures hétérogènes que permettent les modèles d'ordonnancement de graphes de tâches. Réciproquement, ces modèles de tâches ne permettent pas d'exprimer les dépendances spatiales des applications ou même de manipuler simplement leur structure. Les deux approches ne répondent pas totalement aux objectifs fixés. Parmi les travaux similaires relatifs au modèle cible, aucun ne répond à tous les objectifs fixés initialement. Il convient donc de mettre en œuvre un tel modèle.

3. Modèle proposé

3.1. Description

Notre modèle est une extension d'une version simplifiée du modèle L^2C [5] pour l'instant restreinte au parallélisme en mémoire partagée (SMP) en y introduisant la notion de tâche issue de StarPU.

Un type de composant L^2C est un ensemble d'interfaces *uses/provides*, d'interfaces permettant de partager des données ainsi qu'un ensemble d'attributs configurables depuis l'assemblage. Ces propriétés sont nommées permettant de les manipuler ensuite. Les types composants sont ensuite instanciés, fixant ainsi la valeurs de leurs attributs, puis assemblés ensemble en connectant leurs interfaces.

Notre modèle étend les types de composants L^2C en spécialisant les interfaces *uses/provides* pour les tâches afin que les instances de composants puissent utiliser des instances tâches créées par d'autres instances de composants. Ces interfaces spécialisées exposent une unique méthode qui vise à créer et renvoyer une instance de tâche.

Définissons tout d'abord les tâches du modèle. Un type de tâche dans notre modèle est une signature de fonction/méthode. Il peut disposer de plusieurs implémentations. Les implémentations des types de tâche utilisés par un composant doivent être des fonctions/méthodes de ce composant. Les implémentations sont ensuite sélectionnées par un ordonnanceur à l'exécution afin de minimiser le temps de complétion de l'application.

Une instance de tâche est l'application d'une fonction/méthode avec des paramètres (fixés lors de sa création). Une instance de tâche est typée. La fonction qu'une instance exécute est l'une des implémentations de son type. La création des instances est dynamique et relayée à un support exécutif (tel que StarPU). Une fois créée, une instance de tâche est soumise à un ordonnanceur global à l'assemblage qui se chargera de l'exécuter en différé dans un autre fils d'exécution. Exécuter une tâche revient à exécuter la fonction/méthode associée (avec ses paramètres). Cela s'apparente à un mécanisme d'appel de fonction/méthode asynchrone qui produirait une instance de tâche en réponse à cette opération.

Les instances de composants du modèle peuvent créer des instances tâches qui sont invisibles des autres instances de composants. Ces instances de composants définissent ensuite des dépendances temporelles entre les instances de tâches auxquelles ils ont accès formant ainsi un graphe de tâches local aux instances de composants. Afin de permettre la création de graphe de tâches à l'échelle de l'assemblage, les types de composants du modèle disposent des interfaces utilisant ou fournissant des instances de tâches. Les interfaces qui utilisent les instances de tâches peuvent ainsi créer des instances de tâches à la demande en effectuant un appel de

méthode sur une interface. On notera que les composants ne peuvent effectuer que des opérations sur les tâches auxquelles ils ont accès empêchant ainsi toute manipulation globale du graphe de tâches finalement construit.

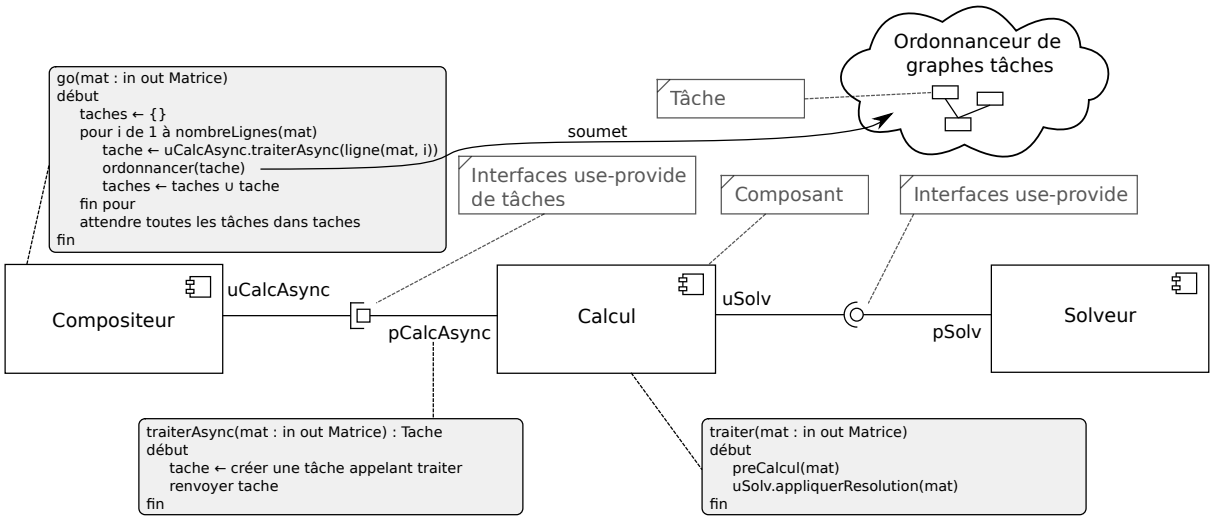

FIGURE 1 – Fonctionnement du modèle sur un assemblage d'exemple

La figure 1 montre un exemple d'assemblage dans lequel l'instance de composant *Compositeur* utilise des tâches fournies par l'instance de composant *Calcul*. L'instance de composant *Compositeur* effectue une boucle parcourant les lignes d'une matrice et soumet des tâches à l'ordonnanceur (via l'interface *pCalcAsync*) traitant chacune une ligne. Lorsque l'instance *Compositeur* fait un appel de méthode sur l'interface *pCalculerAsync*, l'instance *Calcul* crée et renvoie une tâche. La tâche ainsi créée appelle une méthode de l'instance de composant *Calcul* qui appelle à son tour une méthode sur l'interface *pSolv*. L'instance *Compositeur* se contente ici de créer un groupe de tâches et effectue ensuite une synchronisation. Il pourrait éventuellement fixer des dépendances entre les tâches fournies par l'instance *Calcul*.

3.2. Discussion

Nous avons proposé un modèle de composants avec ordonnancement de tâches pour les applications HPC ciblant les architectures SMP. L'idée générale est de construire une application via un assemblage d'instances de composants qui peuvent créer et se partager des tâches afin de produire ensemble un graphe de tâches qui sera ordonnancé efficacement ensuite.

Bien que le modèle vise à permettre la construction d'applications disposant des avantages des composants et de l'ordonnancement de tâches, il ne propose aucun support des données. Il est alors difficile pour l'ordonnanceur de prendre en compte les transferts d'informations entre les tâches. De plus, cela force le développeur à expliciter toutes les dépendances manuellement ce qui peut rapidement devenir fastidieux et source d'erreurs. En plus de résoudre ces problèmes, le support des données conduirait à une prise en charge plus facile des architectures hétérogènes constituant notre prochain défi.

3.3. Implémentation

Le modèle a été implémenté au-dessus de L^{2}C qui offre des composants primitifs et des connexions de bas niveau et au-dessus de StarPU qui fournit une API de bas niveau permettant de manipuler et ordonnancer des tâches de différentes manières.

Afin d'être en accord avec le modèle, il ne faut faire que des opérations locales sur le graphe de tâches. En effet, l'exécution d'opérations d'attente globale de toutes les tâches du graphe aurait par exemple pour effet de rompre le déterminisme d'une application disposant de plusieurs fils d'exécution.

Afin de simplifier la création et le partage des tâches dans les composants L^{2}C, nous avons conçu une API à cet effet. Cette API encapsule les tâches dans des objets et permet d'effectuer des appels de fonctions/méthodes asynchrones simplement. Un appel de fonction/méthode asynchrone crée une tâche qui effectue un appel de fonction ou appel de méthode et renvoie un objet tâche correspondant à la tâche créée et contenant les paramètres nécessaires à l'exécution de la fonction associée. L'API permet entre autres de créer, d'ordonnancer et d'attendre des instances de tâches.

4. Évaluation des performances et des aspects génie logiciel

Cette section évalue le modèle proposé afin de déterminer s'il répond aux objectifs fixés.

4.1. Méthodologie

Nous avons réalisés nos tests de performances sur une machine de 32 cœurs disposant de 4 CPUs Sandy Bridge E5-4620 de 8 cœurs cadencé à 2.20 GHz et de 384 Go de mémoire RAM. Les tests ont été effectués chacun 10 fois et la médiane a été sélectionnée ensuite. Les barres d'erreurs sur les figures correspondent respectivement aux valeurs minimales et maximales.

4.2. Expériences et analyse des résultats

Les performances du modèle sont évaluées en analysant le surcoût qu'il introduit, son passage à l'échelle et l'équilibrage de charge qu'il permet [1].

La figure 2 montre le surcoût d'ordonnancement par tâche introduit par le modèle par rapport au surcoût introduit par StarPU en fonction du nombre de cœurs. L'expérience a été réalisée en mesurant le temps d'ordonnancement de 1 000 000 tâches. Les tâches sont chargées d'exécuter une fonction vide permettant ainsi de mesurer le surcoût du modèle utilisé. On constate que le surcoût du modèle est négligeable face au surcoût de StarPU. Toutefois, on note une forte variation des performances entre 31 et 32 cœurs car StarPU ordonnance les tâches sur n threads et qu'un thread supplémentaire s'occupe de soumettre les tâches à ordonnancer nécessitant ainsi $n + 1$ cœurs.

La figure 3 montre le temps nécessaire pour calculer une discrétisation d'un ensemble de Mandelbrot en fonctions du nombre de cœurs utilisés. Trois implémentations ont été réalisées pour cela : une implémentation utilisant L^{2}C et MPI effectuant une décomposition statique, une autre utilisant StarPU et une dernière implémentée avec le modèle proposé. Les deux dernières versions effectuent une sur-décomposition en tâches. On constate que la version L^{2}C passe moins bien à l'échelle que les deux autres qui sont proches du passage à l'échelle idéal. Étant donné que les calculs sont irréguliers, la version MPI n'équilibre pas bien la charge contrairement aux

1. L'implémentation est disponible à l'adresse suivante : http ://graal.ens-lyon.fr/~jrichard/compas-2015/implementation.tar.gz

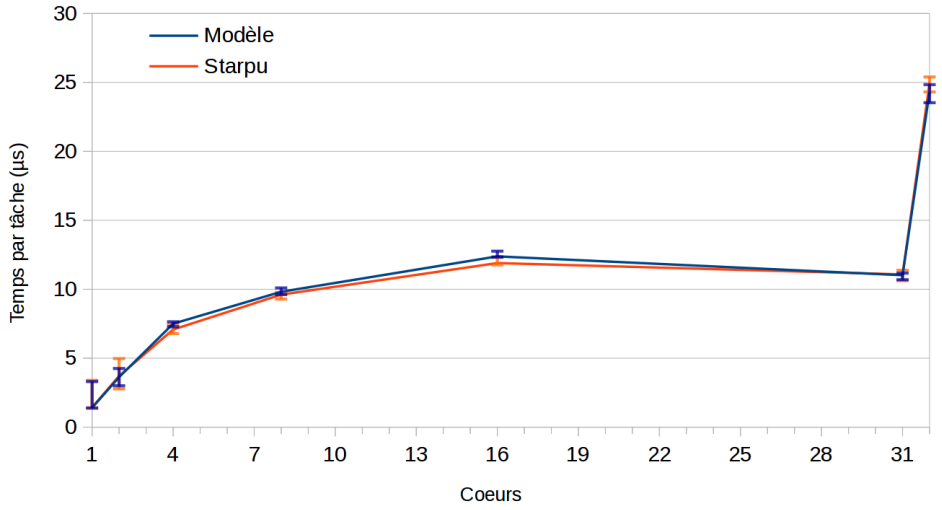

FIGURE 2 – Surcoût du modèle proposé et de StarPU en fonction du nombre de cœurs

deux autres versions dégradant ainsi ses performances. Ainsi, sur 32 cœurs la version MPI est 2.6 fois plus lente que les deux autres, elles même 5% plus lentes que le temps idéal. On en déduit donc que le modèle permet de profiter de l'équilibrage de charge permis par les modèles de tâches.

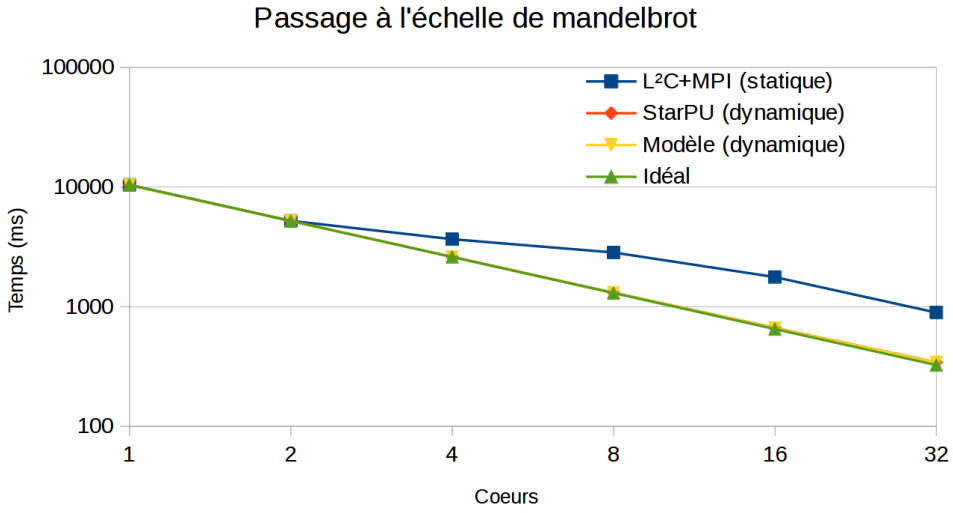

FIGURE 3 – Équilibrage de charge d'un calcul d'un ensemble de Mandelbrot

4.3. Discussion sur les aspects génie logiciel

Les aspects génie logiciel évalués sont la séparation des préoccupations et l'adaptation de l'exemple de la figure 1 de la section 3.

On distingue sur la figure 1 plusieurs parties permettant d'identifier une séparation des préoccupations au niveau de la gestion des tâches. On peut voir que la composition des tâches est

indépendante de leur création et des calculs qu'elles effectuent. En effet, ces trois opérations sont respectivement faites dans trois composants différents et sont indépendantes car leur réalisation nécessite uniquement la connaissance des interfaces *pCalcAsync* et *pSolv*.

La figure 1 montre un exemple dans lequel il est simple d'adapter les calculs réalisés par des tâches. En effet, les tâches créées par l'instance de composant *Calcul* exécutent indirectement une méthode de l'interface *pSolv* via la méthode *traiter*, or l'instance de composant *Solveur* peut être remplacé simplement par un assemblage quelconque disposant d'une instance de composant avec d'une interface du même type que *pSolv*. L'adaptation de cet assemblage permet ainsi d'adapter les calculs réalisés par une tâche. De même, on peut adapter la création et la composition des tâches en remplaçant respectivement les instances de composants *Calcul* et *Compositeur*.

4.4. Résumé de l'évaluation

En termes de performances, les résultats montrent que notre approche introduit un surcoût négligeable par rapport à l'usage de StarPU. Ils montrent aussi que le modèle proposé offre un équilibrage de charge issu des modèles de tâches permettant ainsi de passer très bien à l'échelle sur certaines expériences de nature irrégulière.

En terme de génie logiciel, les résultats montrent que notre modèle profite d'avantages issus des modèles de composants tels que l'adaptation simple des applications ou encore la séparation des préoccupations en séparant explicitement les opérations de création des tâches, de leur composition, des calculs effectués les tâches.

5. Conclusion et perspectives

Afin de concilier les avantages des modèles de composants en terme de génie logiciel et les bénéfices apportés par les modèles d'ordonnancement de tâches en termes d'équilibrage de charge et de portabilité des performances, cet article a proposé et évalué un modèle de composants avec ordonnancement de tâche supportant les architectures SMP.

Les résultats expérimentaux obtenus jusque là montrent que le modèle proposé introduit un surcoût négligeable et combine les avantages issus simultanément des modèles de composants et d'ordonnancement de tâches tels que la séparation des préoccupations ou encore l'équilibrage de charge.

Cet article constitue un premier pas vers une unification d'un modèle de composants et d'un modèle d'ordonnancement de tâches au sein d'un même modèle. Il s'inscrit dans une démarche plus grande visant à offrir aux applications de haute performance les avantages des deux approches.

Les accélérateurs de calcul ainsi que les architectures NUMA, largement adoptés par les supercalculateurs, complexifient le problème en affiliant des données ou des ressources aux unités de traitement. Il convient d'intégrer une certaine localité dans les calculs des applications afin d'utiliser efficacement ces architectures. La prochaine étape de nos travaux cible ainsi l'intégration de la gestion des données dans un modèle unifié afin de prendre en compte ces aspects. En plus des architectures évoquées, nous prévoyons ensuite de supporter des architectures distribués.

Bibliographie

1. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, vol. 23, n2, 2011, pp. 187–198.
2. Baude (F.), Caromel (D.), Dalmasso (C.), Danelutto (M.), Getov (V.), Henrio (L.) et Pérez (C.). – GCM : a grid extension to Fractal for autonomous distributed components. *Annales of Télécommunications*, vol. 64, n1–2, 2009, pp. 5–24.
3. Benkner (S.), Pllana (S.), Träf (J. L.), Tsigas (P.), Dolinsky (U.), Augonnet (C.), Bachmayer (B.), Kessler (C.), Moloney (D.) et Osipov (V.). – PEPPHER : Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, vol. 31, n5, 2011, pp. 28–41.
4. Bernholdt (D. E.), Allan (B. A.), Armstrong (R.), Bertrand (F.), Chiu (K.), Dahlgren (T. L.), Damevski (K.), Elwasif (W. R.), Epperly (T. G.), Govindaraju (M.) et al. – A Component Architecture for High-Performance Scientific Computing. *International Journal of High Performance Computing Applications*, vol. 20, n2, 2006, pp. 163–202.
5. Bigot (J.), Hou (Z.), Pérez (C.) et Pichon (V.). – A low level component model easing performance portability of HPC applications. *Computing*, vol. 96, n12, 2013, pp. 1115–1130.
6. Bosilca (G.), Bouteiller (A.), Danalis (A.), Herault (T.), Lemarinier (P.) et Dongarra (J.). – DAGuE : A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, vol. 38, n1–2, 2012, pp. 37–51.
7. Bouziane (H. L.), Pérez (C.) et Priol (T.). – A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. *In : Euro-Par – Parallel Processing*, éd. par Luque (E.), Margalef (T.) et Benítez (D.), pp. 698–708. – Springer, 2008.
8. Bruneton (E.), Coupaye (T.), Leclercq (M.), Quéma (V.) et Stefani (J.-B.). – The FRACTAL component model and its support in Java. *Software : Practice and Experience*, vol. 36, n11–12, 2006, pp. 1257–1284.
9. Caromel (D.), Delbé (C.), Di Costanzo (A.) et Leyton (M.). – ProActive : an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology*, vol. 12, n1, 2006, pp. 69–77.
10. Duran (A.), Ayguadé (E.), Badia (R. M.), Labarta (J.), Martinell (L.), Martorell (X.) et Planas (J.). – OmpSs : a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, vol. 21, n2, 2011, pp. 173–193.
11. Kale (L. V.) et Krishnan (S.). – CHARM++ : A Portable Concurrent Object Oriented System Based on C++. *ACM SIGPLAN Notices*, vol. 28, n10, 1993, pp. 91–108.
12. Planas (J.), Badia (R. M.), Ayguadé (E.) et Labarta (J.). – Hierarchical Task-Based Programming with StarSs. *International Journal of High Performance Computing Applications*, vol. 23, n 3, 2009, pp. 284–299.
13. Szyperski (C.). – *Component Software : Beyond Object-Oriented Programming*. – Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 2002, 2nd édition.