View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Modulariser les ordonnanceurs de tâches : une approche structurelle

Marc Sergent, Simon Archipoff

▶ **To cite this version:**

Marc Sergent, Simon Archipoff. Modulariser les ordonnanceurs de tâches : une approche structurelle. ComPAS 2014 : conférence en parallélisme, architecture et systèmes, Apr 2014, Neuchâtel, Suisse. hal-00978364v2

HAL Id: hal-00978364

https://hal.inria.fr/hal-00978364v2

Submitted on 14 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modulariser les ordonnanceurs de tâches : une approche structurelle

Marc Sergent,
Simon Archipoff

Inria Bordeaux Sud-Ouest,
200 Avenue de la Vieille Tour
33400 Talence - France
marc.sergent@inria.fr

Résumé
Pour pouvoir exploiter pleinement les architectures contemporaines, une abstraction de la machine appelée support d'exécution se charge de faciliter le travail du programmeur. Les implémentations actuelles des heuristiques d'ordonnancement avancées peuvent présenter des difficultés de passage à l'échelle. À l'inverse, les heuristiques qui passent à l'échelle sont généralement plutôt simples. Nous proposons un modèle d'ordonnanceurs modulaire qui facilite l'exploration de solutions d'ordonnancement. Nous montrons que les performances de notre modèle d'ordonnanceurs sont équivalentes à celles de l'état de l'art tout en offrant des perspectives de passage à l'échelle.

Mots-clés : architectures hétérogènes, support d'exécution, ordonnancement, modularité

---

1. Introduction

De nos jours, les machines se complexifient et deviennent de plus en plus hétérogènes car elles regroupent des types d'unités de calcul de plus en plus variées : GPUs, Cell, manycores, etc. Des solutions ont été trouvées pour exploiter chaque type d'architecture avec par exemple OpenMP, OpenCL ou Cuda, mais aucune d'entre elles n'abstrait la gestion de la mémoire et de l'ordonnancement. De ce besoin a découlé la création d'une couche logicielle dédiée à ces problématiques, nommée support d'exécution.
Un des rôles du support d'exécution est la gestion de l'ordonnancement. Or, il est difficile de mettre en œuvre les algorithmes d'ordonnancement théoriques dans des stratégies d'ordonnancement de supports d'exécution. Il est également difficile de concilier les performances d'une heuristique d'ordonnancement avancée tout en permettant son passage à l'échelle.
Nous proposons donc un modèle d'ordonnanceur totalement modulaire. Nous ne cherchons pas à proposer une nouvelle stratégie d'ordonnancement, ni une méthode pour permettre le choix d'une stratégie plutôt qu'une autre. Nous voulons entamer une réflexion sur la nature même d'un ordonnanceur en proposant une formalisation de ce que peut être sa structure. Les objectifs du modèle que nous proposons sont de permettre le passage à l'échelle des algorithmes d'ordonnancement tout en préservant leurs performances et d'étendre la capacité d'expérimentation de ces algorithmes.

Nous présentons nos travaux comme suit : tout d'abord nous présentons les concepts régissant les ordonnanceurs modulaires. Nous détaillons ensuite la méthode permettant d'implémenter des composants d'ordonnancement. Enfin, nous validons notre modèle en présentant une étude du comportement et des performances des algorithmes d'ordonnancement implémentés selon le modèle que nous proposons.

2. Ordonnancement et scalabilité : deux objectifs difficilement conciliables

L'ordonnancement étant un problème NP-difficile, de nombreuses heuristiques ont été proposées pour ordonnancer de manière performante (i.e la plus proche possible de l'ordonnancement optimal) certaines classes de problèmes, ou sur certaines architectures spécifiques [15]. Mais la littérature n'a pas beaucoup détaillé ce que devrait être la structure d'un ordonnanceur [6]. Ces questions sont cependant connues, notamment au niveau de l'ordonnancement système. Des propositions ont été faites sur la structuration des ordonnanceurs afin de permettre des interactions entre l'ordonnanceur du système et l'utilisateur, notamment afin de pouvoir permettre à l'utilisateur de choisir l'ordonnanceur utilisé [10][13]. Sur cet axe, les travaux qui se rapprochent le plus de ceux présentés dans cet article sont ceux qui concernent Bossa [5][12], un DSL (Domain Specific Language) permettant de programmer des ordonnanceurs pour le système de manière totalement transparente pour l'utilisateur. Ces travaux n'ont cependant été portés qu'au niveau Système d'Exploitation, notamment Linux, pour des threads, mais pas dans des supports d'exécution utilisateur, pour des tâches. La plupart de ceux qui existent [8][11][9][7] n'ont par ailleurs actuellement été conçus que dans le but d'optimiser une classe particulière d'applications, pour lesquelles un ordonnanceur appliquant une heuristique adaptée à ces problèmes est implémenté dans le cœur même du support d'exécution.
Les deux critères qui nous intéressent lors de l'évaluation de l'implémentation d'une heuristique d'ordonnancement sont ses performances et sa scalabilité à grande échelle. Les stratégies d'ordonnancement actuellement implémentées dans les supports d'exécution ne satisfont généralement qu'un seul de ces deux critères : soit l'heuristique est performante mais ne passe que peu à l'échelle, soit l'heuristique est plutôt simple et offre un potentiel de scalabilité important. L'ordonnancement dans un support exécutif est donc difficile et complexe à mettre en œuvre. Le fossé entre la théorie et la pratique dans le domaine des algorithmes d'ordonnancement fait que très peu d'entre eux ont concrètement été implémentés, encore moins dans des supports d'exécution.
Nous présentons un modèle d'ordonnanceur modulaire basé sur la programmation par composants [14] facilement implémentable, qui se veut être un moyen de réconcilier la théorie et la pratique dans le domaine de l'ordonnancement. Il propose aux théoriciens un modèle qui leur permet de s'abstraire des considérations d'implémentation pour se concentrer sur celle du cœur de l'ordonnanceur. Il offre également une abstraction de la structure d'un ordonnanceur. L'objectif de ce modèle est de faciliter l'implémentation d'algorithmes d'ordonnancement satisfaisant les deux critères cités ci-dessus : performance et scalabilité.

3. Présentation du modèle

3.1. Ordonnancement modulaire : présentation

Nous avons basé nos travaux sur le support d'exécution StarPU[3][4]. Ce choix a été motivé par le fait que ce support d'exécution se veut être une plateforme d'expérimentation pour des stratégies d'ordonnancement et veut donc permettre d'implémenter de la manière la plus simple possible des heuristiques d'ordonnancement pourtant complexes. StarPU étant un support

d'exécution basé sur un paradigme de graphe de tâches, à toute tâche manipulée par StarPU sont associées les données de cette tâche ainsi que les dépendances de et vers cette tâche. Pour exécuter ces tâches, StarPU modélise les unités de calcul de la machine sous forme de Workers StarPU. Lorsqu'une tâche est dite prête, i.e quand toutes ses dépendances entrantes sont satisfaites et que ses données sont disponibles, le mécanisme de libération des tâches de StarPU va soumettre la tâche prête à l'ordonnanceur. Les Workers StarPU vont ensuite récupérer le travail à effectuer depuis l'ordonnanceur, puis l'exécuter sur les unités de calcul qui leur sont associées.
Les ordonnanceurs modulaires que nous proposons sont construits à l'aide de composants d'ordonnancement assemblés selon une structure de graphe orienté. Chaque composant effectue une action particulière, comme prioriser les tâches ou décider sur quelle ressource une tâche doit s'exécuter. Cette conception de la structuration des ordonnanceurs facilite leur assemblage et étend la capacité d'expérimentation des algorithmes d'ordonnancement. La figure 1 présente un ordonnanceur modulaire typique.

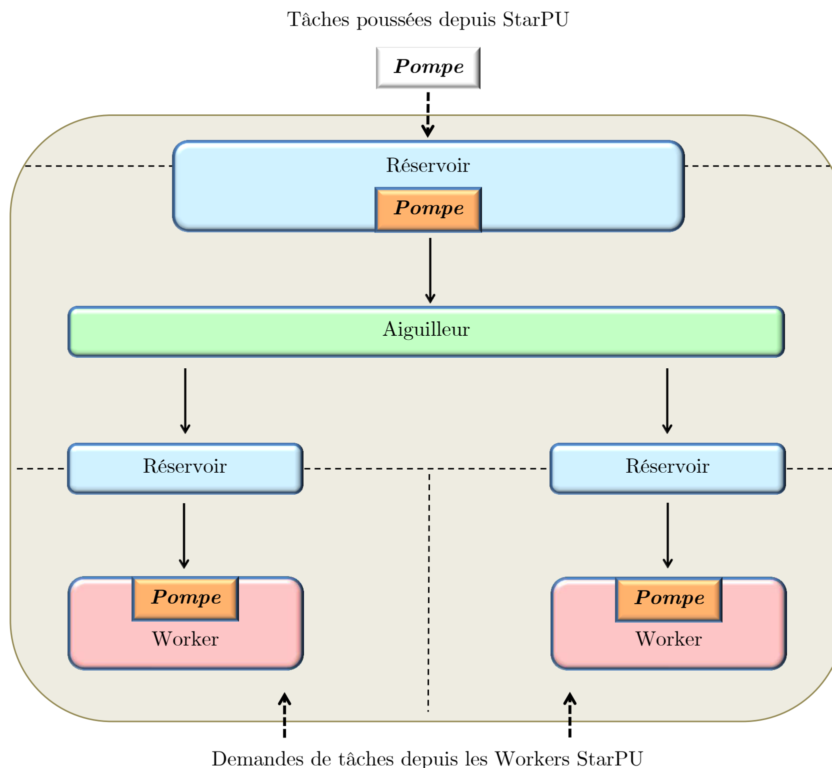

FIGURE 1 – Schéma d'un ordonnanceur modulaire

Les tâches poussées par StarPU dans un ordonnanceur modulaire passent d'un composant à un autre en suivant son graphe orienté, puis sont stockées dans un des réservoirs de l'ordonnanceur. Les pompes servent à faire progresser les tâches entre les différents réservoirs, pour les amener aux réservoirs les plus proches des composants Worker. Lorsqu'un Worker veut récupérer une tâche, le composant Worker qui lui est associé dans l'ordonnanceur tente de

récupérer une tâche depuis ses parents (selon l'orientation du graphe). S'il réussit à récupérer une tâche depuis un réservoir, il la donne au Worker appelant qui peut alors l'exécuter. Sinon, le Worker appelant ne reçoit aucune tâche et c'est au support d'exécution de décider de sa prochaine action.

3.2. Des classes de composants

Un composant d'ordonnancement est un élément logiciel respectant les règles de la programmation par composants : ses données sont encapsulées et il communique avec d'autres composants selon une interface de communication fixe. Il existe plusieurs classes de composants d'ordonnancement :

- Les composants réservoir servent à stocker les tâches, en attendant que des composants en aval des réservoirs les récupèrent. Ils peuvent servir à effectuer du préchargement de données sur les ressources de calcul.
- Les composants d'aiguillage prennent les décisions d'ordonnancement, en choisissant à quel composant fils passer chaque tâche. Les algorithmes d'ordonnancement sont généralement implémentés dans ces composants.
- Les composants à effets de bord, comme leur nom l'indique, appliquent des effets de bord aux tâches qui les traversent. Un exemple est le composant qui sélectionne la meilleure implémentation d'une tâche pour une architecture donnée.
- Les composants Worker sont la représentation des Workers StarPU dans le modèle de l'ordonnanceur modulaire : ils se chargent de tirer des tâches pour les Workers StarPU.

Ces composants peuvent ensuite être assemblés entre eux pour former des stratégies d'ordonnancement, comme celle présentée sur la figure 1. Nous allons maintenant voir quels principes assurent la progression des tâches dans nos ordonnanceurs.

3.3. Principes de l'assemblage de composants

Le graphe de composants est organisé en zones d'ordonnancement, représentées par des pointillés sur la figure 1. Ces zones sont délimitées par les composants réservoirs, chargés d'accumuler les tâches passant par elles. Chaque zone d'ordonnancement doit posséder une pompe permettant de faire progresser les tâches d'une zone à un autre. Il existe deux pompes prédéfinies : les tâches poussées par StarPU dans l'ordonnanceur et la récupération des tâches par les Workers de StarPU. Un assemblage de composants d'ordonnancement est donc un ordonnanceur modulaire s'il assure la présence d'une pompe par zone d'ordonnancement.

Pour mieux comprendre ces principes, nous allons expliquer comment fonctionne un ordonnanceur simple que nous avons implémenté : modular-eager, présenté en figure 2. Lorsqu'une tâche est poussée par StarPU dans ce dernier, elle est stockée dans le composant Fifo C, le premier réservoir, qui joue ici le rôle de fenêtre d'ordonnancement. L'arrivée d'une tâche dans ce composant provoque l'activation de la pompe C qui pousse les tâches du composant Fifo C vers les composants Fifo A et B jusqu'à ce qu'ils soient pleins, leur taille étant un paramètre de l'ordonnanceur qui sera étudié en section 5. Le composant d'aiguillage, ici Eager, décide du composant Fifo vers lequel chaque tâche est poussée. Lorsqu'une tâche est poussée dans une zone d'exécution, le préchargement des données peut commencer. Enfin, lorsque le composant Worker A (resp. B) veut récupérer une tâche pour son Worker, il actionne la pompe A (resp. B) pour tirer une tâche depuis le composant Fifo A. S'il réussit à en obtenir une, il la donne à son Worker.

Lorsqu'un composant Worker veut récupérer une tâche mais que le composant Fifo le plus proche en amont est vide, ce dernier fait une demande de tâches à la fenêtre d'ordonnancement : le composant Fifo C. Celle-ci va alors activer la pompe C pour remplir les composants

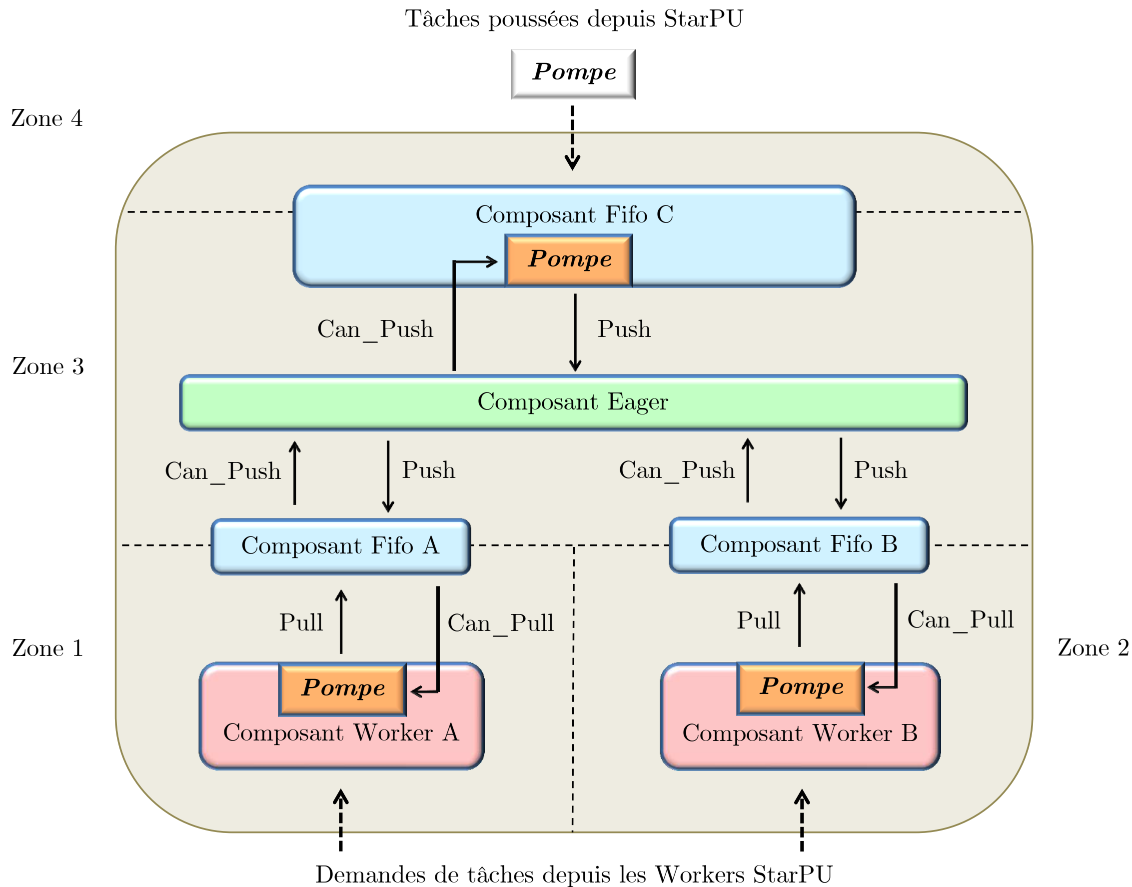

FIGURE 2 – Ordonnanceur modulaire modular-eager

Fifo A et B. Le composant Worker peut ainsi récupérer une tâche.

4. Implémentation d'un composant d'ordonnancement

4.1. Interface et squelette d'un composant d'ordonnancement

Pour pouvoir communiquer entre eux, tous les composants d'ordonnancement doivent respecter une même interface. Elle est composée de quatre fonctions principales :

- `Push` pousse une tâche vers un composant en aval ;
- `Pull` récupère une tâche depuis un composant en amont ;
- `Can_Push` avertit un composant en amont qu'il peut pousser des tâches ;
- `Can_Pull` avertit un composant en aval qu'il peut tirer des tâches.

Prenons en exemple le fonctionnement de la récupération de tâches par les Workers StarPU de la figure 2 : lorsque le composant Worker A veut récupérer une tâche, il appelle la méthode `Pull` du composant Fifo A. Ce dernier prépare une tâche à donner au composant Worker A, puis appelle la méthode `Can_Push` du composant Eager pour demander que des tâches soient poussées vers les composants Fifo A et B. Cet appel remonte jusqu'au composant Fifo C qui

va actionner la pompe C pour pousser des tâches vers les composants Fifo A et B. Par contre, le composant Fifo A ne va pas forcément recevoir de tâches : cette décision revient au composant d'aiguillage, donc à l'algorithme d'ordonnancement, ici Eager. La pompe C continue de pousser des tâches autant qu'elle peut jusqu'à ce que les composants Fifo A et B soient pleins. Enfin, la pile d'appels de fonctions revient jusqu'au composant Fifo A qui va donner la tâche au composant Worker A.

4.2. Instancier un composant d'ordonnancement

Nous fournissons un squelette générique implémentant le fonctionnement de base des composants d'ordonnancement. L'interface de communication du squelette décrit le composant basique comme un simple passeur de tâches. Le squelette n'implémente pas la fonction `Push` de l'interface de communication, qui devra être définie par les composants qui hériteront du squelette.

Pour créer un composant d'ordonnancement, il faut spécialiser le squelette afin d'attribuer une fonctionnalité précise au nouveau composant. Cette instanciation se fait en deux étapes :

- définir une fonction `Push`, qui sera l'« empreinte » du composant d'ordonnancement ;
- redéfinir éventuellement les autres fonctions de l'interface de communication.

Il appartient au programmeur de définir les fonctionnalités qu'il souhaite donner à son composant d'ordonnancement. Par exemple, un composant d'aiguillage définira seulement une fonction `Push` implémentant une heuristique d'ordonnancement. Un composant à effets de bord ne définira également qu'une fonction `Push` qui appliquera l'effet désiré. Un composant réservoir, par contre, implémentera une fonction `Push` permettant de stocker une tâche dans la liste locale du composant et une fonction `Pull` permettant de retirer une tâche de cette même liste.

5. Étude des ordonnanceurs modulaires

5.1. Cadre expérimental

Nous avons utilisé la plateforme Plafrim pour nos expérimentations, et plus particulièrement une des machines du cluster « Mirage ». Son architecture est la suivante :

- CPU : 2 Westmere Intel® Xeon® X5650 @ 2,67 GHz 6 cœurs, 12 en tout ;
- GPU : 3 Nvidia® Tesla® M2070 ;
- RAM : 36 Go.

L'application de test que nous avons choisi est la factorisation de Cholesky, car elle possède les caractéristiques typiques d'un algorithme d'algèbre linéaire dense, qui est un domaine d'application intéressant pour évaluer les performances d'une stratégie d'ordonnancement. Pour nos tests, nous utiliserons la factorisation de Cholesky tuilée simple précision, avec des tuiles de taille fixée après calibration à 960*960.

La factorisation de Cholesky que nous utilisons est tirée de la bibliothèque d'algèbre linéaire Morse[1], dédiée au portage des algorithmes d'algèbre linéaire sur des supports d'exécution. Ce choix est motivé par le fait que cette bibliothèque utilise les noyaux d'algèbre linéaire des bibliothèques connues et reconnues que sont Plasma et Magma[2], ce qui renforce la pertinence des résultats obtenus. Le code de la factorisation en elle-même fait une centaine lignes et consiste principalement en un nid de boucles autour d'appels à des BLAS. Le branchement de ce code au support d'exécution StarPU est fait à l'aide de macros, et prend également de l'ordre d'une centaine de lignes. Une version antérieure à celle utilisée dans nos expériences est disponible dans Magma 1.1.

Cette étude comparera un autre ordonnanceur modulaire que nous avons implémenté, modular-heft, avec l'ordonnanceur DMDAS existant dans StarPU. Ces deux ordonnanceurs, bien qu'étant tous deux basés sur l'algorithme d'ordonnancement Heft[15] (Heterogeneous Earliest Finish Time), diffèrent car DMDAS ne possède pas de fenêtre d'ordonnancement. Il est intéressant de remarquer que la longueur du code de l'ordonnanceur DMDAS est de l'ordre du millier de lignes, alors que chaque composant d'ordonnancement fait quelques centaines de lignes. De plus, la plupart d'entre eux sont réutilisables tels quels pour écrire d'autres ordonnanceurs.

5.2. Paramétrage de l'ordonnanceur modular-heft

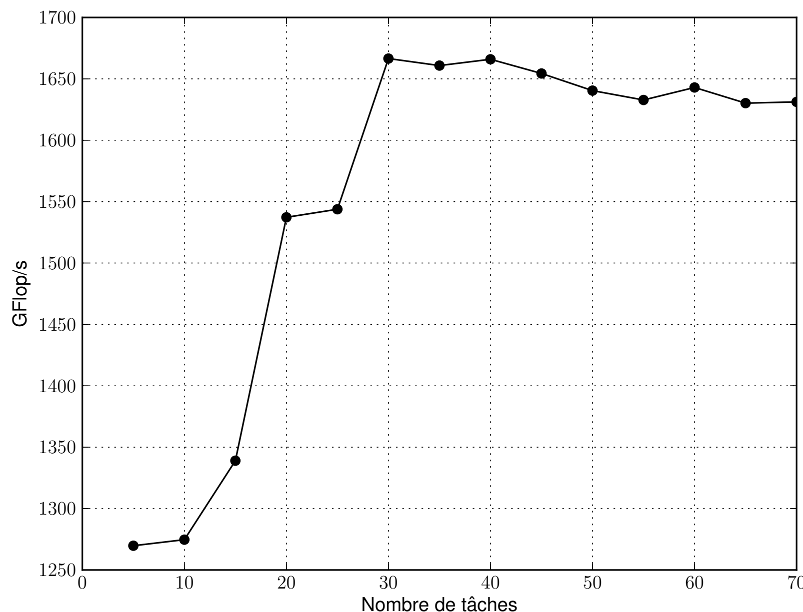

FIGURE 3 – Influence du dimensionnement des réservoirs sur les performances

L'objectif de ce premier test est de mesurer l'influence du dimensionnement des réservoirs associés aux Workers sur les performances de notre ordonnanceur modular-heft, à taille de problème et à algorithme d'ordonnancement fixe. Pour cette expérimentation, la taille de la matrice est donc fixée à 48000*48000 éléments, soit 50*50 blocs.

On observe sur la figure 3 que les performances sont faibles lorsque le dimensionnement des réservoirs est petit (5,10 et 15 tâches), car ils ne peuvent pas stocker assez de tâches pour permettre un préchargement efficace des données ainsi qu'une réelle prise en compte des priorités. Les performances augmentent pour 20 et 25 tâches, car le préchargement devient plus efficace. Un pic de performance est atteint pour 30 tâches, puis les performances diminuent légèrement pour un nombre plus élevé de tâches. Cela s'explique par le fait que les réservoirs ont une taille minimale à avoir pour permettre le préchargement des données tout en prenant correctement en compte les priorités, mais également par le fait que si les réservoirs sont trop gros, les tâches sont ordonnancées trop à l'avance et les priorités ne sont plus correctement respectées.

Pour la suite de nos tests, la taille des réservoirs associés aux Workers de l'ordonnanceur modular-heft sera donc fixée à 30.

5.3. Performances globales

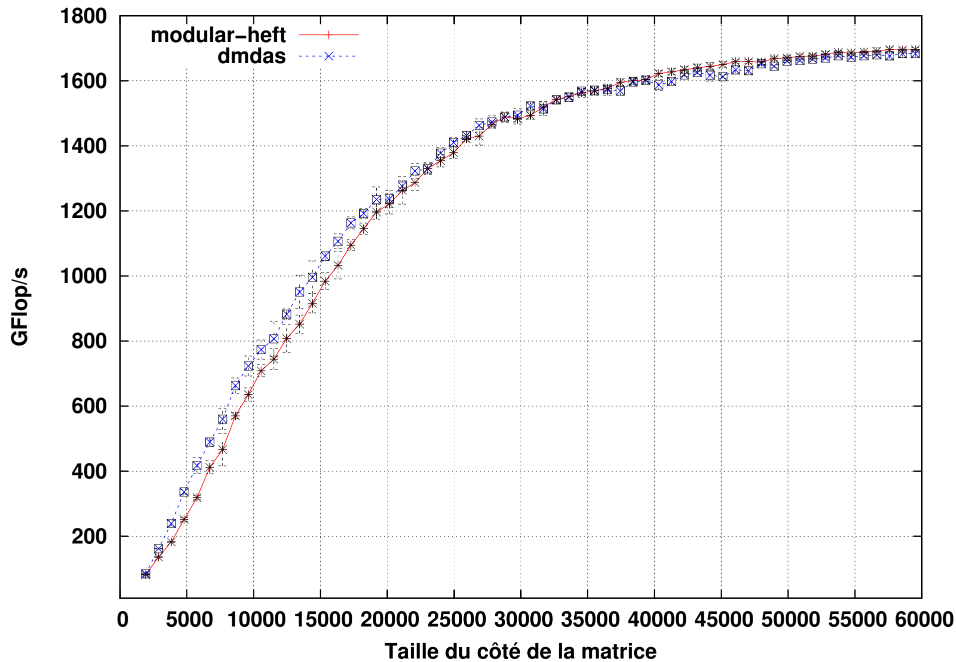

FIGURE 4 – Performances de modular-heft et DMDAS sur la factorisation de Cholesky

L'objectif de ce second test est de mesurer les performances de notre ordonnanceur modular-heft et de les comparer à celles du DMDAS avec priorités existant dans StarPU.
La figure 4 montre les performances obtenues. En régime permanent (pour des tailles de matrice supérieures à 25000*25000 éléments), les deux ordonnanceurs présentent des performances équivalentes, mais notre ordonnanceur perd en performance face à DMDAS pour les petites matrices. Nous expliquons ce phénomène par le surcoût introduit par notre ordonnanceur : les montées/descentes dans le graphe d'ordonnancement au moment de l'activation de la pompe de la fenêtre d'ordonnancement en réponse à un appel à `Can_Push` génèrent un surcoût non négligeable (~100 µs par appel). Nous pensons mettre en place un système inspiré du mécanisme d'hysteresis pour résoudre ce problème : au lieu de demander le remplissage d'un réservoir dès qu'une tâche y est récupérée, il pourrait être intéressant de ne le faire qu'une fois tous les X fois (avec X à déterminer), ou dès que le réservoir descend sous un certain pourcentage de remplissage.

5.4. Comportement des ordonnanceurs

La figure 5 montre l'état des réservoirs associés aux Workers CPU et GPU durant l'exécution de la factorisation de Cholesky tuilée simple précision à taille de problème fixée à 48000*48000 éléments, soit 50*50 blocs. Les schémas présentant le comportement de l'ordonnanceur DMDAS de StarPU montrent qu'il ordonnance toutes les tâches directement vers les réservoirs associés aux Workers, et qu'il charge beaucoup plus les GPUs que les CPUs, avec 380 tâches maximum pour un GPU contre 13 pour un CPU. On peut également remarquer la présence de variations (parfois fortes) dans le nombre de tâches à certains moments, preuve que les priorités ne sont pas toujours prises en compte au plus tôt.

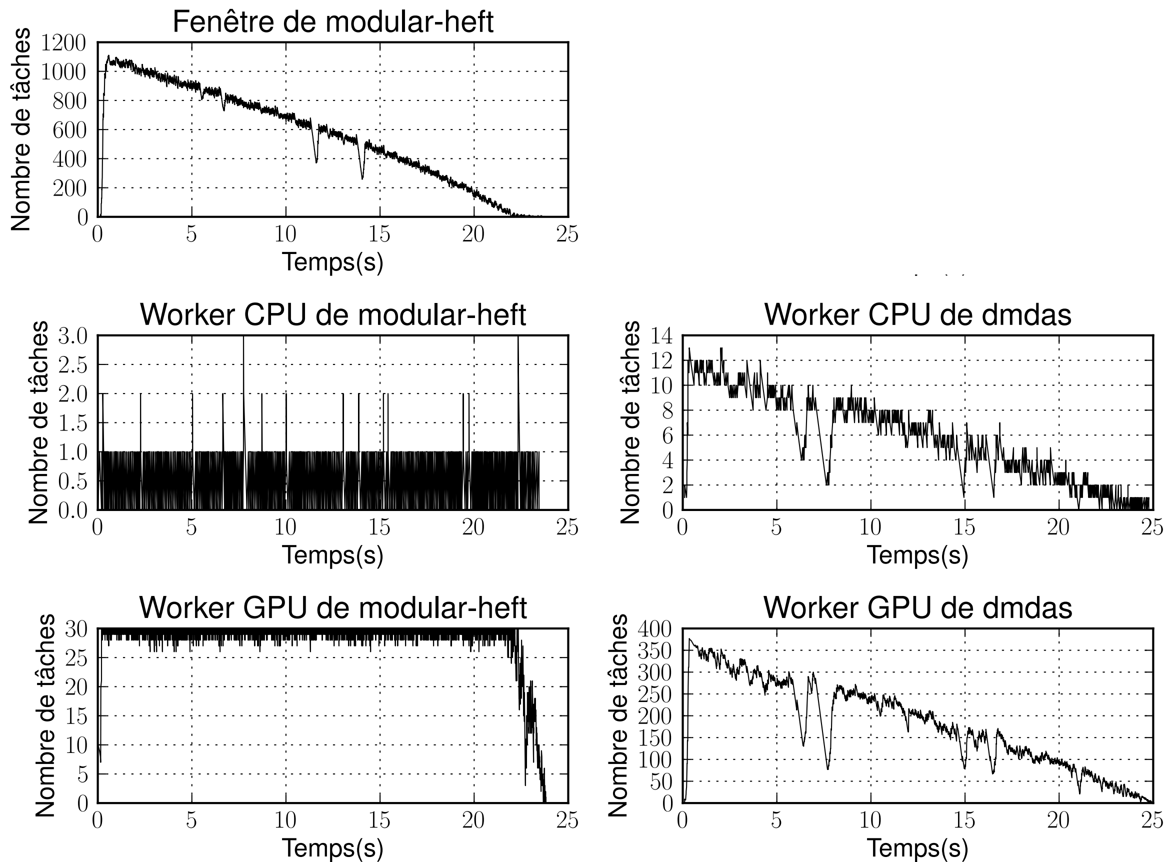

FIGURE 5 – État des réservoirs durant l'exécution de la factorisation de Cholesky tuilée

Notre ordonnanceur modular-heft propose comme solution d'utiliser une fenêtre d'ordonnancement, chargée d'éviter un ordonnancement trop hâtif. Elle permet d'ordonnancer les tâches selon les besoins des Workers, et donc d'éviter le problème rencontré par le DMDAS de StarPU. Les schémas montrant le comportement de notre ordonnanceur attestent de la pertinence de cette proposition : les réservoirs associés aux Workers sont pleins durant tout le régime permanent de l'exécution et le trop-plein de tâches est stocké dans la fenêtre d'ordonnancement. On remarque également que les variations observées dans les réservoirs de l'ordonnanceur DMDAS de StarPU sont très fortement lissées, grâce à l'action de la fenêtre d'ordonnancement.

5.5. Discussion

Ces expériences montrent la pertinence du modèle de structuration des ordonnanceurs que nous proposons par rapport aux performances et au comportement de ces derniers. Nous pouvons maintenant évoquer le problème de la scalabilité de nos ordonnanceurs. Notre modèle permet désormais de combiner facilement plusieurs algorithmes d'ordonnancement, organisés selon différents niveaux. Nous pouvons ainsi imaginer un ordonnanceur qui, par exemple, utilise une heuristique de type Heft pour décider de l'ordonnancement sur des nœuds Numa, puis une heuristique de type vol de travail entre sockets d'un même nœud, et enfin une heuristique de type Eager pour équilibrer la charge entre ressources homogènes d'un même socket.

C'est une étude conjointe avec des experts en algorithmique de l'ordonnancement qui nous permettra d'évaluer la scalabilité des performances de ces futurs ordonnanceurs. Notre modèle ouvre cette possibilité.

6. Conclusion

Nous avons présenté dans cet article une approche composants de la construction d'ordonnanceurs. Dans ce modèle, une stratégie d'ordonnancement est un assemblage de composants d'ordonnancement, chacun étant dédié à une tâche bien précise dans la stratégie globale. Grâce à notre méthode, il est facile d'assembler un nouvel ordonnanceur qui utilise les composants existants ou d'en implémenter de nouveaux.
L'étude du comportement de l'ordonnanceur modular-heft sur la factorisation de Cholesky montre un lissage de la courbe de tâches en attente dans la fenêtre d'ordonnancement, attestant que les priorités sont correctement respectées dans l'application de cette factorisation. De plus, l'étude de performance montre que notre ordonnanceur a des performances équivalentes à l'ordonnanceur DMDAS existant dans StarPU servant de référence à l'étude.
Nos travaux futurs sur ce modèle vont s'axer sur l'amélioration de ses performances, de sa scalabilité pour des exécutions distribuées et sur la vérification de sa validité. Nous prévoyons d'ajouter un composant bouchon pour contrôler le flux de tâches. Cette fonctionnalité permettrait un meilleur ordonnancement des tâches en les regroupant « par paquets », pour éviter le piège qui consiste à ordonnancer une tâche à l'instant même où elle est prête. Nous pensons également implémenter une méthode de gestion de l'ordonnancement distribué. Nous voulons permettre aux ordonnanceurs de communiquer directement entre eux durant l'exécution, au lieu de distribuer statiquement le travail entre plusieurs machines et de laisser chaque ordonnanceur local travailler de manière indépendante. Ces travaux permettent aussi d'envisager et d'expérimenter des ordonnanceurs à plusieurs étages. Il sera possible d'utiliser plusieurs algorithmes d'ordonnancement à différents niveaux, en les associant à la topologie de la machine par exemple. Enfin, nous voulons ajouter un DSL permettant l'assemblage de ces ordonnanceurs à des fins de vérification et de validation notamment, mais aussi à des fins d'intégration dans les contextes d'ordonnancement de StarPU à plus long terme. Il suffira alors à l'utilisateur de décrire sa stratégie d'ordonnancement dans le DSL, qui sera alors validée par un système de preuve, puis compilé sous la forme d'un ordonnanceur modulaire et chargé dans StarPU à l'exécution.

Bibliographie

1. Agullo (E.), Bosilca (G.), Bramas (B.), Castagnede (C.), Coulaud (O.), Darve (E.), Dongarra (J.), Faverge (M.), Furmento (N.) et Giraud (L.). – Matrices over runtime systems at exascale. – In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion :*, p. 1330–1331, 2012.
2. Agullo (E.), Demmel (J.), Dongarra (J.), Hadri (B.), Kurzak (J.), Langou (J.), Ltaief (H.), Luszczek (P.) et Tomov (S.). – Numerical linear algebra on emerging architectures : The PLASMA and MAGMA projects. *Journal of Physics : Conference Series*, vol. 180, juillet 2009, p. 012037.
3. Augonnet (C.). – *Scheduling Tasks over Multicore machines enhanced with acelerators : a Runtime System's Perspective*. – Thèse de PhD, Université Bordeaux 1, 2011.
4. 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, p. 187–198.
5. Barreto (L. P.), Douence (R.), Muller (G.) et Südholt (M.). – Programming OS schedulers with domain-specific languages and aspects : new approaches for OS kernel engineering. – In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, p. 1–6, 2002.
6. Benoit (A.), Marchal (L.) et Robert (Y.). – Who needs a scheduler ? *LIP Research Report RR-2008-34*, octobre 2008.
7. Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.) et Zhou (Y.). – Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, vol. 37, n1, 1996, pp. 55 – 69.
8. 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, janvier 2012, pp. 37–51.
9. Chan (E.), Van Zee (F. G.), Bientinesi (P.), Quintana-Orti (E. S.), Quintana-Orti (G.) et Van de Geijn (R.). – SuperMatrix : a multithreaded runtime scheduling system for algorithms-by-blocks. – In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, p. 123–132, 2008.
10. Gai (P.), Abeni (L.), Giorgi (M.) et Buttazzo (G.). – A new kernel approach for modular real-time systems development. – In *Real-Time Systems, 13th Euromicro Conference on*, p. 199–206, 2001.
11. Gautier (T.), Lima (J. V. F.), Maillard (N.) et Raffin (B.). – XKaapi : a runtime system for data-flow task programming on heterogeneous architectures. – In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.
12. Julia L. Lawall, Gilles Muller et Luciano Porto Barreto. – Capturing OS expertise in an event type system : the bossa experience. – In *Proceedings of the 10th workshop on ACM SIGOPS European workshop, 2002*, p. 54–61, 2002.
13. Pabla (C. S.). – Completely fair scheduler. *Linux J.*, vol. 2009, n184, août 2009.
14. Szyperski (C.). – Component technology : what, where, and how ? – In *Proceedings of the 25th international conference on Software engineering*, p. 684–693, mai 2003.
15. Topcuoglu (H.), Hariri (S.) et Wu (M.-y.). – Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, n3, 2002, p. 260–274.