Optimiser l'Efficacité des Systèmes Parallèles : Adaptation Dynamique des Graphes de Tâches Récursives

Thomas Morin

Compas'2024 : Parallélisme / Architecture/ Système
IETR/LS2N/Nantes-U - Nantes - France, 2-5 juillet 2024

Optimiser l'Efficacité des Systèmes Parallèles : Adaptation Dynamique des Graphes de Tâches Récursives

Thomas Morin

CNRS, Inria, LaBRI, Université de Bordeaux, Bordeaux, France

Résumé
L'efficacité des systèmes parallèles a été significativement améliorée par l'utilisation de modèles de programmation par tâches. Particulièrement, le modèle de Flot Séquentiel de Tâches (FST) est très utilisé grâce à sa simplicité d'utilisation, avec des tâches dont les tailles sont difficilement adaptables à l'exécution, ce qui peut être problématique lorsque déterminer la granularité optimale est complexe. Pour outrepasser cette limitation, nous étendons le modèle des tâches récursives du FST de StarPU pour permettre une transformation dynamique des tâches en sous-graphes. Nous fournissons une évaluation préliminaire en mémoire partagée, qui montre que cette adaptation sur le moment améliore les performances.

Mots-clés : Tâches Granularité Multicoeur

---

1. Introduction

La puissance des supercalculateurs actuels vient en grande partie des architectures hétérogènes. Leur complexité architecturale induit des difficultés de programmation efficace, et donc utiliser efficacement ces architectures pour obtenir des performances portables nécessite dès lors l'utilisation de Systèmes d'Exécutions (SE). Le programmeur décrit ainsi son application à haut niveau, et l'exécution réelle de l'application est déléguée à cet outil. Une grande partie des SE utilise du parallélisme par tâches, où l'application est vue comme un graphe acyclique composé de petites unités de travail, appelées tâches.
Parmi l'ensemble des modèles basés sur des tâches, un des plus utilisés est le Flot Séquentiel de Tâches (FST), comme dans OpenMP, StarPU ou StarSs. Néanmoins, ce modèle présente quelques problèmes, notamment dus à la soumission séquentielle des tâches, qui rend difficile l'adaptation dynamique de la granularité des tâches. Nous apportons une première réponse à cette limite en étendant le modèle FST avec du parallélisme de tâches récursives. Une tâche récursive peut soit être exécutée normalement, soit être découpée, où elle se transforme dès lors en un sous-graphe de tâches durant l'exécution, grâce à une fonction donnée par le programmeur. Cette extension pose deux problèmes majeurs. D'une part, l'implémentation technique des tâches récursives dans un SE, et d'autre part, la sélection des tâches à découper.
Des éléments de réponses ont été précédemment apportés à la première question dans [3]. Dans le présent article, nous nous efforçons de répondre à la deuxième question en utilisant les tâches récursives de StarPU [1]. Ainsi, nos travaux ont abouti aux contributions suivantes :

— Introduction d'un nouvel outil de décision, nommé découpeur, permettant de décider quelle tâche doit être découpée.

— Présentation d'une étude expérimentale préliminaire qui illustre le potentiel de performances de notre approche.

En explorant ces aspects, nous cherchons à améliorer la compréhension du parallélisme par tâches récursives dans le contexte du modèle FST, dans le but d'améliorer l'adaptabilité et la performance des applications pour différentes plateformes hétérogènes.

2. Les défis de la granularité posés par le modèle FST

Dans le modèle FST, les dépendances sont déterminées automatiquement grâce à l'analyse des accès aux données, selon la cohérence séquentielle. Ces dépendances permettent de synchroniser entre elle les tâches : l'exécution d'une tâche ne peut commencer que lorsque toutes ses dépendances entrantes sont relâchées. Les dépendances sortantes sont relâchées lorsque une tâche se termine. Néanmoins, il est important de remarquer que ce modèle a quelques limites intrinsèques malgré son importante utilisation. Premièrement, soumettre des tâches bien avant leur exécution peut induire une congestion du système, rendant complexe la soumission de larges graphes. Essayer de réguler la soumission en la suspendant périodiquement n'est pas sans risque, dans la mesure où une soumission sous-optimale peut impliquer des périodes d'inactivités. Deuxièmement, l'approche une-taille-pour-tous n'est pas raisonnable avec différents types d'unités de calcul. Les GPUs excellent avec des tâches de grosse taille, tandis qu'un coeur d'un CPU va pouvoir atteindre sa performance maximale pour une taille plus petite. Comme il y a généralement un nombre important de coeurs de CPUs, les exploiter tous peut nécessiter un grand nombre de tâches, et donc, pour une taille de problème donnée, de plus petites tâches que pour les GPUs. Troisièmement, même avec des UCs homogènes, varier les granularités est essentiel : si on compare l'exécution d'une tâche à gros grain et l'exécution résultante de la transformation en sous-graphe de cette tâche, alors le temps d'utilisation des unités de calcul va être généralement plus faible pour la tâche à gros grain (elle s'exécute de manière plus efficace), mais sa transformation en sous-graphe peut augmenter le parallélisme, et donc résulter en une diminution du temps de terminaison du calcul. Ainsi, optimiser le temps d'exécution général de l'application peut nécessiter de considérer plusieurs granularités.

Ces problèmes de granularités peuvent être réglés par le SE en autorisant une tâche à être exécutée sur différents coeurs d'un CPU, comme montré dans StarPU [2]. Les coeurs de CPUs sont alors rassemblés et l'exécution d'une tâche est déléguée à un autre SE, qui va être chargé d'exécuter de manière parallèle la tâche sur l'ensemble de coeurs donné. Cela rend les CPUs compétitifs avec les GPUs sur les tâches à gros grain, mais cette technique ne créé pas différentes granularités. Une autre possibilité est de laisser le programmeur soumettre des tâches à différentes granularités, mais cette approche manuelle nécessite beaucoup de travail, et ne rend pas le graphe dynamique, ce qui peut être un problème dans la mesure où la granularité optimale n'est pas forcément connue à la soumission. Heureusement, les SE offrent des caractéristiques qui permettent aux tâches de devenir des sous-graphes à l'exécution. Ces contributions peuvent être classifiées en fonction de leurs possibilités de gérer l'hétérogénéité, leurs expressions des dépendances, et leurs méthodes de gestion de données. OmpSs [5] a introduit les dépendances faibles pour établir des dépendances à grain fin entre des tâches et des sous-graphes dans un contexte homogène. TaskFlow [4] introduit des schémas de tâches avancés pour permettre une génération dynamique des sous-graphes de tâches, mais les dépendances doivent être exprimées manuellement. Enfin, ParSEC autorise des graphes de tâches (DAG) hiérarchiques pour atteindre d'importantes performances sur des systèmes distribués hétérogènes [6].

Les tâches récursives ont été introduites récemment dans StarPU [3], avec lesquelles les pro-

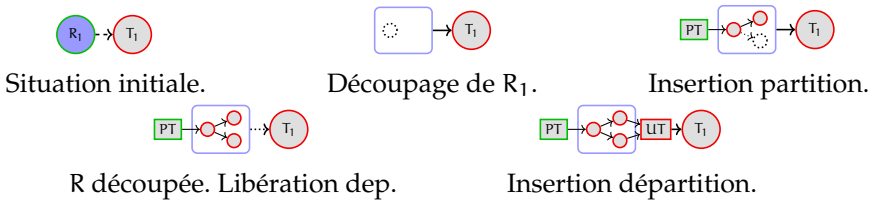

FIGURE 1 – Traitement d'un graphe de tâches récursives par le gestionnaire de données de StarPU.

grammeurs décrivent une hiérarchie de données, et soumettent des tâches qui peuvent être transformés en un DAG qui va travailler sur les sous-données. La caractéristique fondamentale de StarPU est un gestionnaire de données automatique sans synchronisation inutile, qui fonctionne dans un contexte hétérogène. Dans le cas d'un découpage de tâches (la tâche se transforme en DAG), le gestionnaire de données insère de lui même une tâche appelée *Tâche de Partition* (TP) pour partitionner et rendre utilisable les sous-données. À l'inverse, quand une tâche est non découpée, et que les données étaient précédemment partitionnées, le gestionnaire de données insère automatiquement une *Tâche de Départitionnement* (TD) pour attendre les données des tâches des sous-DAGs et les rassembler, comme montré dans la Figure 1. Nous avons d'une part les dépendances *normales*, dépendances introduites par la cohérence séquentielle, et d'autre part les dépendances *récursives*, insérées par les *TP* et les *TD*.

3. Découpage sur le moment des tâches dans StarPU

Cette section présente l'intégration d'un nouveau composant, le découpeur, au sein de la structure d'ordonnancement de StarPU. Le rôle du découpeur est de recevoir une tâche potentiellement récursive et de décider si elle évolue vers une forme récursive, ou si elle reste une tâche normale. Dès lors, nous posons trois question clés.

La première question concerne **quelle** tâche doit être découpée, dans l'idée d'optimiser le temps de terminaison du système. Pour un graphe donné, trouver un équilibre entre le temps d'execution de ses tâches et le parallélisme est crucial, car comme expliqué précédemment, une tâche à gros grain va avoir généralement une exécution plus efficace que le sous-graphe résultant de son découpage, mais elle générera aussi moins de parallélisme, ce qui peut augmenter le temps de terminaison du graphe. Ainsi, notre heuristique permettant de prendre la décision de découper une tâche considère deux critères :

- L'efficacité du découpage, c'est-à-dire le rapport entre le temps séquentiel de la tâche non-découpée et celui du sous-graphe découpé, fixé expérimentalement à 50%.
- Le besoin en parallélisme. Nous découpons lorsque le nombre de tâches disponible est inférieur à 4 fois le nombre de coeurs.

Une fois que les paramètres à prendre en compte sont déterminés, la seconde question est relative à **quand** la décision de découper une tâche doit être prise. Ce moment a une influence significative sur la qualité de l'information utilisée pour prendre une décision. Assurer une certaine synchronisation entre le découpage des tâches et la progression des calculs est essentiel. Découper les tâches trop tôt peut surcharger le système, alors que retarder trop cette décision peut mener à des périodes d'inactivités.

Ainsi, décider de découper peut être fait à différentes étapes du flot du SE, notamment à la soumission. Découper à cet instant minimise les surcoûts, mais la décision peut être faite bien

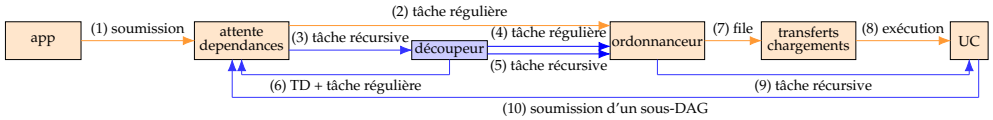

FIGURE 2 – Schéma de l'insertion de notre solution dans l'architecture existante. Les composants et chemins déjà existants sont en oranges, les nouveaux sont en bleus.

avant son exécution, induisant des informations décisionnelles incohérentes avec l'exécution.
Par conséquent, il est avantageux de retarder ce processus de décision tant qu'une tâche récursive n'a pas résolu toutes ses dépendances *normales*. Cependant, cette précaution seule n'est pas entièrement suffisante. Considérons une suite de tâches récursives nommée $R_1 \rightarrow R_2 \cdots \rightarrow R_k$. Supposons que toutes les tâches précédant R_k ont été découpées. La décision de découper R_k peut être faite avant l'exécution des sous-tâches générées par R_1, car toutes les dépendances *normales* ont été relâchées. Ainsi, il est nécessaire d'ajouter des protections pour synchroniser la décision de découpage avec la progression des calculs. Nous relâchons donc les dépendances *normales* sortantes d'une tâche récursive lorsque l'un de ses sous-tâches se termine. Ainsi, la soumission des sous-graphes de la séquence susmentionnée sera faite graduellement : la tâche récursive R_i restera bloquée jusqu'à l'exécution d'une sous-tâche de R_{i-1}.
La troisième question est relative à la conception du SE, et spécifiquement de déterminer **où** placer le découpeur dans le flot du SE. Nous avons déjà vu que le découpeur doit être placé après la phase de soumission d'une tâche. En outre, contrairement aux tâches régulières, l'exécution d'une tâche récursive ne nécessite aucun transfert de données. Par conséquent, pour éviter des transferts potentiellement inutiles, il est de bonne augure de prendre cette décision avant l'étape de préchargement des données.
Comme montré sur la Figure 2, nous positionnons le découpeur à l'étape initiale de l'ordonnancement. La gestion d'une tâche récursive est ainsi : après la soumission par l'application (1), une tâche récursive est donnée au découpeur (3) une fois que ses dépendances sont complétées. Le découpeur prend une décision. Si la tâche nécessite un découpage, elle est directement assignée à l'ordonnanceur (5), qui la place ensuite dans la file d'exécution d'une UC (9). Le traitement de cette tâche récursive entraîne finalement la soumission d'un sous-DAG (10). Dans le cas où la tâche n'est pas découpée, la tâche est transformée en une tâche régulière. Si ses données n'étaient pas préalablement découpées, la tâche régulière sera dès lors exécutée normalement (4). Sinon, ses données sont traitées par des tâches à un niveau plus profond, et une tâche de départitionnement est soumise automatiquement (6) pour maintenir la cohérence séquentielle.

4. Étude de cas : factorisation de Cholesky

Dans cette section, nous effectuons une analyse de performance poussée avec la factorisation de Cholesky, afin de montrer le potentiel de notre solution sur des systèmes homogènes. La plateforme utilisée possède deux Intel Xeon Gold 6240 CPU @ 2.60GHz, avec 18 coeurs chacun. Nous comparons trois tailles de tuile : une large (1120), une moyenne (560) et une fine (280). La taille large correspond à la meilleure taille de tuile sur cette plateforme pour la multiplication de matrices, les autres étant des multiples. Nos critères de découpage sont ceux explicités plus haut, fixés de manière expérimentale, après étude de différentes configurations. L'ordonnanceur locality-aware work-stealing (*lws*) de StarPU a été utilisé pour toutes les expériences. Nous effectuons la moyenne sur dix répétitions.
La Figure 3 compare les versions non-récursives exploitant les tailles de tuiles large, moyenne

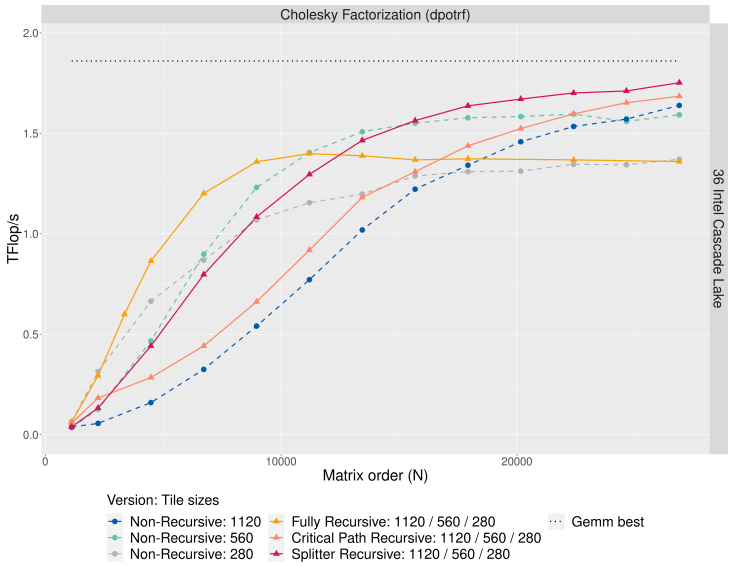

FIGURE 3 – Performance d'une factorisation de Cholesky en fonction de la taille de matrice, de la taille de tuile, et de la version récursive.

et fine, avec des versions récursives, choisissant la granularité dynamiquement. Nous considérons trois variantes : 1) : la version complètement récursive, où chaque tâche est découpée à la granularité la plus fine (280), 2) la version chemin critique, où chaque tâche est découpée si elle est sur le Chemin Critique, 3) la version utilisant les critères explicités précédemment. Nous observons que nos critères permettent une accélération d'approximativement 10% en comparaison de la meilleure non-récursive (large), et une accélération de 5% par rapport à la version récursive chemin critique. La version Chemin Critique améliore légèrement les performances par rapport à la version granularité large, car les tâches sur le chemin critique sont parallèles et accélérées, les rendant moins critiques. Néanmoins, le graphe n'est pas adapté dynamiquement, et donc la quantité de tâches découpées est toujours la même au cours de l'exécution. On ne tire dès lors pas profit de moment où il serait nécessaire d'avoir plus de parallélisme, ou, à l'inverse, moins. La version complètement récursive est la plus rapide pour les petites matrices car toutes les tâches sont découpées, et que la soumission est dès lors parallèle, ce qui réduit le goulot d'étranglement qu'est la soumission. Finalement, en comparant pour différentes tailles de matrices, la version découpeur propose un bon équilibrage entre l'efficacité et le parallélisme, en permettant d'obtenir des performances proches du meilleur possible.
La Figure 4 montre le nombre d'opérations flottantes des tâches prêtes durant l'exécution d'une factorisation de Cholesky pour une matrice carrée de taille 26880. Les couleurs représentent le niveau de l'opération flottante, c'est-à-dire qu'en bleu (respectivement orange, vert), nous représentons le nombre d'opérations flottantes pour les tâches de granularité large (respectivement moyenne, fine). Ainsi, plus la couleur bleue est représentée, et moins il y a de tâches découpées (les tâches sont à granularité large), et plus la couleur verte est représentée, plus il y a de tâches découpées (les tâches sont à granularité fine, et donc le découpeur a décidé d'en découper une grande partie). Nous observons dès lors que notre politique de découpage est capable de réagir lorsque le nombre d'opérations diminue, en créant des petites tâches.

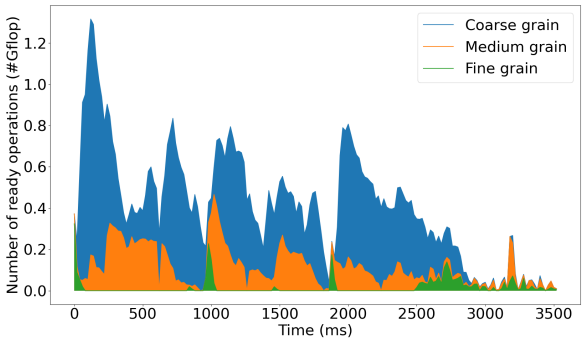

FIGURE 4 – Évolution du nombre d'opérations flottantes au cours d'une factorisation de Cholesky utilisant le découpeur pour une matrice de taille 26 880 en fonction du niveau des tâches.

5. Conclusion

L'accroissement de la complexité des plateformes de calcul a mené au développement de systèmes d'exécutions permettant de séparer l'exécution et l'expression d'un problème. La plupart de ces systèmes utilisent des tâches, et certains utilisent le paradigme par flot séquentiel de tâches. Malgré sa puissance et sa simplicité d'usage, ce paradigme a des limites intrinsèques que nous repoussons en étendant les tâches récursives de StarPU. Cette extension permet de décider si une tâche doit être découpé, et a montré des résultats préliminaires prometteurs.
Nous cherchons à étendre ces résultats au cas hétérogènes, où les tâches peuvent s'exécuter sur CPUs ou sur GPUs, et deux sous-tâches d'une même tâche peuvent aussi s'exécuter sur des unités de calcul différentes, ce qui peut engendrer des communications en surnombre. Notre approche vise à utiliser un programme linéaire permettant de décider si une tâche tend à une affinité CPU ou GPU. En outre, nous explorons l'extension des tâches récursives dans un contexte distribué pour minimiser l'impact des tâches de communication sur le système d'exécution.

Bibliographie

1. Augonnet (C.), , Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. – In *Euro-Par Parallel Processing*, 2009.
2. Cojean (T.), Guermouche (A.), Hugo (A.), Namyst (R.) et Wacrenier (P.-A.). – Resource Aggregation for Task-based Cholesky Factorization on top of Modern Architectures. *Parallel Computing*, vol. 83, 2019.
3. Faverge (M.), Furmento (N.), Guermouche (A.), Lucas (G.), Namyst (R.), Thibault (S.) et Wacrenier (P.-A.). – Programming Heterogeneous Architectures Using Hierarchical Tasks. *Concurrency and Computation : Practice and Experience*, vol. 35, n25, 2023.
4. Huang (T.-W.), Lin (D.-L.), Lin (C.-X.) et Lin (Y.). – Taskflow : A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TPDS*, vol. 33, n6, 2022.
5. Perez (J. M.), Beltran (V.), Labarta (J.) et Ayguadé (E.). – Improving the Integration of Task Nesting and Dependencies in OpenMP. – In *IPDPS*, 2017.
6. Wu (W.), Bouteiller (A.), Bosilca (G.), Faverge (M.) et Dongarra (J.). – Hierarchical DAG scheduling for Hybrid Distributed Systems. – In *IPDPS*, 2015.