Compas'2022 : Parallélisme / Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Programmation des architectures hétérogènes à l'aide de tâches hiérarchiques

Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Samuel Thibault, Pierre-André Wacrenier

Inria Bordeaux Sud-Ouest, 200 Av. de la Vieille Tour, 33405 Talence
prenom.nom@inria.fr

Résumé
Les systèmes à base de tâches ont gagné en popularité du fait de leur capacité à exploiter pleinement la puissance de calcul des architectures hétérogènes complexes. Un modèle de programmation courant est le modèle de *soumission séquentielle de tâches (Sequential Task Flow, STF)* qui malheureusement ne peut manipuler que des graphes de tâches statiques. Ceci conduit potentiellement à un surcoût lors de la soumission, et le graphe de tâches statique n'est pas nécessairement adapté pour s'exécuter sur un système hétérogène. Une solution standard consiste à trouver un compromis entre la granularité permettant d'exploiter la puissance des accélérateurs et celle nécessaire à la bonne performance des CPU. Pour répondre à ces problèmes, nous proposons d'étendre le modèle STF fourni par le support d'exécution StarPU [4] en y ajoutant la possibilité de transformer certaines tâches en sous-graphes durant l'exécution. Nous appelons ces tâches des *tâches hiérarchiques*. Cette approche permet d'exprimer des graphes de tâches plus dynamiques. En combinant ce nouveau modèle à un gestionnaire automatique des données, il est possible d'adapter dynamiquement la granularité pour fournir une taille optimale aux différentes ressources de calcul ciblées. Nous montrons dans cet article que le modèle des tâches hiérarchiques est valide et nous donnons une première évaluation de ses performances en utilisant la bibliothèque d'algèbre linéaire dense Chameleon [1].

1. Introduction

Suite à la récente évolution des systèmes de calcul hautes performances en direction d'architectures multi-cœurs hétérogènes, de nombreux efforts de recherche sont dédiés à la création de supports d'exécution qui prennent en charge des techniques de programmation portables et des outils permettant d'exploiter ces matériels complexes. Des implémentations matures de supports d'exécution sont désormais disponibles pour les systèmes multi-cœurs homogènes et hétérogènes. Des standards comme OpenMP (depuis la version 4.0) utilisent la programmation à base de tâche, avec des applications mises sous la forme de graphes de tâches orientés acycliques (*Directed Acyclic Graph, DAG*). Cependant, utiliser ce paradigme pour exploiter efficacement des architectures hétérogènes pose plusieurs problèmes. Tout d'abord, les ressources de calcul de ces plateformes ont des caractéristiques et exigences diverses. Les accélérateurs GPU favorisent généralement les grands jeux de données, là où les cœurs CPU classiques atteignent leurs performances maximales avec des noyaux à grain fin travaillant sur une empreinte mémoire réduite. De plus, ces systèmes comportent habituellement un nombre de CPU bien supérieur au nombre de GPU, augmenter la quantité de petites tâches pourrait donc améliorer les performances. Dans cet article nous introduisons un nouveau genre de tâche, les *tâches*

hiérarchiques, qui sont des tâches capables de se transformer dynamiquement à l'exécution en un nouveau graphe de tâche. Le programmeur a seulement besoin de fournir des indications sur les tâches pouvant être hiérarchiques. Le support d'exécution peut alors retarder la soumission de certaines parties du graphe de tâches pour paralléliser le processus d'insertion des tâches, sélectionner dynamiquement l'implémentation à exécuter et fortement réduire le nombre de tâches dans le système. Cette approche est similaire à ce qui est fait par le méchanisme de tâche imbriquées d'OPENMP. Cependant, nous l'étendons jusqu'à le rendre capable de gérer les architectures hétérogènes tout en exprimant des dépendances à grain fin. Cela est rendu possible grâce à gestionnaire de données avancé pouvant changer la disposition des données dynamiquement et de manière asynchrone. Le modèle des tâches hiérarchiques répond aux problématiques de dynamisme du graphe, de surcoût du support d'exécution et aux limitations inhérentes au STF.

2. Travaux connexes

De nombreux efforts de recherche ont été dédiés au problème de surcoût des supports d'exécution à base de tâche, ainsi qu'à la question de l'augmentation de la quantité de parallélisme exprimée par ces systèmes. Les auteurs de [3] analysent les facteurs limitants la scalabilité d'un tel support d'exécution et proposent des solutions individuelles pour chacun des problèmes listés, notamment un système de dépendances sans attente et un ordonanceur centré sur la délégation au lieu du vol de travail. D'autres approches s'intéressent à une gestion des dépendances plus avancée. Par example, [6] introduit des tâches employant de façon interne des méthodes de partage de travail. Elles peuvent ainsi exploiter un parallélisme de boucle à grain fin sans recourir à des barrières. Cependant la contribution qui s'approche le plus de notre proposition est le concept de *weak dependencies* présenté par [7]. Il s'agit d'une extension d'OPENMP qui améliore le modèle de flot de donnée d'OPENMP en prenant en charge des dépendances à grain fin, non seulement entre les tâches « sœurs », mais entre des tâches présentant n'importe quel lien de parenté. Notre contribution généralise les *weak dependencies* au cas hétérogène où la cohérence de la mémoire n'est pas assurée par le matériel sous-jacent. Du point de vue de la gestion dynamique des tâches, plusieurs travaux visent à donner aux supports d'exécution à base de tâche une expressivité plus dynamique. Dans *TaskFlow* [5], des modèles de tâche avancés sont introduits. Le modèle dynamique en particulier permet de générer un sous-graphe dynamiquement pour une tâche donnée. Cependant, une synchronisation est ajoutée à la fin de ces tâches pour faciliter la gestion des dépendances et c'est au programmeur de gérer les données, en changeant leur disposition si nécessaire.

3. Contexte

Le modèle de programmation de *soumission séquentielle de tâches (Sequential Task Flow, STF)* consiste à s'appuyer sur la *cohérence séquentielle* des données pour créer des dépendances implicites. De cette manière, le STF permet de soumettre simplement une série de tâches grâce à des appels de fonction non-bloquants qui délèguent l'exécution de la tâche au support d'exécution. À la soumission d'une tâche, le support d'exécution l'ajoute au graphe avec ses dépendances, qui sont automatiquement calculées par analyse des dépendances de données [2]. L'exécution de la tâche est ensuite retardée jusqu'à ce que ses dépendances soit satisfaites. La figure 1 montre un exemple de fonctionnement du STF. Au lieu d'appeler les fonctions (`F`, `G`, `H`), la version STF soumet séquentiellement trois tâches correspondantes et attend leur complétion. On spécifie également les données sur lesquelles ces fonctions travaillent, ainsi que leur mode d'accès (lecture seule, écriture, lecture-écriture). La tâche `G` accèdant à la donnée `x` après que la tâche `F` y ait écrit, le support d'exécution en déduit une dépendance entre les deux tâches.

```
call F(x, y)
call G(x, u)
call H(y, z)
```

```
call submit(F, x:RW, y:RW)
call submit(G, x:R, u:RW)
call submit(H, y:R, z:RW)
call wait_tasks_completion()
```

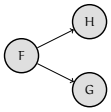


FIGURE 1 – Exemple de pseudo-code pour un algorithme séquentiel (*à gauche*), la version STF correspondante (*centre*) et le graphe qui en résulte (*à droite*).

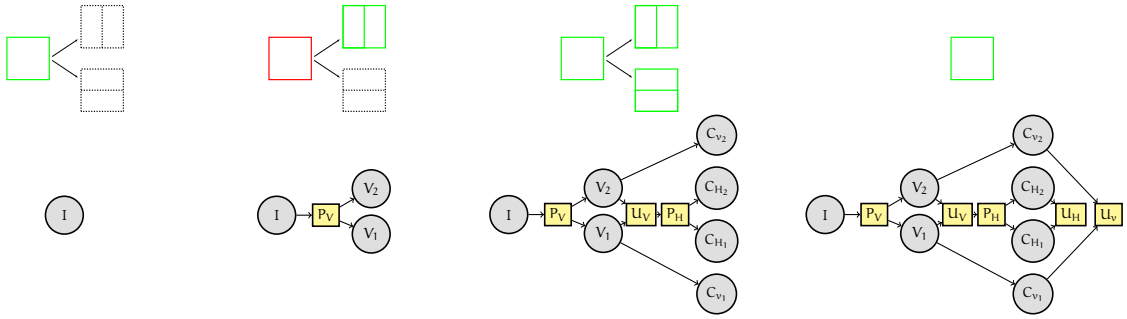
De la même manière, une dépendance est ajoutée entre `F` et `H` à cause de la donnée `y`. Nous présentons dans les sections 4 et 5 comment les tâches hiérarchiques peuvent être incorporées dans ce modèle sans rompre la cohérence séquentielle.

4. Gestion automatique des données

La manipulation des données est fondamentale dans StarPU, à la fois pour automatiquement placer les dépendances du STF et pour gérer les tranferts entre les différentes mémoires d'une architecture distribuée ou hétérogène. Afin de profiter de ces fonctionnalités, une application doit enregistrer les données manipulées par les tâches. StarPU propose pour cela une structure de données opaques appelée *handle*. Pour faciliter la manipulation de données, StarPU utilise des *filtres*, un outil permettant de partitionner une donnée en sous-données, chacune associée à un nouveau *handle*. Nous proposons un mécanisme pour automatiser la gestion de plusieurs partitions simultanées. Grâce à ce mécanisme, StarPU peut automatiquement insérer les tâches de partitionnement ou de départitionnement si nécessaire. Le programmeur doit d'abord définir un schéma de partitionnement grâce à une opération de *plan* qui définit pour StarPU le filtre et les nouveaux *sous-handles*. Le gestionnaire de données sera alors chargé d'insérer les tâches de partitionnement et de garantir la cohérence des données. Quand une tâche est soumise, StarPU s'assure que tous ses *handles* sont correctement partitionnés. Si la donnée est accédée en lecture seule, différents partitionnements peuvent coexister. Dès que la donnée est accédée en écriture, StarPU automatiquement (et potentiellement récursivement) départitionne les sous-données pour ne laisser actif que le *handle* où l'on écrit. La figure 2 montre une matrice sur laquelle deux plans de partitionnement, un vertical et un horizontal, sont déclarés. On initialise le handle via la tâche I, on le modifie dans les tâches V_i en utilisant le partitionnement vertical et enfin on effectue des vérifications (en lecture seule) dans les tâches C utilisant les deux partitionnements. La figure 2a montre l'état du DAG et la disposition des données après l'exécution des opérations de *plan* et l'insertion de la tâche I. Une tâche de partitionnement P_v est insérée par StarPU quand la première tâche V modifiant la donnée est soumise (figure 2b). On applique la même méthode à la soumission des tâches travaillant en lecture seule sur les partitions verticales et horizontales. Contrairement aux C_{V_i}, les tâches C_{H_i} ne partagent aucun *handle* avec les tâches précédantes, le gestionnaire de données s'appuie donc sur l'ancêtre commun de ces *handles* (la matrice entière) et ajoute des tâches de départitionnement et partitionnement pour rendre les données accessibles dans le découpage horizontal (tâches U_V et P_H dans la figure 2c). Enfin, on insére des tâches de départitionnement pour nettoyer les partitions au terme de l'exécution (figure 2d).

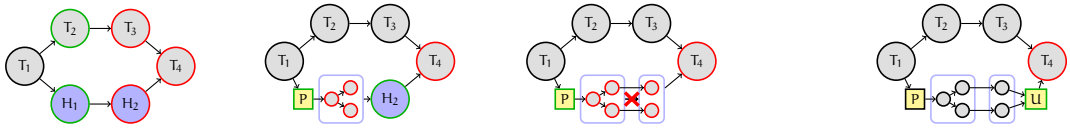
5. Paradigme des tâches hiérarchiques

Une tâche hiérarchique est simplement une tâche ordinaire qui est capable de soumettre un sous-graphe à l'exécution, au lieu d'effectuer des calculs. Traiter une tâche hiérarchique consiste à soumettre le sous-graphe correspondant, les dépendances sortantes pouvant être relachées au terme de cette soumission. L'introduction de tâches hiérarchiques dans un support d'exécution à base de tâches comme StarPU doit respecter les contraintes suivantes afin de préserver la



(a) Étape 1. (b) Étape 2. (c) Étape 3. (d) Étape 4.

FIGURE 2 – Illustration du mécanisme de gestion automatique des données. Les données en pointillés sont inactives. Le trait est plein quand elles sont actives. Un cadre rouge correspond à un état *partionné en lecture-écriture*, un cadre vert à *partionné en lecture seule* ou *départionné*.



(a) DAG initial (b) Traitement de H_1 (c) Traitement de H_2 (d) Insertion automatique de U

FIGURE 3 – Exemple de graphe avec deux tâches hiérarchiques et quatres tâches ordinaires

généralité de l'implémentation.

1. La profondeur de la hiérarchie n'est pas limitée.
2. Les programmeurs expriment leur DAG au plus haut niveau et se contentent d'annoter certaines tâches comme potentiellement hiérarchiques.
3. La gestion des données doit être invisible pour les programmeurs.
4. Les dépendances entre tâches doivent toujours être placées au grain le plus fin (le plus profond dans la hiérarchie).

Les propriétés 1 et 2 permettent d'avoir une interface générale sans limitations du point de vue du programmeur. La propriété 3 fait référence à la gestion des données (voir la section 4). Elle est nécessaire pour le cas hétérogène où la mémoire n'est pas partagée entre les ressources et où des transferts de données auront forcément lieu. Enfin, la propriété 4 est utilisé dans la gestion du parallélisme, nous ne voulons d'une approche *fork-join* où une barrière serait placée à la fin de chaque tâche hiérarchique traitée. La figure 3 est un scénario d'exécution où certaines tâches du graphe ont été marquées comme hiérarchiques (en bleu). L'état de chaque tâche est décrit par sa bordure : une tâche prête a une bordure **verte**, une tâche en attente a une bordure **rouge** (au moins une dépendance n'est pas satisfaite) et une tâche terminée a une bordure **noire**. On peut ainsi voir dans la figure 3a que T_1 est finie, les tâches T_2 et H_1 sont donc prêtes à être exécutées. T_2 et T_3 sont des tâches ordinaires, mais H_1 soumet un sous-graphe, produisant la figure 3b. La dépendance entre H_1 et H_2 est alors relachée et H_2 est prête à être traitée. On peut voir dans la figure 3c comment les dépendances entres les tâches « hiérarchiquement » soumises sont placées au plus bas niveau de la hiérarchie par le support d'exécution. L'utilisation des tâches hiérarchiques nécessite de changer la granularité des données dynamiquement à l'exécution, potentiellement à chaque traitement de tâche hiérarchique. L'approche que nous proposons est d'automatiquement insérer des tâches de gestion de données avant les tâches utilisant des données incorrectement partitionnées. On s'appuie pour cela sur le gestionnaire de données présenté dans 4. On peut voir dans la figure 3b (resp. 3d), l'insertion d'une tâche de partitionnement P (resp. U) devant le sous-graphe généré par H_1 (resp. T_4).

6. Évaluation expérimentale

Pour illustrer le potentiel des tâches hiérarchiques, nous les appliquons dans le cadre de l'algèbre linéaire dense avec la bibliothèque CHAMELEON. Pour ce faire, nous avons étendu les descripteurs de matrice pour décrire un partitionnement hiérarchique des blocs. Comme expliqué dans la section 4, tout ces partitionnements sont seulement planifiés et ne seront appliqués qu'à l'exécution, si nécessaire. Dans les expériences suivantes, la décision de traiter une tâche hiérarchique est prise le plus tôt possible (à la soumission) et se base sur la structure des données des blocs utilisés par l'opération. Si toutes les tuiles peuvent être partitionnée on soumet un sous-graphe, sinon, la tâche utilise le premier niveau de donnée commun entre les blocs. Nous avons travaillé sur une architecture comportant deux INTEL XEON GOLD 6142 de 16 cœurs cadencés à 2.6 GHz, deux NVIDIA V100 et 384 GB of memory. Les tailles de bloc retenues sont celles offrant la meilleure performance asymptomatique pour CPU seul (960) et pour une configuration hybride CPU-GPU (2880). Les courbes utilisant des tâches hiérarchiques utilisent la notation $x/y/z/etc.$, ce qui signifie que chaque bloc mesure initialement x et est ensuite susceptible d'être partitionné en blocs de taille y, eux même pouvant être découpés en blocs de taille z. Plus précisément, nous découpons les blocs diagonaux de chaque matrice (composée de $nb \times nb$ blocs) afin de créer nb sous-graphes. Dans le cas des matrices ayant plus d'un niveau de partitionnement on découpe les sous-blocs de la même manière. STARPU a été configuré pour utiliser un seul stream par GPU (nous avons observé qu'augmenter le nombre de stream a peu d'impact sur les performances avec des blocs de taille 960 ou 2880). Nous utilisons l'ordonanceur DMDA, un algorithme s'appuyant sur des modèles de performances pour minimiser le temps d'exécution. La figure 4 présente le comportement d'un produit de matrice par bloc, en utilisant un ou deux GPU. Dans les deux cas, les versions hiérarchiques sont en retard sur les petites matrices, car le traitement des tâches hiérarchiques ajoute un surcoût. À mesure que la taille de matrice augmente, la quantité de noyaux utilisant de plus petits blocs devient suffisante pour répartir une partie du travail sur les CPU et amortir ce surcoût. Ajouter plusieurs niveaux de partitionnement n'a pas ici un impact très important sur les performances des versions hiérarchiques. À partir d'un certain point, le nombre de tâches de calcul devient assez important pour que la courbe 2880 commence à donner plus de travail aux CPU et rattrape les courbes hiérarchiques. La chute soudaine qu'on peut voir au bout de la courbe 960 s'explique par un conflit entre le préchargement des données effectué par l'ordonanceur STARPU et l'éviction au sein de la mémoire GPU. Ces expériences illustrent la capacité des bulles à produire un meilleur compromis au niveau de la granularité des calculs, et ce malgré un partitionnement de matrice encore simpliste. Nous sommes capables d'augmenter l'expressivité grâce aux tâches hiérarchiques tout en améliorant le comportement général.

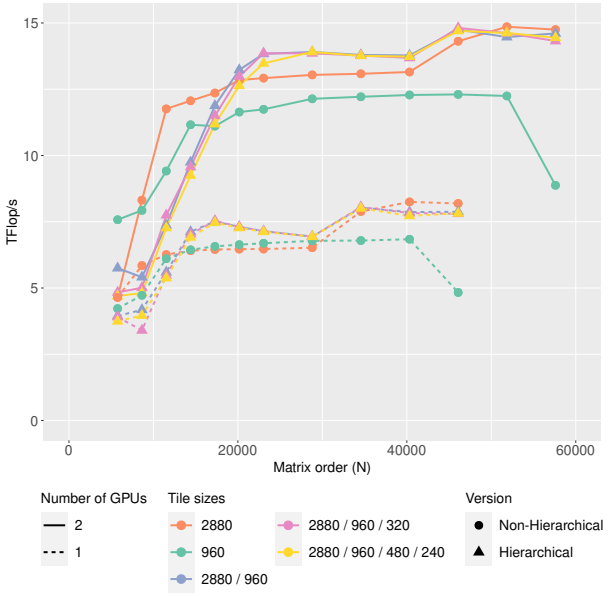


FIGURE 4 – Comparaison des performances du produit de matrice (DGEMM) avec et sans tâches hiérarchiques

7. Conclusion

Dans cet article, nous proposons d'étendre le STF et de mettre à jour le support d'exécution sous-jacent pour dépasser les contraintes inhérentes à ce modèle de programmation. Notre approche introduit un nouveau genre de tâche, les *tâches hiérarchiques*, qui peuvent soumettre un sous-graphe de tâches à l'exécution. En outre, afin d'assurer la justesse du graphe généré par la soumission parallèle, nous présentons un nouveau gestionnaire automatique de données dont le but est d'adapter leur disposition au bon moment en soumettant des tâches de gestion de données. Nous avons prévu d'étendre ce travail de plusieurs façons. Tout d'abord, nous devons appréhender les tâches hiérarchiques du point de vue de l'ordonancement. Il faut répondre à la question « quand doit-on traiter une tâche hiérarchique ? ». La solution de ce problème doit prendre en compte la quantité de tâches déjà dans le système et la charge de chaque ressource de calcul. Par ailleurs, on aimerait pouvoir choisir *quel sous-graphe* soumettre quand une tâche hiérarchique est traitée, il faudra pour cela des modèles de performances avancés permettant de sélectionner l'implémentation la plus adaptée. Enfin, le graphe de tâches résultant du traitement d'une tâche hiérarchique se doit d'être ordonnancé efficacement. Répondre à ces problématiques permettra d'améliorer le comportement des tâches hiérarchiques au sein du support d'exécution. Plus généralement, nous voulons examiner comment ce modèle peut être utilisé pour implémenter des algorithmes irréguliers tel que des solveurs d'algèbre linéaire creux ou utilisant des approximations de rang faible.

Remerciements

Ce travail est soutenu par l'ANR dans le cadre du projet Solharis sous la subvention (ANR-19-CE46-0009). Les expériences présentées dans ce document ont été réalisées à l'aide de la plateforme expérimentale PlaFRIM, financée par l'Inria, le CNRS (LABRI et IMB), l'Université de Bordeaux, Bordeaux INP et le Conseil Régional d'Aquitaine (voir https ://www.plafrim.fr).

Bibliographie

1. Agullo (E.), Augonnet (C.), Dongarra (J.), Ltaief (H.), Namyst (R.), Thibault (S.) et Tomov (S.). – A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. *GPU Computing Gems, Jade Edition*, vol. 2, 2011, pp. 473–484.
2. Allen (R.) et Kennedy (K.). – *Optimizing Compilers for Modern Architectures : A Dependence-Based Approach*. – Morgan Kaufmann, 2002.
3. Álvarez (D.), Sala (K.), Maroñas (M.), Roca (A.) et Beltran (V.). – Advanced Synchronization Techniques for Task-Based Runtime Systems. – In *Proceedings of PPoPP '21*, p. 334–347, 2021.
4. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, vol. 23, février 2011, pp. 187–198.
5. Huang (T.-W.), Lin (D.-L.), Lin (C.-X.) et Lin (Y.). – Taskflow : A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 2021, pp. 1–1.
6. Maroñas (M.), Sala (K.), Mateo (S.), Ayguadé (E.) et Beltran (V.). – Worksharing tasks : An efficient way to exploit irregular and fine-grained loop parallelism. – In *Proceedings of HiPC'19*, pp. 383–394, 2019.
7. Perez (J. M.), Beltran (V.), Labarta (J.) et Ayguadé (E.). – Improving the Integration of Task Nesting and Dependencies in OpenMP. – In *Procceeding of IPDPS'17*, pp. 809–818, 2017.