TER : Modèle d'exécution spéculative glouton pour la parallélisation à base de tâches

Jules Van Assche

03/02/19

1 Contexte et problématique

Le nombre de coeurs présent sur les cpu ne cesse d'augmenter tout comme le potentiel de parallélisation, malheureusement tout les programmes n'exploitent pas ce parallélisme au maximum par choix du développeur ou car le programme ne nécessite pas d'être parallélisé. La parallélisation étant très intéressante en terme de performance, de nombreux paradigmes de parallélisation apparaissent permettant d'exploiter un plus grand nombres de coeurs et de fait accélerer l'exécution du programme.

La parallélisation à base de tâche est l'un d'entre eux et a montré de nombreux points forts.Ce paradigme permet grâce à un niveau d'abstraction élevé de ne pas se soucier du matériel ce qui engendre une plus grande portabilité et évite d'avoir à réecrire le code en fonction du matériel. Cette approche nécessite une bonne séparation des calculs en tâches et aussi la gestion des dépendances entre les différentes tâches. Le résultat de cette approche nous donne un *graphe acyclique* (DAG) de tâches et de leurs dépendances. Encore une fois il y a plusieurs manières de construire le graphe des dépendances, celle qui nous intéresse est le « *sequential task flow* » (STF). Un thread va s'occuper de créer les tâches et d'avertir le moteur d'éxecution des accès aux données pour chacune des tâches. Le découpage en tâches lui est à la volonté du développeur, il doit choisir une granularité qui permette un gain de performance conséquent sans trop complexifier la construction du graphe avec un nombre de tâches trop élevé.
Des moteurs d'executions à base de tâches existent déja, OpenMP est le plus connu. Ses avantages sont sa portabilité, stabilité, néanmoins avec l'introduction d'un mot clé pragma [1] servant à indiquer le type de donnée et un nombre de tâches conséquantes le code peut être sujets à des erreurs et à des une faible lisibilité. De plus le nombre de mises à jours permettant une rétro-compatibilité sont faibles.
Le moteur d'éxecution utilisé dans ce ter est SPETABARU qui est bien entendu un moteur d'éxecution à base de tâches crée en C++. Ce moteur à

la particularité de proposer la *spéculation*. Le développeur à la possibilité d'indiquer si une tâche modifiera ou non la donnée en entrée. Etant donné que nous sommes dans une moteur d'éxecution utilisant un STF, un thread peut avertir le moteur d'éxecution de la modification ou non de la donnée lors de l'exécution d'une tâche.
L'idée d'un modèle d'éxecution serait donc de dupliquer les tâches pouvant potentiellement modifier les données sur les coeurs disponibles. Cela ajouterait des chemins d'exécution qui seraient exécutés en parallèle au chemin original, le chemin emprunté dépendra de la modification des données par les tâches incertaines , le moteur d'exécution devra donc désactiver les tâches qui ne seront pas utilisé suite à la spéculation.

Figure 1: DAG original

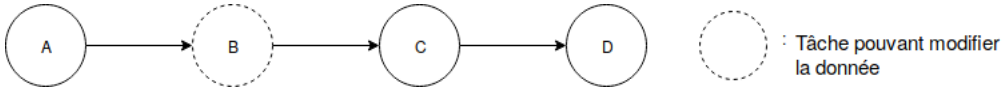
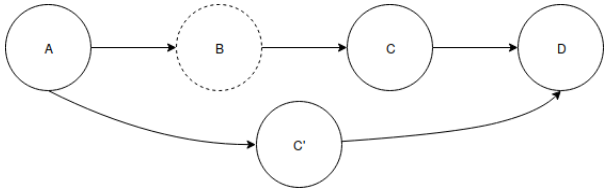
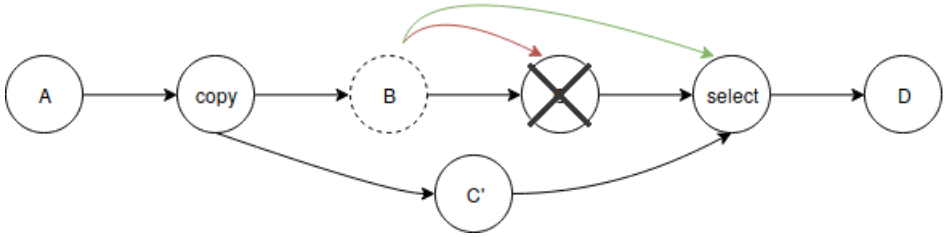


Figure 2: DAG avec tâches dupliqué



On peut voir ci dessus un cas ou nous avons une tâche pouvant potentiellement modifier sa donnée. Nous allons donc accroitre le parallélisme en ajoutant une tâche supplémentaire qui sera exécutée en parallèle. Le chemin emprunté lors de l'éxecution dépendra donc de la modification de la donnée ou non par B.

Figure 3: Activation/désactivation des tâches lors de l'exécution



Ci-dessus, si **B ne modifie pas la donnée** on **désactivera C** et on **laisse C' finir son éxecution**. D'autres paramètres sont à prendre en compte que nous verrons plus tard comme la *cohérance des données* (le select et le copy sur la Figure 3). L'idée ici est d'avoir un aperçu d'un modèle d'éxecution spéculatif et du concept d'activation/désactivation des tâches en fonction de la spéculation, le tout au moment de l'exécution.

2 Objectifs

Comme vu ci-dessus un certain nombre de moteur d'exécutions à base de tâches existent, néanmoins à notre conaissance aucun ne propose un système de spéculation. Le but de ce TER est donc :

- Comprendre la parallélisation à base de tâches et le concept de spéculation dans ce contexte
- Comprendre le moteur d'éxecution SPETABARU et implémenter un modèle d'execution spéculatif.
- Mesurer le gain obtenu grâce à ce modèle d'execution spéculatif en le comparant à d'autres modèles spéculatifs du moteur.

3 Tâches effectuées

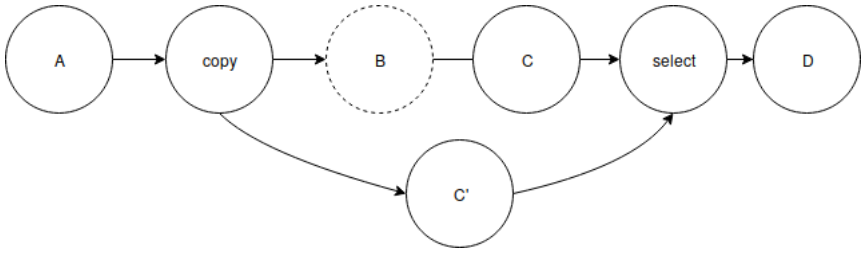
3.1 Comprendre la parallélisation à base de tâches et le concept de spéculation

Pour comprendre la parallélisation je me suis surtout basé sur l'article de mon tuteur qui résume son fonctionnement. [3] J'ai du intégrer des concepts nécessaire à la compréhension de la parallélisation à base de tâche comme le modèle *fork-join*, *le sequential-task-flow* [2] ou le *direct-acyclic-grap*. J'ai pu lire des articles qui utilisait ses concepts lors de l'implémentation de leurs système. J'ai du faire attention à bien distinguer la création du DAG et son éxecution. Il à fallu aussi bien comprendre la notion de spéculation dans ce contexte, ce que ce principe impliquait en terme d'activation/désactivation des tâches.

3.2 Comprendre le fonctionnement du moteur et conceptualisation du modèle d'exécution

J'ai tout d'abord commencé par modéliser des cas d'exécution sur papier et aussi numériquement pour pouvoir poser des questions sur les différents cas possibles. Il à fallut comprendre le concept de cohérance des données, étant donné que nous créeons des tâches parallèles supplémentaires, il faut que la tâche ayant des résultats de 2 tâches puisse choisir quelle donnée est la plus cohérante, ce qui ce fait lors de la spéculation avec l'activation/désactivation de tâches servant à assurer la cohérance des données comme par exemple avec la tâche D sur la figure 2 qui doit choisir quelle donnée utiliser entre C' et C.

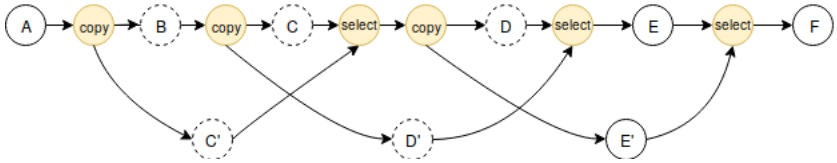
Figure 4: Figure 2 avec cohérence des données



Ci-dessus on peut voir la création de nouvelles tâches *copy* et *select*, ses tâches comme leurs noms l'indique explicitement servent à respectivement copier la donnée qui sera utilisé puis à récuperer la donnée qui sera utilisé en restant cohérent.
J'ai ensuite pu éxecuter des cas plus complexe sur le papier pour tenter de lister tout les paramètres à prendre en compte lors de la création du DAG et de son éxecution.

Figure 5: Exemple de cas plus complexe, graphe original

Figure 6: Exemple de cas plus complexe, graphe transformé



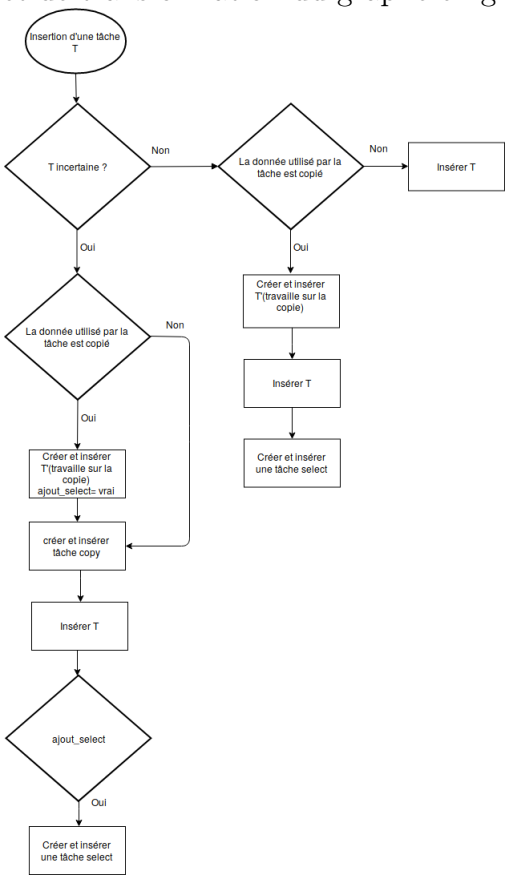
J'ai pu me questionner lors de la réalisation de ses graphes et de me échanges avec mon tuteur sur plusieurs points lors de la création du graphe mais aussi de son éxecution.

- Qu'est ce qui dépend de la compilation et de l'éxecution ?
- Quelle tâche va s'occuper d'activer/désactiver les autres tâches ?
- Quand faut-il assurer la cohérence des données et qui doit s'en occuper ?

J'ai réalisé plusieurs grafcet pour poser des potentiels solutions à ses questions. J'ai demandé l'avis de mon tuteur sur les différents grafcet pour ne pas me diriger vers une solution fausse et perdre du temps.

J'ai ensuite réaliser des grafcet plus concis pour chacun des cas probable lors de l'exécution du graphe. Par exemple que se passe t-il lorsqu'une tâche dupliqué est incertaine et se finit ? Que faut il faire ? (par exemple la tâche D' sur la figure 6.

Figure 7: Grafcet de transformation du graphe en graphe parallélisé



4 Tâches prévisionnel

4.1 Comprendre le fonctionnement du moteur d'éxecution

J'ai déja compris une partie du fonctionnement du moteur d'éxecution notamment grâce à la compréhension de la parallélisation à base de tâches et de ses concepts sous-jacent. Néanmoins le modèle d'exécution que je dois implémenter n'utilise pas certaine notions du moteur d'exécution tels que les groupes de tâches spéculatif. Je dois donc pour l'instant faire abstraction de cette notion dans le moteur tout en comprenant son fonctionnement.
Le moteur étant conçu pour être utilisé pour des tâches différentes avec des type multiples son écriture est assez complexe car il doit garantir une généricité. De ce fait il y a de nombreux concept C++ à assimilier ou ré-assimilier

tels que les templates variadic, les tuples, de la manipulation de réferences.

4.2 Implémenter le modèle d'exécution

Une fois que j'aurais compris les concepts utilisés dans le moteur et que j'aurais rédigé du pseudo code suite à mes grafcet je pourrais me lancer dans l'implémentation du modèle d'exécution. Cette partie est probablement la plus compliqué étant donné qu'elle nécessite la compréhension du moteur, du modèle d'exécution et des concepts C++.

4.3 Mesurer les performances du modèle d'éxecution par rapport à d'autres modèles présent dans le moteur d'éxecution

Par la suite il faudra mesurer les performance du modèle que j'aurais implémenté dans plusieurs cas différents. Il faudra comparer ses résultats avec les autres modèles déja existants dans le moteur d'éxecution et pouvoir peut être déterminer l'utilisation d'un modèle spécifique dans un cas précis.

References

[1] Openmp tasking. `https://www.nersc.gov/users/software/programming-models/openmp/openmp-tasking/`.

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. *ACM Transactions on Mathematical Software*, July 2016.

[3] Bérenger Bramas. Increasing the degree of parallelism using speculative execution in task-based runtime systems. *CoRR*, abs/1803.04211, 2018.