Programmation des architectures hétérogènes à l'aide de tâches divisibles

Marc Sergent

- Rapport de stage -
Programmation des architectures hétérogènes à l'aide de tâches divisibles

Stage de Master 1 Informatique réalisé par
Marc Sergent
du 2 Mai au 31 Juillet 2012
au sein de l'INRIA Bordeaux - Sud-Ouest à Talence

Directeur de stage : Raymond Namyst
Encadrant : Samuel Thibault

1er septembre 2012

Table des matières

Chapitre I

Introduction

Les ordinateurs multicoeurs équipés d'accélérateurs réalisent une percée remarquable dans le paysage du calcul haute performance. Parmi les machines parallèles les plus puissantes au monde (selon le classement `www.top500.org`), une sur deux est équipée d'accélérateurs. Cette récente évolution vers des architectures hétérogènes a entraîné un regain d'efforts de recherche visant à concevoir des outils permettant de programmer facilement des applications capables d'exploiter efficacement toutes les unités de calcul de ces machines.

Le support d'exécution StarPU, développé dans l'équipe Runtime, a été conçu pour servir de cible à des compilateurs de langages parallèles et des bibliothèques spécialisées (algèbre linéaire, développements de fourier, etc.). La fonction principale de StarPU est d'ordonnancer des graphes dynamiques de tâches de manière efficace sur l'ensemble des ressources hétérogènes de la machine. Pour ce faire, le support s'appuie sur des modèles adaptatifs de prédiction de coût des calculs et des transferts de données, ainsi que sur une mémoire virtuellement partagée destinée à minimiser les mouvements de données entre les différentes mémoires. StarPU est avant tout une plateforme pour expérimenter de nouvelles stratégies d'ordonnancement, celles-ci pouvant aisément être construites en redéfinissant les fonctions appelées en réaction à certains événements (nouvelle tâche prête, unité de calcul oisive, etc.). StarPU a récemment été utilisé avec succès pour l'implémentation d'algorithmes parallèles en algèbre linéaire sur configurations multi-GPU, en collaboration avec d'autres équipes françaises et étrangères.

L'un des aspects les plus difficiles, lors du découpage d'une application en graphe de tâches, est de choisir la granularité de ce découpage, qui va typiquement de pair avec la taille des blocs utilisés pour partitionner les données du problème. Les granularités trop petites ne permettent pas d'exploiter efficacement les accélérateurs de type GPU, qui ont besoin de mettre en oeuvre un parallélisme massif pour « tourner à plein régime ». À l'inverse, les processeurs traditionnels exhibent souvent des performances optimales à des granularités beaucoup plus fines. Le choix du découpage est donc non seulement difficile,

mais il a en outre une influence sur la quantité de parallélisme disponible dans le système : trop de petites tâches risque d'inonder le système en introduisant un surcoût inutile, alors que peu de grosses tâches risque d'aboutir à un déficit de parallélisme. Fixer un découpage manuellement demande donc de nombreux tâtonnements avant de trouver le bon compromis.

L'objectif de ce travail de recherche est d'introduire dans StarPU la notion de tâches divisibles, c'est-à-dire de tâches que le support d'exécution pourra décider (ou non) de redécouper en plusieurs sous-tâches à l'exécution, en fonction de différents critères tels que la quantité de parallélisme que l'on souhaite générer, l'opportunité d'exploiter certains types d'unités de calcul à un moment donné, etc. Une grande partie du travail consistera à étudier comment gérer efficacement une partition non uniforme des données (pour autoriser la co-existence de sous-données de différentes granularités) ainsi qu'une gestion des dépendances entre tâches s'adaptant à des raffinements locaux du graphe de tâches.

On s'intéressera dans un premier temps à une version simple du découpage, où le support d'exécution sait à l'avance en combien de sous-tâches il peut découper chaque tâche (voire même contrôler leur taille), ce qui permettra de se concentrer dans un premier temps sur l'ordonnancement, la calibration de la stratégie de découpage et la gestion des granularités multiples.

Chapitre II

Contenu

II.1 Granularité adaptative et tâches divisibles : le contexte

Depuis l'avènement des architectures hétérogènes, la granularité est un problème récurrent dans l'optimisation des performances des applications. De nombreux travaux ont déjà été menés pour étudier ce problème, et essayer de mettre en place des solutions plus ou moins efficaces.

En Octobre 1997 sortait la première API d'OpenMP en Fortran. OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel. Les prémisses de la question de granularité adaptative apparaissent dans une des possibilités d'ordonnancement des calculs parallèles d'OpenMP appelée "guided" : cet ordonnancement donne au premier thread une partie du travail de taille donnée, puis chaque thread suivant prendra une partie plus petite de travail que le précédent.

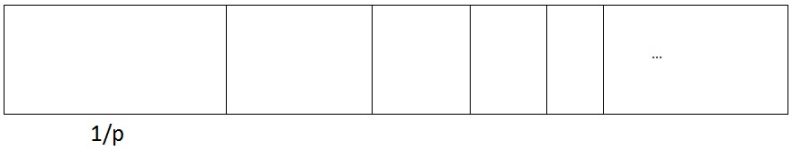

FIGURE II.1 – Répartition du travail en ordonnancement "guided" dans OpenMP

Cette forme d'ordonnancement commence à mettre en place l'idée que le dimensionnement des parties du travail doit être décidé dynamiquement, afin d'améliorer l'équilibrage de charge durant l'exécution.

Intel a également développé un langage de programmation pour le calcul parallèle appelé Cilk. Initialement créé au MIT en 1994, ce langage est l'un des premiers à mettre en place une façon automatique de gérer la granularité pendant l'exécution : le vol de travail par division de tâches.
Lorsqu'un thread n'a plus de travail (idle), il va voler du travail à un autre thread pour pouvoir continuer à travailler. Concrètement, Cilk dispose d'un compilateur développé pour détécter des parties de code qui sont "volables" par d'autres threads. Cependant, ce procédé reste très coûteux à l'exécution.

Depuis les années 2000, des travaux de recherche se sont centrés sur un certain paradigme de codage des applications : par graphe de tâches. Ces travaux mettent en lumière 4 façons distinctes de penser la granularité dynamique des tâches, en fonction du nombre d'unités de calcul :

- **Les tâches rigides :** elles s'exécutent sur un nombre fixe d'unités de calcul.

- **Les tâches modelables :** elles peuvent s'exécuter avec différents nombres d'unités de calcul, mais lorsque l'exécution a commencé, il est impossible de changer ce nombre. Cette façon de penser a déjà été implémentée dans le support d'exécution StarPU, sous le nom "Parallel tasks" (tâches parallèles).

- **Les tâches malléables :** le nombre d'unités de calcul sur lesquelles la tâche s'exécute peut varier durant toute l'exécution. Ces tâches sont les plus permissives, mais en pratique les tâches malléables sont des amas de tâches séquentielles, regroupées pour pouvoir être exécutées en parallèle.

- **Les tâches divisibles :** elles fonctionnent dans la même idée que les tâches malléables, sauf qu'au lieu d'accumuler des petites tâches séquentielles pour créer des tâches parallèles, elles peuvent être divisées si besoin durant l'exécution, pour alimenter des unités de calcul oisives.

Pour mettre en évidence des résultats dans ces travaux, il est courant d'utiliser les algorithmes de type LU (Lower / Upper) : opérations sur les matrices par décomposition inférieure / supérieure, car ces algorithmes servent à mesurer la puissance de calcul des machines du top500.

Parmi ces algorithmes, nous avons choisi d'utiliser la factorisation de Cholesky : elle consiste, pour une matrice symétrique définie positive A, à déterminer une matrice triangulaire inférieure L telle que : $A = LL^T$, où L^T est la transposée de L.

Dans l'algorithme informatique de cette factorisation, utilisant la librairie LAPACK, nous avons choisi de nous baser sur la factorisation tuilée de Cholesky :

cet algorithme est composé de quatre tâches principales : POTRF (qui est la factorisation de Cholesky), TRSM, GEMM et SYRK.

Cet algorithme est itératif : une itération consiste à, sur une colonne donnée de la matrice triangulaire basse, calculer le POTRF sur la tuile qui est sur la diagonale, puis calculer le TRSM sur toutes les autres tuiles de la colonne, et enfin apporter les contributions de ces calculs au reste de la matrice, à l'aide des tâches GEMM et SYRK, en fonction du type de tile (GEMM pour une tuile carrée, SYRK pour une tuile sur la diagonale).

Voici un exemple d'exécution de la factorisation de Cholesky sur une matrice 4*4 avec cet algorithme :

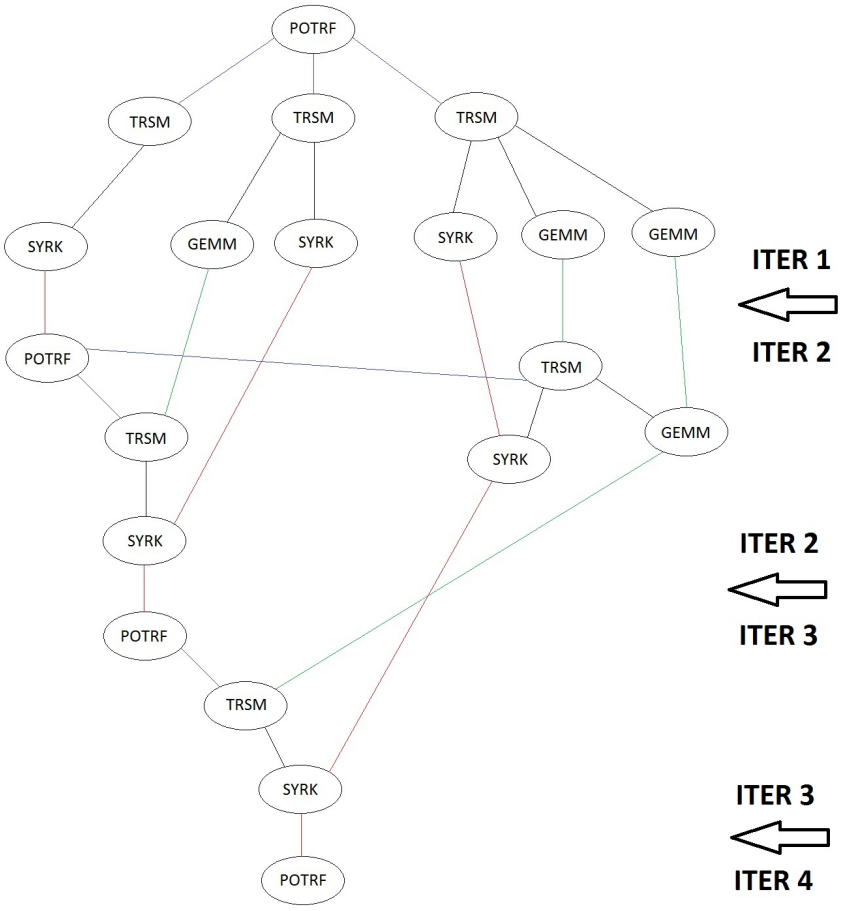

FIGURE II.2 – Graphe de tâches de Cholesky tuilé sur matrice 4*4

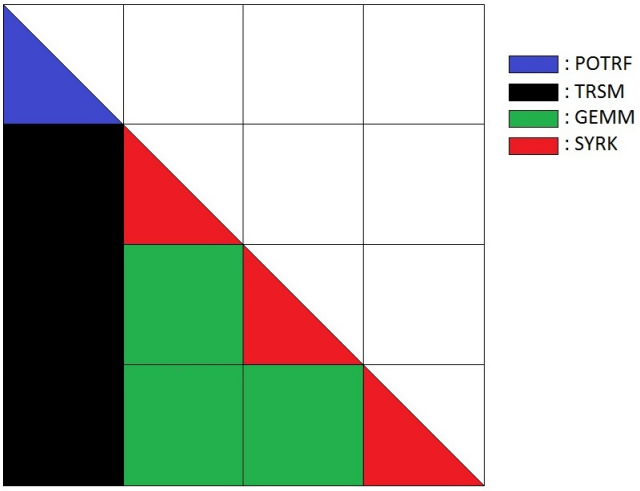

FIGURE II.3 – Itération 1 de Cholesky tuilé sur matrice 4*4

II.2 Ma contribution : les tâches divisibles dans StarPU

II.2.1 Préambule : un choix de perspective.

Pour ce stage, nous avons pris le parti, avec Raymond Namyst et Samuel Thibault, de ne se préoccuper que de la division de l'exécution d'une tâche, et pas de ses données.
En effet, la gestion des dépendances de données et du partitionnage des données dans StarPU fait qu'il est actuellement impossible de diviser récursivement des données durant l'exécution, car il interdit de changer le partitionnement alors que des tâches sont déjà lancées sur les données.
Nous avons donc choisi de diviser les données en grain fin, et de donner aux tâches un ensemble de données correspondant à un bloc de données plus gros, qu'elles pourront "distribuer" à leurs enfants lors de la division, afin de pouvoir se concentrer sur la façon de diviser l'exécution des tâches.

Nous avons également choisi de fixer l'ordonnanceur que nous allons utiliser dans StarPU : HEFT (Heterogeneous Earliest Finishing Time), car c'est un algorithme d'ordonnancement très classiquement utilisé.

II.2.2 Tâches divisibles : l'idée de base

Dans StarPU, les tâches suivent un chemin précis durant leur exécution. La décision de découpage d'une tâche ne peut être prise qu'à un seul endroit : lors du passage de la tâche dans l'ordonnanceur.

Ce choix se fait en deux étapes :

- Est-ce que l'ordonnanceur peut se passer de calculer si la tâche divisée termine avant la tâche complète, car il dispose d'informations lui permettant de savoir que la division est inutile ?
  Si non, on "prédécoupe" la tâche, en appelant la fonction de division associée à la tâche, qui rend un tableau contenant les tâches filles à l'ordonnanceur.
  Cette première étape est déterminée grâce à l'heuristique de régime permanent, que je détaillerais plus loin.

- Est-ce que la durée d'exécution des tâches filles sur la machine est plus petite que celle de la tâche père ?
  Si oui, on soumet les tâches filles à StarPU.
  Cette seconde étape est en rapport avec l'heuristique HEFT, que j'expliquerai en détails un peu plus loin.

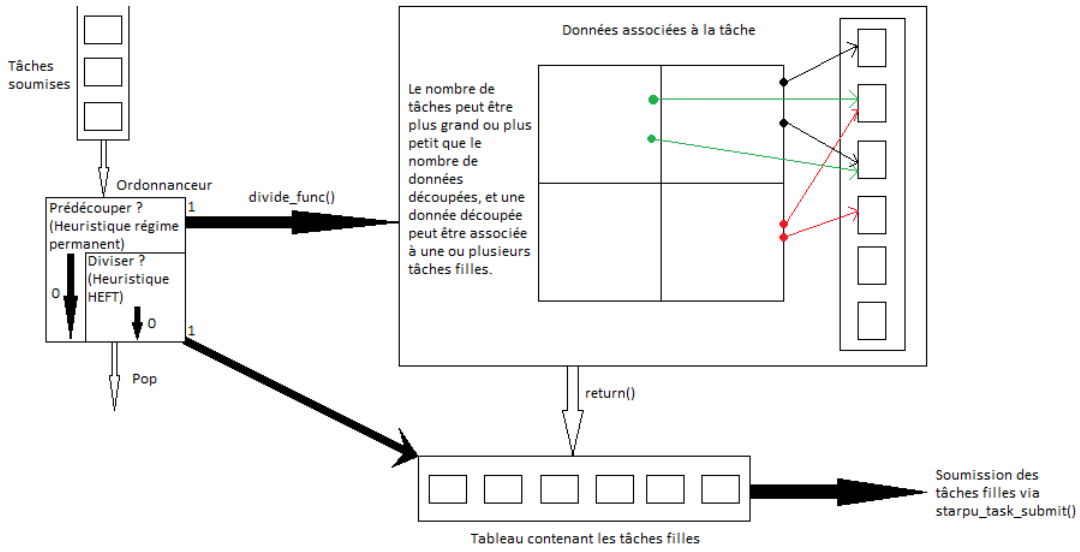

FIGURE II.4 – Tâches divisibles : comment ça fonctionne ?

Ce fonctionnement a été mis en place de sorte que la ou les tâches filles puissent récupérer directement les données détenues par leur père afin de pouvoir s'exécuter directement, sans perdre de temps avec la gestion des dépendances de données.

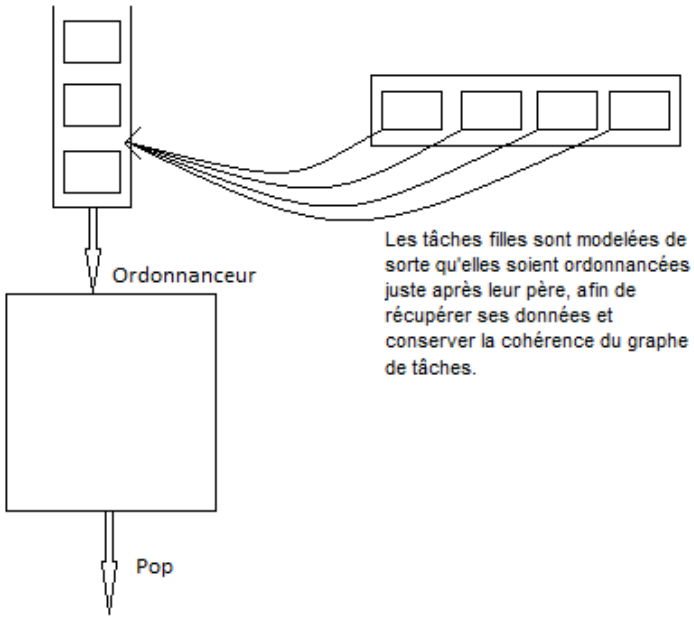

FIGURE II.5 – Les tâches filles sont ordonnancées directement après leur père

Les tâches filles forment donc un sous-graphe de tâches, "interne" à la tâche père.
Lorsque les tâches filles sont soumises, la tâche père est mise en "veille", attendant un message de ses enfants lui disant qu'il peut terminer. Dans ce but, une tâche supplémentaire est toujours ajoutée au graphe des tâches filles, appelée tâche de synchronisation : son but est de transmettre au père le message de terminaison de tâche dès que les tâches filles ont terminé.

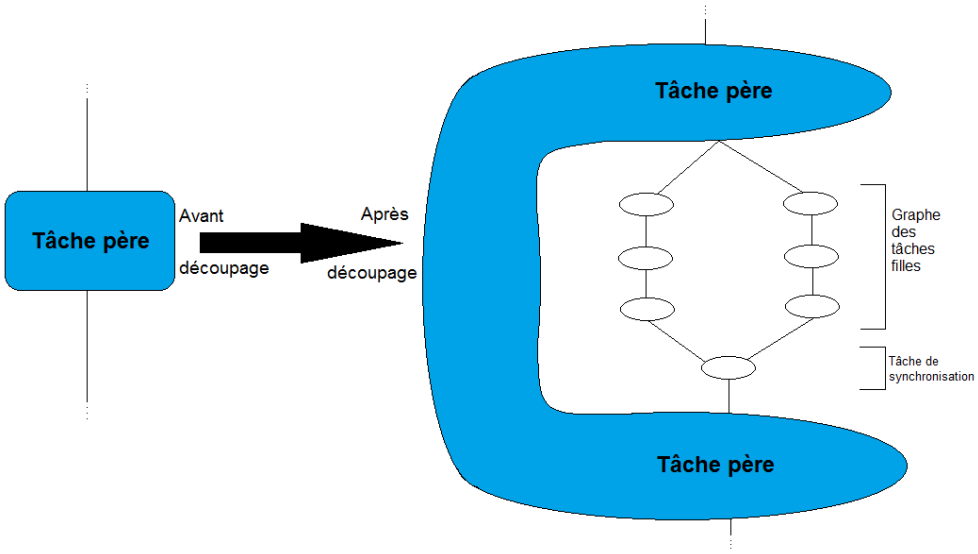

FIGURE II.6 – Tâches divisibles : dans le graphe des tâches

II.2.3 Pouvoir "prédire l'avenir" : la fenêtre d'ordonnancement et le bouchon

Pour pouvoir ordonnancer efficacement les tâches divisibles sur des architectures hétérogènes, il est important de pouvoir choisir quelle tâche diviser : en effet, on peut choisir de diviser une petite tâche sur-le-champ alors qu'une très grosse tâche arrive après, et perdre en parallélisme car il aurait plutôt fallu découper la grosse tâche et conserver la petite en l'état.
Il peut être intéressant donc de disposer d'un moyen d'analyser les tâches qui vont arriver, et en déduire des informations, comme constater qu'il reste assez de tâches à venir pour satisfaire toutes les unités de calcul et se passer donc de calculer un possible découpage, qui serait un surcoût de calcul inutile, par exemple.

Il nous faut donc des outils pour pouvoir "prédire l'avenir" : j'ai donc intégré à StarPU la notion de fenêtre d'ordonnancement.

Cette fenêtre a été associée à un système de gestion d'équilibrage de charge. Il permet de contrôler le nombre et le temps d'exécution des tâches déjà ordonnancées, afin de conserver un équilibrage de charge intéressant à chaque instant de l'exécution d'une application. Lorsque ce système détecte que l'équilibrage est bon, il stocke toutes les tâches arrivant dans l'ordonnanceur dans la fenêtre d'ordonnancement.
Grâce à la fenêtre d'ordonnancement, l'ordonnanceur peut regarder les tâches

qui vont lui être présentées à l'avance, et faire des choix d'ordonnancement et de découpage adaptés à l'état courant et futur de la machine.

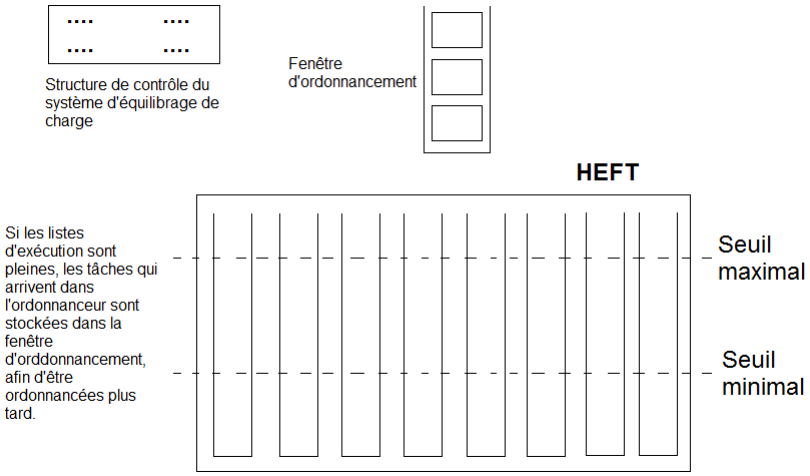

FIGURE II.7 – La fenêtre d'ordonnancement

De plus, j'ai ajouté à la fenêtre d'ordonnancement un système de "bouchon" : tant qu'un ou plusieurs bouchons sont actifs (ils sont contrôlés par des fonctions push/pop), toutes les tâches passant par l'ordonnanceur sont directement stockées dans la fenêtre d'ordonnancement, même si l'équilibrage de charge n'est pas optimal. Une fois le bouchon désactivé, les tâches reprennent le chemin normal, et l'équilibrage de la charge de la machine est demandé.
Le but de ce système de bouchon est de permettre d'ordonnancer des paquets de tâches ensemble, afin de pouvoir choisir quelles tâches découper seulement lorsque l'ensemble du paquet de tâches a été soumis : il fait attendre l'ordonnanceur jusqu'à ce que toutes les tâches soumises soient présentes dans la fenêtre d'ordonnancement, afin de faire des choix de découpage les plus intéressants possible.
Le bouchon est également utilisé dans StarPU afin d'ordonnancer ensemble les tâches ayant les mêmes dépendances de données.

II.2.4 Choix du découpage : heuristique de régime permanent, heuristique "HEFT"

Maintenant que les éléments nécessaires à l'ordonnanceur pour faire des choix intéressants ont été introduits, il reste à définir les heuristiques permettant de faire le choix : découper ou ne pas découper ?
J'ai choisi de me baser sur deux heuristiques que j'ai appelées : heuristique de

régime permanent, et heuristique HEFT.

La première est une heuristique très simple basée sur le nombre de tâches disponibles à l'instant courant. Lorsqu'on sait que le graphe de tâches est assez large à un instant t, on dit qu'on est en "régime permanent" : il n'est pas nécessaire de découper la moindre tâche, car il reste assez de tâches à venir pour nourrir toutes les unités de calcul.
Cette heuristique est très peu coûteuse mais très intéressante, car elle permet de se passer d'autres calculs d'heuristiques bien plus coûteux alors qu'ils ne sont pas nécessaires, et donc de gagner en efficacité.

La deuxième heuristique est basée sur le système d'ordonnancement HEFT : elle simule sur une copie locale (à l'instant courant) de l'état de la machine l'exécution de HEFT avec les tâches filles et la tâche père, et choisit l'état où la date de terminaison critique est la meilleure.

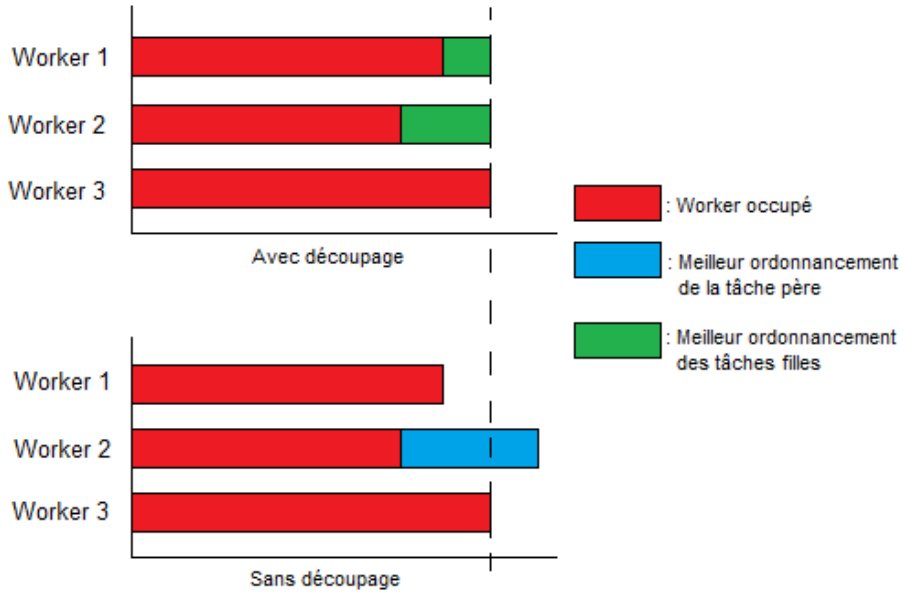

FIGURE II.8 – Une fois la simulation du découpage effectuée, on la compare au choix fait sans découpage, et on choisit l'option la plus intéressante. Ici, le découpage semble plus intéressant, la tâche va donc être découpée.

Cette heuristique reste très coûteuse, mais elle permet d'excellents choix d'ordonnancement.

II.3 Eléments d'implémentation : comment créer une fonction de division dans StarPU ?

Je vais vous présenter comment faire pour ajouter une fonction de division à une tâche dans StarPU.

Je vais vous expliquer comment j'ai ajouté à l'algorithme de la factorisation de Cholesky de StarPU les fonctions de division. Nous allons prendre l'exemple du premier noyau de calcul de cet algorithme, le POTRF.

Tout d'abord, il faut donner au codelet associé à la tâche la fonction de division que l'on veut appeler. Il existe un champ pour cela désormais, le champ division_func :

```
static struct starpu_codelet cl11 =
{
    .where = STARPU_CPU,
    .cpu_funcs = {chol_cpu_codelet_update_u11, NULL},
    .model = &chol_model_11,
    .division_func = &divide_func_u11,
    .numchild = 4
};
```

On encadre la soumission des tâches initiales de l'algorithme par un bouchon dans le code principal du programme : un cork_push/pop, afin que l'ordonnanceur les voient toutes d'un coup.

```
/* We need a cork here to make sure that all the Cholesky tasks are scheduled together */
starpu_cork_push();

for (k = 0; k < nblocks; k++)
{
    ...
}
...
starpu_cork_pop();
```

Une fonction de division se résume en trois parties principales : créer les tâches filles (sans oublier la tâche de synchronisation), initialiser les tâches (paramètres de la tâche / du codelet, quelles données, quelles dépendances), et stocker les tâches dans le tableau passé en paramètre, ainsi que leur nombre, afin de fournir ces informations à l'ordonnanceur.

```
/* Cholesky divide function for U11 computation : POTRF */
int divide_func_u11(struct starpu_task *task,
    struct starpu_task *tab_ret[STARPU_NMAXCHILDTASKS],
```

```
    int *numtasks)
{
    unsigned nchild = task->cl->numchild;
    unsigned nbufsperchild = (task->cl->nbuffers)/nchild;

    /* Creating the final task : it has no codelet, it's an empty
     * task which aims to synchronize the small tasks, and call the
     * callback function which will end their father. */
    struct starpu_task *final_task = starpu_task_create();

    final_task->nodeps = task->nodeps + 1;
    final_task->priority = task->priority;
    final_task->callback_func = callback_func;
    final_task->callback_arg = (void *) task;

    /* Creating child tasks, filling child tasks fields from
     * their father's fields, and expressing dependencies
     * between them and the final task. */
    struct starpu_task *child_task[nchild];

    for (i=0; i < nchild; i++)
        child_task[i] = starpu_task_create();

    for (i=0; i < nchild; i++)
    {
        (child_task[i])->nodeps = task->nodeps + 1;
        (child_task[i])->priority = task->priority;
        (child_task[i])->cl->where = task->cl->where;
        (child_task[i])->cl->numchild = nchild;
        (child_task[i])->cl_arg_size = task->cl_arg_size;
    }

    (child_task[0])->cl->nbuffers = nbufsperchild;
    (child_task[1])->cl->nbuffers = 2*nbufsperchild;
    (child_task[2])->cl->nbuffers = 3*nbufsperchild;
    (child_task[3])->cl->nbuffers = nbufsperchild;

    (child_task[0])->cl->cpu_funcs[0] = chol_cpu_codelet_update_u11;
    (child_task[0])->cl->division_func = &divide_func_u11;
    (child_task[0])->cl->model = &chol_model_11;

    (child_task[1])->cl->cpu_funcs[0] = chol_cpu_codelet_update_u21;
    (child_task[1])->cl->division_func = &divide_func_u21;
    (child_task[1])->cl->model = &chol_model_21;

    (child_task[2])->cl->cpu_funcs[0] = chol_cpu_codelet_update_u22;
```

```
    (child_task[2])->cl->division_func = &divide_func_u22;
    (child_task[2])->cl->model = &chol_model_22;

    (child_task[3])->cl->cpu_funcs[0] = chol_cpu_codelet_update_u11;
    (child_task[3])->cl->division_func = &divide_func_u11;
    (child_task[3])->cl->model = &chol_model_11;

    for (j=0; j < nbufsperchild; j++)
    {
        /* .. Affecting handles to child tasks .. */
    }

    /* Enforcing child tasks dependencies */
    starpu_task_declare_deps_array(child_task[1],1,&(child_task[0]));
    starpu_task_declare_deps_array(child_task[2],1,&(child_task[1]));
    starpu_task_declare_deps_array(child_task[3],1,&(child_task[2]));

    /* Enforcing synchronization dependency between child tasks and the final task */
    starpu_task_declare_deps_array(final_task,nchild,child_task);

    tab_ret[0] = child_task[0];
    tab_ret[1] = child_task[1];
    tab_ret[2] = child_task[2];
    tab_ret[3] = child_task[3];
    tab_ret[4] = final_task;

    *numtasks = nchild;

    return 0;
}
```

Et c'est tout ! Le reste est directement effectué par l'ordonnanceur.

II.4 Validation du travail : des résultats intéressants

Pour obtenir des résultats valides, nous avons décidé de partir de la version de Cholesky en tuiles, mais avec la matrice allouée en un bloc contigu en mémoire afin de faciliter le partitionnement des données. Pour comparer les résultats, nous nous baserons donc sur les résultats obtenus avec l'algorithme similaire intégré dans les exemples de StarPU (examples/cholesky/cholesky_tag).
Nous avons décidé de nous concentrer sur des machines à architectures homogènes (CPU uniquement), afin de pouvoir obtenir une idée de base du comportement des tâches divisibles sur ces machines simples, avant de les transposer sur des architectures hétérogènes.
Nos machines de tests font partie du groupes de machines Infini(1-4) se trouvant sur le réseau du CREMI, appartenant à l'Université Bordeaux 1.

J'ai obtenu les résultats suivants :

- J'ai constaté un gain de temps par rapport à la version des exemples de StarPU de l'ordre de 5%.

- J'ai constaté un gain en équilibrage de charge pendant tout le calcul, et particulièrement à la fin de l'algorithme : les tâches divisibles y prennent toute leur importance, en permettant un très bon équilibrage de charge jusqu'à la toute fin de l'exécution.

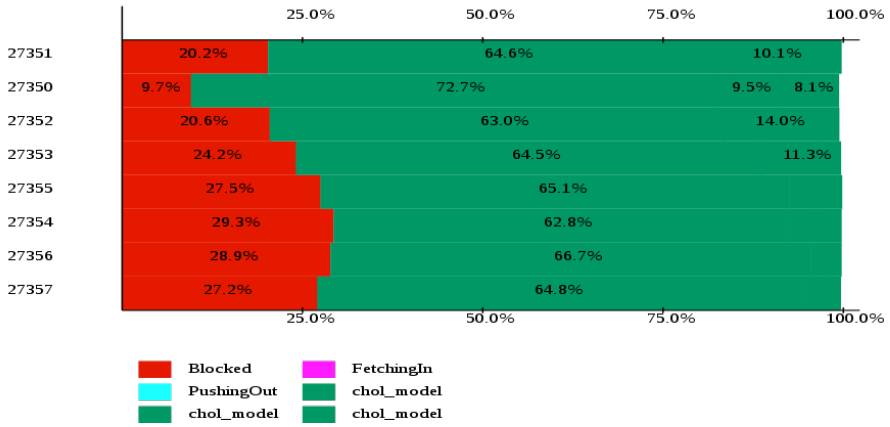

FIGURE II.9 – Equilibrage de charge original sur Cholesky tuilé - dernières 1.5 secondes. Le temps en rouge est le temps d'inactivité.

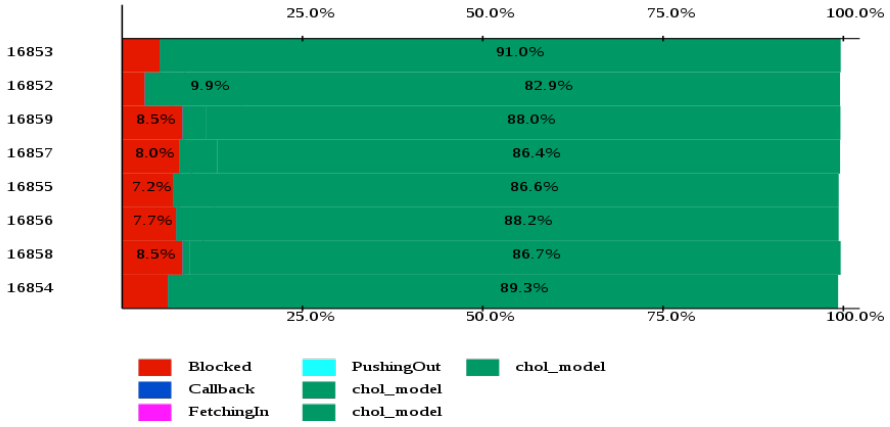

FIGURE II.10 – Equilibrage de charge avec tâches divisibles sur Cholesky tuilé - dernières 1.5 secondes. Le temps en rouge est le temps d'inactivité.

– J'ai aussi mis en évidence que les tâches divisibles sont utilisées pendant toute l'exécution, permettant une mixité entre plusieurs granularités de tâches.

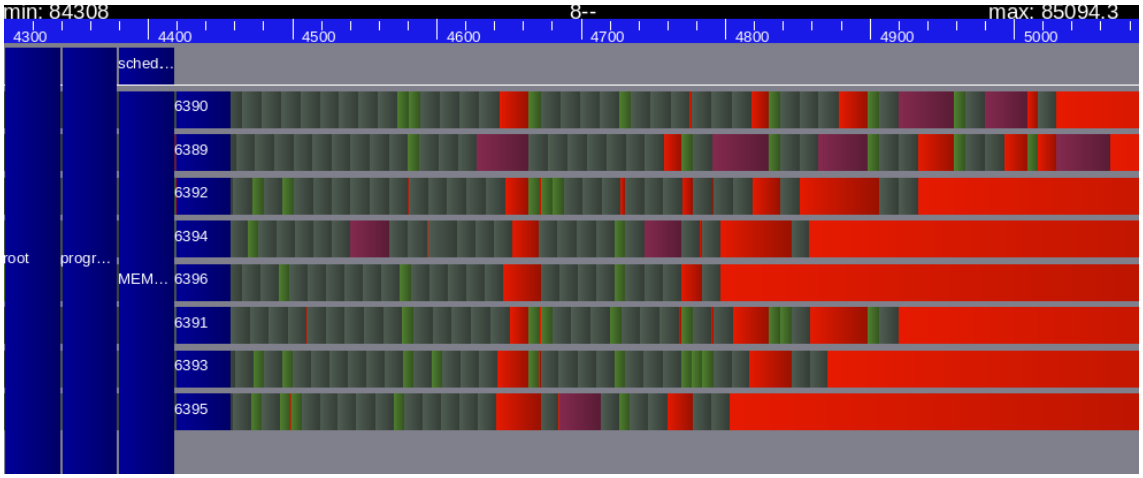

FIGURE II.11 – Equilibrage de charge original sur Cholesky tuilé - dernières 800 ms

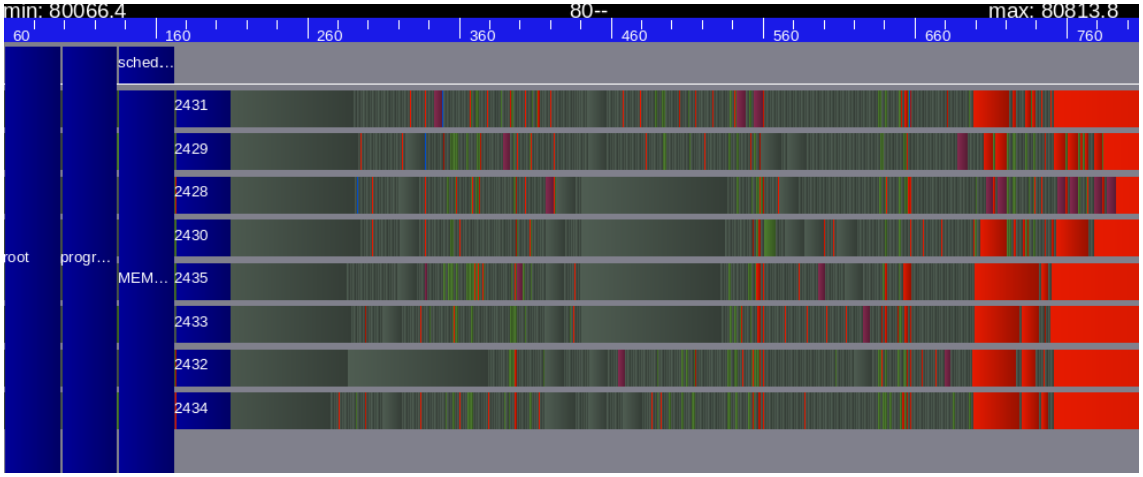

FIGURE II.12 – Equilibrage de charge avec tâches divisibles sur Cholesky tuilé - dernières 800 ms

– Grâce à cela, j'ai pu établir une courbe mettant en évidence le gain en équilibrage moyen pendant toute l'exécution :

FIGURE II.13 – Equilibrage de charge moyen durant l'exécution de Cholesky tuilé

J'ai remarqué que, lors du régime permanent, le nombre de tâches par worker se stabilise autour de 4, ce qui est le seuil d'équilibre de charge minimal défini dans les paramètres du système d'équilibrage de charge lors de cette exécution.
J'ai également remarqué que ce nombre ne dépasse jamais 8, soit le double du seuil d'équilibre, ce qui garantit un déséquilibre maximal très acceptable compte tenu de la nature de l'algorithme, qui a tendance à débloquer de grandes quantités de tâches à la fin de certaines tâches clé, notamment les POTRF.

- J'ai remarqué que le temps d'exécution des tuiles découpées exécutées séquentiellement est plus élevé que le temps d'exécution des tuiles non découpées, mais si les tuiles découpées peuvent être exécutées en parallèle, le gain est très important :

  Exemple : tâche TRSM divisée en 4 blocs, se fait en 6 tâches : 4 tâches TRSM, 2 tâches GEMM.

  - Taille : 8388608 octets de données.

    - Tâche père : 5.251836e+04 us

    - Tâches filles :

      - TRSM : 7.280215e+03 us par tâche.
      - GEMM : 1.278062e+04 us par tâche.
      - Somme des tâches filles : 5.468208e+04 us
      - Perte due au découpage : 4%
      - Gain si parallélisme : 411%

  - Taille : 2097152 octets de données.

    - Tâche père : 7.280215e+03 us

    - Tâches filles :

      - TRSM : 1.107089e+03 us par tâche
      - GEMM : 1.800407e+03 us par tâche
      - Somme des tâches filles : 8.02917e+03 us
      - Perte due au découpage : 10%
      - Gain si parallélisme : 404%

- J'ai mis en évidence qu'actuellement, il y a un seuil du système d'équilibrage de charge, une taille des blocs et une profondeur de découpage maximale optimales pour cet algorithme. Cela peut être causé par le fait que l'heuristique HEFT reste très coûteuse, et qu'il faut donc trouver un compromis entre taille de blocs, profondeur de découpage et équilibrage de charge, pour que le temps de calcul de l'heuristique ne surpasse pas le gain de performance obtenu.

- J'ai également remarqué qu'il existe un dimensionnement optimal de l'heuristique de régime permanent, car il faut assurer qu'il n'y ait pas de contensions à cause d'un nombre de tâches par worker autorisé trop faible, ou au contraire un engorgement car le nombre de tâches par worker autorisé est trop grand.

Chapitre III

Conclusion

III.1 Les tâches divisibles dans StarPU, ce qui a été fait

Durant ce stage, la notion de tâches divisibles a été implémentée dans le support d'exécution StarPU.
Cette insertion s'est faite autour de trois axes :

- L'idée de base des tâches divisibles est de permettre à l'ordonnanceur, lors de l'analyse d'une tâche, d'appeler une fonction lui donnant l'ensemble des tâches filles de la tâche courante si elle devait être découpée et, en fonction de ces informations, décider s'il faut soumettre les tâches filles selon le schéma de soumission des tâches divisibles, ou ne pas découper, et occulter ces tâches pour continuer normalement.

- Pour que cette idée soit effective, il ne faut pas se contenter de regarder la tâche courante, mais aussi dans le "futur", et analyser les tâches qui vont être ordonnancées, afin d'être sûr qu'une tâche à venir ne serait pas plus intéressante à découper que la tâche courante.
  C'est dans cette idée que la fenêtre d'ordonnancement a été mise en place : assurer un équilibrage de charge suffisant en permanence grâce au système conçu à cet effet, tout en permettant de stocker les tâches "en attente" dans la fenêtre d'ordonnancement, analysable par l'ordonnanceur afin d'effectuer les choix nécessaires pour le découpage des tâches.
  Un système de bouchon est venu compléter la fenêtre d'ordonnancement, afin de pouvoir remplir la fenêtre par paquets, et permettre de meilleurs choix de découpage.

- Pour effectuer le choix concret, deux heuristiques différentes sont utilisées :

– Une heuristique dite de "régime permanent", consistant en une affirmation : s'il reste assez de tâches dans la fenêtre d'ordonnancement pour nourrir toutes les unités de calcul, on est sûrs qu'on ne divisera pas la tâche courante, ou les tâches suivantes. On peut donc se passer de calculer si la division est intéressante ou non.
Peu coûteuse, elle permet de se passer le plus possible d'utiliser la seconde heuristique.

– Une heuristique dite "HEFT", qui consiste à simuler sur une copie locale de l'état courant de la machine une répartition selon HEFT des tâches filles, et vérifier si cette dernière est plus intéressante qu'une affectation par HEFT de la tâche père ou non.
Cette heuristique est très performante, mais également très coûteuse.

III.2 Une perspective choisie, des débouchés immenses

Dès le départ, nous avons choisi une perspective : nous avons pris le parti de ne nous occuper que du découpage de l'exécution des tâches, et d'écarter le problème du découpage de données récursif, en découpant toutes les données en grain fin, puis en donnant des paquets de données à chaque tâche.
Ceci implique que les données doivent être allouées en un bloc entier, puis partitionnées en grain fin. Cette allocation a donc un défaut : toute les données sont allouées en un bloc contigu, à un seul endroit en mémoire, à l'opposé de la version tuilée de Cholesky, qui alloue tuile par tuile, ce qui permet au système de distribuer les morceaux de la matrice sur les différents noeuds mémoire de la machine et donc gagner en performances sur les temps d'accès mémoire.
Cette perspective soulève donc son lot de débouchés, car le découpage récursif des données devrait permettre une allocation distribuée de la matrice, et donc permettre de bons gains de performances.

Actuellement, seule la tâche courante est analysée pour savoir s'il faut la découper ou la conserver à la granularité actuelle, sans regarder ce qui va arriver après. On pourrait commencer à regarder le "futur" dans la file d'attente de la fenêtre d'ordonnancement, mais l'heuristique HEFT est déjà très coûteuse actuellement.
Si l'heuristique HEFT est suffisamment améliorée au niveau des performances, il pourrait être possible d'améliorer drastiquement les performances en permettant de regarder le "futur", mais aussi en permettant de découper plus loin sans être limité par le coût de l'heuristique, ou moins en tout cas.

Le débouché le plus intéressant serait une conception nouvelle des algorithmes :

par exemple pour Cholesky, on ne soumettrait plus que la tâche principale avec la matrice à calculer, et le support d'exécution découperait le problème tout seul à la granularité optimale pour l'équilibrage de charge de la machine sur laquelle l'exécution se déroule, et ce durant toute l'exécution, sans avoir besoin de s'occuper des problèmes d'optimisation de la granularité, de la génération d'un graphe de tâches, etc.

Ce stage a permis d'ouvrir des portes vers cet idéal.

Table des figures