Université de Bordeaux : Sciences et Technologies

Rapport de stage optionnel de Master 1

Ordonnancement d'applications de type stencil sur cluster hybrides CPU/GPU

Loris Lucido

Supervisé par :
Olivier Aumage, Samuel Thibault

Équipe Storm, Inria - LaBRI, centre de recherche Inria Bordeaux - Sud-Ouest

9 septembre 2016

Table des matières

Remerciements

Je tiens à remercier mes Maîtres de stage Olivier Aumage et Samuel Thibault pour leurs conseils, l'aide apportée, leurs explications détaillées ainsi que leur relecture. Je remercie aussi Jérome Clet-Ortega et Raymond Namyst de m'avoir accueilli dans leur bureau. Enfin, je remercie le centre INRIA et en particulier l'équipe *STORM* pour le bon accueil qui m'a été fait.

L'équipe *STORM* Inria Sud-ouest

L'INRIA (Institut Nationnal de Recherche en Informatique et en Automatique) est un centre de recherche français basé dans huit villes en France : Bordeaux, Grenoble, Lille, Nancy, Paris, Rennes, Saclay, Sophia.

L'équipe *STORM* (*STatic Optimizations, Runtime Methods*) fait parti du centre INRIA Bordeaux Sud-ouest et effectue des travaux de recherche dans le domaine du Calcul Distribué et Haute Performance.

1 Introduction

1.1 Le domaine du Calcul Haute Performance

Dans le domaine du Calcul Haute Performance, il est d'usage d'utiliser plusieurs unités de calculs en parallèle afin de traiter des problèmes de plus en plus rapidement. Ces unités de calculs peuvent être des CPUs (processeur) ou des GPUs (carte graphique). Dans la suite de ce rapport, nous appellerons *ouvrier* une unité de calcul (CPU ou GPU).

Afin d'utiliser au mieux ce matériel, il convient d'utiliser des paradigmes de programmation adaptés. Une méthode qui a déjà était le sujet de nombreuses recherches est la programmation en tâches. Une tâche représente un sous-ensemble du problème à résoudre. Programmer avec des tâches consiste donc à construire plusieurs sous-ensembles du problème et définir comment chaque sous-ensemble est lié aux autres sous-ensembles à l'aide de dépendances de tâches. Nous obtenons alors un graphe de dépendances de tâches. Un exemple est illustré figure 1.1. Enfin nous les donnons à résoudre à des ouvriers différents, dans un ordre qui dépend du graphe de tâches construit.

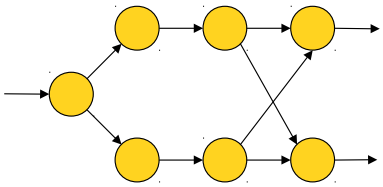


Figure 1.1 – Graphe de dépendances de tâches.

1.2 Qu'est-ce qu'une application stencil ?

Il s'agit d'une classe d'algorithmes beaucoup utilisé en simulation (telle que simulation de gaz ou de propagation de chaleur). Un tel algorithme permet généralement de discrétiser des phénomènes physiques locaux sur des grilles régulières.

La dimension temporelle est exprimée par des itérations. Une itération consiste à appliquer une petite fonction sur chaque cellule d'une grille. Cette fonction ayant la particularité de mettre à jour le contenu d'une cellule en fonction de son voisinage. Un stencil n-points est un stencil où la mise à jour

d'une cellule dépend de (n-1) cellules voisines. Un stencil 5-points est illustré figure 1.2.

Pour résoudre un problème stencil à l'aide de tâches, nous pouvons regrouper plusieurs mises à jour de cellules voisines en une tâche. Il faut alors regrouper suffisamment de mises à jour pour que cette tâche représente un calcul conséquent.

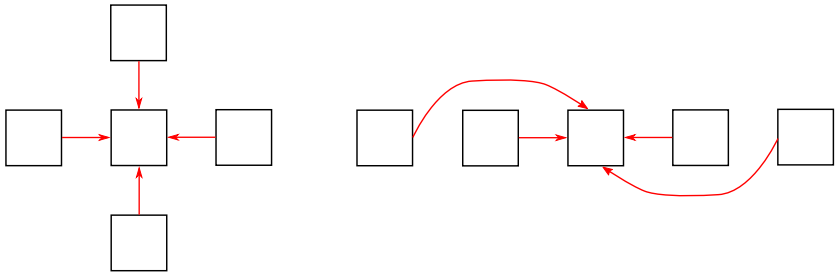


Figure 1.2 – Stencil 5-points en 1D (droite) et 2D (gauche) - La mise à jour d'une cellule se fait ici à partir des 4 cellules voisines.

1.3 Sujet d'étude et plan

Un support d'exécution est une couche logicielle permettant de traiter des problèmes de Calcul Haute Performance de manière générique. Au cours du stage nous avons cherché à savoir si *StarPU*, le support d'exécution développé dans l'équipe *STORM*, non spécifique aux stencils, peut aussi traiter des applications stencil efficacement sur des cartes graphiques dédiées au calcul. Nous souhaitons savoir s'il est capable de bien ordonnancer des tâches stencils, c'est-à-dire de choisir la bonne carte graphique pour la bonne tâche.

Pour résoudre ce problème, nous allons considérer les données utilisées par chaque tâche, puis vérifier que *StarPU* exécute le maximum de tâches possibles liées à un petit ensemble de données (localité des données). Ceci devrait permettre de réduire les transferts (coûteux) de données entre plusieurs cartes graphiques. Nous exploiterons l'ajout d'information dans l'application pour détecter et observer la localité des données.

Dans la section 2 nous discuterons du contexte du stage et ferons un bref État de l'art. Dans la section 3 nous détaillerons les travaux principaux réalisés durant le stage. Dans la section 4 nous évaluerons l'ordonnancement du support d'exécution *StarPU* lorsque confronté à une application stencil.

Enfin, dans la section 5 nous résumerons les points importants du stage et nous discuterons des perspectives qui en résultent.

2 Contexte

Dans cette section, nous présentons le contexte du stage. Nous revenons plus en détail sur la programmation parallèle en tâches au sein de supports d'exécution. Nous discuterons des particularités des applications stencil. Nous terminerons par un État de l'art dans le domaine.

2.1 Ordonnancer un ensemble de tâches

Ordonnancer une tâche signifie demander à un ouvrier (une unité de calcul) d'exécuter cette tâche. L'objectif étant de minimiser le temps de calcul global de toutes les tâches, une question récurrente lorsqu'on programme des tâches est de savoir à qui demande-t-on de calculer une tâche parmi tous les ouvriers disponibles. Lorsque l'on souhaite n'utiliser que des CPUs, une solution consiste à laisser l'ordonnancement des tâches au système d'exploitation. Cette solution perd de son sens lorsqu'on y ajoute un ou plusieurs GPUs. De plus, les politiques d'ordonnancement du système d'exploitation ne sont jamais adaptées à l'application.

Une autre solution consiste à laisser l'application assigner, une à une, les tâches à un ouvrier. Enfin, et c'est le sujet de ce stage, nous pouvons laisser l'ordonnancement à des supports d'exécution ou *runtime* qui vont décider pour chaque tâche sur quel ouvrier il convient de la calculer.

L'objectif du stage consiste à étudier l'ordonnancement d'application de type stencils dans un support d'exécution. Nous étudierons en particulier l'ordonnancement de tâches dans le support d'exécution *StarPU*, développé dans l'équipe *STORM* Inria Sud-ouest. Les applications de type stencil ayant des particularités que nous discuterons section 2.4.1, il est intéressant d'évaluer si les outils déjà disponibles dans le support d'exécution *StarPU* permettent de traiter de telles applications ou dans le cas contraire, d'identifier quels seraient les ajouts nécessaires.

2.2 Architecture hétérogène

Les machines de calculs sont souvent composées de dizaines de CPUs (*Central Processing Unit*) et d'une, deux voir trois cartes graphiques. On parle alors d'architecture hétérogène. Les GPUs (*Graphics Processing Unit*) sont les ouvriers des cartes graphiques. Ils sont généralement bien plus rapide que des CPUs pour certaines tâches comme le graphisme. Ils sont aussi beaucoup utilisés en Calcul Haute Performance pour faire du calcul généraliste.

Comme illustré figure 2.1, chaque carte graphique dispose de sa propre mémoire. Celle-ci est cependant plus limitée en volume que la mémoire principale, soit quelques gigaoctets pour une carte graphique contre plusieurs dizaines de gigaoctets pour la mémoire principale. De plus, les ouvriers d'une carte graphique ne peuvent pas directement avoir accès à la mémoire principale. Lorsque l'on souhaite accéder à une donnée, il faut la copier depuis la mémoire principale vers la mémoire de la carte graphique. La latence étant plus élevée qu'un accès par un CPU.

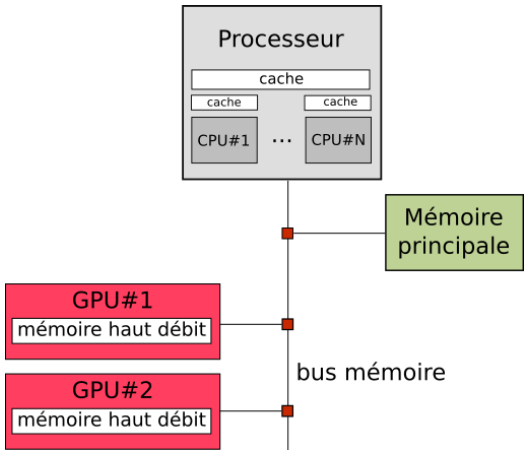


Figure 2.1 – Topologie mémoire d'une machine avec deux cartes graphiques.

Cette latence étant un facteur temporel limitant important, il convient de réguler au mieux les transferts vers les mémoires des GPUs. Pour cela, nous pouvons essayer de :

— limiter l'occupation du bus mémoire en réduisant les transferts au strict minimum
— limiter le temps passé à attendre qu'une donnée arrive en mémoire en

faisant parallèlement des transferts et des calculs. On parle alors de recouvrement des transferts mémoires.

2.3 Support d'exécution

Un support d'exécution est un composant qui vient se placer entre le système d'exploitation et l'application comme illustré figure 2.2. Son rôle est d'ordonnancer finement les processus parmi les différents ouvriers. Pour ce faire, il doit demander au système d'exploitation de laisser le support d'exécution se charger de l'ordonnancement des processus. Il joue aussi le rôle d'interface avec les accélérateurs GPU.

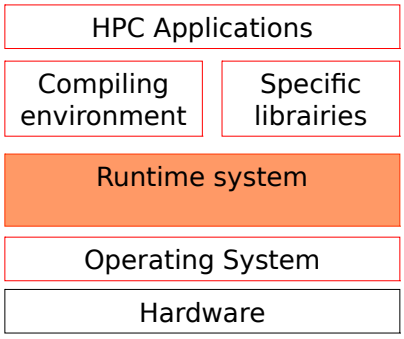


Figure 2.2 – Hiérarchie logicielle et support d'exécution.

L'application doit construire un ensemble de tâches pour résoudre un problème, et soumettre toutes ces tâches au support d'exécution. Ce dernier peut ainsi décider de la répartition des tâches de calcul sur les ouvriers à disposition. Pour prendre ces décision, il utilise des politiques d'ordonnancement ou ordonnanceurs. Nous détaillons quelques exemples section 3.2.

2.4 Applications stencils

Pour notre étude, nous utiliserons comme stencil un prototype jouet simple afin de faciliter l'analyse des expériences.

2.4.1 Schéma de calcul particulier des stencils

Les nombreux accès aux cellules voisines lors d'une mise à jour peuvent avoir comme principale conséquence un ratio accès mémoire par mise à jour

de cellule très élevé. Le facteur temporel limitant n'est alors plus le temps de calcul mais le temps d'accès mémoire des cellules voisines. De plus, il devient alors difficile de recouvrir totalement les temps de transferts entre les différents noeuds mémoires par du calcul. Il faut alors faire de la réutilisation des données.

2.4.2 Économiser de la bande passante

Nous nous intéresserons surtout aux problèmes de taille supérieure à la mémoire disponible. Ainsi, il ne sera pas possible de stocker l'intégralité des données du problème étudié en mémoire GPU, il faudra donc évincer régulièrement des données de la mémoire GPU et faire des transferts depuis la mémoire principale, ce qui a un coût.

Pour limiter le volume de données transférées, il est nécessaire de réutiliser les données déjà présentes en mémoire plutôt que d'en transférer de nouvelles. Pour cela, nous pouvons travailler sur la **localité en espace** des données en se basant sur l'affirmation suivante : si un emplacement mémoire est référencé, il est très probable que les emplacements mémoires voisins soient aussi référencés dans un futur proche. Cette affirmation se vérifie pour les stencils lors de la mise à jour d'une cellule.

De plus, du à la nature algorithmique itérative d'un stencil, chaque emplacement mémoire est accédé à chaque nouvelle itération sur le stencil. Nous aimerions alors travailler sur la **localité temporelle** des données. Ainsi, plutôt que de mettre à jour tous les éléments les uns à la suite des autres, nous pouvons faire évoluer un petit morceau du domaine. Cela en traitant plusieurs itérations de manières successives, comme illustrée figure 2.3, pour un stencil 3-points.

2.5 Répartition de charge

Pour certain type d'applications stencil comme la prévision météo, il est courant d'utiliser des heuristiques plus ou moins précises (et donc coûteuses) selon le domaine. Par exemple, on utilisera une heuristique plus approximative pour prévoir le temps qu'il fera sur une région ensoleillée que sur une région nuageuse.

Dans la section 4.5.3 nous évaluerons les performances de certaines méthodes face à une répartition de charge déséquilibrée. Nous regarderons en particulier une répartition de charge statique (où le déséquilibre est fixée au

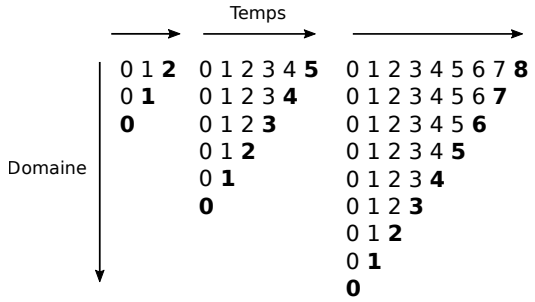


Figure 2.3 – Exemple d'évolution d'un morceau de domaine pour un stencil 1D. Les chiffres représentent le numéro d'itération. On progresse en diagonale sur un petit morceau du domaine, plutôt que de traiter une itération complète d'un seul coup.

départ) et dynamique (où le déséquilibre évolue au cours du temps, comme cela peut-être le cas pour la prévision météo).

2.6 État de l'art

2.6.1 Les contraintes des applications stencils

Dans le domaine du Calcul Haute Performance, le paradigme de programmation parallèle en tâches est très utilisé. Cependant, pour des machines ayant une architecture hétérogène, il peut devenir très complexe de trouver la meilleure répartition des tâches sur les différents ouvriers. Pour pallier ce problème, les supports d'exécution sont une solution. Ils essayent alors de prendre les décisions d'ordonnancement appropriées.

La localité des données étant un facteur temporel réducteur important pour les applications stencils, il est intéressant d'étudier quelles peuvent être les décisions prises par le support d'exécution *StarPU*, développé par l'équipe *STORM*, et quelles améliorations nous pouvons envisager.

2.6.2 Solutions existantes au problème

Le domaine des stencils reste un problème très étudié en calcul haute performance. Cependant, la répartition des calculs est en général fait à la main. Des travaux ont aussi portés sur des algorithmes caches oublieux ou *oblivious* adaptés aux stencils [1]. Ceux-ci tentent de profiter du cache sans

en connaître la taille. Pour cela, on favorise la réutilisation des données. Cela nécessite aussi une charge de travail supplémentaire pour l'application.

De précédents travaux dans l'équipe *STORM* portant sur des applications d'algèbre linéaire dense, creuse et compressée ont montré que le support d'exécution *StarPU* pouvait apporter des performances similaires qu'un code écrit spécifiquement. En revanche, peu de travaux ont été réalisés dans l'équipe concernant les stencils. Nous voulons savoir si les supports d'exécution peuvent traiter ce problème de manière **générique**.

Pour s'assurer que le support d'exécution prend des décisions d'ordonnancement favorisant la localité, nous avons besoins d'outils de visualisation adaptés au problème. Ces outils doivent nous permettre d'évaluer les ordonnanceurs et donc éventuellement de proposer des améliorations.

3 Contribution du stage

Dans cette section nous ferons un état des lieux des propriétés de recouvrement des transferts mémoires et de prise en compte de la localité des données dans les ordonnanceurs de *StarPU*. Nous les évaluerons section section 4. Afin de pouvoir les évaluer, nous discuterons d'une méthode de référence, puis détaillerons les outils de visualisation développés qui nous permettent d'observer les décisions prises par les ordonnanceurs pour une application stencil.

3.1 Résumé de la contribution

Nous avons étudié le comportement du support d'exécution *StarPU* pour des applications stencils. Pour les évaluer, nous les comparons avec une méthode très spécifique aux stencils (et donc non générique) qui nécessite des changements dans l'application : il s'agit de l'algorithme de soumission de tâches cache oublieux.

Pour comprendre les performances obtenues avec *StarPU*, nous avons besoin d'observer la localité en espace et en temps des données. Ceci permettant de réduire le volume des transferts mémoires. Nous avons écrit des outils de visualisation adaptés détaillés section 3.4.

Enfin, nous avons assemblé un ordonnanceur simple adapté aux applications de type stencil détaillé section 3.2.5.

3.2 Ordonnanceurs : recouvrir les transferts mémoires et localité des données

3.2.1 Politique d'éviction des données en mémoire

En plus des politiques d'ordonnancement, *StarPU* dispose aussi d'une politique d'éviction des données. Lorsque la mémoire GPU commence à devenir pleine, il faut rapatrier des données en mémoire principale. Pour cela, une politique *LRU* ou *Least Recently Used* est utilisée. Celle-ci garde les données les plus récemment référencées et évince les plus anciennement référencées.

3.2.2 *Eager* (`eager` et `prio`)

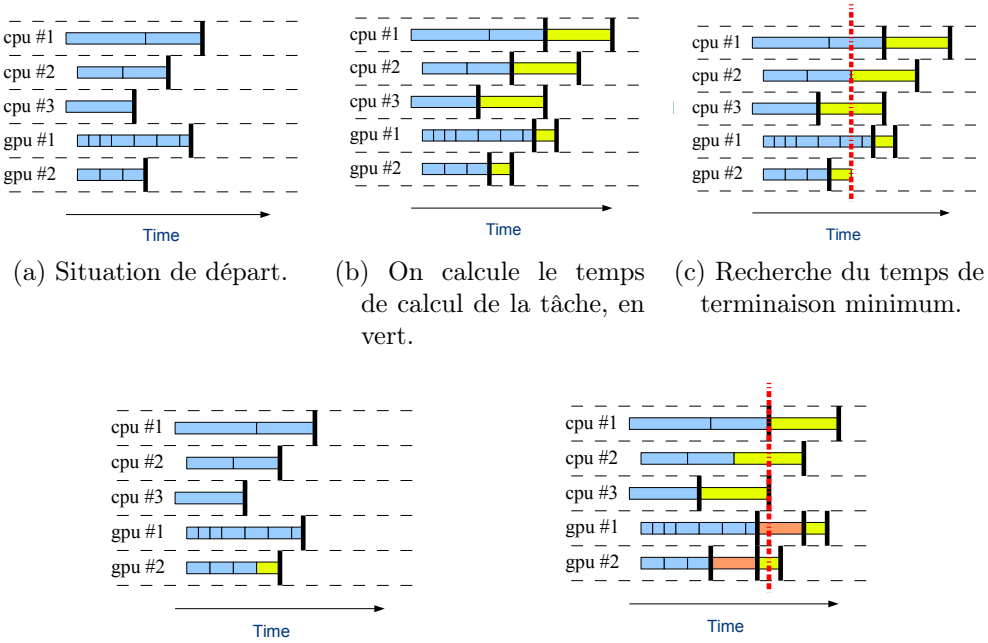
L'ordonnanceur `eager` utilise une seule liste centralisée pour tous les ouvriers. Il distribue les tâches aux ouvriers suivant un ordonnancement tourniquet ou *round-robin*. Il existe une variante de `eager` appelée `prio`. Celle-ci prend en compte les priorités que l'application peut assigner aux tâches. Elle ordonnance les tâches à priorité élevée en premier.

Cet ordonnanceur ne travail pas du tout sur la localité des données. Cependant, en utilisant cet ordonnanceur, nous pouvons feindre de la localité des données depuis l'application. Pour cela nous attribuons une priorité croissante avec le numéro d'itération d'une tâche. Nous espérons ainsi que l'ordonnanceur ait tendance à faire évoluer des groupes de cellules proches comme illustré sur la figure 2.3. Nous vérifions cela section 4.6.

3.2.3 *Deque Model Data Aware* (`dmda` et `dmdar`)

Chaque ouvrier dispose de sa propre file de tâches. Lorsque l'ordonnanceur `dmda` souhaite assigner une tâche, il calcule pour chaque ouvrier une date de terminaison de la tâche. Cette date dépend du nombre de tâches pas encore terminées présentes dans la file, de la durée de la tâche et de la vitesse de calcul de l'ouvrier. `dmda` prend aussi en compte la durée nécessaire au transfert des données de la tâche comme illustrée figure 3.1. Cela réduit les transferts entre mémoire GPU et mémoire principale.

`dmdar` est une variante de `dmda`. Chaque ouvrier va sélectionner en premier, dans sa file, les tâches ayant le plus de données déjà disponibles dans sa mémoire. Cela permet de maximiser le recouvrement des transferts par du calcul.



(a) Situation de départ.

(b) On calcule le temps de calcul de la tâche, en vert.

(c) Recherche du temps de terminaison minimum.

(d) Le GPU 2 semble être le meilleur candidat.

(e) Cette fois-ci, on prend en compte les temps de transferts, en rouge. Le meilleur candidat est donc le CPU 3.

Figure 3.1 – Ordonnancement d'une tâche pour l'ordonnanceur `dmda`.

3.2.4 *Locality Work Stealing* (`lws`)

L'ordonnanceur `lws` distribue les tâches aux différents ouvriers en fonction de la charge (durée de la tâche) et de la localité des données. Un ouvrier pourra faire du vol de travail s'il se retrouve inactif. Il pourra alors aller prendre une tâche d'un ouvrier voisin déjà bien chargé. L'ordonnanceur `lws` est illustré figure 3.2.

De plus, lorsqu'un ouvrier doit prendre une tâche dans une liste de tâches (la sienne où celle d'un autre), il choisira en priorité une tâche dont les données se trouvent déjà dans la mémoire de l'ouvrier.

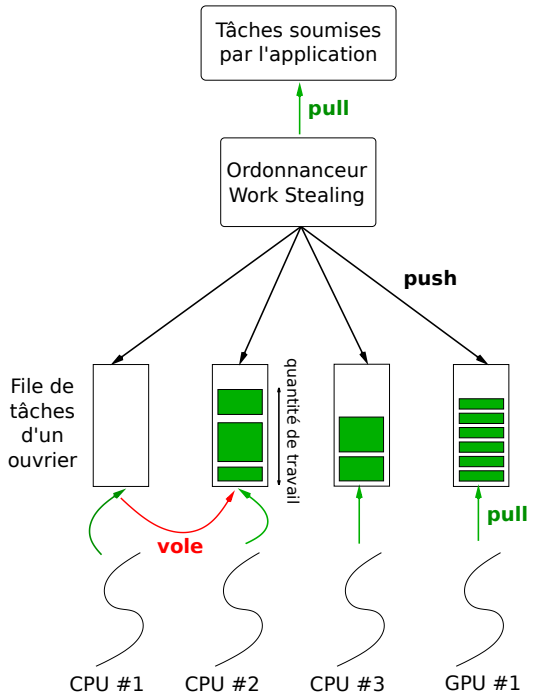


Figure 3.2 – Ordonnanceur `lws` (*Locality Work Stealing*).

3.2.5 *Modular Heft* (`modular-ready`)

Cet ordonnanceur est illustré figure 3.3. Sa particularité est qu'il s'agit d'un ordonnanceur modulaire, construit par assemblage de composants. Ainsi, il est aisé de construire un nouveau ordonnanceur en ajoutant ou modifiant

un composant. Notre contribution a été de remplacer les composants racines et feuilles par des composants *tâches prêtes*.

Le principe est le suivant. Chaque ouvrier dispose de sa file de tâche. Tant qu'il reste des tâches dans sa file, on récupère une tâche prête dans la liste et on l'exécute. Une tâche prête est une tâche dont les données se trouvent déjà en mémoire pour un noeud donné. Le recouvrement des transferts mémoires est donc crucial ici.

Lorsque la liste est vide, on va aller notifier le composant racine. Celui-ci va alors essayer de pousser des tâches dont les données se trouvent déjà dans la mémoire de cet ouvrier.

Le composant *mct* (*Minimum Completion Time*) récupère les tâches poussées par la racine, et estime une date de terminaison de la tâche pour chacun de ses composants fils (les composants ouvriers). Cette date dépend de la durée de la tâche, de la vitesse de l'ouvrier, de la durée des éventuels transferts et du nombre de tâches dans la file de l'ouvrier. Il sélectionne la date la plus proche dans le future et envoie la tâche à l'ouvrier correspondant.

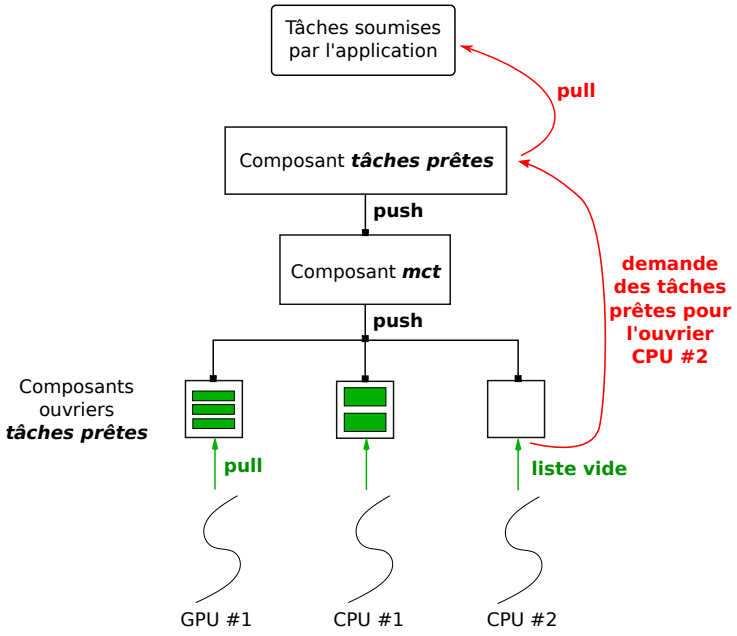


Figure 3.3 – Ordonnanceur `modular-ready`.

3.3 Méthode de référence : algorithme cache oublieux de soumission de tâches

Pour évaluer les ordonnanceurs du support d'exécution *StarPU*, nous avons besoin d'une méthode de référence. Pour cela, nous utilisons le résultat de précédentes recherches portant sur un algorithme cache oublieux adaptés au stencil [1]. Nous avons implémenté cet algorithme tel qu'il y est décrit.

Cet algorithme décrit un ordre dans lequel faire les mises à jour des cellules qui favorise la réutilisation des données. Nous n'utilisons pas d'ordonnanceurs et laissons l'application décider quelle tâche sera exécutée sur quel ouvrier et dans quel ordre. Nous soumettons les tâches dans l'ordre dicté par l'algorithme cache oublieux afin de limiter le volume de données transférées. Il devrait s'agir d'une méthode proche de l'idéal lorsque le problème étudié est trop grand pour rentrer en mémoire GPU.

L'algorithme ayant été pensé pour une exécution en séquentiel, il a fallu l'adapter pour une exécution en parallèle. Pour cela, nous découpons le problème en autant de sous-domaines que nous disposons d'ouvriers. Nous appliquons un algorithme cache oublieux de manière locale pour chaque sous-domaine. Nous soumettons donc des tâches en parallèle.

Pour que le calcul du stencil soit correct, il faut respecter les dépendances de tâches. Comme les tâches sont soumises en parallèle il faut faire attention à ce que les algorithmes cache oublieux, local à un ouvrier, soient synchronisés.

Pour cela, il ne faut pas que deux cellules frontières entre deux GPUs soient à plus de deux itérations de décalage. Nous utilisons des sémaphores pour bloquer l'exécution de tâches d'un GPU lorsque les cellules frontières du GPU voisin ne sont pas à la même itération (voir figure 3.4).

Dans la suite du rapport, nous référerons à cette algorithme parallèle de soumission de tâches cache oublieux par simplement `cache-oublieux`.

3.4 Outils de visualisation

Maintenant que nous avons une méthode de référence, il faut pouvoir comprendre ce que les ordonnanceurs font de bien ou mal. L'outil *StarPU* dispose déjà d'outils d'analyse de traces qui permettent d'observer si un ouvrier est actif ainsi que les transferts mémoires, et cela pour chaque instant de l'exécution. Cela ne nous suffit pas, en effet, nous voulons observer la localité des données.

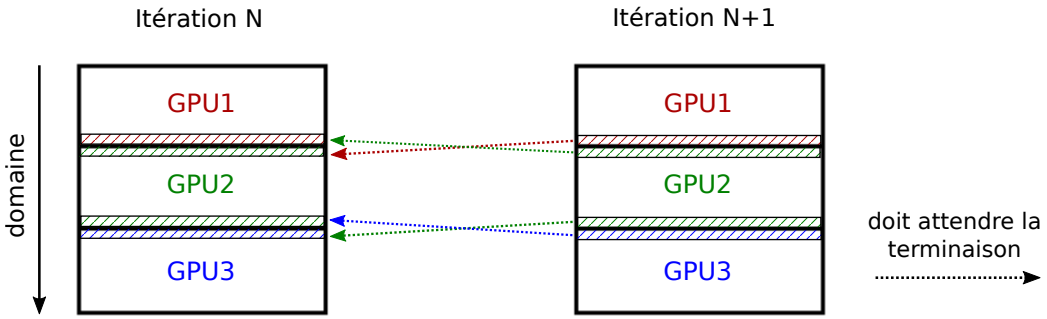


Figure 3.4 – Dépendances croisées à la frontière du domaine assigné aux ouvriers lors d'une soumission de tâches en parallèle pour `cache-oublieux`.

Pour arriver à nos fins nous avons besoin de l'aide de l'application. *StarPU* n'a aucun moyen de savoir de lui-même quelles sont les données dont nous souhaitons observer de la réutilisation. L'application doit donc ajouter un drapeau "localité" sur chaque donnée. En pratique, très peu de modifications sont à faire.

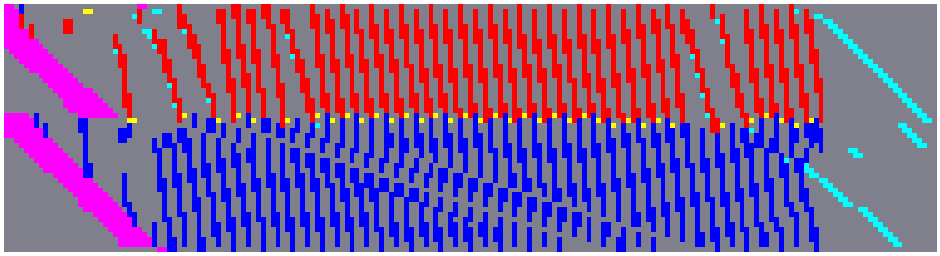
3.4.1 Diagramme d'exécution : domaine en fonction du temps

Un exemple d'un tel diagramme est donné figure 3.5. Sur ce diagramme nous pouvons observer quel ouvrier (couleur rouge et bleu) a travaillé sur quel morceau du domaine (axe des ordonnées) et à quel instant dans l'exécution (axe des abscisses). Nous pouvons aussi voir les transferts mémoires (couleur cyan, magenta et jaune).

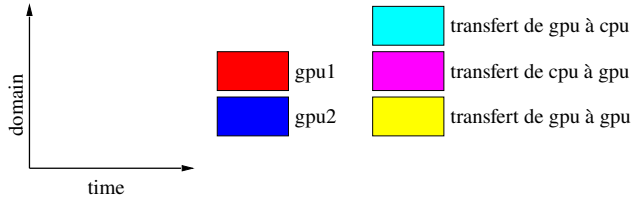
Ce diagramme nous permet de voir dans un premier temps quel est la répartition du domaine faite par l'ordonnanceur. Sur cet exemple, la répartition est très régulière. Cela nous permet aussi de voir dans quel ordre sont traitées les différentes tâches et les différentes itérations. En particulier, si l'ordonnanceur fait de la réutilisation des données en traitant plusieurs itérations successives d'un morceau de domaine, nous devrions voir des points de la couleur d'un ouvrier progresser à l'horizontale.

3.4.2 Diagramme d'exécution : domaine en fonction des itérations

Un exemple d'un tel diagramme est donné figure 3.6. Sur ce diagramme nous pouvons observer quel ouvrier a travaillé sur quel morceau du domaine



(a) `dmdar` - Exécution avec deux GPUs, pas de limite mémoire.



(b) Légende.

Figure 3.5 – Exemple de diagramme d'exécution des tâches : domaine en fonction du temps.

et à quelle itération du stencil. En noir sont aussi affichées des courbes isochrones. Elle permettent d'intégrer une dimension temporelle dans le diagramme. Tout ce qui se trouve entre deux courbes isochrones à la verticale s'est déroulé durant le même laps de temps. Sur l'exemple de la figure 3.6, la durée entre deux courbes isochrones est de 100ms.

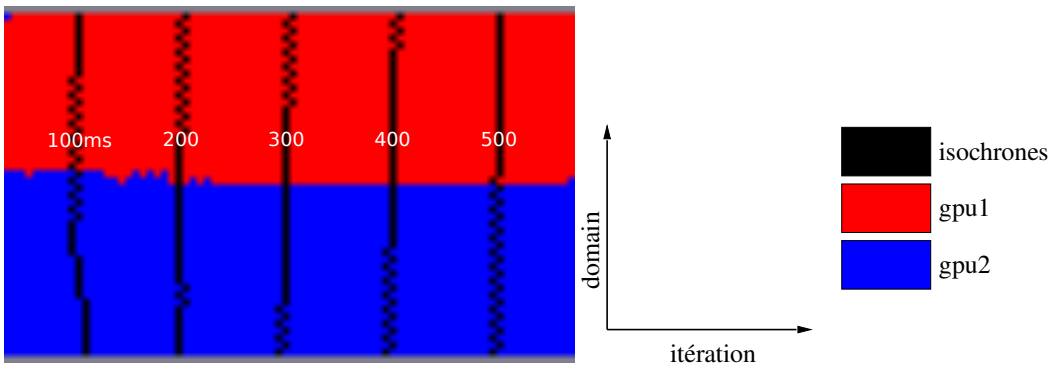
Ce diagramme à l'avantage par rapport au précédent d'avoir une vision précise de quel ouvrier travaille sur quelle itération du stencil. Nous aimerions observer des courbes isochrones formant des motifs triangulaires, signe de localité comme expliqué figure 2.3.

4 Évaluation

Lorsque nous évaluons un ordonnanceur, nous pouvons considérer trois métriques :

— le temps d'exécution ;
— le surcoût lié au temps passé à ordonnancer les tâches ;
— le volume des transferts vers mémoire GPU.

Nous rappelons que nous nous intéresserons en particulier aux tailles de problèmes trop grandes pour être stockées entièrement en mémoire.



(a) `dmdar` - Exécution avec deux GPUs, pas de limite mémoire.

(b) Légende.

Figure 3.6 – Exemple de diagramme d'exécution des tâches : domaine en fonction des itérations.

4.1 Le programme de test utilisé : un stencil 1D

Nous utilisons un stencil jouet 1D 3-points pour faciliter les tests et l'analyse des traces des outils de visualisations. Pour rappel, un stencil 3-points est un stencil où la mise à jour d'une cellule dépend de 3 cellules voisines.

Les temps de calcul et les transferts mémoires sont simulés avec *SimGrid*, un simulateur d'application de Calcul Haute Performance décrit section 4.2. Pour obtenir des tâches conséquentes sur notre stencil 1D factice, nous spécifions statiquement la durée d'une tâche ainsi que la taille des données d'une tâche. Nous ajustons ces deux métriques pour que la durée de transfert des données d'une tâche soit égale à deux fois le temps de calcul d'une tâche, ce qui est raisonnable pour un stencil.

4.2 Expérimentation simulées à l'aide de *SimGrid*

SimGrid est un composant logiciel qui permet de simuler des expériences dans le domaine du Calcul Haute Performance ou dans le *cloud computing* (informatique en nuage). Toutes les variables d'une expérience peuvent être simulés : architecture de la machine ou du réseau de machines, temps de calculs, temps de transferts mémoires, etc.

Cela permet de tester rapidement des expériences même extrême (taille du problème très grand par exemple). De plus, la simulation apporte un

aspect reproductible et déterministe, ce qui n'est pas forcément le cas pour certaines heuristiques.

La validation de ces expériences sur de vraies machines, en utilisant un stencil moins factice que celui utilisé, ne sera pas faite durant le stage.

4.3 Amélioration de notre algorithme parallèle cache-oublieux

Sur le diagramme de la figure 4.1a, nous pouvons voir un exemple d'exécution de `cache-oublieux` sur 1 GPU. Nous pouvons observer la réutilisation des données que fait l'algorithme en faisant évoluer au maximum un morceau de domaine avant de passer à la suite du domaine. C'est ce phénomène que nous aimerions observer pour les ordonnanceurs de *StarPU*.

Une exécution de notre `cache-oublieux` en parallèle est montré sur le diagramme de la figure 4.1b. Nous pouvons voir ici un problème de conception pour notre version parallèle du parcours cache oublieux. En effet, le deuxième GPU, en bleu, doit attendre que le GPU rouge traite la partie du domaine frontière avec le GPU bleu (illustré par la flèche en noir). En effet, pour calculer la n-ième itération de sa première cellule, le GPU bleu a besoin que la (n-1)-ième itération de la dernière cellule assignée au GPU rouge soit terminée. Nous avons utilisé des sémaphores pour nous assurer que le calcule reste correct (illustré figure 3.4).

Une solution consiste à inverser l'ordre cache oublieux pour le deuxième GPU. Sur la figure 4.1c, nous pouvons voir que les deux GPU progressent à la même vitesse. Cette solution est facilement généralisable pour un nombre arbitraire de GPUs.

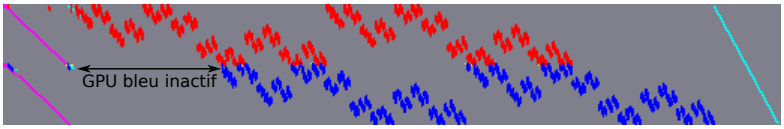
Un diagramme en itération de `cache-oublieux` est disponible en annexe figure A2.

4.4 Résultats d'expériences pour tous les ordonnanceurs

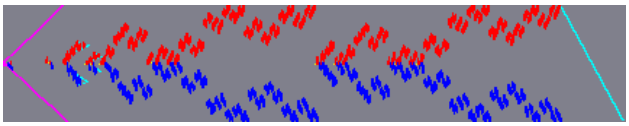
La figure 4.2 montre une simulation de 100 itérations de notre stencil, sur 1 GPU. La figure 4.3 montre la même simulation mais sur 2 GPUs.

Nous observons que `dmda` est environ 6 fois plus lent avec un GPU que `cache-oublieux` et 10 fois plus lent avec deux GPUs. Cependant, `dmdar` qui favorise l'exécution de tâches dont les données se trouvent déjà en mémoire, est *seulement* 1.5 fois plus lent environ avec deux GPUs. Nous voyons bien

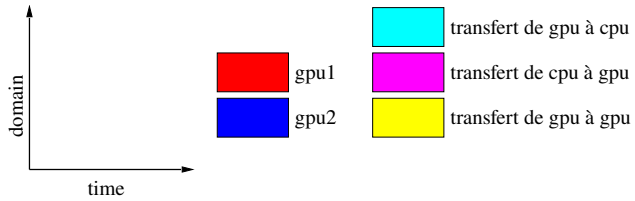
(a) `cache-oublieux` - 1 GPU, pas de limite mémoire.



(b) `cache-oublieux` - 2 GPU, pas de limite mémoire.



(c) `cache-oublieux`, version miroir - 2 GPU, pas de limite mémoire.



(d) Légende.

Figure 4.1 – Soumission de tâches via un parcours cache-oublieux.

ici l'importance de la réutilisation des données afin d'éviter des transferts inutiles.

L'ordonnanceur `modular-ready` est, lui aussi, proche de notre méthode de référence (1.5 fois plus lent) pour l'exécution à un GPU. L'implémentation de l'ordonnanceur `modular-ready`, décrit section 3.2.5, n'a pas encore été écrite pour fonctionner avec plus de un composant ouvrier (un GPU). Il n'apparaît donc pas sur la figure 4.3.

L'ordonnanceur `lws` est lui aussi très lent. Le principal facteur est que des transferts mémoires sont fait quasiment pour chaque tâche. Il n'y a donc pas de réutilisation des données. Il devient impossible de recouvrir les transferts par du calcul car, pour le problème étudié, le transfert est deux fois plus long que le calcul. Un échantillon d'exécution est donné en annexe figure A1.

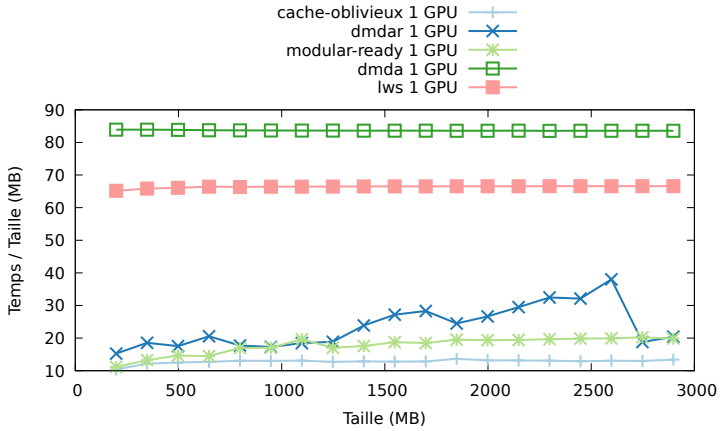


Figure 4.2 – Évolution du temps d'exécution en fonction de la taille du domaine - 1 GPU, limite mémoire fixée à 200MB par GPU.

4.5 Étude de `dmdar`

Nous avons vu dans la section 4.4 que `dmdar` était *seulement* 1.5 fois plus lent que notre méthode de référence. En regardant le volume de données transférées au total de plusieurs exécutions, nous avons relevé que `dmdar` transfère deux à cinq fois plus de données que `cache-oblivieux` (pour une limite mémoire fixée à 200MB et une taille de données variant entre 500MB et 6GB). C'est un facteur qui pourrait être amélioré. Cependant, `dmdar` compense ce problème par du recouvrement de transferts comme justifié section

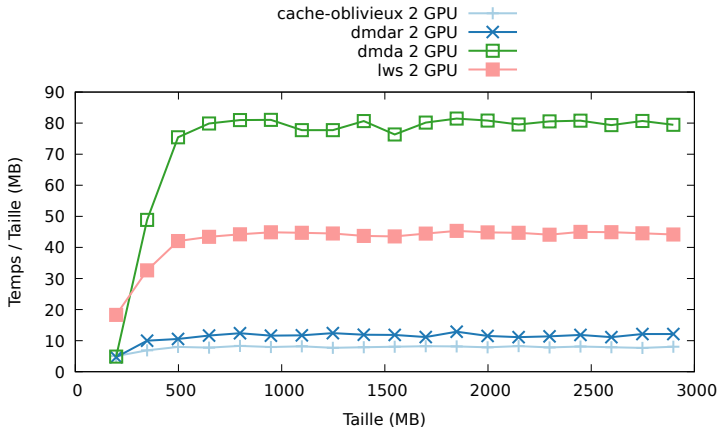


Figure 4.3 – Évolution du temps d'exécution en fonction de la taille du domaine - 2 GPU, limite mémoire fixée à 200MB par GPU.

4.5.1.

4.5.1 Formule du calcul de la durée d'une tâche

Lorsque nous souhaitons estimer le temps de d'exécution T d'une tâche sur un ouvrier n nous utilisons la formule simple suivante : $T = \alpha T_c + \beta T_t$ où T_c est le temps de calcul de la tâche et T_t est le temps de transfert de toutes les données de la tâches vers le noeud mémoire n.

Les constantes α et β peuvent être modifiées pour laisser plus de souplesse à *StarPU* lors de l'ordonnancement. En particulier, nous avons testé des valeurs raisonnables de β qui réduisent de moitié ou doublent le temps de transfert estimé. Nous avons pu observer que peu importe l'attention porté à la durée des transferts, les performances de `dmdar` restent identiques. Cela montre bien que `dmdar` effectue beaucoup de recouvrement des transferts par du calcul.

4.5.2 Mise en avant des décisions de localité des données

Sur les figures 4.4 et 4.5 apparaissent le résultat d'une exécution de notre stencil pour une taille de problème 9 fois supérieure à la mémoire disponible sur chaque GPU.

Regardons d'abord le diagramme en itérations de la figure 4.4. Nous pouvons remarquer que la répartition du domaine choisie par `dmdar` avant le

premier isochrone est alternée entre le GPU bleu et le GPU rouge de façon anarchique. Cela augmente la probabilité de devoir transférer des données d'un GPU à l'autre car on augmente le nombre de cellules frontières à deux GPUs. En revanche, rapidement, dès le troisième isochrone, la répartition devient plus ordonnée. Le rouge occupe le domaine du milieu et le bleu les bords.

Un autre élément que l'on peut noter est la façon dont progressent les itérations. Nous pouvons apercevoir des motifs en formes de dents de scie. Cela signifie que `dmdar` essaye de faire progresser un morceau de domaine au maximum (comme évoqué 2.3). La pointe des dents de scie correspond à l'instant où `dmdar` ne peut plus faire progresser ce domaine tant que les itérations précédentes du voisinage ne sont pas calculées.

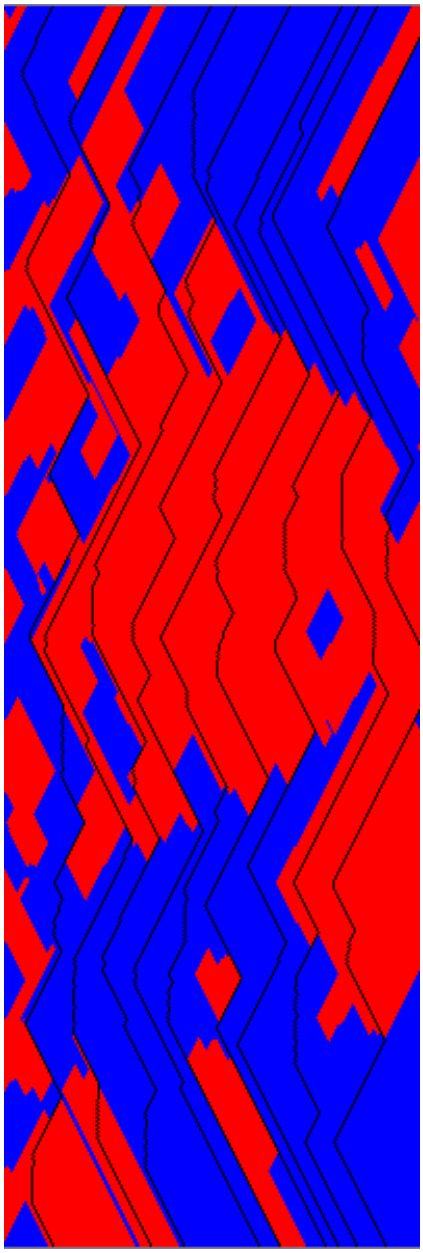
À titre de comparaison, un diagramme en itération de `cache-oublieux` est disponible en annexe figure A2.

Sur le diagramme en temps de la figure 4.5, nous pouvons voir un petit échantillon d'une exécution de `dmdar` dont les motifs apparents se répètent sur toute l'exécution. Sur la partie agrandie, nous pouvons voir que `dmdar` effectue des transferts de données sur le GPU bleu, puis va faire progresser un petit morceau du domaine sur plusieurs itérations. Ce sont ces motifs, signes de localité et réutilisation des données, qui nous montrent que `dmdar` prend des décisions d'ordonnancement adaptées à des applications stencils mais encore imparfaites : la taille des motifs pourrait être plus élevée, leur fréquence plus importante.

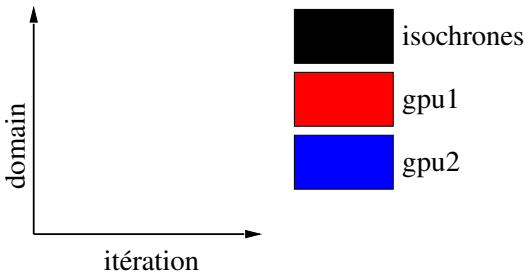
4.5.3 Répartition de charge inégale

Pour cette expérience, nous avons effectué deux répartitions de charge différentes parmi deux GPUs : une statique et une dynamique. La répartition statique diminue de moitié le temps de calcul des tâches pour la première moitié du domaine d'étude. La répartition dynamique diminue de moitié le temps de calcul des tâches pour un quart du domaine, et nous faisons évoluer au cours du temps la section du domaine concernée. Contrairement à ce qui est évoqué dans la formule de la section 4.5.1, nous ne modifions pas la constante α mais modifions le temps de calcul d'une tâche (la variable T_c de la formule).

La soumission statique des tâches de `cache-oublieux` a pour désavantage de ne pas être adapté à une répartition de charge inégale. Cela se vérifie sur les expériences de la figure 4.6. En effet, `dmdar`, qui était environ 1.5 fois plus

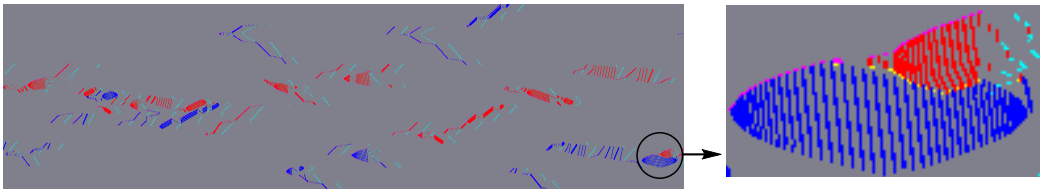


(a) `dmdar` - 2 GPU, taille du problème 1800MB, limite mémoire fixée à 200MB par GPU.

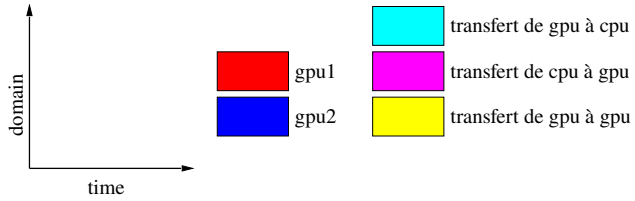


(b) Légende.

Figure 4.4 – Diagramme en itérations d'une exécution de `dmdar`.



(a) `dmdar` - 2 GPU, taille du problème 1800MB, limite mémoire fixée à 200MB par GPU.



(b) Légende.

Figure 4.5 – Échantillon d'exécution de `dmdar`.

lent que `cache-oublieux` (figure 4.3), est ici plus rapide pour la répartition statique et dynamique. Nous pouvons cependant noter que, pour la répartition dynamique, `cache-oublieux` rattrape progressivement son retard à mesure que la taille du domaine augmente.

Une explication pourrait être que, pour un domaine grand, le tri des tâches par nombre de données prêtes est moins efficace. On dispose alors de beaucoup de tâches qui se distinguent peu sur ce critère du nombre de tâches prêtes.

Pour la répartition statique, nous avons choisi un cas extrême pour `cache-oublieux`. Un GPU va travailler sur des tâches deux fois plus rapides que celles de l'autre GPU. Il serait intéressant de confirmer qu'en redessinant le découpage du domaine pour `cache-oublieux`, nous retrouvons des performances meilleures que `dmdar`.

Il est possible de faire de même pour la répartition dynamique. Dans les deux cas, cela nécessite des changements de l'application. Or, nous voulons des méthodes génériques. Ce problème de rééquilibrage de charge dynamique s'apparente au domaine de recherche du repartitionnement. Il existe des partitionneurs comme *PARAMESH* [2] capables de découper un problème en morceaux équitables (faire évoluer les frontière) mais cela reste très coûteux.

`dmdar` peut lui adapter son ordonnancement dynamiquement, en fonction de la charge, et cela à moindre coût.

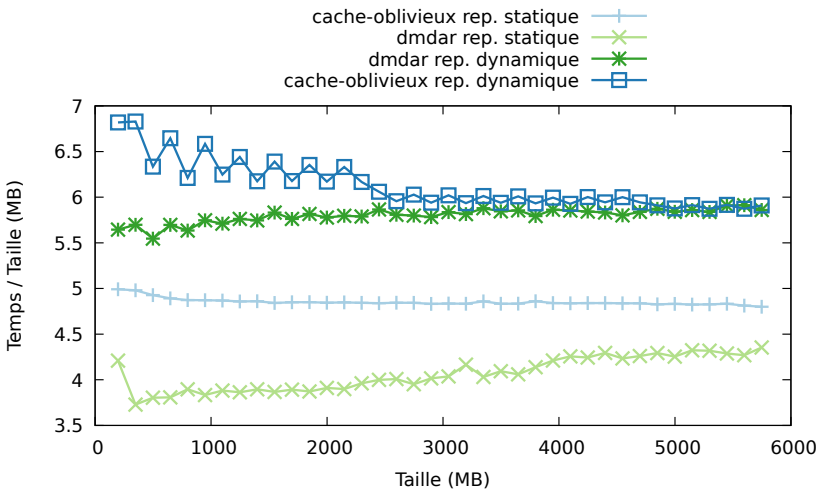


Figure 4.6 – Influence de répartitions de charge inégales sur l'ordonnanceur `dmdar` et sur `cache-oublieux` - deux GPU, pas de limite mémoire.

4.6 Feindre de la localité avec la priorité des tâches

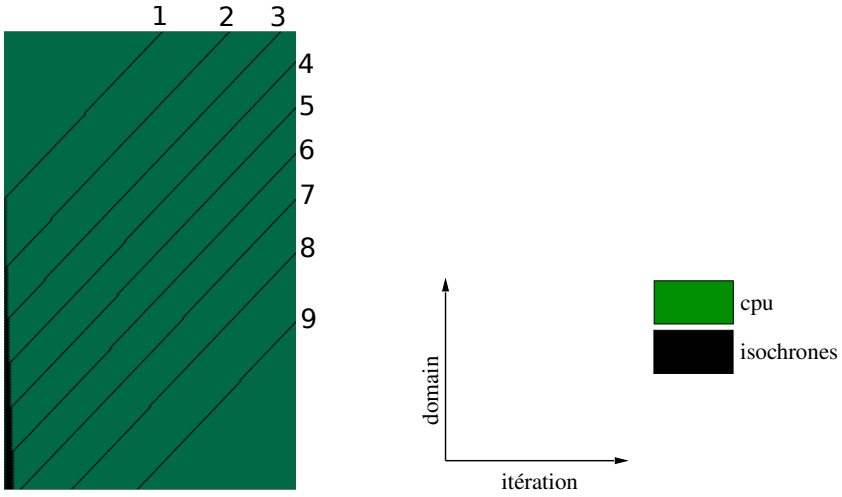
Pour cette expériences nous avons tenté de feindre de la localités des données à moindre coût, en utilisant l'ordonnanceur `prio`. Pour chaque tâche, nous attribuons une priorité égale au numéro d'itération. Nous voulons ainsi favoriser la progression de plusieurs itérations d'un morceau de domaine.

Sur la figure 4.7, nous pouvons voir que assez peu de réutilisation des données est faite avant la première courbe isochrone. De plus, à mesure qu'on progresse dans le temps, `prio` travaille sur une bande en diagonale de plus en plus grosse, si bien qu'au bout d'un certain temps, il n'y a plus de réutilisation de données. Par l'exemple, entre les isochrones 4 et 5, `prio` parcourt la quasi totalité du domaine.

5 Conclusion

5.1 Résumé des points importants

Nous nous sommes intéressé à l'ordonnancement de tâches d'applications stencil au sein du support d'exécution *StarPU* de l'équipe *STORM*. En particulier, nous avons étudié un ordonnancement sur deux GPUs. Pour des problèmes ne tenant pas en mémoire, il est nécessaire de transférer des données



(a) `prio` - 1 CPU, pas de limite mémoire.

(b) Légende.

Figure 4.7 – Localité des données feinte avec la priorité des tâches.

ce qui est coûteux. Pour réduire ce coût, nous pouvons faire du recouvrement de transferts par du calcul.

Les applications stencils ont des propriétés particulières. Beaucoup d'accès mémoires sont fait pour mettre à jour une cellule. Dans notre cas, les transferts mémoires des données d'une tâches durent deux fois plus longtemps que le calcul d'une tâche. Il est donc difficile de recouvrir les transferts sans faire de réutilisation de données.

Nous avons voulu observer la localité des données au sein des ordonnanceurs de *StarPU*. Après avoir fait un bref état des lieux de leurs propriétés, nous avons décrit un outil de visualisation adaptés aux stencils. Ces outils nous ont permis de mieux cerner les décisions d'ordonnancement.

Pour évaluer ces ordonnanceurs, nous avons écrit une méthode de référence qui est la soumission de tâches cache oublieux. Cette méthode maximise la réutilisation des données.

Nous avons ensuite réalisé plusieurs expériences afin de comparer les ordonnanceurs à `cache-oublieux`. Nous avons pu voir que si certains ordonnanceurs ne sont pas adaptés, d'autres comme `dmdar` et `modular-ready` exploite la localité des données et apporte des performances proches, mais encore perfectible, de `cache-oublieux` et de manière générique, ce qui était

l'objectif de départ. Cette généralités est importance aussi pour des problèmes avec une charge déséquilibrée. En effet, pour résoudre des problèmes à charge déséquilibrée, la méthode `cache-oublieux` nécessite des changements de l'application pour partitionner le domaine. Il existe des partitionneurs mais ce sont des outils coûteux.

5.2 Perspectives

5.2.1 Court terme

Il serait intéressant de valider les résultats obtenus en utilisant un stencil 2D 5-points moins artificiel que le stencil 1D 3-points utilisé ou à partir d'applications de simulation nucléaire. Nous pourrions aussi refaire les expériences sur de vraies machines et non via une simulation.

Il aurait aussi été intéressant de revoir la politique d'éviction de données de *StarPU* utilisée lorsque la mémoire se rapproche de la limite. En particulier, nous aurions pu travailler sur un algorithme *LRU* (*Least Recently Used*) optimal pour un stencil et adapté à une exécution en parallèle sur plusieurs GPUs.

5.2.2 Long terme

Nos outils de visualisation pour les stencils ne sont pas adaptés à des stencils 2D. Comment représenter, sur une dimension, un domaine en deux dimensions, avec pour objectif d'observer la localité des données ? Il serait intéressant de voir comment généraliser nos outils à des stencils 2D, 3D voir pour un nombre arbitraire de dimensions. Des pistes d'exploration sont la génération de plusieurs images ou d'un petit film.

Beaucoup de travaux de questionnement sur la visualisation ont été réalisés dans l'équipe *Polaris* d'Inria Grenoble, qui pourront développer des outils mieux adaptés.

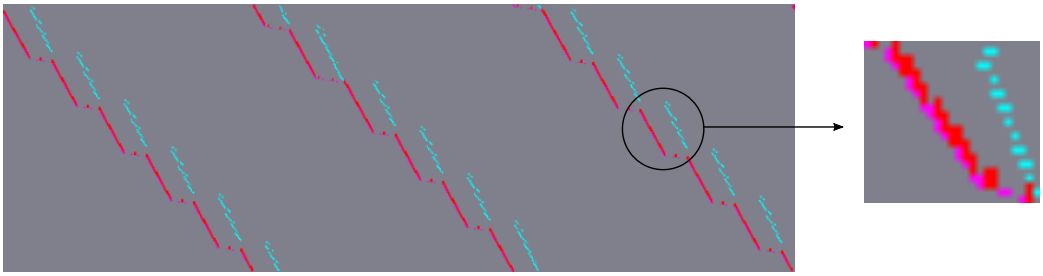
Pour les problèmes où la taille des données est trop importante pour rentrer en mémoire principale. Il faut donc faire de l'*out of core* et transférer régulièrement des données entre la mémoire principale et le disque dur. La principale différence avec notre problème est la latence d'un accès mémoire sur disque est environ 10^6 fois plus élevée.

Une autre particularité des stencils non évoquée dans ce stage est qu'avec le temps, un stencil converge vers un état stable, où une mise à jour de tous

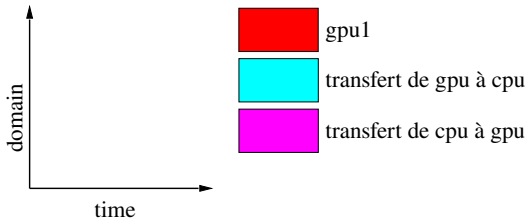
les éléments ne modifient plus aucune cellules. Il est courant de détecter la convergence d'un stencil au niveau applicatif à l'aide de barrières. Il s'agit d'une construction pour des programmes parallèles permettant à tous les ouvriers de s'attendre les uns les autres. On peut ainsi, une fois que tous les ouvriers se sont arrêtés sur la barrière, vérifier si le stencil est stable. Si cette méthode est simple, elle est aussi très coûteuse. Peut-être est-il possible de détecter la convergence dans *StarPU* pour arrêter l'exécution quand nécessaire.

Un autre aspect non étudié est la programmation distribuée en réseau. Pour résoudre des problème de taille de grande, il est possible de partager le travail entre plusieurs machines reliées en réseau et communiquant avec un protocole de communication adapté (e.g. *Message Passing Interface*). Il faut ainsi bien penser la redistribution des données et éviter les transferts trop petit. En effet, la latence est environ 10^3 plus élevée qu'un accès depuis une machine vers sa mémoire principale.

Annexes

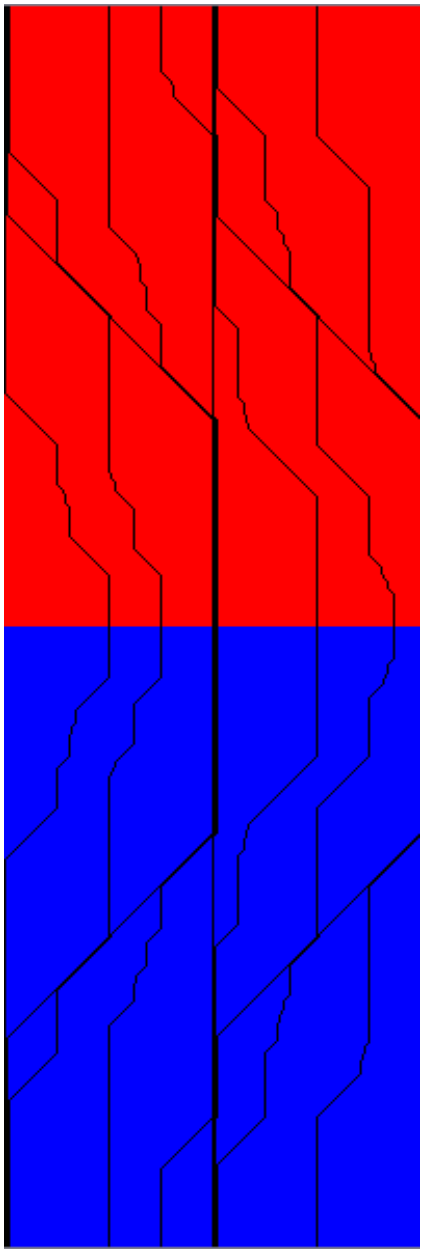


(a) Sur la partie grossie nous pouvons voir environ un transfert (en violet) par accès à une donnée - 1 GPU, limite mémoire fixée à 200MB par GPU.

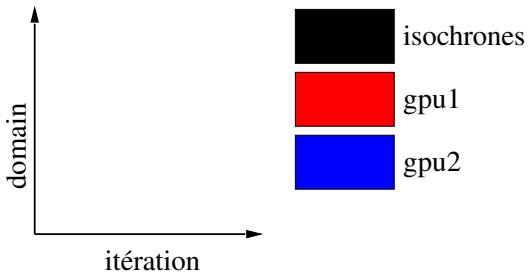


(b) Légende.

Figure A1 – Échantillon d'exécution pour l'ordonnanceur `lws`.



(a) `cache-oublieux` - 2 GPU, taille du problème 1800MB, limite mémoire fixée à 200MB par GPU.



(b) Légende.

Figure A2 – Diagramme en itérations d'une exécution de `cache-oublieux`.

Références

[1] Matteo FRIGO et Volker STRUMPEN. « Cache Oblivious Stencil Computations ». In : *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts : ACM, 2005, p. 361–366. ISBN : 1-59593-167-8. DOI : 10.1145/1088149.1088197. URL : http://doi.acm.org/10.1145/1088149.1088197.

[2] Peter MACNEICE et al. « PARAMESH : A parallel adaptive mesh refinement community toolkit ». In : *Computer Physics Communications* 126.3 (2000), p. 330–354. ISSN : 0010-4655. DOI : http://dx.doi.org/10.1016/S0010-4655(99)00501-9. URL : http://www.sciencedirect.com/science/article/pii/S0010465599005019.