HAL
open science

Utilisation efficace des accélérateurs GPU – Ordonnancement sur machines hybrides

Raphaël Bleuse

To cite this version:

Raphaël Bleuse. Utilisation efficace des accélérateurs GPU – Ordonnancement sur machines hybrides. Calcul parallèle, distribué et partagé [cs.DC]. 2013. hal-00858233

HAL Id: hal-00858233

https://inria.hal.science/hal-00858233v1

Submitted on 5 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Grenoble INP – Ensimag
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Rapport de projet de fin d'études

Utilisation efficace des accélérateurs GPU

Ordonnancement sur machines hybrides

Stagiaire

Raphaël Bleuse
3ème année – Option Ingénierie des Systèmes d'Information

Responsable de stage

Grégory Mounié

Tuteur école

Frédéric Pétrot

Laboratoire

LIG, équipe-projet Moais

Centre de recherche Inria Grenoble – Rhône-Alpes
655 avenue de l'Europe
Montbonnot
38334 Saint-Ismier Cedex France

Durée

11 février 2013 – 5 juillet 2013 (5 mois)

10 juin 2013

Résumé. La course à la puissance de calcul dans les super-calculateurs pose la problématique de la consommation énergétique de ces machines. Les systèmes hybrides – composés de processeurs et d'accélérateurs GPU (*Graphics Processing Unit*) – sont une réponse prometteuse à cette question. Actuellement, l'allocation des tâches sur des telles machines est réalisée par le programmeur de manière statique. Nous étudions le problème de l'ordonnancement de tâches indépendantes sur ces architectures. Nous proposons un algorithme d'approximation bi-critère – de faible complexité algorithmique – optimisant simultanément localité et temps de complétion avec des garanties de performance. Les performances de son implémentation dans l'environnement de calcul parallèle XKaapi sont ensuite validées par une étude expérimentale.

Mots clefs : Calcul Haute Performance ; architectures hybrides ; algorithme d'approximation ; multi-objectif ; localité

Abstract. The race for ever more computing power raises the issue of supercomputers' power consumption. Heterogeneous architectures – composed of processor and GPU accelerators – seem to be a promising answer. Scheduling on such machines is nowadays relying on the programmer's skill set and made in a static way. We study the problem of scheduling of independent tasks on such architectures. We propose a bi-objective approximation algorithm which simultaneously optimizes the makespan and the affinity. The provided algorithm has a low complexity. We then validate the performance of its implementation within the framework XKaapi.

Keywords: High Performance Computing; heterogeneous architectures; approximation algorithm; multi-objective; data locality

Table des matières

Résumé / *Abstract* **iii**

Table des matières **v**

1 Introduction **1**

2 État de l'art **3**
2.1 Des spécificités des machines hybrides 3
2.2 Du problème d'ordonnancement 4
2.3 Des techniques d'ordonnancement 6
2.3.1 Algorithmes de liste 6
2.3.2 Hétérogénéité 7
2.3.3 Vol de travail 7
2.4 De l'approche duale 9
2.5 De l'optimisation multi-objectif 9
2.5.1 Définition 9
2.5.2 Critères 10

3 Ordonnancement bi-critère de tâches indépendantes sur machine hybride **13**
3.1 Définition du problème 13
3.2 Affinité 14
3.2.1 Définition 14
3.2.2 Prise en compte de l'affinité 14
3.3 Algorithme proposé : MultiDual 15
3.3.1 GDual 15
3.3.2 MultiDual 15
3.4 Implémentation 17
3.4.1 *Runtime* XKaapi 17
3.4.2 Intégration de l'algorithme 18
3.5 Validation expérimentale 19
3.5.1 Conditions expérimentales 19
3.5.2 Validation 19

4 Organisation du travail **23**
4.1 Structure d'accueil 23
4.2 Cahier des charges 23

4.3 Plan de travail 24

5 Conclusion **27**

Bilan personnel **29**

Bibliographie **31**

Chapitre 1

Introduction

Le classement du Top500 [1] recense les machines les plus puissantes de la planète. Outre leur complexité, leur consommation énergétique devient un enjeu majeur. Un autre classement – celui du Green500 [2] – met l'emphase sur la puissance de calcul développée en regard de l'énergie fournie. Utiliser des architectures dédiée pour certains calculs semble être une réponse à ce nouveau genre de défi. Cependant la conception de super-calculateurs avec des architectures hétérogènes rend difficile leur exploitation. Premièrement, le développement d'applications capable de tirer parti de cette hétérogénéité nécessite d'écrire plusieurs versions d'un même code de calcul – une par architecture présente. Il faut de surcroit choisir où et quand exécuter le code au sein de la machine. Trouver l'ordonnancement optimal est intrinsèquement difficile puisque c'est un problème $\mathcal{NP}$-difficile. Il est par conséquent plus raisonnable de chercher à trouver une approximation de cet optimal. Néanmoins, trouver un ordonnancement performant reste difficile car de nombreux facteurs entre en compte : certaines portions de code sont plus efficaces sur certaines architectures, le transfert des données peut entrainer des phénomènes de contention, etc.

L'objectif de ce projet de fin d'études est de proposer un nouvel algorithme – à faible coût – multi-critère pour l'ordonnancement de tâches indépendantes sur des architectures multi-CPU / multi-GPU. Nous considérons une application modélisée par un ensemble de tâche indépendantes caractérisées par leurs temps d'exécution et une affinité basée sur la localité. L'ordonnancement déterminé cherche à minimiser le temps de complétion de l'application tout en maximisant l'affinité.

Contexte

Ce projet de fin d'études a été réalisé au sein de l'équipe-projet Moais. J'ai été encadré par Grégory Mounié, qui est maître de conférence dans cette équipe. La structure d'accueil est décrite plus en détail dans la section 4.1.

Organisation du manuscrit

Le manuscrit s'articule autour de trois chapitres. Le chapitre 2 intitulé « État de l'art » expose les notions générales et les résultats connus. La section 2.1 introduit

1. http://www.top500.org/
2. http://www.green500.org/

ce chapitre en présentant brièvement les spécificités des machines multi-CPU / multi-GPU. Ensuite, la section 2.2 présente la problématique de l'ordonnancement. La section 2.3 étudie plusieurs algorithmes et techniques classiques utilisés pour résoudre le problème d'ordonnancement dans des délais raisonnables. Dans la section 2.4, nous passons en revue l'approche duale proposée par Hochbaum *et al.* Nous terminons l'état de l'art en étudiant – dans la section 2.5 – les problèmes multi-objectifs.

La problématique d'ordonnancement bi-critère sur machine hybride est abordée dans le chapitre 3. Les sections 3.1 et 3.2 présentent le problème et les notions utilisées de manière formelle. La section 3.3 expose l'algorithme que nous proposons ainsi que ses performances théoriques. L'implémentation et la validation expérimentale sont respectivement détaillées dans les sections 3.4 et 3.5.

Le chapitre 4 intitulé « Organisation du travail » explicite dans quel cadre et comment le travail a été réalisé durant ce stage. Nous terminons ce manuscrit par un bilan des travaux effectués pendant ce stage. Nous évoquons aussi les pistes qui semblent les plus intéressantes à suivre. Enfin nous dressons un bilan personnel de cette expérience.

Chapitre 2

État de l'art

Nous présentons brièvement les spécificités des machines multi-CPU / multi-GPU, aussi appelées machines hybrides. Nous discutons ensuite différent aspects du problème d'ordonnancement. Pour plus d'informations, on pourra se référer à [27].

2.1 Des spécificités des machines hybrides

La course à la puissance évoquée dans l'introduction amène les constructeurs à construire des machines hybrides : les machines embarquent plusieurs CPUs et plusieurs GPUs.

Alors que les CPUs sont des unités de calcul généraliste, les GPUs sont des unités de calcul très optimisées pour les calculs avec beaucoup de parallélisme de données. Ces différences rendent les machines hybrides hétérogènes par ensemble, d'une part un ensemble de processeurs identiques, d'autre part un ensemble de GPUs identiques. Ainsi, il est fréquent d'observer pour une même tâche des temps d'exécution cinquante fois plus rapides sur une des architectures [11].

De plus – à l'heure actuelle [1] – les CPUs et GPUs sont séparés physiquement et ne partagent pas de mémoire. L'algorithmique utilisée ne peut donc pas se reposer sur des propriétés de cohérence mémoire. Il est en outre nécessaire de transférer les données entre les différentes unités de calcul. L'agencement des ces différentes unités au sein de la machine est important puisque cela peut mener à des contentions sur les bus de données [13, 14]. La figure 2.1 représente l'architecture d'une machine hybride du laboratoire : `idgraf`. Une telle configuration est classique pour une machine hybride.

Ces différences rendent difficile la programmation et la modélisation de ces nouvelles architectures. Tout d'abord, du fait de leur nouveauté, ces architectures sont encore très changeantes. Les capacités de calcul des matériels changent à chaque génération, rendant difficile la stabilisation des codes. En outre, le découpage des applications ainsi que le placement des tâches reposent principalement sur les compétences des développeurs.

L'ordonnancement de ces tâches par un *runtime* permet de mieux prendre en compte la complexité des architectures et peut être réalisé de manière dynamique.

1. AMD a annoncé en mai 2013 sa volonté de fondre des puces dans lesquelles CPU et GPU ont accès à l'intégralité de la mémoire disponible (architecture hUMA). `http://www.amd.com/us/products/technologies/hsa/Pages/hsa.aspx#3`

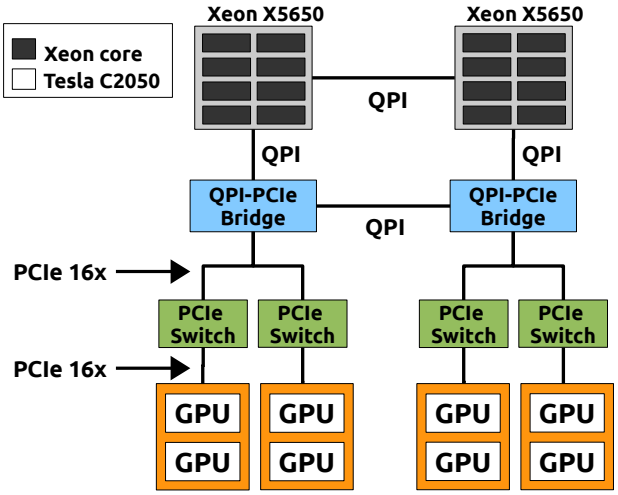


FIGURE 2.1 – Architecture matérielle d'une machine hybride (`idgraf`) [13]. Les CPUs et les GPUs ne partagent pas de mémoire et communiquent via des bus PCI.

2.2 Du problème d'ordonnancement

La théorie de l'ordonnancement est une branche de la recherche opérationnelle. Un problème d'ordonnancement consiste à planifier dans le temps la réalisation de tâches tout en respectant des contraintes – par exemple de délai ou de précédence. Une solution à un problème d'ordonnancement est l'allocation, à une date donnée, d'une tâche à une ressource chargée de la réaliser. Le principal critère considéré est le temps de complétion, c.-à-d. le temps nécessaire pour que toutes les tâches aient été réalisées. D'autres critères peuvent être pris en compte pour élaborer une solution, ils sont évoqués plus en détail dans la sous-section 2.5.2.

L'ordonnancement de tâches sur plusieurs processeurs est une classe particulière des problèmes d'ordonnancement. Le problème d'ordonnancement multi-processeur considère n tâches et m processeurs. À chaque couple (tâche, processeur) est associé un temps de calcul [2]. On considère aussi souvent des contraintes de précédence entre certaines tâches. Ces contraintes de précédence forment une relation d'ordre partielle sur les tâches, souvent représentée par un graphe orienté acyclique (cf. figure 2.2) Toute exécution des tâches doit respecter cet ordre. Résoudre le problème d'ordonnancement multi-processeur consiste à déterminer sur quel processeur et dans quel ordre exécuter les tâches.

Exemple 1. Considérons $n = 8$ tâches et $m = 3$ processeurs identiques. Les temps de calcul sont respectivement $p_1 = 1$, $p_2 = 3$, $p_3 = 4$, $p_4 = 2$, $p_5 = 2$, $p_6 = 3$, $p_7 = 1$ et $p_8 = 5$. Un ordonnancement dont le temps de complétion est 9 et respectant les contraintes de précédence illustrées sur la figure 2.3(a) est reporté sur la figure 2.3(b).

2. Il est distingué trois modèles en fonction des relations existant ou non entre les processeurs. On pourra se référer à [27, 30] pour plus de détail.

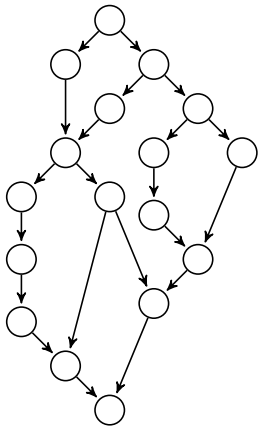
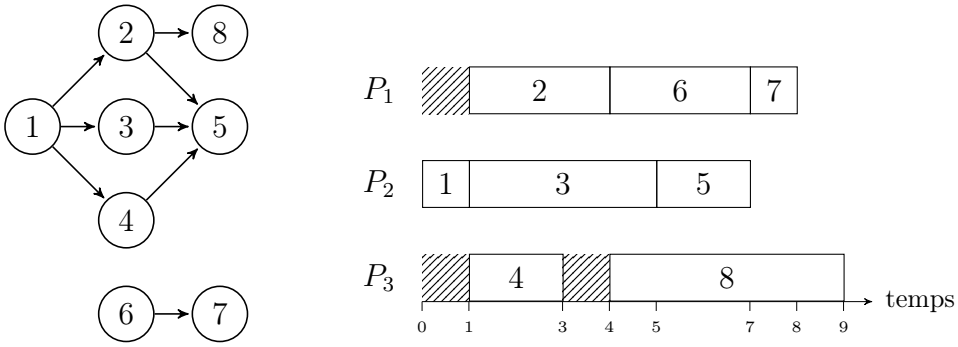


FIGURE 2.2 – Exemple de programme parallèle à base de tâches. Les contraintes de précédence sont représentées par les arcs. L'ordre d'exécution doit respecter ces contraintes.



(a) Contraintes de précédence.

(b) Ordonnancement réalisable avec un temps de complétion de 9.

FIGURE 2.3 – Ordonnancement sur plusieurs processeurs identiques.

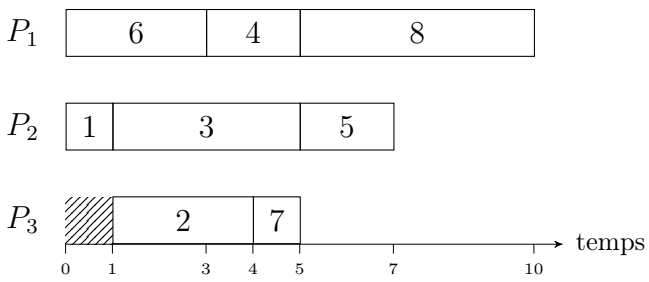


FIGURE 2.4 – Ordonnancement obtenu avec l'algorithme de liste pour le graphe de la figure 2.3(a) avec trois processeurs.

2.3 Des techniques d'ordonnancement

Le problème d'ordonnancement de tâches sur des processeurs est connu pour être un problème combinatoire difficile : minimiser le temps de complétion est $\mathcal{NP}$ [35]. Pour passer outre les phénomènes d'explosion combinatoire, la recherche de solution non-optimale est une réponse. Les algorithmes d'approximation fournissent – à l'aide d'heuristique – des solutions proches de l'optimum. Dans la mesure du possible, ces heuristiques cherchent à être bien fondées théoriquement : des garanties de performance caractérisent la solution fournie.

Nous étudions ici différents algorithmes à faible coût : prendre des décisions rapides est crucial afin de ne pas retarder l'exécution parallèle des programmes. Nous nous intéresserons aussi à leurs garanties et à leur analyse dans le pire cas.

Une manière de classifier les algorithmes est de distinguer les algorithmes *on-line* et *off-line*. Les algorithmes *off-line* requièrent une connaissance complète du problème pour déterminer un ordonnancement. À l'inverse, les algorithmes *on-line* sont capables d'ordonnancer les tâches sans avoir une vision globale du problème.

Shmoys *et al.* proposent un schéma très général permettant de transformer un algorithme *off-line* en algorithme *on-line* tout en conservant des garanties de performance [30]. Les algorithmes *on-line* par conception ont néanmoins généralement de meilleurs ratios d'approximation que les algorithmes rendus *on-line* grâce à ce schéma. De cette façon il est possible – dans un premier temps – de restreindre le champ de recherche aux algorithmes *off-line*. Par conséquent nous chercherons en premier lieu à concevoir un algorithme *off-line*.

2.3.1 Algorithmes de liste

Sous l'hypothèse que tous les processeurs utilisés sont identiques, Graham a proposé et étudié le premier algorithme d'approximation garantie : l'algorithme de liste [16,17]. L'algorithme tient à jour la liste des tâches prêtes, c.-à-d. celles dont les contraintes de précédence sont satisfaites. Dès qu'un processeur devient inactif, il retire la tâche dont le temps d'exécution est le plus long et l'exécute. Les tâches découvertes pendant l'exécution sont ajoutées dynamiquement à la liste. Cet algorithme génère une solution dont le temps de complétion ne dépasse pas le double du temps optimum. La figure 2.4 montre l'ordonnancement de l'exemple 1 obtenu en appliquant cet algorithme.

L'étude de Graham pose les notions fondamentales qui seront les bases de l'analyse des algorithmes parallèles et leur ordonnancement. En distinguant le travail total de toutes les tâches W – c.-à-d. le temps séquentiel – et le travail des tâches sur un plus long chemin de précédences D – c.-à-d. le chemin critique – il montre que le temps d'exécution T_m sur m processeurs est majoré par

$$T_m \leq \frac{W}{m} + \left(1 - \frac{1}{m}\right) \cdot D \tag{2.1}$$

Les valeurs W/m et D sont des bornes inférieures du temps de complétion. Le facteur 2 de l'algorithme de Graham vient d'ici.

2.3.2 Hétérogénéité

Les machines actuelles embarquent des ressources de calcul qu'il n'est plus possible de considérer comme identiques. De plus, ces différentes ressources n'ont pas – dans le cas général – de mémoire commune : il n'est plus possible de négliger les délais induits par les transferts de données.

Pour le problème plus général de l'ordonnancement hétérogène – prenant en compte ces différences – Topcuoglu *et al.* proposent avec l'algorithme HEFT [34] une extension de l'approche de Graham (cf. sous-section 2.3.1). Ici, à chaque tâche est associé, pour chaque processeur, un temps de calcul. On mesure aussi les temps de transfert entre les différents processeurs. L'algorithme trie les tâches par un ordre ascendant défini de manière récursive (cf. définition 1). Cet ordre prend en compte les délais de transfert entre les ressources et les différents temps de calcul. Les décisions de placement des tâches se font alors comme dans l'algorithme de liste *off-line*.

Définition 1 (Rang ascendant (*upward rank*) [34])**.** Le rang ascendant d'une tâche n_i est défini récursivement par la relation

$$\text{rang}(n_i) = \overline{w_i} + \max_{n_j \in \text{succ}(n_i)} (\overline{c_{i,j}} + \text{rang}(n_j)) \tag{2.2}$$

où $\overline{w_i}$ est le temps moyen d'exécution de la tâche n_i, $\overline{c_{i,j}}$ le coût moyen de communication entre les tâches n_i et n_j et $\text{succ}(n_i)$ l'ensemble des successeurs immédiats de n_i dans le graphe de précédence.

Cet algorithme est néanmoins entaché de plusieurs défauts. Tout d'abord, il ne permet de travailler qu'avec des graphes de précédence statiques. Enfin, son approche gloutonne peut l'amener à se tromper sévèrement dans ses choix d'ordonnancement, d'autant plus que seul les moyennes sont considérées. Ceci n'est pas pénalisant lorsque les processeurs sont identiques puisqu'il est impossible de faire un mauvais choix, mais devient très pénalisant avec des processeurs différents. En effet, le ratio d'approximation de HEFT dans le pire cas est au moins de $m/2$, où m est le nombre de CPUs [21].

2.3.3 Vol de travail

Toutefois, les implantations naïves des algorithmes présentés jusqu'à présent ne passent pas bien à l'échelle. Même si en pratique seules les tâches prêtes sont ordonnancées, l'information est centralisée. De ce fait, lorsque le nombre de processeurs devient important, des phénomènes de contention mémoire apparaissent. Ces algorithmes sont, de plus, difficiles à distribuer.

Une technique efficace pour pallier aux faiblesses évoquées auparavant est le vol de travail [3, 5, 33]. Plutôt que de garder dans une liste commune à tous les processeurs les tâches prêtes à être exécutées, chaque processeur tient à jour une liste locale qui lui est propre. Tant que la liste locale n'est pas vide, le processeur exécute les tâches présentes et y ajoute celles créées dynamiquement. Dès que la liste devient vide, le processeur essaie de voler des tâches dans la liste d'un autre processeur choisi au hasard. Cette technique passe mieux à l'échelle pour plusieurs raisons. Premièrement, elle est distribuée par nature : chaque processeur ayant sa liste de tâches. D'autre part, la probabilité qu'une tentative de vol ait lieu est petite. De ce fait, il est peu fréquent que deux processeurs cherchent à accéder à la même liste simultanément : la gestion des tâches prêtes n'est plus un goulot d'étranglement. De plus, en utilisant des codes séquentiels très optimisés pour un grain fin de parallélisme, cette famille d'algorithme possède de bonnes qualités : la charge globale se répartit automatiquement entre les processeurs et ceci indépendamment de leur structuration. On parle dans ce cas de schéma *processor oblivious*. Enfin, Blumofe *et al.* ont montré que l'ordonnancement généré par leur algorithme est aussi bon que celui de Graham puisque le temps de complétion est borné par

$$T_m \leq \frac{W}{m} + O(D) \tag{2.3}$$

Néanmoins, dans sa version naïve, le vol de travail a tendance à dégrader les performances de certaines applications. Les vols aléatoires sont la source de défauts de cache inexistants dans les versions séquentielles des programmes.

Plusieurs approches ont été étudiées afin, notamment, de prendre en compte la localité des ressources.

Une façon naïve de prendre en compte la localité est de supposer que les régions d'un programme proches dans le graphe de précédence accèdent aux mêmes données. Toutefois, cette supposition ne concerne qu'une classe très restreinte de calculs.

Acar *et al.* font office de précurseurs dans [1] en présentant une modification heuristique du vol de travail. Des indications de localité sont fournies à l'algorithme et permettent de mieux prendre en compte les accès à la mémoire. Ceci permet de mieux utiliser la plus grande quantité de mémoire disponible et de limiter les défauts de cache. Ils sont ainsi capable d'atteindre des *speedup* super-linéaires.

Une autre approche est de prendre en compte la topologie de l'architecture sur laquelle est effectuée le calcul. Quintin *et al.* [26] exploitent ainsi la hiérarchie existante et pénalise les vols distants. L'impact sur les délais de communication des vols distants est aussi amoindri en volant plusieurs tâches.

Enfin, l'approche de XKaapi se base sur la constatation que la compréhension fine des dépendances entre les données est nécessaire pour exécuter efficacement des programmes parallèles. Gautier *et al.* proposent dans [15] un modèle de programmation permettant de décrire les usages des variables au sein d'un programme. Une tâche correspond à l'exécution d'une fonction sans effet de bord. Elle déclare les types d'accès qu'elle effectue pour chaque variable partagée : lecture, écriture. Ainsi le moteur d'exécution est capable de déterminer les synchronisations nécessaires entre les tâches.

Les travaux de Bender *et al.* [4] ainsi que ceux sur XKaapi [12–14] montrent que l'algorithme de vol de travail s'adapte bien aux architectures distribuées et hétérogènes.

On citera aussi les travaux de Narang *et al.* [23] qui proposent une implémentation distribuée tout en bornant la mémoire utilisée.

2.4 De l'approche duale

Les algorithmes présentés jusqu'à présent cherchent à déterminer des ordonnancements réalisables et sous-optimaux. La performance de ces algorithmes est mesurée par le degré de sous-optimalité toléré. De manière antagoniste, l'approche duale introduite par Hochbaum *et al.* [18] cherche des ordonnancements irréalisables et meilleurs que l'optimal.

Définition 2 (Algorithme d'approximation duale [18])**.** Un algorithme d'approximation duale – on parle d'algorithme g-dual – accepte comme entrée un nombre λ : l'estimation du temps de complétion. Soit l'algorithme détermine une solution dont le temps de complétion est au plus $g\lambda$, soit il n'existe pas de solution dont le temps de complétion soit inférieur à λ et l'algorithme l'indique sans se tromper.

Il est ensuite possible d'utiliser un algorithme d'approximation duale pour déterminer un algorithme d'approximation. En effet, ces algorithmes permettent – grâce à une dichotomie – d'approcher l'estimation de la valeur de l'optimal aussi proche que souhaité. L'estimation du temps de complétion est une information de valeur et permet à ce schéma de construire des algorithmes de faible complexité.

L'algorithme 1 présenté par la suite est une 2-approximation duale pour le temps de complétion.

2.5 De l'optimisation multi-objectif

Optimiser un seul objectif – par exemple le temps de complétion – a été longuement étudié. Il peut cependant être pertinent de s'intéresser à plusieurs objectifs simultanément. Une raison de s'orienter vers de l'ordonnancement multi-objectif peut être la présence de plusieurs utilisateurs qui ne souhaitent pas optimiser le même critère. Une autre raison est l'enjeu économique majeur que devient l'énergie nécessaire pour faire fonctionner les super-calculateurs. Considérer à la fois le temps de complétion et l'énergie utilisée permet à l'utilisateur de faire des compromis et de mieux gérer sa machine.

2.5.1 Définition

La variété des objectifs qu'il est possible de prendre en compte rend difficile la définition du problème d'optimisation multi-objectif. Nous présentons ici la définition proposée dans [10] : sous l'hypothèse non-réductrice que tous les objectifs sont à minimiser, un problème d'optimisation multi-objectif consiste à déterminer où et quand exécuter chaque tâche tout en minimisant chacune des fonctions objectifs.

Ce type de problème est intrinsèquement plus difficile que l'ordonnancement avec un seul objectif. Le problème peut être $\mathcal{NP}$-dur même si chacun des ordonnancements avec un objectif unique est calculable en temps polynomial.

Ensemble de Pareto

Pour caractériser les ordonnancements réalisables au vu des différents objectifs, on définit la notion de dominance de Pareto.

Définition 3 (Dominance de Pareto)**.** Soient σ et σ' deux ordonnancements réalisables et $(f_l)_l$ l'ensemble des fonctions objectif considérées.

$$\sigma \text{ domine } \sigma' \text{ au sens de Pareto } \iff \forall l \ f_l(\sigma) \leq f_l(\sigma') \tag{2.4}$$

La dominance de Pareto définit une relation d'ordre partielle sur les ordonnancements réalisables. Un ordonnancement réalisable est dit optimal au sens de Pareto si il n'est dominé par aucun autre. L'ensemble de Pareto d'un problème d'ordonnancement est défini comme l'ensemble des ordonnancements réalisables optimaux.

Intuitivement, l'ensemble de Pareto d'un problème représente les compromis intéressants pour les différentes solutions à ce problème. Il est cependant difficile de générer l'ensemble de Pareto d'un problème puisqu'il est dans le cas général de taille exponentielle en la taille de l'entrée. Il est néanmoins possible de se restreindre à un sous-ensemble de taille polynomiale sous certaines conditions [25].

2.5.2 Critères

Nous passons ici en revue divers critères qui présentent un intérêt pour l'ordonnancement de tâches. Les critères d'énergie, d'équité et de fiabilité sont mentionnés pour ouvrir un horizon plus large sur les critères qu'il est possible de considérer. Néanmoins ces derniers critères semblent trop complexes à appréhender durant un stage de fin d'études.

Localité

Les programmes séquentiels sont généralement optimisés pour exploiter la localité spatiale et temporelle des données. Lorsque le code est parallélisé, cette localité peut être perdue. La recherche de la localité dans les programmes parallèles peut être réalisée à deux niveaux : le *runtime* et l'application.

Au niveau du *runtime*, le vol de travail naïf est connu pour altérer les performances en ne prenant pas en compte la localité préexistante dans l'exécution séquentielle. Les travaux d'Acar *et al.* montrent que la prise en compte de la localité – au travers d'annotations du code – permet de conserver les optimisations séquentielles [1].

Au niveau applicatif, la manière dont sont stockées les données affecte directement les performances. Ainsi le phénomène de faux partage d'une seule ligne de cache peut dégrader les performances d'une application. L'utilisation de courbe de couverture de l'espace pour indexer des tableaux de données a été beaucoup étudiée et permet de limiter les défauts de cache sans prendre en compte la taille des caches : on parle de technique *cache oblivious* [2, 6, 19, 32]. Pour plus de détail sur les courbes de couverture de l'espace, on pourra se référer au livre de Sagan [28].

Temps

Le principal critère optimisé dans les problèmes d'ordonnancement pour le calcul haute performance est le temps de complétion de la dernière tâche. Il en existe d'autre, les plus populaires étant :

- *Minsum* : $\sum_i C_i$. On cherche à minimiser la somme des temps de complétion C_i de toutes les tâches.
- Nombre de tâches en retard (*tardiness*) : $\sum_i U_i = \sum_i \delta(C_i - d_i)$ avec δ la fonction de Heaviside. On impose une limite de temps d_i pour chaque tâche et on cherche à minimiser le nombre de tâches qui dépassent leur limite.

Mémoire

En introduisant le vol de travail [5], Blumofe *et al.* se sont intéressés au surcoût mémoire engendré. Ils ont montré que l'utilisation mémoire de l'exécution parallèle est borné par pS_1, où p est le nombre de processeurs et S_1 la mémoire nécessaire à l'exécution séquentielle. Ainsi, avec les quantités de mémoire disponibles actuellement, ce critère ne semble pas avoir d'impact majeur sur l'ordonnancement. On peut tout de même mentionner les travaux de Narlikar qui propose un algorithme permettant de réduire les besoins en mémoire [24].

Les considérations de mémoire prennent tout leur intérêt lorsque les processeurs possèdent une mémoire locale ou en calcul distribué. Il est alors possible de raisonner sur les quantités de mémoire utilisée afin de limiter les transferts. Une telle approche peut être plus simple que d'optimiser directement les transferts.

Recouvrement des temps de calcul / communication

Les premiers algorithmes d'ordonnancement proposés font l'hypothèse que les temps de transferts sont nuls, ce qui est le cas dans les architectures à mémoire partagée. Il faut toutefois prendre en compte les phénomènes de contention qui peuvent apparaître lors de la lecture ou l'écriture en mémoire ainsi que les hiérarchies de cache [8, 9]. En outre la complexification des machines et la demande de puissance de calcul ne permet plus d'ignorer les temps de transfert des données. Pour améliorer les performances des applications il faut donc considérer les temps de transfert des données comme une entrée des algorithmes à part entière. Une technique envisageable est de pipeliner les transferts avec les calculs. Ceci va dans le sens des évolutions matérielles actuelles : les constructeurs intègrent maintenant dans les cartes graphiques des moteurs de copie indépendant des unités de calcul. Afin de gagner en efficacité, le recouvrement des temps de calcul et de communication s'utilise de concert avec les optimisations de localité.

Énergie

Afin de pouvoir atteindre des puissances de calcul de plus en plus importantes tout en restant dans des enveloppes énergétiques raisonnables, l'énergie doit être prise en compte dans l'ordonnancement. Cependant, plusieurs écueils rendent difficiles la prise en compte de ce critère. Il est en effet difficile de mesurer la consommation énergétique de manière fine. En outre il n'existe pas de modèle générique et reconnu par la communauté scientifique pour estimer cette consommation énergétique.

Équité

La recherche de puissance de calcul de plus en plus importante oblige les utilisateurs de super-calculateur à unir leurs efforts. Partager les ressources de manière équitable

devient alors un enjeu en soi. Deux niveaux de performance sont alors distinguables, l'efficacité globale de la plate-forme de calcul et celle perçue par chacun des utilisateurs. L'hétérogénéité des utilisateurs et de leurs objectif fait de l'optimisation de ce critère un problème multi-objectif à part entière. Cordeiro étudie dans sa thèse comment encourager cette collaboration indispensable [7].

Fiabilité

Alors que la puissance de calcul des machines va croissant, leur complexité fait de même. Les machines les plus puissantes selon le classement du Top500 [3] embarquent de l'ordre de 10^5 à 10^6 cœurs et cela a des conséquences sur la fiabilité. Ainsi, le temps moyen entre deux pannes est de l'ordre de la journée et ne fera que diminuer avec la complexification des machines. Il est ainsi impossible d'utiliser ces machines pour y faire des calculs sans prendre en compte l'éventualité d'avoir à recommencer ou dupliquer certaines opérations [22].

3. `http://www.top500.org/`

Chapitre 3

Ordonnancement bi-critère de tâches indépendantes sur machine hybride

Optimiser le placement de tâches en ne prenant en compte que leur temps d'exécution ne permet pas de considérer de manière satisfaisante la complexité des machines hybrides. En effet, l'architecture matérielle rend difficile la compréhension du comportement des applications. Par exemple les GPUs effectuent leurs calculs dans une mémoire dédiée, impliquant des transferts de données complexes à modéliser. L'approche envisagée ici est de permettre au développeur ou au *runtime* de décrire simplement une affinité entre une tâche et l'endroit où elle sera exécutée.

Shmoys et Tardos ont étudié l'ordonnancement de tâches sur des machines hétérogènes en considérant des coûts [29]. Leur technique pourrait être utilisée dans le cadre de notre approche. Cependant, même si l'algorithme qu'ils proposent permet de construire des ordonnancements avec d'excellents ratios d'approximation sur le coût et le temps de complétion, ce dernier est basé sur la programmation linéaire en entier. Ainsi l'implémenter est difficile et sa complexité algorithmique est grande.

Les travaux de Jeannot *et al.* sur l'ordonnancement bi-critère fiabilité / temps de complétion [20] utilisent une approche similaire à la notre. L'objectif pour optimiser la fiabilité est très proche de celui que nous proposons pour l'affinité. Cependant son étude n'est valide que dans le cadre de machines uniformes et ne s'applique pas à notre problème.

3.1 Définition du problème

Nous considérons une architecture multi-cœurs constituée de CPUs et de GPUs. Nous noterons $\mathcal{M}$ l'ensemble des ressources de calcul de l'architecture. Cet ensemble est divisé en deux parties notées $\mathcal{M}^C$ et $\mathcal{M}^G$ représentant respectivement l'ensemble des CPUs et l'ensemble des GPUs. Toutes les ressources appartenant à une partie sont identiques, mais aucune relation n'est supposée entre des ressources n'appartenant pas à la même partie. Nous définissons une application comme un ensemble $\mathcal{T}$ de tâches séquentielles indépendantes. À chaque tâche t sont associés deux temps : $p^C(t)$ le temps d'exécution sur un CPU et $p^G(t)$ le temps d'exécution sur un GPU. Par commodité nous noterons $p(t, m)$ le temps d'exécution de la tâche t sur la ressource m. Nous

considérons enfin une fonction d'affinité binaire notée aff_b comme définie ci-après (cf. section 3.2).

Pour le problème considéré, l'objectif est de minimiser le temps de complétion tout en maximisant l'affinité de l'ordonnancement. Nous rappelons que le temps de complétion C_{max} est défini comme le temps de terminaison de la dernière tâche. L'affinité Φ d'un ordonnancement est quant à elle définie dans la sous-section 3.2.

3.2 Affinité

3.2.1 Définition

Nous considérons un ensemble de tâches indépendantes $\mathcal{T}$ et un ensemble de processeurs $\mathcal{M}$. L'affinité est définie comme une fonction $\text{aff} : \mathcal{T} \times \mathcal{M} \to \mathbb{R}$. Elle associe à chaque couple (tâche, processeur) une valeur réelle. Plus cette valeur est grande, plus il est intéressant d'ordonnancer la tâche sur le processeur en question.

Définition 4 (Affinité binaire)**.** Une fonction d'affinité binaire est une fonction d'affinité $\text{aff}_b : \mathcal{T} \times \mathcal{M} \to \mathbb{R}$ vérifiant la propriété suivante :

$$\forall t \in \mathcal{T}, \exists!\, m \in \mathcal{M} \mid \text{aff}_b(t, m) = 0 \tag{3.1}$$

L'unique processeur pour lequel l'affinité binaire est non nulle sera noté $m_{\text{aff}}(t)$ dans la suite de l'exposé. De même la valeur de l'affinité prise en $(t, m_{\text{aff}}(t))$ sera notée $\text{aff}_b(t)$.

Un ordonnancement est caractérisé par une fonction d'allocation $\pi : \mathcal{T} \to \mathcal{M}$ qui associe à une tâche le processeur qui l'exécute. Nous pouvons alors définir $\Phi(\pi)$ – l'affinité d'un ordonnancement – comme :

$$\Phi(\pi) = \sum_{t \in \mathcal{T}} \text{aff}(t, \pi(t)) \tag{3.2}$$

3.2.2 Prise en compte de l'affinité

La fonction objectif naturellement associée à l'affinité est : $\max \sum_{t \in \mathcal{T}} \text{aff}(t, \pi(t))$. Optimiser ce seul critère est simple : l'affinité maximale d'un ordonnancement est obtenue en plaçant chaque tâche sur le processeur pour lequel $\text{aff}(t, m)$ est maximale. Considérer l'affinité de pair avec la minimisation du temps de complétion est beaucoup plus difficile.

Proposition 1. Il n'existe pas d'algorithme d'approximation pour le problème d'optimisation bi-critère affinité / temps de complétion dont les ratios d'approximation soient constants pour une unique solution.

Démonstration. La preuve utilise le même schéma que Jeannot *et al.* dans [20]. Les valeurs des fonctions objectif ont été adaptées pour correspondre à notre problème.

Considérons une instance du problème avec deux machines : un CPU m_C et un GPU m_G. Soit $k \in \mathbb{R}^{+*}$. Considérons alors une unique tâche t telle que $p^C(t) = k$ et $p^G(t) = 1$. D'autre part choisissons aff telle que $\text{aff}(t, m_C) = k$ et $\text{aff}(t, m_G) = 1$. Il n'y a que deux ordonnancements possibles : π_C et π_G, pour lesquels t est respectivement

placée sur le CPU ou sur le GPU. Nous remarquerons que π_G est optimal pour le temps de complétion alors que π_C maximise l'affinité.

$C_{max}(\pi_C) = k$ et $C_{max}(\pi_G) = 1$. Ainsi $\frac{C_{max}(\pi_C)}{C_{max}(\pi_G)} = k$: ce ratio est aussi grand que souhaité. De même, $\Phi(\pi_C) = k$ et $\Phi(\pi_G) = 1$. Il vient $\frac{\Phi(\pi_C)}{\Phi(\pi_G)} = k$: ce ratio diverge vers l'infini avec k.

Aucun des ordonnancements n'est en mesure de fournir – pour les deux objectifs – une approximation avec un ratio constant. □

La proposition 1 montre qu'il est impossible de trouver une unique solution tout en garantissant des ratios d'approximation constants. Afin de contourner cette difficulté, nous nous intéresserons aux ordonnancements dont le temps de complétion est supérieur à un seuil arbitraire B [1]. Nous comparerons leur affinité à $\Phi^*(B)$, définie comme l'affinité maximale parmi tous les ordonnancements partiels dont le temps de complétion est inférieur à B.

Pour la suite de l'étude du problème nous nous restreindrons aux fonctions d'affinité binaire.

3.3 Algorithme proposé : MultiDual

Nos travaux composent les résultats de deux études : la prise en compte de la fiabilité pour l'ordonnancement sur machines homogènes [20] et l'ordonnancement à faible complexité pour des machines hybrides [21].

Nous proposons un algorithme (cf. algorithme 2) avec des garanties de performance et une complexité raisonnable pour résoudre ce problème. L'algorithme est inspiré du schéma de Stein et Wein [31] et repose sur la technique d'approximation duale [18]. Nous détaillons dans un premier temps l'algorithme sur lequel MultiDual repose : GDual (cf. algorithme 1). Nous étudions plus en détail MultiDual dans la sous-section suivante.

3.3.1 GDual

L'algorithme GDual a été proposé dans [21]. Pour déterminer un ordonnancement, l'algorithme trie les tâches par *speedup*. La stratégie utilisée est de charger en priorité les GPUs avec les tâches qui ont la plus grande accélération sur cette architecture. Une fois qu'il n'est plus possible de placer des tâches sur les GPUs, l'algorithme essaie de placer les tâches restantes sur les CPUs.

Lemme 1. GDual détermine un ordonnancement dont le temps de complétion est au plus $2C^*_{max} + \epsilon$ [21].

3.3.2 MultiDual

L'élaboration de l'ordonnancement est divisé en deux phases. Dans un premier temps, nous nous autorisons – de manière mesurée – à perdre du temps pour placer

1. Ce seuil arbitraire peut être interprété comme le temps de complétion visé.

Algorithme 1 : GDual

Entrées : paramètre dual λ ; tâches $\mathcal{T}$; temps d'exécution p^C, p^G
Pré-Requis : $\forall t \in \mathcal{T},\ p^C(t) < \lambda \wedge p^G(t) < \lambda$ `/* Rejet trivial */`

début
- soit $\mathcal{T}_G = \{t \in \mathcal{T} \mid p^C(t) > \lambda\}$ `/* Tâches trop longues sur CPU */`
- **pour chaque** $t \in \mathcal{T}_G$ **faire**
  - ordonnancer t sur le GPU le moins chargé m_G
  - $\mathcal{T} = \mathcal{T} \setminus \{t\}$
  - **si** m_G *est chargé au delà de* λ **alors**
    - **retourner** "Rejet : $C^*_{max} \geq \lambda$"
- soit $\mathcal{T}_C = \{t \in \mathcal{T} \mid p^G(t) > \lambda\}$ `/* Tâches trop longues sur GPU */`
- **pour chaque** $t \in \mathcal{T}_C$ **faire**
  - ordonnancer t sur le CPU le moins chargé m_C
  - $\mathcal{T} = \mathcal{T} \setminus \{t\}$
  - **si** m_C *est chargé au delà de* λ **alors**
    - **retourner** "Rejet : $C^*_{max} \geq \lambda$"
- trier $\mathcal{T}$ par $\frac{p^C(t)}{p^G(t)}$ décroissants
- **tant que** $\mathcal{T} \neq \emptyset$ **et** $p^G(t) < \lambda$ **faire**
  - $t = \mathcal{T}$.pop()
  - ordonnancer t sur le GPU le moins chargé m_G
  - **si** m_G *est chargé au delà de* λ **alors**
    - **break**
- **tant que** $\mathcal{T} \neq \emptyset$ **et** $p^C(t) < \lambda$ **faire**
  - $t = \mathcal{T}$.pop()
  - ordonnancer t sur le CPU le moins chargé m_C
  - **si** m_C *est chargé au delà de* λ **alors**
    - **break**
- **si** $\mathcal{T} \neq \emptyset$ **alors**
  - **retourner** "Rejet : $C^*_{max} \geq \lambda$"

les tâches en maximisant l'affinité de l'ordonnancement. La deuxième partie de l'algorithme place les tâches restantes avec pour objectif de minimiser le temps de complétion de l'ordonnancement. Nous utilisons pour cette phase l'algorithme GDual.

Algorithme 2 : MultiDual(α, B)

Entrées : paramètre dual λ ; tâches $\mathcal{T}$; affinité aff_b ; temps d'exécution p

début

```
                                                    /* Phase 1: affinité */
```

soit $\mathcal{T}_A = \{t \in \mathcal{T} \mid p(t, m_{\text{aff}}(t)) < B\}$

trier $\mathcal{T}_A$ par $\frac{\text{aff}_b(t)}{p(t, m_{\text{aff}}(t))}$ décroissants

pour chaque $t \in \mathcal{T}_A$ **faire**

  si $m_{\text{aff}}(t)$ *n'est pas chargé au-delà de* αB **alors**

    ordonnancer t sur $m_{\text{aff}}(t)$

```
                                                    /* Phase 2: Cmax */
```

ordonnancer les tâches restantes avec GDual

Lemme 2. MultiDual(α, B) détermine un ordonnancement vérifiant $\Phi^*(B) \leq \frac{1}{\alpha}\Phi$.

Idée de démonstration. Les tâches considérés pour le pré-placement sont celles qui ont le meilleur apport d'affinité par rapport à leur temps d'exécution. En plaçant ces tâches en premier, il est possible de garantir le ratio. Le détail de la preuve peut être trouvé dans [20]. □

Lemme 3. MultiDual(α, B) détermine un ordonnancement dont le temps de complétion est au plus $(1+\alpha)B + 2C^*_{max}$.

Démonstration. Pour montrer cette garantie, il nous suffit de montrer que le placement des tâches par la première phase de l'algorithme ne leur permet pas de terminer après $(1+\alpha)B$. Supposons, par l'absurde, qu'il existe une tâche t placée durant la première phase qui termine après $(1+\alpha)B$. Par construction seules les tâches dont le temps d'exécution est inférieur à B sont considérées. Ainsi la tâche t n'a pu commencer son exécution au plus tôt après αB, ce qui est amène la contradiction.

En vertu du lemme 1 l'algorithme GDual détermine un ordonnancement ne dépassant pas $2C^*_{max}$. La garantie globale de notre algorithme en découle directement. □

Lemme 4. MultiDual est un algorithme de complexité $O(n \cdot log(n))$.

Démonstration. Le pré-placement des tâches nécessite le parcours d'une liste triée. D'autre part l'ordonnancement par GDual a recours lui aussi à un parcours de liste triée. Par conséquent la complexité est $O(n \cdot log(n))$. □

3.4 Implémentation

3.4.1 *Runtime* XKaapi

XKaapi est un projet logiciel d'environ 250 000 lignes de code dont le noyau – écrit en C – est constitué de 30 000 lignes de code. La partie qui s'occupe de l'ordonnancement

compte pour 10 % du noyau. Comme mentionné rapidement dans la sous-section 2.3.3, XKaapi est basé sur une approche *dataflow* pour représenter les applications parallèles. Le *runtime* a vocation à être utilisé pour faire du calcul distribué, ce qui influe sur son architecture.

À chaque CPU est associé une file contenant les tâches locales qu'il doit exécuter. Dès que le CPU finit d'exécuter une tâche, il appelle l'ordonnanceur. Du fait de la nature distribuée du *runtime*, l'algorithme d'ordonnancement n'a connaissance que des tâches locales et de celles qui ont été poussées. Prendre des décisions de manière distribuée n'est pas pénalisant en pratique, la plupart des applications reposent sur quelques tâches « maître » qui construisent le graphe par itérations.

Les GPUs de la machine sont pilotés par un CPU dédié. Les versions GPU des tâches sont encapsulées dans des tâches spéciales à destination du CPU associé. Ce dernier est chargé de la gestion de ces tâches spéciales et des transferts vers / depuis le GPU qu'il gère. Il est à noter que contrairement aux tâches CPU classiques, des limitations techniques empêchent ces tâches d'en créer d'autre.

Définir une politique d'ordonnancement dans XKaapi revient à définir des fonctions qui seront appelées par le *runtime* pendant le cycle de vie des tâches. L'ordonnanceur peut agir à différents moments : avant ou après l'exécution d'une tâche ; lors d'un vol de tâche ; à la création d'une tâche ou lorsqu'une tâche devient active (c.-à-d. lorsque ses contraintes de précédence sont satisfaites).

3.4.2 Intégration de l'algorithme

Le travail d'intégration dans XKaapi de l'algorithme que nous proposons (cf. section 3.3) représente un ajout d'une centaine de lignes de code par rapport à l'implémentation de GDual [2]. Cette contribution s'est divisée en trois phases.

Tout d'abord, il nous fallu choisir une fonction d'affinité aff. Notre choix s'est porté vers une affinité basée sur la localité des données pour plusieurs raisons. Premièrement, de nombreuses références (notamment [1, 24]) indiquent qu'optimiser la localité est un moyen simple d'améliorer les performances. Enfin, XKaapi est un *runtime* avec une approche *dataflow*. Une telle approche – basée sur les données des tâches – permet d'avoir facilement une bonne connaissance du placement des données au sein de la machine.

La conception et l'analyse théorique de l'algorithme MultiDual utilise un seuil arbitraire B qui permet de séparer les deux phases d'ordonnancement. Le choix que nous avons fait pour l'implémentation dans XKaapi est de prendre B égal au paramètre dual. Ce seuil arbitraire peut être vu comme le temps de complétion visé comme mentionné plus tôt. Étant donné que le paramètre dual est l'estimation du temps de complétion, cette décision semble cohérente pour une première implémentation.

D'autre part, nous avons modifié l'implémentation déjà existante de GDual. En effet, MultiDual peut être vu comme une exécution de GDual précédé d'un préplacement de certaines tâches.

2. L'implémentation de GDual compte 650 lignes de code.

#CPU	#cœurs / CPU	fondeur	modèle	fréquence	mémoire
2	6	Intel	Xeon X5650	2,67 GHz	72 Gio

#GPU	fondeur	modèle	fréquence	mémoire	Capacité CUDA
8	NVIDIA	Tesla C2050	1,15 GHz	3Go GDDR5	2.0

TABLE 3.1 – Caractéristiques détaillées des ressources de calcul d'`idgraf`.

3.5 Validation expérimentale

3.5.1 Conditions expérimentales

Hardware

La validation expérimentale des performances de l'implémentation a été réalisée sur `idgraf`, un serveur de Calcul Haute Performance géré par Grid'5000. Ce serveur contient douze cœurs de calcul généraliste et huit GPUs. Les capacités d'*hyper-threading* et de *thermal throttling* n'ont pas été activées pour nos expériences. Les caractéristiques détaillées des ressources de calcul sont reportées dans la table 3.1. La topologie d'`idgraf` est visible sur la figure 2.1 (cf. page 4). Il est à noter que les huit GPUs sont organisés de manière hiérarchique et qu'ils partagent – par groupe de deux – un même bus PCI. Une telle configuration peut créer des phénomènes de contention lorsque deux GPUs partageant un bus cherchent à communiquer simultanément.

Software

Afin de conserver un environnement identique pour toutes les expériences, une image contenant tous les logiciels nécessaires a été déployée[3] sur `idgraf`. L'image déployée utilise Debian squeeze au dessus du noyau Linux 3.2.0.2-amd64. Les applications ont été compilées et *linkées* avec les bibliothèques du NVIDIA CUDA Toolkit v5.0 ainsi que celles du PLASMA Installer 2.4.6. Ces applications proviennent du jeu d'exemples fournis par XKaapi.

3.5.2 Validation

Dans le but de valider notre implémentation, nous avons utilisé une décomposition de Cholesky. Nous avons réalisé trente exécutions pour chaque configuration. Sur chaque graphe l'intervalle des valeurs prises est représenté par une barre verticale. Nous comparons les performances de l'implémentation de notre algorithme à celles de GDUAL et HEFT. Nous nous intéressons plus particulièrement au placement initial des tâches par l'algorithme, à cet effet le vol de tâche a été désactivé.

Nous validons dans un premier temps l'analyse théorique sur les temps de complétion puis nous étudions l'impact du choix de α sur l'ordonnancement déterminé.

3. `https://www.grid5000.fr/mediawiki/index.php/Deploy_environment-OAR2`

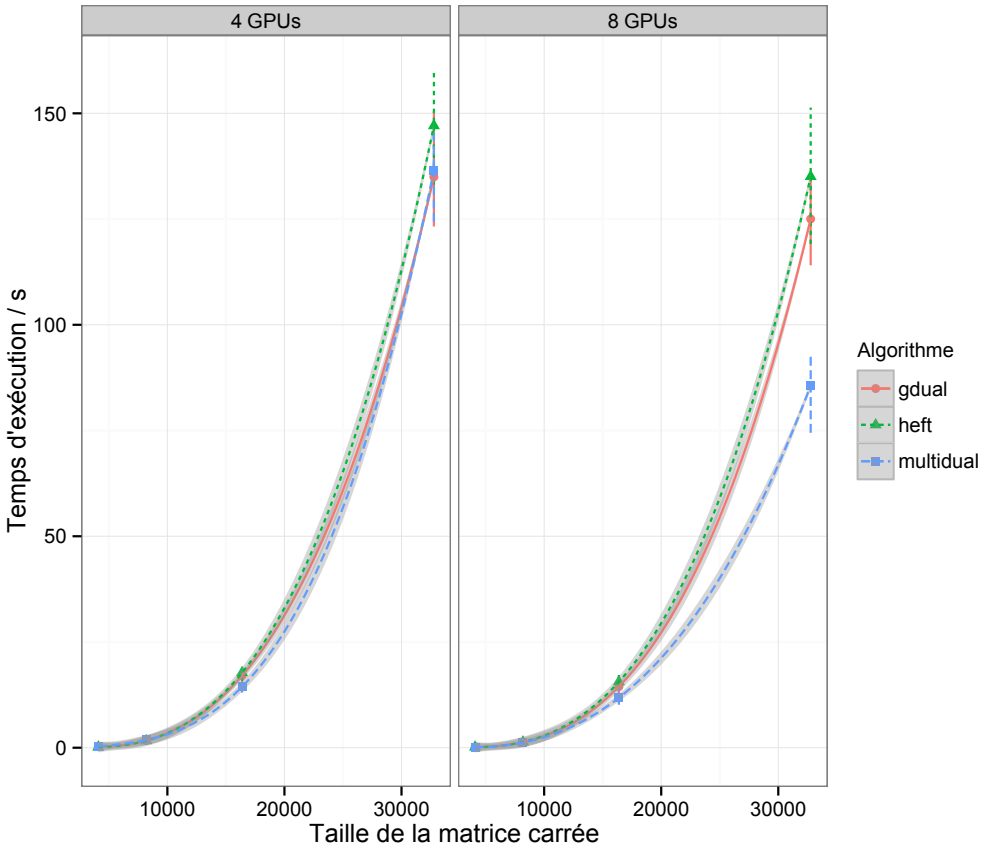


FIGURE 3.1 – Comparaison de MULTIDUAL à deux algorithmes de référence en fonction du nombre de GPUs. Le nombre de CPUs est fixé à quatre.

Temps de complétion

Les deux courbes de la figure 3.1 reportent les temps d'exécution des algorithmes HEFT, GDUAL et MULTIDUAL (avec $\alpha = 0.5$) pour deux configurations de calcul. Nous rappelons que la borne théorique sur le temps de complétion est $(1+\alpha)B+2C^*_{max}$. Les temps de complétion des placements des tâches par MULTIDUAL sont comparables à ceux de GDUAL lorsqu'il y a quatre GPUs et sont en deçà en présence de huit GPUs : ceci valide notre borne théorique.

Nous remarquons qu'avec quatre GPUs, les performances des trois algorithmes sont comparables. Par contre, les performances de MULTIDUAL sont bien meilleures avec huit GPUs. Un tel comportement s'explique par la topologie d'`idgraf`. Avec huit GPUs activés, les bus PCI arrivent à saturation, provoquant de la contention sur les communications. L'algorithme GDUAL considère les GPUs comme un ensemble et génère des communications supplémentaires au sein de ce groupe.

Impact du paramètre α

Pour étudier l'influence du paramètre α sur l'ordonnancement, nous avons réalisé plusieurs exécutions de l'application en activant tous les GPUs. La figure 3.2 reporte les résultats de ces exécutions pour différentes valeurs de α entre 0 et 1. Plus la valeur

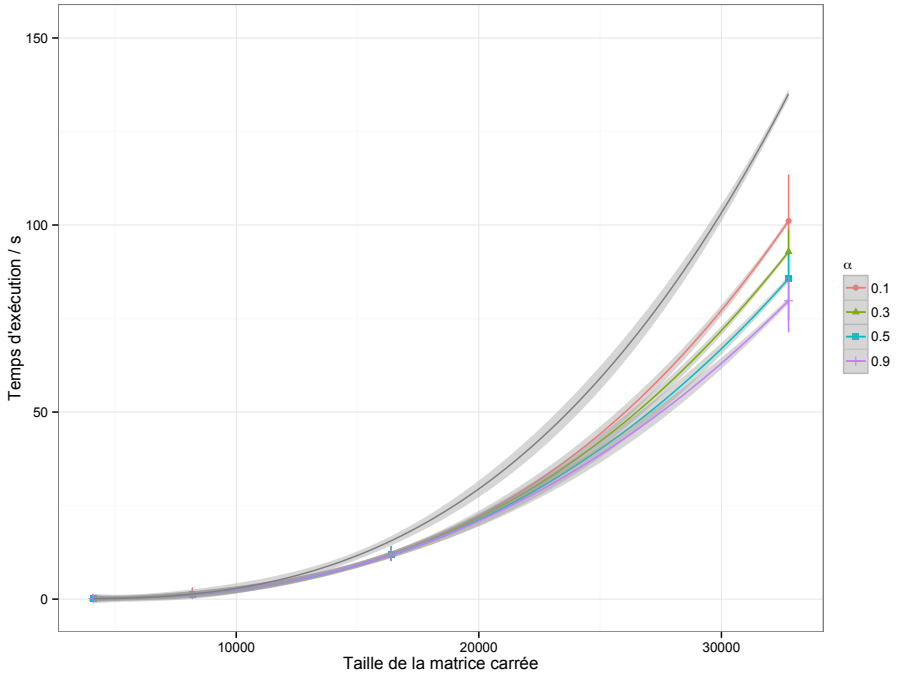


FIGURE 3.2 – Influence du paramètre α sur le temps d'exécution de l'application. Les exécutions ont utilisé quatre CPUs et huit GPUs. La courbe la plus haute représente les temps obtenus avec HEFT.

de α est petite, moins l'affinité est prise en compte. L'apport de la prise en compte de l'affinité n'est pas significatif pour les petites matrices. En effet la quantité de communication est relativement faible. Par contre, l'algorithme arrive à mieux mobiliser les ressources pour les grandes matrices. Ainsi les temps d'exécution sont significativement plus grands quand $\alpha = 0.1$, corroborant la présence de contention. À l'inverse, alors que la borne théorique en pire cas prévoit une augmentation du temps total, une grande valeur de α permet un gain substantiel sur les temps d'exécution.

Cette tendance se remarque mieux sur les débits applicatifs (cf. figure 3.3). L'algorithme HEFT a de meilleures performances sur les petites matrices, qui ne nécessitent pas suffisamment de communication pour saturer les pus PCI. Par contre, les besoins en communication induits par la prise en compte de la localité moins fine de HEFT le pénalise.

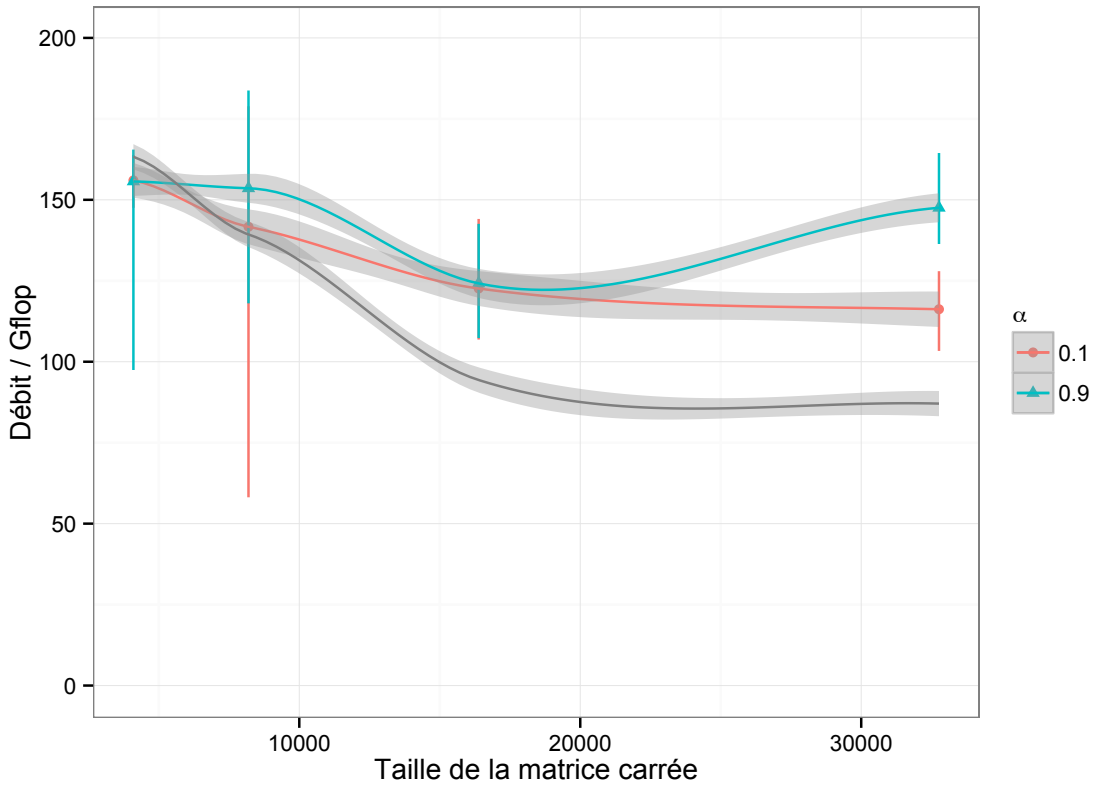


FIGURE 3.3 – Influence du paramètre α sur les débits applicatifs. La courbe la plus basse représente les débits obtenus HEFT.

Chapitre 4

Organisation du travail

4.1 Structure d'accueil

L'objectif de l'équipe-projet MOAIS est de développer les fondations techniques et scientifiques permettant d'améliorer les performances des architectures parallèles distribuées.

Les efforts de recherche sont centrés sur les problèmes d'ordonnancement multi-critères : précision, réactivité, utilisation des ressources, fiabilité. L'originalité de l'approche est d'utiliser l'adaptivité de l'application pour permettre son contrôle par l'ordonnancement. Quatre axes sont principalement explorés par l'équipe :

Ordonnancement Formaliser et analyser les problèmes d'ordonnancement.

Algorithmique Concevoir et analyser des algorithmes qui peuvent donner le contrôle de leur exécution à l'ordonnancement.

Programmation parallèle Spécifier et implémenter les interfaces permettant d'exprimer des contraintes de synchronisation.

Interactivité Afin d'améliorer l'interactivité, l'emphase est mise sur la scalabilité.

4.2 Cahier des charges

La finalité de ce stage est de proposer un algorithme d'ordonnancement prenant en compte les spécificités des machines hybrides, c.-à-d. contenant des processeurs généralistes et des accélérateurs de calcul de type GPU. Le cahier des charges comportent quatre points :

1. Étudier et concevoir un algorithme d'ordonnancement multi-critère pour des machines hybrides.
2. Réaliser une étude théorique des performances dans le pire cas de l'algorithme proposé.
3. Valider de manière expérimentale les performances de l'algorithme proposé.
4. Diffuser au sein de la communauté scientifique les résultats obtenus.

4.3 Plan de travail

Le stage se découpe naturellement en trois phases principales : faire une étude bibliographique et cerner l'existant en premier lieu ; concevoir puis implémenter un algorithme et enfin diffuser les résultats.

Organisation prévisionnelle

Le planning prévisionnel a été établi aux alentours du 2 avril 2013. La figure 4.1 reporte le diagramme de Gantt reprenant les différents jalons et étapes. À cette époque, l'état de l'art ainsi que la prise en main de l'environnement technique avaient été réalisés. Il était prévu de passer trois semaines à concevoir un algorithme multi-critère, puis trois semaines à analyser ses performances. Nous envisagions un mois pour réaliser l'implémentation et trois semaines pour valider expérimentalement ses performances. Nous prévoyions de rédiger ce rapport en parallèle du travail d'expérimentation.

Organisation effective

Le stage se découpe en trois phases comme prévu initialement. Toutefois, l'étude d'un nouvel algorithme accuse des différences avec le planning prévu. La phase de conception a été très courte : une fois le schéma de Stein et Wein identifié et le critère choisi, l'algorithme est pour ainsi dire conçu. La phase d'analyse a par contre pris beaucoup plus de temps qu'escompté. D'une part, la recherche d'un meilleur ratio d'approximation garanti a demandé plus d'efforts qu'envisagé. L'implémentation a été plus rapide que prévu : l'existence d'une implémentation de l'algorithme GDual dans XKaapi nous a permis de compenser une partie du temps pris en plus pour l'analyse.

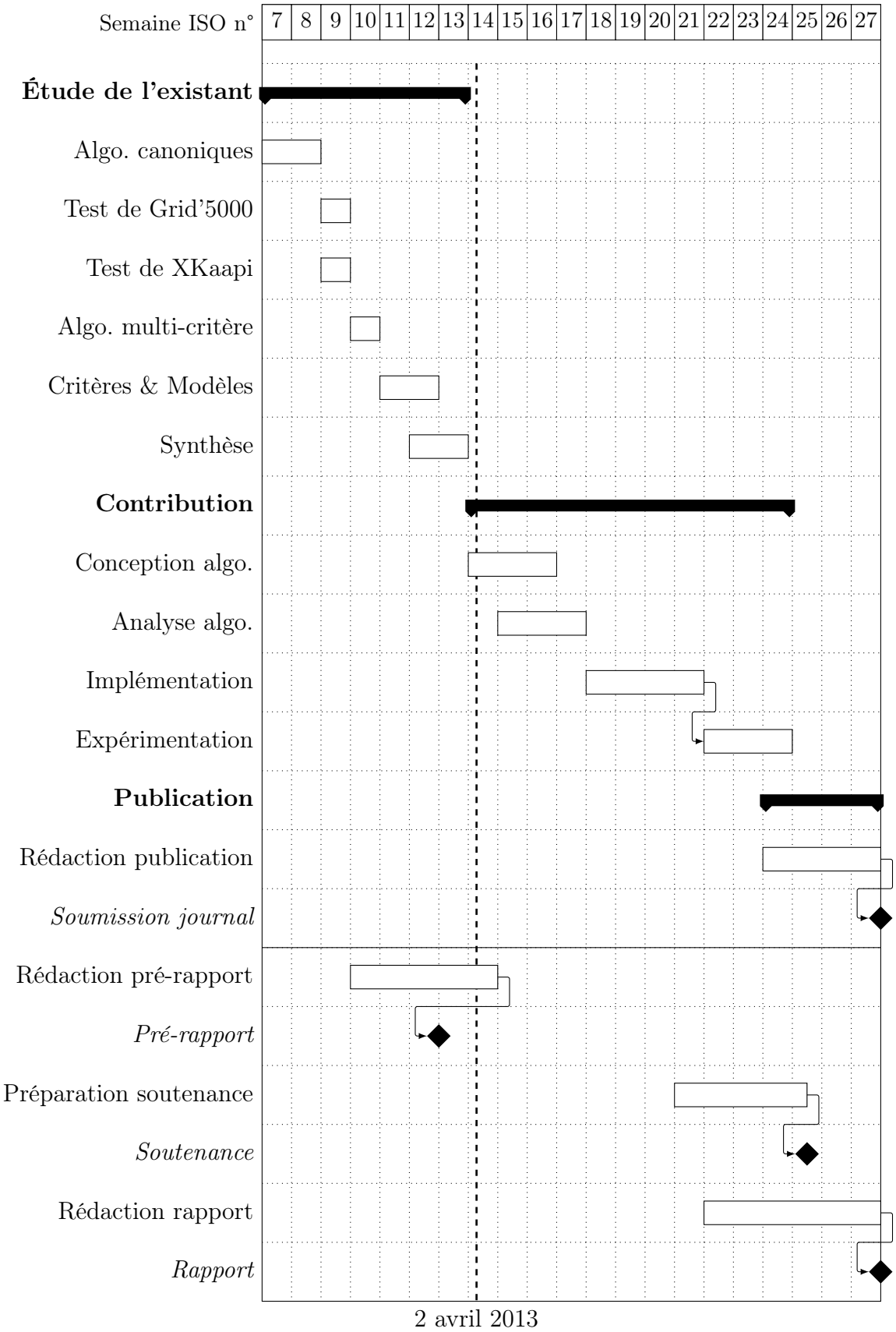


FIGURE 4.1 – Organisation prévisionnelle en date du 2 avril 2013.

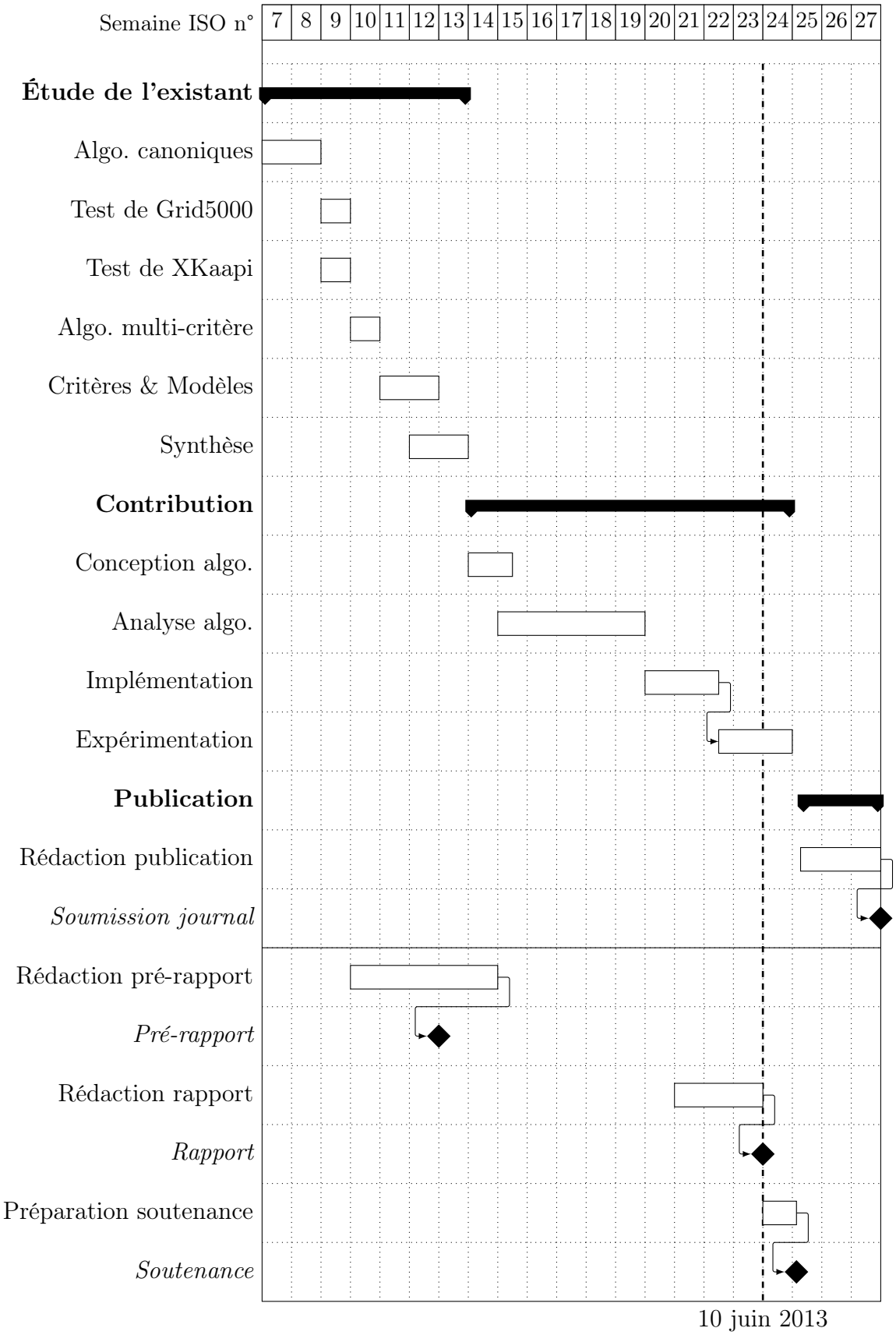


FIGURE 4.2 – Organisation effective en date du 10 juin 2013.

Chapitre 5

Conclusion

Au sein des super-calculateurs, l'utilisation efficiente des ressources est indispensable. L'ordonnancement – étudié depuis une cinquantaine d'années par la communauté de la recherche opérationnelle – permet d'allouer efficacement le travail aux ressources de calcul. Le problème étant $\mathcal{NP}$-dur dans le cadre général les recherches se sont portées vers la création d'heuristiques dont les performances sont garanties tout en ayant des complexité de calcul raisonnables. Les principales heuristiques utilisées pour minimiser le temps de complétion sont les algorithmes de liste et le vol de travail, respectivement introduites par Graham et Blumofe. L'approche duale de Shmoys et les algorithmes multi-critères sont des schémas algorithmiques fréquemment utilisés dans notre cadre. Afin de gagner en performance à des coûts raisonnables, les constructeurs intègrent maintenant des accélérateurs de calcul GPU. Alors que cette évolution entraîne une complexification des machines, les techniques utilisées aujourd'hui par la communauté du Calcul Haute Performance commencent à montrer leurs limites. Nous avons étudié pendant ce projet de fin d'études le problème de l'ordonnancement de tâches indépendantes sur des machines multi-CPU / multi-GPU avec pour objectifs la maximisation de l'affinité et la minimisation du temps de complétion. Nous avons conçu l'algorithme MultiDual et exhibé des bornes théoriques sur ses performances : $\Phi^*(B) \leq \frac{1}{\alpha}\Phi$ pour l'affinité et $C_{max} \leq (1+\alpha)B + 2C^*_{max}$ pour le temps de complétion. Enfin, nous avons validé de manière expérimentale les performances de l'algorithme proposé.

Plusieurs pistes n'ont pas été abordées et mériteraient d'être approfondies. Concernant l'algorithme, l'évolution logique serait d'augmenter le type de ressources considérées ou de considérer des fonctions d'affinité non binaire. Au niveau de l'implémentation, la manière de choisir les paramètres α et B influe sur les performances. Il serait judicieux d'étudier comment choisir ces paramètres au plus juste en fonction de l'application et de la taille des données traitées.

Bilan personnel

Ces quatre mois passés au sein de l'équipe-projet MOAIS me laisse une impression très positive vis à vis du monde de la recherche. Travailler sur des sujets de pointe et novateurs demande de la créativité et de l'initiative personnelle. Il est d'autre part très stimulant de travailler sur de tels sujets : en effet cela demande de la rigueur et il faut savoir remettre en cause son cheminement intellectuel.

Sur le plan technique, ce projet de fin d'études m'a permis d'approfondir ma connaissance du monde du Calcul Haute Performance. J'ai notamment perfectionné mes compétences de programmation en C en abordant les problématiques de l'écriture de code performant.

Mais plus que tout, ce stage me conforte dans l'idée qu'il faut chercher à élargir son panel de connaissance au-delà de son sujet premier d'étude. Ceci permet d'avoir le recul indispensable à l'étude approfondie dudit sujet.

Bibliographie

[1] Acar, U. A., Blelloch, G. E., and Blumofe, R. D. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures* (2000), pp. 1–12. (cf. pages 8, 10 et 18)

[2] Alber, J., and Niedermeier, R. On multidimensional curves with Hilbert property. *Theory of Computing Systems 33*, 4 (2000), 295–312. (cf. page 10)

[3] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures* (1998), pp. 119–129. (cf. page 8)

[4] Bender, M. A., and Rabin, M. O. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems Special Issue on SPAA 35*, 3 (2002), 289–304. (cf. page 8)

[5] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on* (1994), pp. 356–368. (cf. pages 8 et 11)

[6] Butz, A. R. Alternative Algorithm for Hilbert's Space-Filling Curve. *Computers, IEEE Transactions on C-20*, 4 (1971), 424–426. (cf. page 10)

[7] Cordeiro, D. *The impact of cooperation on new high performance computing platforms*. PhD thesis, Université de Grenoble, Feb 2012. (cf. page 12)

[8] Culler, D. E., Karp, R. M., Patterson, D., Sahay, A., Santos, E. E., Schauser, K. E., Subramonian, R., and von Eicken, T. LogP : towards a realistic model of parallel computation. *SIGPLAN Not. 28*, 7 (jul 1993), 1–12. (cf. page 11)

[9] Culler, D. E., Karp, R. M., Patterson, D., Sahay, A., Santos, E. E., Schauser, K. E., Subramonian, R., and von Eicken, T. LogP : a practical model of parallel computation. *Commun. ACM 39*, 11 (nov 1996), 78–85. (cf. page 11)

[10] Dutot, P.-F., Rzadca, K., Saule, E., and Trystram, D. Multi-Objective Scheduling. In *Introduction to Scheduling*, Y. Robert and F. Vivien, Eds., Chapman & Hall/CRC Computational Science. CRC Press, Nov 2009, pp. 219–251. (cf. page 9)

[11] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. Parallel Computing Experiences with CUDA. *Micro, IEEE 28*, 4 (2008), 13–27. (cf. page 3)

[12] Gautier, T., Besseron, X., and Pigeon, L. KAAPI : A thread scheduling runtime system for data flow computations on cluster of multi-processors. In

Proceedings of the 2007 international workshop on Parallel symbolic computation (2007), pp. 15–23. (cf. page 8)

[13] Gautier, T., Ferreira Lima, J. V., Maillard, N., and Raffin, B. XKaapi : A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (Boston, Massachusetts, États-Unis, 2013). (cf. pages 3, 4 et 8)

[14] Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures. In *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)* (2013). (cf. pages 3 et 8)

[15] Gautier, T., Roch, J.-L., and Wagner, F. Fine Grain Distributed Implementation of a Dataflow Language with Provable Performances. In *Computational Science – ICCS 2007*, Y. Shi, G. Albada, J. Dongarra, and P. A. Sloot, Eds., vol. 4488. Springer Berlin Heidelberg, 2007, ch. Lecture Notes in Computer Science, pp. 593–600. (cf. page 8)

[16] Graham, R. L. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal 45*, 9 (1966), 1563–1581. (cf. page 6)

[17] Graham, R. L. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (1969), 416–429. (cf. page 6)

[18] Hochbaum, D. S., and Shmoys, D. B. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM) 34*, 1 (1987), 144–162. (cf. pages 9 et 15)

[19] Hungershöfer, J., and Wierum, J.-M. On the quality of partitions based on space-filling curves. *Computational Science—ICCS 2002* (2002), 36–45. (cf. page 10)

[20] Jeannot, E., Saule, E., and Trystram, D. Bi-objective Approximation Scheme for Makespan and Reliability Optimization on Uniform Parallel Machines. In *Euro-Par 2008 – Parallel Processing*, E. Luque, T. Margalef, and D. Benítez, Eds., vol. 5168. Springer Berlin Heidelberg, 2008, ch. Lecture Notes in Computer Science, pp. 877–886. (cf. pages 13, 14, 15 et 17)

[21] Kedad-Sidhoum, S., Monna, F., Mounié, G., and Trystram, D. Scheduling Independent Tasks on Multi-Cores with GPU Accelerators. Submitted to HeteroPar workshop of Europar 2013, 2013. (cf. pages 7 et 15)

[22] Kovatch, P., Ezell, M., and Braby, R. The Malthusian Catastrophe Is Upon Us ! Are the Largest HPC Machines Ever Up ? In *Euro-Par 2011 : Parallel Processing Workshops*, M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. Vallée, and J. Weidendorfer, Eds., vol. 7156. Springer Berlin Heidelberg, 2012, ch. Lecture Notes in Computer Science, pp. 211–220. (cf. page 12)

[23] Narang, A., Srivastava, A., Kumar, N., and Shyamasundar, R. Affinity Driven Distributed Scheduling Algorithm for Parallel Computations. In *Distributed Computing and Networking*, M. Aguilera, H. Yu, N. Vaidya, V. Srinivasan, and R. Choudhury, Eds., vol. 6522. Springer Berlin Heidelberg, 2011, ch. Lecture Notes in Computer Science, pp. 167–178. (cf. page 9)

[24] Narlikar, G. J. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems 35*, 2 (2002), 151–187. (cf. pages 11 et 18)

[25] Papadimitriou, C. H., and Yannakakis, M. On the approximability of trade-offs and optimal access of web sources. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on* (2000), pp. 86–92. (cf. page 10)

[26] Quintin, J.-N., and Wagner, F. Hierarchical Work-Stealing. In *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds., vol. 6271. Springer Berlin Heidelberg, 2010, ch. Lecture Notes in Computer Science, pp. 217–229. (cf. page 8)

[27] Robert, Y., and Vivien, F. *Introduction to scheduling.* Chapman & Hall/CRC Computational Science. CRC Press, 2009. (cf. pages 3 et 4)

[28] Sagan, H. *Space-Filling Curves.* Universitext. Springer New York, 1994. (cf. page 10)

[29] Shmoys, D. B., and Tardos, E. Scheduling unrelated machines with costs. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms* (Philadelphia, PA, USA, 1993), SODA '93, Society for Industrial and Applied Mathematics, pp. 448–454. (cf. page 13)

[30] Shmoys, D. B., Wein, J., and Williamson, D. P. Scheduling parallel machines on-line. *SIAM Journal on Computing 24*, 6 (1995), 1313–1331. (cf. pages 4 et 6)

[31] Stein, C., and Wein, J. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operations Research Letters 21*, 3 (1997), 115–122. (cf. page 15)

[32] Tchiboukdjian, M., Danjean, V., and Raffin, B. Binary mesh partitioning for cache-efficient visualization. *IEEE Trans Vis Comput Graph 16*, 5 (2010), 815–28. (cf. page 10)

[33] Tchiboukdjian, M., Gast, N., Trystram, D., Roch, J. L., and Bernard, J. A tighter analysis of work stealing. In *International Symposium on Algorithms and Computation (ISAAC)* (2010), pp. 291–302. (cf. page 8)

[34] Topcuoglu, H., Hariri, S., and Wu, M.-Y. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth* (1999), pp. 3–14. (cf. page 7)

[35] Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 3 (1975), 384–393. (cf. page 6)