



HAL
open science

Scheduling Under Memory Constraint in Task-based Runtime Systems

Maxime Gonthier

► **To cite this version:**

Maxime Gonthier. Scheduling Under Memory Constraint in Task-based Runtime Systems. Distributed, Parallel, and Cluster Computing [cs.DC]. Ecole normale supérieure de lyon - ENS LYON, 2023. English. NNT: 2023ENSL0061 . tel-04260094

HAL Id: tel-04260094

<https://theses.hal.science/tel-04260094v1>

Submitted on 26 Oct 2023

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.



THÈSE
en vue de l'obtention du grade de Docteur, délivré par
l'ÉCOLE NORMALE SUPÉRIEURE DE LYON

École Doctorale N°512
École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

Soutenue publiquement le 25/09/2023, par :

Maxime GONTHIER

**Scheduling Under Memory Constraint in
Task-based Runtime Systems**

*Ordonnancement Sous Contrainte Mémoire dans un Modèle de
Programmation à Base de Tâches*

Devant le jury composé de :

Camille	COTI	Professeure, École de Technologie Supérieure Montréal	<i>Rapporteuse</i>
Julien	LANGOU	Professeur, University of Colorado Denver	<i>Rapporteur</i>
Thomas	HERAULT	Professeur, The University of Tennessee Knoxville	<i>Examineur</i>
Isabelle	TERRASSE	Personnalité scientifique, Airbus	<i>Examinatrice</i>
Loris	MARCHAL	Chargé de Recherche HDR, CNRS, LIP	<i>Directeur de thèse</i>
Samuel	THIBAUT	Professeur des universités, Université de Bordeaux	<i>Directeur de thèse</i>

Scheduling Under Memory Constraint in Task-based Runtime Systems

Abstract: Hardware accelerators, such as GPUs, now provide a large part of the computational power used for scientific simulations. GPUs come with their own limited memory and are connected to the main memory of the machine via a bus with limited bandwidth. Scientific simulations often operate on very large data, to the point of not fitting in the limited GPU memory. In this case, one has to turn to out-of-core computing: data are kept in the CPU memory, and moved back and forth to the GPU memory when needed for the computation. This out-of-core situation also happens when processing on multi-core CPUs with limited memory huge datasets stored on disk.

In both cases, data movement quickly becomes a performance bottleneck. Task-based runtime schedulers have emerged as a convenient and efficient way to manage large applications on such heterogeneous platforms. They are in charge of choosing which tasks to assign on which processing unit and in which order they should be processed.

During this thesis, we worked on the problem of scheduling for a task-based runtime to improve data locality in an out-of-core setting, in order to reduce data movements. We designed strategies for both task scheduling and data eviction from limited memories. We implemented them in the STARPU runtime and compared them to existing scheduling techniques in runtime systems. Our strategies achieves significantly better performance when scheduling tasks on multiple GPUs with limited memory, as well as on multiple CPU cores with limited main memory.

We also worked on batch scheduling of IO intensive workloads. Similarly, we used data locality techniques to reduce the average latency of a job.

Keywords: Memory-aware scheduling, Eviction policy, Tasks sharing data, Runtime systems, Data locality, GPUs, CPUs, Job input sharing, Job scheduling

Research Units

ROMA, ENS Lyon, 46 Allée d'Italie, 69007 Lyon and STORM, Inria Bordeaux Sud-Ouest, 200
Avenue de la Vieille Tour, 33400 Talence

Remerciements

Le début de cette thèse a été pour moi comme une navigation à l'aveugle dans un océan d'incertitudes. Je ne connaissais ni mes encadrants, ni les villes de Lyon et de Bordeaux, ni personne dans ces régions, j'avais un peu peur. Finalement, ce fut un voyage extrêmement enrichissant en expériences, rencontres et apprentissages, tous plus positifs les uns que les autres. Tout cela a été possible grâce à une multitude de personnes que je tiens à remercier.

En premier lieu, je souhaite exprimer ma gratitude envers le jury. Je tiens à remercier Julien et Camille pour avoir pris le temps de relire attentivement ce long manuscrit. J'espère que cette lecture vous a été agréable et je vous remercie pour vos retours. Un grand merci également à Thomas et Isabelle pour leur rôle d'examineurs. J'ai été ravi de répondre à vos questions, toutes particulièrement intéressantes.

En second lieu, je voudrais remercier toutes les personnes qui ont pris le temps de relire et de contribuer, de près ou de loin, à l'élaboration de ce manuscrit. Je tiens donc à remercier Loris, Samuel, Mihail, Olivier, Radja, Laércio, Gwenolé, Amina, Emmanuelle, Lionel, Nathalie, Nicolas, Laureline et ma mère.

La première étape de mon périple fut de descendre le Rhône et d'accoster à Lyon. C'est là que j'ai fait la rencontre de Loris. J'ai grandement apprécié nos échanges. Grâce à tes encouragements, tes retours positifs et ta capacité inépuisable à générer de nouvelles idées, j'ai toujours travaillé en toute confiance. J'avais alors l'assurance que j'atteindrai bon port à l'issue de ces trois années. Je te suis reconnaissant pour le temps que tu as investi tout au long de cette thèse, que ce soit lors de nos nombreuses réunions ou par le biais de tes multiples contributions. J'avais ainsi le sentiment de ne pas travailler pour un directeur de thèse, mais bel et bien aux côtés d'un chercheur. Un grand merci également à toute l'équipe ROMA pour son accueil chaleureux. Grégoire, Anne, Yves, Frédéric, Bora, Anthony et Lucas, merci beaucoup.

Un an plus tard, j'ai achevé ma descente du Rhône, traversé le détroit de Gibraltar pour rejoindre le port de la Lune. C'est là que j'ai été chaleureusement accueilli par Samuel. Merci infiniment, Sam, de m'avoir fait ressentir le sentiment de chez-moi si rapidement. Je te suis reconnaissant pour le temps que tu as consacré à travailler avec moi sur tous mes problèmes liés à StarPU, parfois pendant des heures. J'ai véritablement apprécié nos échanges. Tu es une source inépuisable de conversations, toujours intéressantes. Merci à toute l'équipe STORM pour ces innombrables heures de discussions passées dans la salle café. Merci Emmanuelle pour tous tes précieux conseils sur la vie en postdoc. Mihail, merci pour tes séances de psychothéra-courses (en tout bien tout honneur). Amina je te remercie pour ta bienveillance et tes encouragements. Merci aussi à Olivier, Nathalie, Raymond, Marie-Christine, PAW, Laércio, Van Man et Denis.

J'ai aussi rencontré les membres de l'open space, qui ont rendu cette thèse mille fois plus amusante et j'aimerais les remercier. Vincent, merci à toi. Ton pardessus te donne un air de méchant tout droit sorti d'un film de James Bond, j'aime beaucoup. Merci Romain, finalement, j'aime bien ta tasse. Baptiste, je te remercie pour tes multiples propositions de pauses café quotidiennes. Gwenolé, merci pour ton sens de l'humour, ton professionnalisme et ta créativité en matière de déguisements. Merci Diane pour nos séances d'escalades véloces. Alice, tes fanfictions sont toujours un régal, merci. Merci Lana pour les super chaussons requins. Merci Kun de me forcer à faire des cookies. Merci Célia pour ta bonne humeur contagieuse. Radja, merci de courir plus vite que le bus. Merci Thomas de reprendre le flambeau des thèses StarPU avec autant de brio. Merci Albert pour nos sorties aux BT.

J'ai également eu l'occasion de discuter avec de nombreuses personnes d'autres équipes. Je remercie Alycia pour les montagnes de poils de Fifi que tu m'as généreusement laissées. Merci aussi à Lionel, Abdou, Mathieu, Olivier B, Philippe S, Philippe V, ainsi qu'à l'ensemble des équipes TADAAM, TOPAL et CONCACE.

I then sailed my boat toward the Baltic Sea for a 3-month stay at Uppsala University. It was a wonderful time that taught me how to work with new people and how to adapt to a new environment. I discovered the natural beauty of Sweden, but more importantly, the kindness of its people.

Thank you, Marina, for facilitating my arrival at Uppsala and for organizing so many events for me to join. Watching the big fire in Gamla is still vivid in my memory. Thanks to all my friends on the 6th floor of Sernanders väg 4. Thanks to Camille for introducing me to outdoor climbing. Thanks to Ivo, Lukas, Sonja, Gesina, Andreas, and everyone at the Division of Scientific Computing.

And, of course, thank you, Carl, for all your insightful advice on the direction our project should take. A special thanks to Elisabeth for being exceptionally welcoming and for inviting me to her birthday celebration in the beautiful Kallmyr. I hold my time in Uppsala dear, and much of that is thanks to you.

Une embarcation ne navigue pas sans une charpente solide. Celle-ci a été construite au fil du temps par ma famille. Merci à mes parents de m'avoir toujours inculqué l'amour de la science et de la culture. J'apprécie toujours autant de visiter des musées et de m'intéresser à la géographie, et c'est cette curiosité qui m'a aussi grandement aidé à mener à bien cette thèse. Je vous remercie également de m'avoir assisté lors de deux (bientôt trois) déménagements. À mon frère, un grand merci pour m'avoir montré la voie de la science et de la recherche. Grâce à toi, il m'a été beaucoup plus facile de me lancer dans cette aventure. Je te remercie aussi de m'avoir écouté parler de mes problèmes pendant trois ans, et d'être toujours aussi attentionné. À ma sœur, je tiens à te remercier pour ta bonne humeur constante, nos soirées sous les étoiles filantes et d'être toujours là pour moi. Tu es une grande sœur en or.

Un voyage en mer c'est encore mieux lorsqu'on peut rester en contact avec des amis sur le continent. Merci à Guillaume, pour être un ami extraordinaire, et ce depuis presque 10 ans (et merci pour les 70 euros). À Amandine et Bylitis, merci d'être toujours des amies fidèles, et ce depuis 13 ans ! C'est toujours un plaisir de vous voir et de savoir que je peux compter sur vous. Merci à Ben, je passe toujours d'excellents moments avec toi, même après de longues périodes de séparation. Merci à tous les membres de DOàT pour nos soirées du Nouvel An, la béchamel, le limoncello et les anniversaires. Un merci tout particulier également à Doruntine, Joseph, Fares, Hamza, Stéphane et tant d'autres que j'oublie sûrement.

Enfin je voudrais remercier Laureline, sans qui mon navire aurait sombré à coup sûr. Tu as rendu ces trois années absolument merveilleuses. À la maison, je me sentais bien, j'étais écouté, soutenu et con-

seillé. Merci d'avoir accepté de déménager à deux reprises avec moi. Merci de ne m'avoir jamais jugé et de me comprendre comme personne d'autre. Notre voyage ensemble comporte encore de nombreux écueils, mais je suis sûr que nous atteindrons de plus beaux rivages dans un futur proche.

Merci à Fifi d'être aussi mignon et d'avoir servi de petit canard pour mes problèmes dans StarPU tout au long de cette thèse, ce qui fait de toi, sans aucun doute, un ingénieur StarPU hors-pair.

Mon voyage m'amène maintenant à traverser l'Atlantique, et grâce à vous tous, je n'ai plus peur.

Résumé en français



LES SCIENTIFIQUES mènent des efforts constants afin d'améliorer les performances de leurs applications, que ce soit pour accroître leur précision ou pour s'attaquer à des problèmes plus grands. Une puissance de calcul importante est nécessaire pour obtenir des résultats précis dans les domaines tels que la prévision météorologique, la prédiction de tremblements de terre ou la simulation de flux d'air. Les applications susmentionnées dépendent également de l'utilisation de grandes quantités de données d'entrée. Pour les applications modernes, la puissance de calcul et les besoins mémoire sont si importants que des machines dédiées, appelées supercalculateurs, sont devenues une nécessité. Les supercalculateurs ont donné naissance au domaine du calcul haute performance (HPC), qui se concentre sur l'optimisation de leur utilisation afin de maximiser leurs capacités. Une réalité bien connue dans le domaine du calcul haute performance est qu'il existe un lien intrinsèque entre la puissance de calcul des supercalculateurs et leur mémoire.

Depuis ENIAC [70], le premier ordinateur électronique et programmable, construit en 1945, la gestion de la mémoire a toujours été une contrainte. La mémoire était alors constituée de 18 000 tubes à vide, et il en fallait 36 pour stocker un seul nombre en base 10. Les tubes à vide nécessitaient tellement d'énergie que plusieurs d'entre eux grillaient presque chaque jour, ce qui rendait l'ordinateur inutilisable la moitié du temps. La partie la plus critique du fonctionnement de l'ordinateur était donc le stockage de données.

Au fil des progrès technologiques, la nécessité d'une gestion efficace de la mémoire est restée d'actualité. On peut citer par exemple l'ordinateur Cray X-MP [41], construit en 1983 pour être l'ordinateur le plus rapide du monde. Il était composé de quatre CPUs avec 64 mégaoctets de mémoire partagée et pouvait atteindre une performance de pointe de près de 1 gigaflop/s, c.-à-d., 10^9 opérations à virgule flottante par seconde. Lorsque la taille de l'ensemble des données d'entrée dépassait la mémoire du CPU, il était alors nécessaire de transférer les données depuis le disque dur à l'aide d'un câble dont la bande passante ne dépassait pas quelques mégaoctets par seconde. Afin d'éviter de ralentir les calculs, il était donc important de gérer le moment et la manière d'accéder aux données.

Afin d'atteindre des vitesses de calcul sans précédent, la tendance de la dernière décennie consiste à utiliser des GPUs en plus des CPUs. Depuis 2022, grâce à l'utilisation de plus de 37 000 GPUs, le supercalculateur Frontier [53] est la machine la plus puissante du monde, avec une performance de 1 exaflop/s, c.-à-d., 10^{18} opérations à virgule flottante par seconde. Les GPUs sont rapides et massivement parallèles, mais n'intègrent qu'une mémoire relativement limitée. Lorsque les utilisateurs essaient de résoudre des applications très volumineuses, il est courant que toutes les données d'entrée du problème ne peuvent pas tenir dans la mémoire des unités de calcul. Pour des GPUs, cela signifie qu'il faut transférer les données depuis la mémoire d'un CPU en utilisant un bus dont la bande passante est limitée. Bien que les GPUs aient une puissance de calcul de l'ordre de milliers de gigaflop/s, les bus ont com-

munément une bande passante de l'ordre d'une douzaine de milliers de mégaoctets/s. Cette différence de rapport de 100 est un parfait exemple de l'écart grandissant entre la vitesse de calcul et les bandes passantes de communication. Cette évolution s'accompagne également d'une diminution de la mémoire par gigaflop/s. Ce qui signifie que le problème ne peut pas être résolu en chargeant toutes les données d'entrée en mémoire. Cet écart constitue un goulot d'étranglement pour les performances. Cela motive les chercheurs en calcul haute performance à travailler sur l'optimisation des techniques de gestion de la mémoire afin d'utiliser tout le potentiel des supercalculateurs.

Dans cette thèse, nous visons à combler cette lacune en répondant au problème suivant : *Comment minimiser le temps d'exécution d'une application dont l'ensemble des données est plus grand que la mémoire des unités de calcul utilisées ?* Une réponse possible consiste à améliorer le matériel utilisé. Cependant, l'ajout de mémoire est coûteux et ne constitue pas une solution permanente : si l'ensemble des données devient de plus en plus important, on ne peut pas ajouter de la mémoire indéfiniment. L'ajout de bus supplémentaires peut être une option pour améliorer les performances. Cependant, cela est difficile car les bus sont coûteux, consomment beaucoup d'énergie et prennent de la place dans un nœud de calcul déjà physiquement limité.

La réponse que nous proposons est plus générique et repose sur l'amélioration du logiciel d'un support d'exécution. Les optimisations logicielles génériques peuvent être appliquées à n'importe quel matériel et constituent donc un objectif complémentaire aux améliorations matérielles. Pour exploiter la puissance des plateformes hétérogènes complexes, il est devenu très courant d'utiliser la programmation à base de tâches, c.-à-d., d'exprimer le calcul de l'application sous la forme d'un graphe acyclique dirigé (DAG), et de laisser un support d'exécution dynamique gérer l'exécution du graphe de tâches sur de telles plateformes. La charge de l'allocation des données en mémoire, du choix de l'ordre de traitement des tâches et de leur répartition sur les unités de calcul est ainsi retirée au programmeur de l'application. Cette charge est alors gérée par le support d'exécution sous la forme d'un problème d'ordonnement de tâches. Étant donné que le support d'exécution gère à la fois les données et les tâches, il est alors possible de créer des ordonnanceurs qui minimisent les mouvements de données. Cela améliore les performances lorsque la mémoire est une contrainte. La création de politiques d'ordonnement est une approche plus générale car elle peut être appliquée à n'importe quelle architecture matérielle ou application à base de tâches. Notre objectif est donc de construire un *ordonneur générique qui partitionne et ordonne des tâches sur une ou plusieurs unités de calcul à mémoire limitée, et qui apporte de meilleures performances que les stratégies d'ordonnement actuelles*. Pour atteindre cet objectif, dans le Chapitre 1 nous étudions d'abord le contexte dans lequel cette thèse a eu lieu, nous passons en revue les travaux connexes, et nous détaillons comment nous avons l'intention de résoudre le problème énoncé précédemment. Le Chapitre 2 présente la manière dont nous simplifions et modélisons le problème. Le Chapitre 3 présente une solution algorithmique pour l'ordonnement statique de tâches indépendantes sur une seule unité de calcul. Nous étendons ensuite cette solution à plusieurs unités de calcul dans le chapitre 4 et proposons un nouvel ordonnanceur dynamique. Le Chapitre 5 décrit les améliorations apportées à notre ordonnanceur dynamique et les expériences menées avec des ensembles de tâches avec dépendances. Dans le Chapitre 6, nous transférons les leçons tirées de l'ordonnement avec localité à l'ordonnement *batch*¹ en introduisant de nouveaux algorithmes pour les systèmes batch. Les chapitres sont résumés ci-dessous.

¹Le concept *batch* désigne les techniques appliquées à l'échelle d'un supercalculateur pour gérer des *travaux*. Les travaux sont des applications ou codes soumis par des utilisateurs à un supercalculateur. Ainsi batch peut désigner de l'ordonnement de travaux sur un supercalculateur appelé ordonnancement batch, des systèmes de gestion de travaux appelé systèmes batch ou alors un simulateur de travaux appelé simulateur batch.

Chapter 1: Contexte et Revue de la Littérature

Dans ce premier chapitre, nous présentons le contexte dans lequel cette thèse a été conduite : le besoin d'exécuter des applications d'algèbre linéaire dont les données d'entrée ne tiennent pas dans la mémoire d'une unité de calcul. Nous expliquons également la motivation qui justifie l'approche que nous avons choisie pour résoudre ce problème. Le Chapitre 1 développe ensuite les travaux existants sur ce problème et comment ils interviennent à différents niveaux de l'architecture matérielle (du cache aux systèmes batch). Nous justifions ensuite notre choix de positionnement par rapport à la proximité de l'architecture matérielle. Enfin, nous expliquons comment nous avons réduit l'écart entre l'ordonnancement théorique et l'ordonnancement pratique.

Chapter 2: Énoncé de la Problématique et Intégration dans un Support d'Exécution

Le Chapitre 2 exprime notre problème à l'aide de différents modèles théoriques. Le modèle le plus simple est basé sur l'ordonnancement de tâches indépendantes sur une seule unité de calcul à mémoire limitée. En utilisant cette simplification, nous prouvons l'optimalité d'une politique d'éviction et démontrons la complexité de notre problème. Nous complexifions ensuite notre modèle en ajoutant plusieurs unités de calcul, des poids hétérogènes et des ensembles de tâches avec dépendances. Nous présentons également STARPU, le support d'exécution que nous avons utilisé pour implémenter nos algorithmes et mener nos expériences. Pour répondre à nos besoins, nous introduisons dans STARPU la capacité d'ajouter des politiques d'éviction personnalisées ainsi qu'un nouvel outil de collecte de données et de visualisation.

Chapter 3: Ordonnancement Statique pour une Seule Unité de Calcul [R1, IP1, W1, J1]

Une première solution algorithmique est proposée en considérant un problème d'empaquetage, que nous prouvons être NP-complet. À partir de ce problème d'empaquetage, nous avons développé un ordonnanceur appelé Hierarchical Fair Packing (HFP), qui regroupe des tâches partageant des données. Pour l'évaluer, nous adaptions deux méthodes de la littérature que nous estimons pertinentes par rapport à notre problème. Les trois stratégies mentionnées sont implémentées dans le support d'exécution STARPU. Nous évaluons expérimentalement les trois méthodes ainsi qu'un ordonnanceur de référence et un ordonnanceur de pointe et utilisons des variantes de la multiplication matricielle. Le Chapitre 3 contient ensuite une description des évaluations expérimentales et une explication des résultats grâce à notre outil de visualisation présenté dans le chapitre précédent.

Chapter 4: Exploiter la Puissance de Plusieurs GPUs [IP2, C1]

Alors que le chapitre précédent utilisait une seule unité de calcul, le Chapitre 4 se concentre sur les défis de l'ordonnancement de tâches sur plusieurs unités de calcul, chacune avec sa propre mémoire limitée. Le chapitre commence par une description du nœud de calcul utilisé. Cette description souligne l'importance de la localité spatiale lorsque sont utilisés plusieurs mémoires locales, en montrant comment l'ordonnanceur de pointe utilise une telle architecture. Nous présentons ensuite une technique de partitionnement de graphe, que nous avons adapté à notre problème en l'étendant avec du vol de tâches. Nous expliquons ensuite comment HFP est adapté pour devenir une stratégie de partitionnement et d'ordonnancement pour plusieurs unités de calcul : en y ajoutant des techniques d'équilibrage de charges et de vol de tâches. Nous présentons également une nouvelle stratégie dynamique appelée DARTS (Dynamic Data-Aware Reactive Task Scheduling), également implémenté dans STARPU. L'intuition de DARTS est de prendre en compte la localité des données avant l'allocation des tâches. DARTS utilise

l'état de la mémoire des unités de calcul afin de choisir les données à charger et maximiser la réutilisation des données. Pour illustrer les performances de nos ordonnanceurs, le chapitre présente des résultats expérimentaux sur différentes variantes de la multiplication matricielle et des sous-ensembles de la factorisation de Cholesky.

Chapter 5: Ordonnement Dynamique pour les Graphes de Tâches [N1, P1, R3]

Ce chapitre se concentre sur le problème d'ordonnement d'ensembles de tâches avec dépendances sur plusieurs unités de calcul à mémoire limitée. Nous décrivons des algorithmes existants et largement utilisés dans les support d'exécution : une politique de vol de travail avec localité et un ordonnanceur du support d'exécution PARSEC. Nous retravaillons la version de DARTS présentée dans le dernier chapitre afin de favoriser le transfert rapide de données, gérer les dépendances et inclure les priorités dans la prise de décision. Nous avons également travaillé à réduire la complexité de DARTS. Pour démontrer l'efficacité de ces techniques d'ordonnement, le chapitre présente des études expérimentales utilisant GEMM, les factorisations Cholesky et LU, et en utilisant un ou plusieurs GPUs ou des cœurs CPU.

Chapter 6: Tirer Parti de la Localité pour les Ordonneurs Batch [R2]

Les travaux présentés dans ce chapitre ont été menés dans le cadre d'une collaboration avec Elisabeth LARSSON et Carl NETTELBLAD. Cette collaboration s'est accompagné d'un séjour de trois mois à l'université d'Uppsala en Suède. Nous avons adapté notre sujet d'étude afin de répondre à leurs besoins de recherche tout en gardant l'accent sur l'ordonnement sous contrainte mémoire. L'université d'Uppsala exploite une plateforme de calcul à haute performance utilisée par des chercheurs pour soumettre des travaux utilisant de grandes quantités de données. Ces travaux nécessitent le chargement de fichiers d'entrée de plusieurs gigaoctets avant de pouvoir commencer leurs exécutions. Les ordonneurs batch traditionnels ne sont généralement pas conçus pour gérer des ensembles de travaux intense en Entrée/Sortie. Avec ces ensembles de travaux, les temps de chargement peuvent devenir significatifs, augmentant ainsi le temps d'attente pour tous les travaux. C'est pourquoi, dans le Chapitre 6, nous proposons de modéliser les avantages de la réutilisation de données entre des travaux successifs. Nous avons développé un simulateur batch et introduit de nouveaux ordonneurs batch qui ajoutent une telle réutilisation de données. En suivant quelles données sont chargées sur quel nœud, ils sont capables de réduire de la quantité de transfert de données, améliorant ainsi à la fois l'utilisation des ressources et la satisfaction des utilisateurs. Nous évaluons ces algorithmes en utilisant des traces de soumissions de travaux réellement observées sur la plateforme de calcul de l'université d'Uppsala et étudions les performances obtenues après avoir ordonné près de 2 millions de travaux.

Contents

Remerciements	v
Résumé en français	ix
Introduction	1
1 Background and Literature Review	5
1.1 Context	6
1.1.1 Three real-world examples	6
1.1.2 Hardware will not save us...	7
1.1.3 ... but maybe software can	9
1.1.4 What happens when the memory is full?	9
1.1.5 Problem statement	10
1.2 Related works	11
1.2.1 Cache management	11
1.2.2 Partitioned global address space	12
1.2.3 Solutions in runtime systems	12
1.2.4 An out-of-core middleware	13
1.2.5 Scheduling for distributed platforms	14
1.2.6 Out-of-core and communication-avoiding algorithms	14
1.2.7 Locality-aware mapping	15
1.2.8 Locality-aware mapping and ordering	15
1.3 Positioning in the hardware hierarchy	15
1.4 Bridging the gap between theoretical scheduling and runtime schedulers	17
2 Problem Statement and Integration into a Runtime System	21
2.1 Simplifying our optimization problems	22
2.1.1 Expressing applications as task graphs	22
2.1.2 Avoiding the conflicting goals of using multiple processing units	23
2.1.3 Considering homogeneous processing time and data size	23
2.1.4 Making the model complex again	24
2.2 Simplified model with an independent task set and a single processing unit	24
2.2.1 Expressing applications as a bipartite graphs	24
2.2.2 Simplified optimization problem	25

2.2.3	Optimal eviction policy proof	26
2.2.4	Complexity of finding an optimal task order	27
2.3	Making the model parallel	28
2.3.1	Adding the partitioning problem to the bipartite graph	28
2.3.2	Optimization problem in parallel	29
2.4	Extension to heterogeneous task and data weights	29
2.5	Adding dependencies to the model	30
2.6	The STARPU Runtime System	31
2.6.1	Task and data	32
2.6.2	Tasks submission	32
2.6.3	Task flow	34
2.6.4	New functionality to add custom eviction policies	34
2.6.5	New logging and visualization tool	37
2.7	Summary	41
3	Static Scheduling for a Single Processing Unit	43
3.1	Schedulers from the STARPU runtime system	44
3.1.1	A greedy baseline: EAGER	44
3.1.2	Deque Model Data Aware Ready (DMDAR)	44
3.2	Adapted strategies from the literature	45
3.2.1	Reverse-Cuthill-McKee (RCM)	45
3.2.2	Maximum Spanning Tree (MST)	47
3.3	Hierarchical Fair Packing (HFP)	48
3.3.1	Intuition	48
3.3.2	An NP-complete problem	48
3.3.3	Strategy	49
3.3.4	Complexity of HFP	51
3.3.5	Improving HFP with package flipping	52
3.3.6	Optimal eviction policy	53
3.3.7	Adaptation to heterogeneous data sizes	54
3.3.8	Improving the beginning of the schedule with the <i>Ready</i> re-ordering	54
3.4	Experimental settings	54
3.5	Experimental results and analysis	57
3.5.1	Results on the 2D matrix multiplication	57
3.5.2	Results on the 3D matrix multiplication	62
3.5.3	Results on the task set of the Cholesky factorization	65
3.5.4	Results on the 2D matrix multiplication with randomized task order	66
3.5.5	Results on the randomized pairs with 2D inputs	68
3.5.6	Results on the sparse 2D matrix multiplication	69
3.6	Conclusion on static scheduling for a single processing unit	70
4	Harnessing the Power of Multiple GPUs	73
4.1	State-of-the-art schedulers	74
4.1.1	Leveraging expected communication time with DMDAR	74
4.1.2	Using (hyper-)graph partitioning	76
4.2	Hierarchical Fair Packing adaptation to multiple processing units (mHFP)	77
4.2.1	Strategy	77

4.2.2	Additional unused solutions explored for mHFP	78
4.3	A dynamic data-aware scheduler: DARTS	81
4.3.1	Intuition	81
4.3.2	STARPU's task flow with DARTS	81
4.3.3	Strategy	81
4.3.4	Eviction policy	83
4.3.5	Dealing with more input data per task	84
4.3.6	Reducing the scheduling overhead	84
4.3.7	Faster code with fewer mutex	84
4.4	Experimental evaluation with multiple processing units	86
4.4.1	Settings	86
4.4.2	Results on the 2D matrix multiplication with a single GPU	86
4.4.3	Results on the 2D matrix multiplication with multiple GPUs	87
4.4.4	Result on the 2D matrix multiplication with randomized task order and 2 GPUs	93
4.4.5	Result on the 3D matrix multiplication with 4 GPUs	96
4.4.6	Result on the task set of the Cholesky factorization with 4 GPUs	96
4.4.7	Results on the sparse 2D matrix multiplication with 4 GPUs	97
4.5	Conclusion on scheduling for multiple processing units	97
5	Dynamic Scheduling for Task Graphs	99
5.1	Existing runtime schedulers	100
5.1.1	A work stealing policy: LWS	100
5.1.2	A priority-based scheduler from the PaRSEC runtime: AP	101
5.2	Improving the DARTS scheduler	101
5.2.1	Intuition	101
5.2.2	Strategy	101
5.2.3	Eviction policy	103
5.3	Experimental settings	103
5.4	Cholesky factorization with GPUs	104
5.4.1	Overview	104
5.4.2	Optimal data access pattern	105
5.4.3	Single GPU case	105
5.4.4	With multiple GPUs	107
5.4.5	With multiple GPUs and no memory limitation	110
5.5	LU factorization with GPUs	111
5.5.1	Results on 4 GPUs	111
5.5.2	Results on a single GPU and no memory limitation	112
5.6	3D matrix multiplication with GPUs	112
5.7	LU factorization on a multi-core CPU	114
5.8	Conclusion on dynamic scheduling of task sets with dependencies	116
6	Leveraging Locality for Batch Schedulers	117
6.1	Motivation	118
6.2	Related work	119
6.2.1	Scheduling jobs on large clusters	119
6.2.2	Using distributed file systems to deal with data-intensive workloads	119
6.2.3	Using schedulers to deal with data-intensive workloads	120

6.2.4	Reducing I/O contention	120
6.3	Framework	120
6.4	Schedulers	122
6.4.1	Two schedulers from the state of the art: FCFS and EFT	123
6.4.2	Data-locality-based schedulers	123
6.4.3	Adding backfilling to all strategies	125
6.5	Experimental settings	126
6.5.1	Platform description	126
6.5.2	Workloads description	126
6.5.3	Usage of real cluster logs	127
6.5.4	Simulator description	128
6.6	Experimental evaluation and analysis	129
6.6.1	Results on an underutilized cluster	129
6.6.2	Results on a saturated cluster	131
6.6.3	Complete results	132
6.7	Conclusion on locality-aware batch scheduling	135
	Conclusion and Perspectives	137
	Bibliography	143
	Publications	153
	Acknowledgement	155

List of Figures

1.1	Examples of datasets used for applications with dense linear algebra as the main computation phase.	7
1.2	Sources of computing performance over the last 50 years.	8
1.3	Platform topology of a multi-core CPU with shared memory.	9
1.4	Platform topology of multiple GPUs with distributed memory.	10
1.5	The hierarchy of solutions for the MIN-EXEC-TIME-LIMITED-MEM problem.	11
2.1	Task set from the Cholesky factorization.	23
2.2	Example of bipartite graph and processing order with one processing unit.	24
2.3	Example of bipartite graph and processing order with two processing units.	28
2.4	Task insertion of the Cholesky factorization within the STARPU runtime system.	33
2.5	Task flow within the STARPU runtime.	35
2.6	Function definitions needed for custom eviction policy within the STARPU runtime.	36
2.7	Visualization of the processing order on a 2D matrix multiplication. Side of the input matrices $N = 4$. The shading, from lighter to darker, represents the ordering. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches. With multiple GPUs, each color is a set of tasks assigned to a GPU.	37
2.8	Visualization of the processing order on a 3D matrix multiplication with 1 GPU. Side of the input matrices $N = 4$. The shading, from lighter to darker, represents the ordering. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.	38
2.9	Visualization of the processing order on the Cholesky factorization with 1 GPU. Side of the input matrices $N = 10$. The first 50 tasks processed are in red, the next 50 in green, then blue, yellow, magenta and cyan. The shading, from lighter to darker, represents the ordering within each set of 50 tasks. The black area represents the amount of tiles that can be loaded in memory.	39
2.10	Visualization of the processing order on the Cholesky factorization with 2 GPUs. Side of input the matrices $N = 10$. The first 50 tasks processed on each GPU are in red, the next 50 in green, then in blue. The shading, from lighter to darker, represents the ordering within each set of 50 tasks. The black area represents the amount of tiles that can be loaded in memory.	40

3.1	Reverse-Cuthill-McKee ordering on a symmetric sparse matrix.	46
3.2	Data sharing among tasks that reach worst-case complexity for HFP.	52
3.3	Flipping packages to improve HFP.	52
3.4	HFP's processing order on a 2D matrix multiplication with $M = 4$	53
3.5	Difference between HFP and HFP with <i>Ready</i> ordering on a 2D matrix multiplication with a single GPU.	54
3.6	Data dependencies on a 2D matrix multiplication.	56
3.7	Conventions used in experimental evaluation figures.	56
3.8	Results on the 2D matrix multiplication in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	57
3.9	Performance on the 2D matrix multiplication in real execution with 1 Tesla V100 GPU while varying the memory size.	58
3.10	Visualization of RCM's processing order on the 2D matrix multiplication.	60
3.11	Visualization of DMDAR's processing order on the 2D matrix multiplication.	60
3.12	Visualization of HFP's processing order on the 2D matrix multiplication.	61
3.13	Results on the 3D matrix multiplication in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	62
3.14	Performance on the 3D matrix multiplication in simulation with the performance model of 1 Tesla V100 GPU. Memory limited to 500 MB.	63
3.15	Visualization of HFP's processing order on the 3D matrix multiplication.	64
3.16	Performance on the task set of the Cholesky factorization in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	66
3.17	Performance on the random task order from from 2D matrix multiplication in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	67
3.18	Visualization of the processing order on the random task order from from 2D matrix multiplication.	67
3.19	Performance on the randomized pairs with 2D inputs in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	69
3.20	Performance on the sparse 2D matrix multiplication in real execution with 1 Tesla V100 GPU. Memory limited to 500 MB.	70
4.1	Topology of the Gemini node.	75
4.2	Difference between mHFP and mHFP ² on the 2D matrix multiplication with 3 GPUs.	79
4.3	Using interlacing with HFP.	80
4.4	Simplified task flow within the STARPU runtime when using DARTS.	82
4.5	Results of DARTS mutex policies on the 2D matrix multiplication with 4 Tesla V100 GPUs.	86
4.6	Results on the 2D matrix multiplication in real with 1 Tesla V100 GPU. Memory limited to 500 MB.	87
4.7	Performance on the 2D matrix multiplication in simulation with the performance models of 2 Tesla V100 GPUs.	88
4.8	Results on the 2D matrix multiplication in real with 2 Tesla V100 GPUs. Memory limited to 500 MB per GPU.	89
4.9	HFP's ordering on the 2D matrix multiplication with 2 Tesla V100 GPUs.	90
4.10	HFP's ordering on the 2D matrix multiplication with 8 Tesla V100 GPUs.	91
4.11	DMDAR's ordering on the 2D matrix multiplication with 2 Tesla V100 GPUs.	92
4.12	DARTS' ordering on the 2D matrix multiplication with 2 Tesla V100 GPUs.	94

4.13	Performance on the 2D matrix multiplication in real with 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.	95
4.14	Performance on the 2D matrix multiplication with randomized task order in real with 2 Tesla V100 GPUs. Memory limited to 500 MB per GPU.	95
4.15	Performance on the 3D matrix multiplication in simulation with the performance models of 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.	96
4.16	Performance on the task set of the Cholesky factorization in real with 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.	97
4.17	Performance on the sparse 2D matrix multiplication in real with 4 Tesla V100 GPUs.	98
5.1	Tiles computed by a triangle block in an iteration of the Cholesky factorization.	105
5.2	Results on the Cholesky factorization with 1 Tesla V100 GPU. Memory limited to 2000 MB.	106
5.3	DMDAS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU.	106
5.4	DARTS' ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU.	107
5.5	Results on the Cholesky factorization with 8 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.	108
5.6	DARTS' ordering on the first two iterations of the Cholesky factorization with 8 Tesla V100 GPUs.	109
5.7	Results on the Cholesky factorization with 8 Tesla V100 GPUs. Hardware limitation of each GPU memory at 32 GB.	110
5.8	Results on the LU factorization with 4 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.	111
5.9	Results on the LU factorization with 1 Tesla V100 GPU. Hardware memory limitation at 32 GB.	112
5.10	Performance on the 3D matrix multiplication with Tesla V100 GPUs. Memory limited to 2000 MB.	113
5.11	Performance on the 3D matrix multiplication in simulation with the performance models of Tesla V100 GPUs. Memory limited to 2000 MB.	114
5.12	Results on the LU factorization with an AMD EPYC 7642 CPU. Memory limited to 2000 MB.	115
6.1	Platform representation.	122
6.2	Methodology followed to schedule and evaluate jobs from a specific week while avoiding edge effects.	128
6.3	Results on the workload of week 40.	129
6.4	Stretch times of each user session from week 40 compared to FCFS.	131
6.5	Results on the workload of week 43.	132
6.6	Stretch times of each user session from week 43 using LEA compared to FCFS.	133
6.7	Results without backfilling on all evaluated weeks.	133
6.8	Results with backfilling on all evaluated weeks.	134

Introduction



SCIENTISTS are constantly striving for improved performance in their research applications, whether for enhancing accuracy or for tackling larger problems. For instance in domains such as weather forecasting, earthquake prediction or airflow simulation, significant computing power is required to achieve accurate results. All of the above applications are also highly dependent on large amounts of input data. The significance of computing power and memory requirements in modern applications is such that dedicated machines called supercomputers have become a necessity. With supercomputers came the field of High Performance Computing (HPC), which focuses on optimizing their utilization to maximize their capabilities. A well-known reality in HPC is that there has always been an intrinsic connection between the computing power of supercomputers and their memory.

Since ENIAC [70], the first programmable electronic computer built in 1945, dealing with memory has always been a constraint. The memory then consisted of 18 000 vacuum tubes, and it took 36 of them to store a single decimal number. Vacuum tubes required so much energy that several of them burned out almost every day, resulting in a non-functioning computer half the time. The most critical part when running the computer was data storage.

Throughout technological progress, the need for effective memory management remained relevant. For example, we can mention the Cray X-MP computer [41], released in 1983 to be the world's fastest computer. It consisted of four CPUs with a shared memory of 64 megabytes and could achieve a peak performance of almost 1 gigaflop/s, i.e., 10^9 floating-point operations per seconds. A dataset larger than the CPUs memory required to transfer data from a hard drive through a cable with a limited bandwidth of a few megabytes per second. Therefore, it was important to manage how and when the data was accessed in order to avoid computational slowdowns.

The last trend from the past decade is to leverage GPUs in addition to CPUs, to achieve unprecedented computing speed. Since 2022, the Frontier supercomputer [53], thanks to the use of more than 37 000 GPUs, is the most powerful machine in the world, with an achieved performance of 1 exaflop/s, i.e., 10^{18} floating-point operations per seconds. GPUs are fast and massively parallel but embed only relatively limited memory. With users trying to solve larger systems, it becomes common to encounter situations where all the input data of the problem cannot fit into the memory of the computing units. For GPUs, this means transferring data from a CPU memory using a bus with limited bandwidth. While they have a computing power on the order of thousands of gigaflop/s, the bus can typically exhibit a bandwidth on the order of a dozen thousands of megabyte/s. This ratio of 100 is a perfect example of the widening gap between computing speed and communication bandwidth. This is also accompanied by a decrease in memory per gigaflop/s, which means that problems cannot be solved by fitting all the input data into memory. This gap is a performance bottleneck and motivates HPC researchers to work on optimizing memory management techniques to unleash the full potential of supercomputers.

In this thesis, we aim to fill this gap by answering the following problem: *How to minimize the execution time of an application whose dataset is larger than the memory of the processing units being used?* One possible answer is to improve the hardware. However, the addition of more memory is expensive and not a permanent solution: if the dataset gets larger and larger, one cannot add memory indefinitely. Adding more buses to enhance the hardware can be an option for improving performance. However, this is difficult because buses are expensive, consume a lot of energy, and take up space in an already physically limited compute node.

Our proposed answer is more generic and relies on improving the software of a runtime system. Generic software optimizations can be applied to any hardware and are thus a complementary goal to hardware improvements. To harness the power of complex heterogeneous computing platforms, it has become very common to use task-based programming, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system manage the execution of the task graph over such distributed and heterogeneous platforms. The burden of allocating data in memory, choosing task processing order and mapping them is thus offloaded from the application programmer to the runtime system, in the form of a task scheduling problem. Since the runtime system handles both data and tasks, this provides an opportunity to build schedulers that minimize data movement, resulting in improved performance when memory is a constraint. Creating scheduling policies is a more general approach as it can be applied to any hardware or task-based application. Thus, our goal is to build *a generic scheduler that partitions and orders tasks across one or more processing units with limited memory, and brings higher performance than current scheduling strategies*. To achieve this goal, in Chapter 1 we first study the context in which this thesis took place, review related work, and detail how we intend to address the previously stated problem. Chapter 2 presents how we simplify and model the problem. Chapter 3 presents an algorithmic solution for static scheduling of independent tasks on a single processing unit. We then extend this solution to multiple processing units in Chapter 4 and propose a new dynamic scheduler. Chapter 5 describes improvements to our dynamic scheduler and experiments with dependent task sets. We take the opportunity to transfer the lessons learned from locality-aware scheduling to batch scheduling by introducing new algorithms for batch systems in Chapter 6. Each chapter is summarized below.

Chapter 1: Background and Literature Review

In this first chapter, we introduce the context in which this thesis was conducted: the need to execute linear algebra applications that does not fit in the memory of a processing unit. We also explain the motivation behind the approach chosen to solve this problem. Chapter 1 then elaborates on existing work on how this problem is addressed at different hardware levels (from cache to batch systems), before justifying our choice of positioning with regard to hardware proximity. Finally, we aim at bridging the gap between theoretical and practical scheduling.

Chapter 2: Problem Statement and Integration into a Runtime System

Chapter 2 expresses our stated problem with different theoretical models. The simplest model is based on the scheduling of independent tasks on a single processing unit with limited memory. Using this simplification, we prove the optimality of an eviction policy and demonstrate the complexity of our problem. We then complexify our model by adding multiple processing units, heterogeneous weights and task sets with dependencies. We also present STARPU, which is the runtime we used to implement our algorithms and conduct our experiments. To meet our needs, we introduce in STARPU the ability to use custom eviction policies as well as a new logging and visualization tool.

Chapter 3: Static Scheduling for a Single Processing Unit [R1, IP1, W1, J1]

A first algorithmic solution is provided by considering a packing problem, which we prove to be NP-complete. From the packing problem, we developed a scheduler called Hierarchical Fair Packing (HFP), which groups together tasks sharing data. To evaluate it, we adapt two methods from the literature that we believe may be relevant to our problem. The three mentioned strategies are implemented into the STARPU runtime. We experimentally evaluate the three methods along with a baseline and a state-of-the-art-runtime scheduler using variants of matrix multiplication and subsets of the Cholesky factorization. Chapter 3 discusses the results of experimental evaluations, with the help of the visualization tool presented in the previous chapter.

Chapter 4: Harnessing the Power of Multiple GPUs [IP2, C1]

While the previous chapter used a single processing unit, Chapter 4 focuses on the challenges of task-based scheduling on multiple processing units, each with its own limited memory. The chapter begins with a description of the used computing node. This description emphasizes the importance of spatial locality in a setting with multiple local memories by showing how the state-of-the-art runtime scheduler exploits such an architecture. We then introduce a graph partitioner, which we adapted to our problem by extending it with task-stealing methods. We then discuss how the HFP scheduler is adapted into a partitioning and ordering strategy by adding load-balancing and task-stealing to the strategy. We also introduce a new dynamic strategy called DARTS (Dynamic Data-Aware Reactive Task Scheduling), also implemented in STARPU. DARTS' intuition is to consider data locality before task allocation: it uses the state of the processing unit's memory to choose which data should be loaded to increase data reuse. To illustrate the performance of our schedulers, the chapter presents experimental results on different variants of matrix multiplication and subsets of the Cholesky factorization.

Chapter 5: Dynamic Scheduling for Task Graphs [N1, P1, R3]

This chapter focuses on the problem of scheduling task sets with dependencies on multiple processing units with limited memory. We describe existing algorithms that are widely used in runtime systems: a locality work stealing policy and a scheduler from the PaRSEC runtime. Based on the version of DARTS presented in the last chapter, we rebuilt it to favor fast data transfer, deal with dependencies, and include priorities in its decision making. We also worked on reducing the computational complexity of DARTS. To demonstrate the effectiveness of these scheduling techniques, the chapter presents studies on GEMM, the Cholesky and LU factorizations, using one or more GPUs as well as CPU cores.

Chapter 6: Leveraging Locality for Batch Schedulers [R2]

The work presented in this chapter was conducted as part of a research collaboration with Elisabeth LARSSON and Carl NETTELBLAD, during a three-month stay at Uppsala University in Sweden. We adapted our study to meet their specific research needs while keeping the focus on locality-aware scheduling under memory constraints. Uppsala University operates a high-performance computing platform that is used by researchers to submit highly data-dependent workloads. These workloads are jobs that require input files of multiple gigabytes to be loaded prior to computation. Traditional batch schedulers are generally not designed to handle data-intensive workloads. With those workloads, the load times can become significant, increasing the wait time for all jobs. Therefore, in Chapter 6 we propose to model the benefits of reusing data loads between successive jobs. We developed a batch simulator and introduce new batch schedulers that add such data reuse to the scheduling balance. By tracking which

data is loaded on which node for the scheduled jobs, they are able to significantly reduce data loads, thereby improving both resource utilization and user satisfaction. We evaluate these algorithms by using traces of actual job submissions observed on the Uppsala University cluster and study the performance obtained after scheduling nearly 2 million jobs.

Chapter 1

Background and Literature Review

Contents

1.1 Context	6
1.1.1 Three real-world examples	6
1.1.2 Hardware will not save us...	7
1.1.3 ... but maybe software can	9
1.1.4 What happens when the memory is full?	9
1.1.5 Problem statement	10
1.2 Related works	11
1.2.1 Cache management	11
1.2.2 Partitioned global address space	12
1.2.3 Solutions in runtime systems	12
1.2.4 An out-of-core middleware	13
1.2.5 Scheduling for distributed platforms	14
1.2.6 Out-of-core and communication-avoiding algorithms	14
1.2.7 Locality-aware mapping	15
1.2.8 Locality-aware mapping and ordering	15
1.3 Positioning in the hardware hierarchy	15
1.4 Bridging the gap between theoretical scheduling and runtime schedulers	17



SCIENTIFIC RESEARCH involves complex applications that require substantial computational resources. As scientific datasets continue to grow exponentially in size, researchers face a critical challenge: how to efficiently execute applications whose data exceeds the memory capacity of modern supercomputers? This is the foundation of this thesis. The context provided in Section 1.1 details the main challenge that modern scientific applications face: data. It also outlines different paths that can be taken to deal with such a challenge. After extracting a problem statement from it, we describe in Section 1.2 existing solutions from related work. Then, we explain in Section 1.3 how we choose to address the stated problem at two different levels of the hardware/software stack: runtimes and batch systems. Finally, in Section 1.4, we delve into the underlying challenges of using theoretical studies to build applied schedulers that work in runtime systems.

1.1 Context

Challenging and impactful problems are the driving factors of modern scientists. Examples of engaging and significant areas of study include weather forecasting, acoustic and airflow simulations and earthquake prediction. These examples have three characteristics in common.

Computational intensity: These domains heavily rely on sophisticated computational models to simulate and understand complex physical processes. For instance, weather forecasting models simulate atmospheric conditions, acoustic simulations replicate sound propagation, and seismic models analyze earth movements. These models involve intricate mathematical models and require significant computational resources to produce accurate results.

Solvable with linear algebra: Linear algebra provides a fundamental mathematical framework that enables the representation and resolution of such mathematical models.

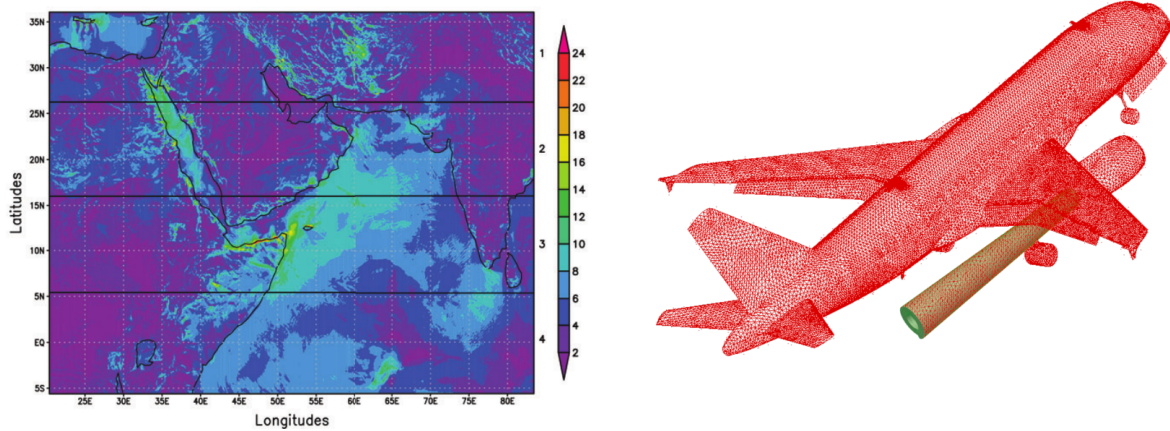
Data intensive: All of these examples require the processing of large amounts of data. These datasets consist of measurements, observations, or simulated models with data volumes that can sometimes reach thousands of gigabytes.

We can point to three real-world examples that are impactful, computational intensive, rely on the resolution of large linear algebra applications, and require a large amount of input data.

1.1.1 Three real-world examples

First, seismic tomography. It is a technique used in geophysics to create images of the Earth's subsurface structures by analyzing seismic waves. Seismic tomography plays a critical role in the assessment and prediction of natural hazards. It is also used to monitor volcanic activity. The recorded seismic data used as input is processed, typically reduced to solving a linear system such as the QR factorization, as shown in [27]. In this same paper, the authors state that the resolution matrix with $267\,520 \times 267\,520$ elements requires over 200 GB of memory. The more collected data, the better the resolution and accuracy of the resulting subsurface model. Therefore, seismic tomography requires large inputs, is useful to the masses, and is resolved by linear algebra.

Second, geostatistics. It is a field of science where scientists try to accurately model and predict environmental phenomena. For example, some scientists use past wind speed measurements as inputs and aim to output accurate wind speed predictions for the coming days. An example of wind speed data used is shown in Figure 1.1a. This is important because wind speed has an impact on various sectors of activity, such as the energy production of a wind farm or the planning of construction activities.



(a) Example of wind speed data used for geostatistics. Figure from [2].

(b) Example of discretization used for acoustic simulations. Figure from [10].

Figure 1.1: Examples of datasets used for applications with dense linear algebra as the main computation phase.

The EXAGEOSTAT software [1] is able to process such forecasts. In the case of wind prediction, EXAGEOSTAT uses large dense symmetric matrices that are solved using the Cholesky factorization [2]. They observed that they needed to move 3032 GB of data to produce the expected prediction.

Third, the simulation of acoustic waves generated by an aircraft engine. Measuring such propagation is a challenging and sometimes dangerous task. Therefore, numerical simulations are widely used in such cases. Airbus is interested in the simulation of aeroacoustic phenomena such as the propagation of acoustic waves generated by an aircraft during take-off or landing. This enables researchers to work on noise reduction for prototype airplanes. Such physical models involve concepts that cannot be modeled on a computer. Therefore, an approximation of the physical expression must be made using a discretization technique. Figure 1.1b shows an example of discretization. Discretization enables the simulation to be computed as large linear systems, which can then be partially solved using LU factorizations [10] and with matrices so large that they are divided into blocks of 160 GB across 80 nodes [20].

As the cited applications require lots of computing power, have large memory requirements, and must be resolved in a reasonable amount of time, they require the use of supercomputers. A supercomputer is a set of interconnected nodes, each containing memory, I/O systems, and processing units that effectively perform the computations. Since all of these applications are common scientific topics of study and rely on linear algebra for their completion, we can deduce that linear algebra occupies a non-negligible fraction of modern supercomputers. Solving linear algebra applications faster would free up many core hours for other important projects that do not rely on linear algebra results and require customized refinement. From this statement we can extract our first problematic: *What are the available solutions to solve large linear algebra applications faster on modern supercomputers?*

1.1.2 Hardware will not save us...

A common solution to improve the execution time of scientific applications is an upgrade of the supercomputers hardware. This means adding more CPUs, more memory, more GPUs, or making the chipsets smaller so that more can fit in the same space. It can also mean using more transistors. This has been a reasonable solution for the last five decades for three reasons. First, the Moore Law, an empirical observation that microprocessor technology improves +100% every 18 months, supported the

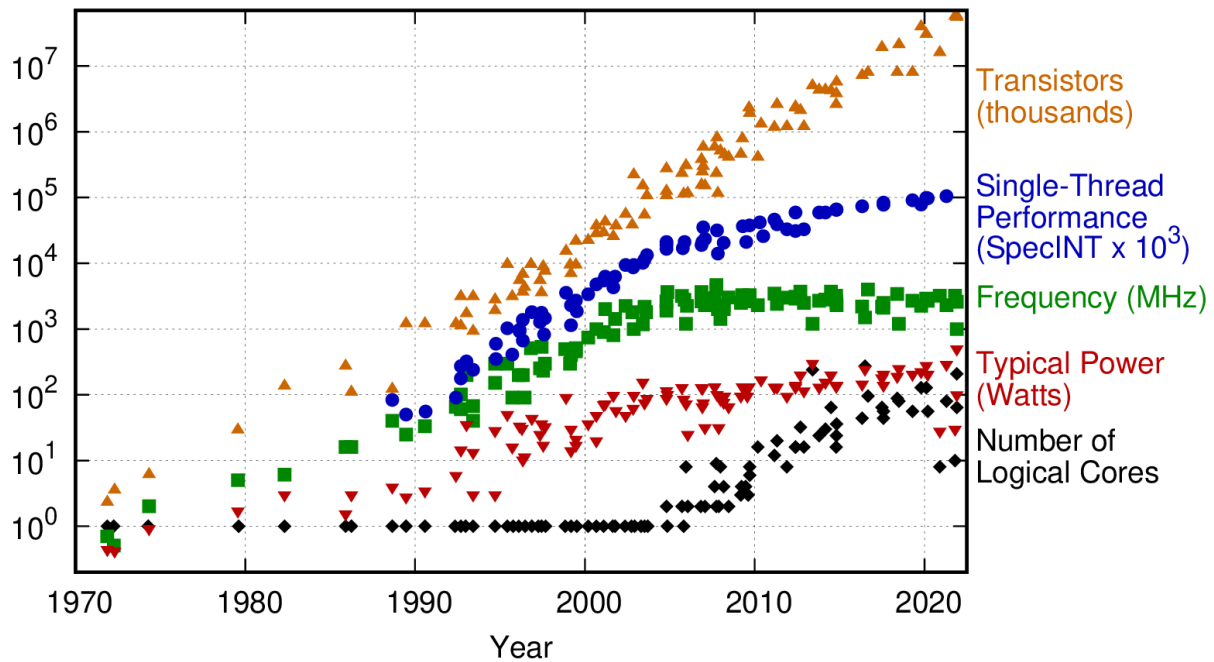


Figure 1.2: Sources of computing performance over the last 50 years. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Plot and data collected for 2010-2021 by Karl Rupp [109].

use of more cores. Second, the Dennard scaling, which states that as transistors get smaller, their power density remains constant, meaning that more can be added to a microprocessor. Third, large investments from the industry supported spending money on larger and larger computing platforms. Today, however, these three reasons are no longer in effect. Nowadays, we are at the end of the easy way to get more performance: Moore's Law is slowly reaching its limit. As we can see in Figure 1.2, the CPU frequency is reaching a ceiling and thread performance is increasing more slowly than before, eventually reaching an asymptote. Even if transistors keep getting smaller, we will run out of money before we run out of physics. It is possible to build smaller transistors, but the cost is not justified now that the progress is slowing down. Since there is a diminishing return on the investments in supercomputers, the large investments from the industry that we mentioned earlier are less and less profitable. To understand where HPC research is headed, we can draw a parallel with aircraft technologies. Nathan Myhrvold made the following statement: *"The way Moore's Law occurs in computing is really unprecedented in other walks of life. If the Boeing 747 obeyed Moore's Law, it would travel a million miles an hour, it would be shrunk down in size, and a trip to New York would cost about five dollars. Those enormous changes just aren't part of our everyday experience."* If airplanes obeyed Moore's Law, we would have supersonic airplanes for every known destination. We have seen the opposite since the Concorde. The economic cost of reaching higher and higher speeds was not realistic, so aeronautical engineers turned their attention to other issues, such as reducing fuel consumption, aircraft weight, or engine noise. Similarly, HPC must now pursue new goals. Some are studying energy efficiency or portability. Others solve applications faster by resorting to mixed precision when possible. If the application does not require floating-point operations, quantum computing could be used, but it is not currently applicable to high-performance computing. To solve linear algebra systems faster without sacrificing the precision of the results, a reasonable approach is to *optimize the software that runs the applications*.

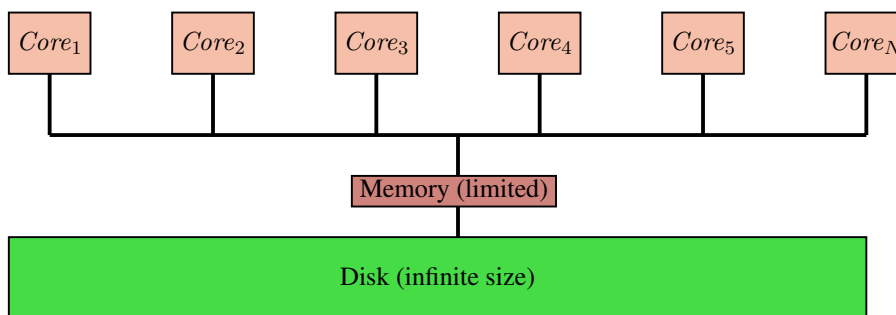


Figure 1.3: Platform topology of a multi-core CPU with shared memory.

1.1.3 ... but maybe software can

In software optimization, it is often very effective to optimize code for a specific architecture or even a specific processing unit. For example, some aim to optimize programs specifically for GeForce 8800 GTX GPUs [110], while others target specific supercomputing platforms. This makes sense, as it can meet the needs of users quite quickly. However, computing architectures and trends change rapidly. From the first high-performance computing platform (the Livermore Atomic Research Computer in 1956 [55]), to the use of PlayStation 2 connected together as a cluster in 2004 [102], and finally the Frontier supercomputer in 2022 [53], only 5 decades have passed. Moreover, in just half a generation, the popular trends in HPC have shifted from using CPUs to adding more and more GPUs or FPGAs to lastly focusing on quantum computing. These rapidly changing trends and architectures make permanent and generic approaches much more valuable. Such persistent solutions include algorithms that rely on theoretical models and are extensible to a large number of different situations on top of being architecture agnostic.

We give here two algorithmic solutions used to increase performance on linear algebra systems without relying on specific architectures or trends. First, one can increase parallelism, i.e., perform computations on multiple processing units at the same time. For example, this has been proposed for dense [72] or sparse matrix multiplication [19]. Ideally, parallelism is the perfect solution. Unfortunately, it is not possible to perfectly parallelize every application due to dependencies between operations or lack of available computing resources. So we turn to the second algorithmic solution: optimizing the partitioning and scheduling of applications on the different processing units of a supercomputer. Partitioning is defined as assigning different parts of the application to different processing units. Scheduling is defined as a method by which a task, specified as a subset of a larger application, is assigned to a processing unit to be computed at a given time. Since it is a solution that does not rely on hardware peculiarities or trends, we choose to focus on this partitioning and scheduling solution throughout this thesis. To optimize linear algebra applications through partitioning and scheduling, it is necessary to understand the major performance bottleneck of modern supercomputers: data movement. Hence, we ask ourselves: *How is the performance of linear algebra applications on modern supercomputers hampered by data movements?*

1.1.4 What happens when the memory is full?

There are two main processing units in modern supercomputers: CPUs and GPUs. All cores of a multi-core CPU share a common (limited) memory, as depicted on Figure 1.3. With users trying to solve larger systems, it becomes common to encounter a situation where all input data of the problem cannot

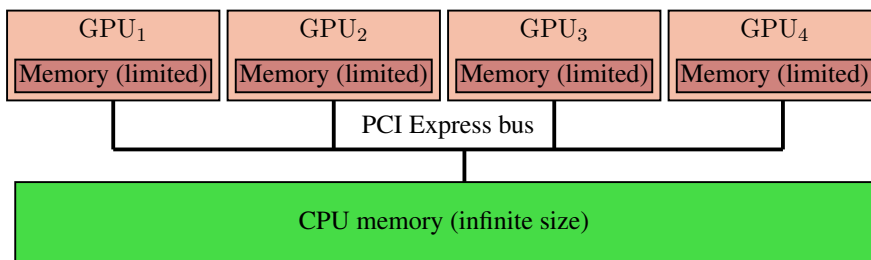


Figure 1.4: Platform topology of multiple GPUs with distributed memory.

fit into the memory of the computing units. In such a case, data must be loaded from the disk (which we consider to be infinite in size). The bandwidth between the disk and the CPU memory is only of a few hundred MB/s, which has a strong impact on performance if data movements are not carefully handled to overlap them with computation. Similarly, one should avoid loading several times the same data but favor data reuse, by improving temporal data locality, i.e., by performing all computations on the same data before its eviction from the memory. Recently, the trend has been to leverage GPUs in addition to CPUs, to achieve unforeseen computation speed and power efficiency.

GPUs are fast because they are massively parallel but also because their local memory (or VRAM) is located on the GPU chip itself and is specifically designed to provide fast access to data for the GPU cores. However, the memory embedded in GPUs is relatively limited, and the bus that connects them to the main memory has a limited bandwidth of a few dozens thousands MB/s. GPUs can achieve multiple teraflops of performance, but when many GPUs are connected to the main memory, the bandwidth of a few dozens thousand megabytes per second becomes the main performance bottleneck. In effect, the GPUs are idle while waiting for the data they need to be loaded into their VRAM. A node architecture with GPUs is described in Figure 1.4. Contrary to common usage, we do not use the term *distributed memory* to describe multiple nodes connected in a network. Instead, in this thesis, *distributed memory* describes the setup of a single node with multiple GPUs, each equipped with its own local memory and connected to a CPU via a PCI bus. In such a distributed memory context, data reuse is even more crucial as one has also to focus on spatial data locality, that is to aim at gathering all computations using the same data on the same GPU. To make matters worse, recent technology trends show a widening gap between peak compute speed and communication bandwidth as well as decrease in memory per gigaflop [42, 45]. This growing disparity result in underutilization of computational resources and longer execution times for data-intensive applications.

1.1.5 Problem statement

In summary, a lot of important applications can be resolved through the resolution of linear algebra systems. However, they are often very computational- and data-intensive. Consequently, they are processed on supercomputers. Even in modern supercomputers, the full application dataset cannot be loaded in the memory of its processing units. This results in data transfers between main memory and processing unit local memory. Because of a gap between communications and computations speed, data transfers become the bottleneck for performance. Improving the hardware is not a permanent solution, so in order to free up core hours on a supercomputer, the following problem must be solved:

Problem 1 (MIN-EXEC-TIME-LIMITED-MEM). *For a given linear algebra application \mathbb{A} with a data set of size $M(\mathbb{A})$ and a set of processing units \mathbb{P} with a cumulated memory of size $M(\mathbb{P})$, how to minimize the execution time of \mathbb{A} on \mathbb{P} even if $M(\mathbb{A}) > M(\mathbb{P})$?*

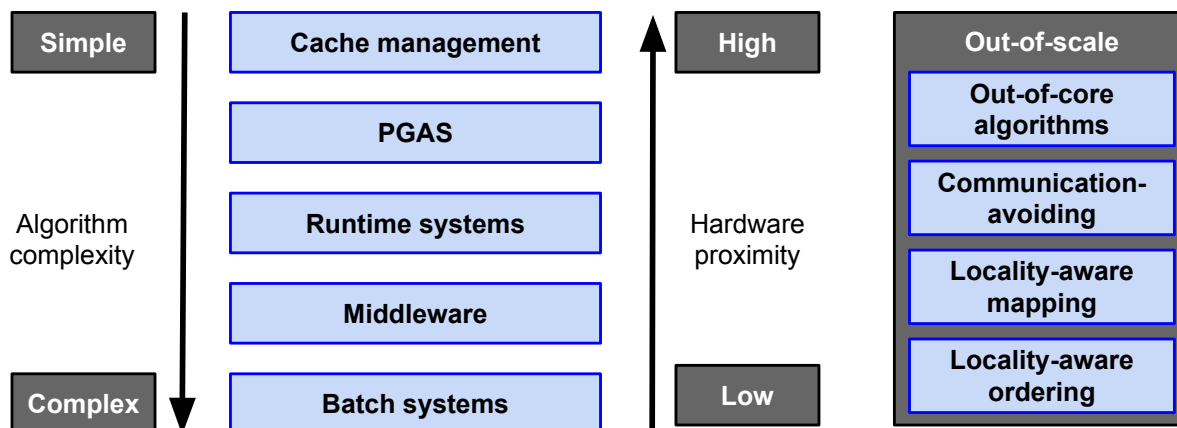


Figure 1.5: The hierarchy of solutions for the MIN-EXEC-TIME-LIMITED-MEM problem.

1.2 Related works

To address this problem of fitting a working set in a given amount of memory [52], researchers explored various strategies, such as advanced caching mechanisms, partitioned global address spaces, task graph scheduling and batch scheduling. All these solutions appear at different levels of the hardware hierarchy, and we present them in the current section. As illustrated in Figure 1.5, a strategy close to the hardware has very little time to make a decision, forcing it to use simple algorithmic solutions, while solutions far from the hardware can have more complex schemes.

In Figure 1.5, we refer to some topics as "out-of-scale". Those are generally offline methods that decide how to process the application in advance and operate without knowledge of future events. For this reason, they can be applied at any hardware level and do not rely on their position on the hardware hierarchy. This includes out-of-core and communication-avoiding algorithms that rely on modifying the application implementation to reduce data transfers. Other examples involve mapping and ordering strategies that carefully assign subsets of an application to workers to improve spatial or temporal locality.

1.2.1 Cache management

Cache management solutions are closest to the hardware as they are directly controlled at compilation time or even within hardware, and have effects on CPU cores, the smallest computational unit. However, such strategies cannot be too complex, since the time available to make a decision is short and the CPU caches are tiny. Cache management can be easily translated to our problem, as it requires *data evictions*: when a cache is saturated, a choice must be made about which data to remove to make room for another. The most common strategy for choosing which data to evict is to rely on the Least Recently Used (LRU) eviction policy. LRU evicts the data that has not been accessed for the longest period of time, as it is considered to be the least likely one to be used in the near future. A more sophisticated strategy at the cache level is to use past accesses of a data¹ to predict which data is more likely to be used again in a given time frame [89]. With this information, a decision can be made to evict the data that has the lowest probability of being used again.

¹Throughout this thesis, for the sake of simplicity, when we refer to "a data," we mean "a piece of data," i.e., an input used for a computation or task.

Some studies in cache management focus on optimizing access to *page caches* (i.e. within the operating system) by answering the following question: when a page cache of limited size is used to store pages requested by an application, how long a page should stay in the cache, and when needed, which page should be evicted from the cache to make room for a new one? Such management is another case of data evictions. In most cases, the sequence of future requests is not available, which makes it very difficult to choose a data to evict. One solution is to try to predict the future in the short term. The operating system can get help by providing system calls such as `madvise`. `Madvise` lets application tell in the short term the future accesses of memory zones, which can be used to evict a data that will not be used again in the near future. Some studies show that knowing just a fraction of the sequence of future requests allows to improve the competitive ratio [13]. There have also been some studies on how to reduce the page loads when one is allowed to reorder some of the next requests [12, 59]. However, in our scope, we are able to reorder a much larger set of requests (tasks), whereas cache studies usually consider that only a very limited number of future cache requests are known in advance, and may possibly be reordered. If the full sequence of page requests is known in advance, Belady's rule minimizes the number of page loads [26]. If a scheduler plans the entire task set in advance, such a solution can be applied to optimally minimize data transfers. Therefore, we introduce such a strategy in Section 2.2.3 and apply it to a scheduler in Section 3.3.6.

A major difference is that operating systems are generally oblivious to future thread accesses. We want to express our applications as a graph of tasks, which allows us to know, through task ordering, which thread is accessing which set of data. Another difference between our work and existing cache management studies is that each task typically does not request a single piece of data (or page in the case of cache management), but multiple pieces of data (e.g. different matrix tiles). This means that one data load or eviction affects only a portion of what a task requires, adding another layer of complexity. Indeed, the decision of evicting data from memory becomes challenging when determining whether to evict data from a task with almost all its data already loaded, or from a task with only one data already loaded.

1.2.2 Partitioned global address space

A Partitioned Global Address Space (PGAS) is a programming model that provides a shared memory abstraction across a distributed memory system. This is used by some programming models like X10 [39] or the HPX [82] runtime to make locality explicit to the user: one can control which objects are located close to each other. This communication model significantly improves the asynchronism and the overlap of communications and computations. The Chapel programming language [33] has recently been enhanced with locality-based optimizations [88] at the compiler level. These PGAS however do not allow for a refined schedule that would for instance manage evictions. We aim to be able to precisely control which data is removed from or added to memory as it is critical under severe memory constraints.

1.2.3 Solutions in runtime systems

To tackle the concern of using GPUs in addition to CPUs and deal with complex data management among heterogeneous processing units, it has become very common to use the task-based programming paradigm, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system such as OmpSs [32], PaRSEC [31], or STARPU [18] manage the execution of the task graph over such distributed and heterogeneous platforms. The burden is thus offloaded from the application programmer to the runtime system, in the form of a task scheduling problem.

Some runtimes have been striving to improve data locality for better performance. For example, runtime systems such as OmpSs and XKaapi [62] rely on work-stealing for load balancing. XKaapi also gives some guarantee on data locality [3] by providing a lower bound on the number of data accesses required by its scheduler. They also show that their locality-guided work-stealing policy performs significantly better than standard work-stealing. We will use a similar work-stealing policy to evaluate our own strategies with the LWS scheduler introduced in Section 5.1.1.

The STARPU runtime system automatically calibrates performance models to predict task execution times. Based on these predictions, the DMDA scheduler (presented in Section 3.1.2), based on the HEFT scheduler [123] strategy, schedules tasks on the resource on which they are expected to complete at the earliest, which also takes data transfers into account. These predictions, however, only rely on the current state of the memory, and do not take into account its limited size: when some new data are loaded in memory, other data may be evicted, which is not taken into account and may lead to incorrect predictions.

With Legion [22, 23], the user explicitly specifies locality thanks to data regions. Then, Legion provides a data mapping strategy to ensure that data is not moved around when it is not necessary. Hence, it is the responsibility of the programmer to describe data dependencies to improve locality. We wish to build something more modular that can be applied to any application without additional requirements.

PaRSEC, previously known as DAGuE [30], uses a different DAG representation, compared to STARPU and other runtimes: it uses parameterized task graphs [44]. The advantage of such a model is the concise representation of the DAG: each task is an algebraic representation that indicates which type of task is to be executed after a task is finished. Thus, the memory required for the DAG is only relative to the number of different task types. However, PaRSEC strategies do not specifically address the problem of scheduling under memory constraints. For the specific case of computing matrix products on multiple GPUs, the PaRSEC runtime pays attention to the memory limitation and implements a control-flow in order to avoid critically overflowing the GPUs memory [75]. We compare our proposed scheduler to a PaRSEC scheduler in Chapter 5, but also to the control-flow strategy specific to the matrix products.

The Python-based Parla runtime [92] provides special data wrappers (Parla Arrays) which allows flexible memory management. Data locality is then considered when scheduling computations through a cost function that mixes the time required for moving data and the load of each node. This is very similar to the DMDA scheduler mentioned for STARPU. Additionally, Parla does not specifically manage limited memory.

Data distribution can also be managed with modern high-level libraries. PHAST [106] or SYCL coupled with the Celerity API [121] can automate parallelization while leaving room to the programmer to direct data distribution on nodes. However, it must be manually programmed and does not specifically address memory limitation.

In summary, no existing runtime system is able to automatically deal with both data locality and limited memory.

1.2.4 An out-of-core middleware

A generic middleware aimed at computing applications on multiple nodes named DOoc [111] aims at reducing data transfers. It has a global scheduler that assigns a task to the node where most of the input data is already located. Then, a greedy local algorithm orders tasks based on the amount of additional input data that must be brought into local memory to make each task ready for execution. The local

algorithm is very similar to the DMDAR scheduler which we compare ourselves to in Chapters 3 and 4. However, such global scheduler is not relevant in our case as we only work with a single node at a time.

1.2.5 Scheduling for distributed platforms

As we move away from hardware proximity, more time is available to schedule applications, as shown in Figure 1.5. Scheduling on large distributed platforms is typically handled by workload managers such as SLURM [127]. However, some strategies aim to replace such managers with more data-aware strategies.

In particular Giersch et al. [69] studied how to allocate and schedule tasks sharing input files on a distributed platform, when the communication between the server holding the input files and the workers is limited. Senger et al. [113] proposed a hierarchical strategy for data distribution in order to improve scalability. Kaya et al. refined the problem by considering that input files are initially distributed on the platform, but may also be transferred through the network if required. They proposed heuristics using hypergraph partitioning [86] and a three-phase approach using initial task placement, refinement, and task ordering [87]. There is additional literature available on the topic of improving data management in batch scheduling. This is presented in Chapter 6, a chapter devoted to this topic.

1.2.6 Out-of-core and communication-avoiding algorithms

Since the seminal work of Hong & Kung [77], which models the complexity of data transfers and prove communication lower bounds for some linear algebra applications, many studies have focused on improving the performance of scientific applications by considering the memory as the primary source of contention. By considering the memory as a limited resource, it is possible to improve locality by rewriting the application or adjusting the data processing logic to increase parallelism by using different data chunk sizes. This is done typically by out-of-core algorithms and communication-avoiding techniques.

Out-of-core algorithms address scenarios where the data cannot fit entirely within the available memory of a processing unit. It allows the input data to stay on slow but large storage (typically disk) and to be loaded into the memory of the processing units (typically CPUs, as presented in Figure 1.3) whenever needed for the computation. With GPUs, out-of-core computing consists in keeping the whole input data in the (larger) memory of the CPU, and loading data in the memory of a GPU only when it is needed for computation (as depicted in Figure 1.4). Toledo [122] surveys such out-of-core for dense linear algebra operations. Direct sparse solvers are known to produce large amount of temporary data, which makes out-of-core computing the only solution to factorize very large sparse matrices [11]. For example, it is necessary to focus on reducing data transfers for task trees arising in sparse direct solvers [97].

In the topic of sparse linear algebra, Demmel et al. [51] propose to use *communication avoiding* algorithms to reduce the amount of data transfers both between processing elements and from/to the main memory. Communication-avoiding algorithms have been studied for a wide range of linear algebra applications. For parallel and sequential dense Strassen's matrix multiplication [21] or QR [49] and LU [71] factorizations, algorithms have been studied and proved to be optimal in the amount of communication they entail. Some communication-optimal algorithms are even implemented on very limited machines, such as laptops [50], to evaluate them in situations where the RAM cannot hold the entire matrix and the bandwidth connecting the disk is slow. This is relevant to our scope, as we also want to test our proposed strategies under major hardware constraints, such as small virtual memory and/or slow communication with the disk.

Kwasniewski et al. [90] work on a parallel matrix multiplication algorithm that is nearly communication-optimal based on a red-blue pebble game to define data shares. However, it is specific

to the matrix multiplication application. Beaumont et al. [24] provide a communication-optimal algorithm specific to the Cholesky factorization in sequential out-of-core systems.

While the cited solutions propose optimizations for linear algebra applications, they often involve the need to partially or completely rewrite the code of the targeted applications. In contrast, our goal is to reduce data transfers and improve performance in a non-communication-optimal way, but without modifying the target applications. This way our algorithms can be more easily integrated into different supports of linear algebra applications. To avoid modifying the applications, we are more interested in applying locality-aware techniques to the ordering and mapping problem, that we review in the next paragraphs.

1.2.7 Locality-aware mapping

The limited memory of GPUs have motivated many studies on how to efficiently access data stored outside the GPU memory. These studies focus on the case of computing on multiple GPUs, and hence concentrate on the mapping problem (which data to put on which GPU). This has been done for several specific applications such as stencil computations [81], sorting problems [126], large scale graph processing [114] or graphic computations [93]. The solution generally consists in building data blocks that each fit within the GPU memory.

Other approaches aim to solve the mapping problem by relying on graph theory and to model the problem either as a matching problem in a bipartite graph composed of workers and tasks [63] or as a partitioning problem in a graph where edges represent the amount of shared data between two tasks [124].

As these solutions concentrate on the mapping, they generally ignore how to order tasks within one GPU. We aim to solve both problems simultaneously, as they cannot be separated if the goal is to minimize data transfers under memory constraints. If one aims only at ordering tasks, one faces the problem of replicating the load of similar data on multiple GPUs, which could be avoided with more sophisticated task partitioning. If the goal is to partition only, the reuse of data on each processing unit will not be optimal and will generate unnecessary data loads.

1.2.8 Locality-aware mapping and ordering

The work of Yoo et al. [128] is the closest to our problem, as they optimize the scheduling of independent tasks sharing input data and tackle both a mapping and ordering problem for multi-core CPUs. For the mapping problem, it make use of the METIS [84] graph partitioner to group tasks in different groups. For the ordering problem, they apply Prim's algorithm [43] to build a maximum spanning tree of tasks and then order the vertices (tasks) according to their order of inclusion in the spanning tree. We will draw inspiration from their work in the next chapters to build strategies that will be used to evaluate our proposed schedulers.

1.3 Positioning in the hardware hierarchy

To address the issue of managing applications with datasets larger than the memory of the workers, we place ourselves on two different hardware hierarchy layers of Figure 1.5; runtimes and batch systems. We explain here how the runtime position in the hardware hierarchy is advantageous:

Proximity to computation units: Runtimes can decide on which computational unit to place each individual task. Such flexibility is important for heterogeneous platforms: some tasks have a larger acceleration factor on GPUs than others. In addition, data movement can be minimized by carefully placing together tasks using common input data.

Architecture awareness: Runtimes are aware of the characteristics of the architecture, including the different memory levels and their access latencies. With this knowledge, precise decisions can be made during execution to place data in the most appropriate processing unit memory and take into account data transfer time.

Dynamic load balancing: The runtime can analyze workload distribution across different processing units and dynamically redistribute tasks and data to achieve a more balanced utilization of resources.

For these reasons, our goal is to propose a *generic task scheduler for runtime systems managing several computing units with shared or distributed limited memory*. We also aim at managing data movements: loads and evictions. More precisely, we want to determine (i) the assignment of the tasks to the processing units to reach a good load balance and spatial data locality and (ii) the order in which tasks must be processed on each processing unit to optimize temporal data locality as well as maximize overlap between computations and data movements. The *generic* keyword means that we want our scheduler to be generic with respect to:

- Memory availability. It must perform both when memory is unconstrained and when it is a scarce resource.
- Number of processing units. It must be efficient with a single or multiple processing units.
- Type of processing unit. We want to tackle any processing unit that has a local memory and can be managed by a runtime. In this thesis we present results on both CPU cores and GPUs.
- Task-based linear algebra applications. Any linear algebra application translated into a set of tasks must be executable by our scheduler.
- Limited knowledge. Our approach does not rely on complete knowledge of the task graph. This means that we do not have information about future tasks that are not yet ready for computation. The ability to work with limited knowledge makes the scheduler applicable to a wider range of applications.

We have also concentrated our efforts on building solutions at the level of batch systems. Batch systems are interesting to study because (i) batch scheduling in HPC systems plays a crucial role in the effective utilization of large supercomputers and (ii) many challenges remain in this area [57]. These include hardware failures that require careful checkpointing of the application, improving user runtime estimates, power- and cooling-aware scheduling, and the management of data-intensive workloads. Targeting batch systems does not specifically solve the MIN-EXEC-TIME-LIMITED-MEM problem, but better scheduling on a cluster indirectly solves faster linear algebra applications. Moreover, it will require locality-aware techniques since we are focusing on data-intensive workloads. Indeed, some scientific domains need to run jobs that are both compute and data intensive. Such jobs use large input files that must be loaded onto a node before they can begin to compute. They are typically submitted as a batch of a few hundred jobs that share the same few input files. Loading the input file, which can be hundreds of gigabytes in size, can take anywhere from a few dozen minutes to 1 or 2 hours for exceptionally large files. Either way, this is a significant cost and greatly increases the average wait time for all other users. Retaining a job's input file and reusing it for a future job eliminates its load time and has the potential to reduce the average latency. Locality-aware strategies maximize such reuse. Our goal here is to *propose new algorithms for I/O-intensive workloads that focus on data reuse to reduce data loads and thus improve both resource utilization and user satisfaction*.

1.4 Bridging the gap between theoretical scheduling and runtime schedulers

Bridging the gap between theoretical scheduling and partitioning algorithms and their implementation in runtime systems is essential if we really want to improve the performance of applications on supercomputers. Theoretical algorithms provide optimal or near-optimal solutions to scheduling problems. However, this is achieved at the cost of limited considerations. Consequently, their implementation in runtime systems faces several challenges. We describe these challenges here and suggest some possible solutions that we believe are of interest.

Queues necessity: Theoretical schedulers and partitioning algorithms are often developed based on simplified assumptions and idealized models. A major simplification concerns the tasks queues used by the runtime system to overlap GPU kernel submission costs. Theoretical schedulers do not consider the queue in which tasks are stored between a scheduling decision and the execution of the task on the processing unit. Because of such queues, there is a delay between the selection of the task to be processed and its actual processing time. Thus, if the decision is made according to the state of memory at time t , at the actual processing time of the task, $t_{real} > t$, the memory will be in a different state because other tasks have been processed in the meantime. To deal with this time discrepancy, we propose to introduce intermediate queues of tasks, which are sets of tasks that are not submitted to the processing units and thus can be freely manipulated. Although it may seem that we are adding queues to a queuing problem, this solution actually works because the intermediate queues are controlled by the scheduler: the scheduler can adapt to the unexpected effects of the main queue of task and consequently change its schedule by adding or removing tasks in the intermediate queue.

Scheduling overhead: The scheduling overhead has two effects on an execution.

Firstly, because it takes time to make a scheduling decision, the state of the execution may have changed between the decision and the actual mapping or ordering of tasks. This can lead to problems such as data being thrown out of memory even though the scheduler planned to use it for the next task computation. The best answer to this is to develop dynamic schedulers that adapt to the state of execution at all times.

Secondly, when evaluating a scheduler on a real computing platform, performance is calculated as the number of flops (floating point operations) divided by the total execution time, which includes the scheduling overhead. Thus, the complexity of the scheduler can negate its benefits. Complexity comes from the number of reads, comparisons, or sorts, but also from the data structures used. This last detail, sometimes overlooked, is often critical in achieving peak performance and must be considered in the design phase.

One great tool available to researchers for the two undesired effects mentioned above is the ability to simulate the execution of an application. A simulation mimics the execution of an application on a computing node that is itself simulated using accurate performance models. Simulations enable one to evaluate the performance of a scheduler without considering the scheduler's overhead. This makes it easier to observe the expected behavior of a theoretical scheduler. Because a simulation can mimic any compute node, it is then possible to compare the simulated results with results on the same, real node. This provides valuable information about, for example, how the scheduling overhead affects the scheduling quality. Another possible solution related to the overhead of a schedule is to degrade the quality of the schedule, for example, by limiting the number of reads or comparisons it can do. It is then important to find a good compromise between the quality of the

schedule and its complexity. This can be easily found using simulation. A simulation provides controlled settings and does not require a reservation on an actual cluster. These two elements make simulation perfect for quickly testing a scheduler under different settings and workload sizes to experimentally find the optimal tradeoff.

Inaccuracy of expected performance: Theoretical models aim at building schedulers that well balance the workload across processing units and overlap communications and computations. To do this, they compute the expected completion time of a task on a given processing unit and the expected transfer time of data from one memory to another and use this information to build static schedulers. However, these predictions are usually inaccurate for two main reasons.

(i) Theoretical models often overlook the fact that two supposedly-identical processing units can still have slightly different computing power. For example, due to uneven wear of the processing units or material defects. Theoretical models can use calibrated performance models to account for real processing unit performance, but there are still some uncertainties. Performance models do not account for unforeseen disturbances that may occur in the system. For example, if a processing unit overheats, the processing time for a task can be extended. Also, contention on a bus used for data transfer can result in reduced bandwidth, causing delays in the completion of data transfers.

(ii) Theoretical models that manage data eviction usually assume that a node memory can be partially or completely flushed in an instant. This is usually not true, which makes it difficult to predict the exact transfer time of a data. After the scheduler informs the runtime that it wants to evict a piece of data, the eviction is not performed immediately. This is due to a delay in the process of evicting data, which involves, for example, checking that the data is not still being used by another worker. There is also inaccuracy in predicting a data transfer time. Indeed, it is almost impossible to predict the exact transfer time of a data before execution, because there are multiple ways to transfer a data: PCI buses, NVLinks, interconnects, etc. Predicting the path taken by a data is not possible with the inaccuracies described in this section. As a result, data are loaded at a different time from what the scheduler expected, and tasks are consequently processed at a different time because they need their data to be loaded before their computation.

These unpredictable factors are not taken into account by theoretical algorithms and result in suboptimal performance. For example, it makes it difficult to balance the load correctly. This is especially critical when the workload is small compared to the number of processing units: putting more work on the fastest unit is usually faster.

The most interesting solution is that schedulers can address these limitations by being dynamic. A dynamic scheduler can compensate for the fact that tasks may take longer or shorter than expected, and it can also take into account that data may be delayed during a transfer. A dynamic scheduler can solve the additional constraint of unforeseen disturbances mentioned in (i) by adjusting the load on a processing unit based on its performance on previously assigned tasks. A dynamic scheduler can also take I/O contention into account by modifying its schedule for nodes connected to contended buses. Regarding the problem mentioned in (ii), a dynamic scheduler can adapt to the current state of memory and decide which data to evict "at the last minute", i.e. just before a data needs to be loaded. Another solution with a dynamic scheduler is to, at runtime, ask for a data eviction either as soon as possible or after the computation of a particular task.

Prefetch system: Another issue of expected performance inaccuracies comes from prefetch systems. Runtime systems are usually equipped with a prefetch system that allows data to be preloaded prior to the computation of the task using that data. Prefetching is the key element that allows communications and computations to overlap. Because of the unexpected disturbances mentioned

above, a theoretical scheduler cannot plan for them in advance. One solution, although not perfect, is to organize the task order in such a way that the data loads are distributed over time. The prefetch is then managed by the runtime, following the data requirements of the tasks. With temporally spread transfers, there will naturally be overlap between communications and computations. Thus, in addition to using a dynamic scheduler that adapts to disturbances, we propose to rely on task ordering and mapping rather than using exact times and placements to maximize overlap.

The schedulers presented in this thesis aim to exploit intuitions from theoretical models and algorithms, while managing the constraints of their implementation in a runtime system, by utilizing some or all of the solutions proposed here.

Chapter 2

Problem Statement and Integration into a Runtime System

Contents

2.1	Simplifying our optimization problems	22
2.1.1	Expressing applications as task graphs	22
2.1.2	Avoiding the conflicting goals of using multiple processing units	23
2.1.3	Considering homogeneous processing time and data size	23
2.1.4	Making the model complex again	24
2.2	Simplified model with an independent task set and a single processing unit	24
2.2.1	Expressing applications as a bipartite graphs	24
2.2.2	Simplified optimization problem	25
2.2.3	Optimal eviction policy proof	26
2.2.4	Complexity of finding an optimal task order	27
2.3	Making the model parallel	28
2.3.1	Adding the partitioning problem to the bipartite graph	28
2.3.2	Optimization problem in parallel	29
2.4	Extension to heterogeneous task and data weights	29
2.5	Adding dependencies to the model	30
2.6	The STARPU Runtime System	31
2.6.1	Task and data	32
2.6.2	Tasks submission	32
2.6.3	Task flow	34
2.6.4	New functionality to add custom eviction policies	34
2.6.5	New logging and visualization tool	37
2.7	Summary	41



IN this chapter we introduce our task sharing data model. This model is used in Chapters 3, 4 and 5. We describe our main objective: building a generic scheduler capable of reducing data transfers and increasing performance by partitioning and scheduling a set of tasks (with and without dependencies) sharing data on one or more processing units with limited distributed or shared memory. Such a complex objective is a combination of different optimization problems that cannot be solved all at once. To succeed, we plan to tackle these optimization problems one at a time.

Therefore, in Section 2.1, we describe how to build a model that simplifies these optimization problems. We then make our model complex again to more accurately describe our main objective. Section 2.2 showcases a simplified version of our model with a simple problem statement, independent tasks and a single processing unit. It allows us to prove that an optimal eviction policy exists, and we use this result to prove the complexity of the problem. Section 2.3 adds multiple processing units to the model. Section 2.4 adds heterogeneity to the data sizes and task processing times. Section 2.5 adds dependencies and presents the model corresponding to our main objective.

The framework used to implement, test in simulation, and test on real high-performance-computing platforms our algorithmic contributions is the STARPU runtime system. Section 2.6 describes the STARPU runtime system and our contribution to its source code.

2.1 Simplifying our optimization problems

Here we describe the three steps we take to simplify our optimization problems.

2.1.1 Expressing applications as task graphs

The applications we are using in this thesis are expressed by the programmer as directed acyclic graphs (DAGs) of tasks, where vertices represent tasks and edges represent data dependencies between tasks. The programmer provides the code for each task as well as the description of its input and output data, allowing the runtime system to assign tasks to processing units and to move data around when needed. Figure 2.1 provides a small example of such a task graph using the Cholesky factorization. Task POTRF_0 is the first task available for computation. Arcs coming out of POTRF_0 indicate the set of tasks that will be made available upon its completion. If multiple arcs point to a task (such as GEMM_3_1_0), all of its predecessors must be completed in order for it to be computed.

Solving the final objective stated earlier from such DAGs would be very ambitious. Applications are often designed to maximize parallelism, as parallel computation greatly increases performance. This involves breaking the work into as many *independent* tasks as possible, which are then spread across the various workers for concurrent execution. Some linear algebra workflows are easily translated into a completely independent set of tasks, such as matrix multiplication. But it can also be the case for applications with dependencies such as the Cholesky and LU factorizations which expose a fair amount of parallelism. As a result, when using a dynamic runtime, the scheduler is exposed at a given time to an already-fairly large subset of tasks which are independent of each other. So, at a given time of the computation, some of the tasks have been completed, and some tasks are not available for computation as their input data has not been computed yet (tasks with pale colors on Figure 2.1). The tasks available for computation form a subset of independent tasks in the graph, called the *ready tasks* (bright tasks in the figure). Several of these available tasks depend on common predecessors (e.g., both GEMM_3_1_0 and GEMM_3_2_0 depend on the result of TRSM_3_0), which means that they share a common input data produced by the common predecessor. Reducing the optimization problem only on the currently available tasks can already lead to a large reduction in data transfers and hence a performance increase.

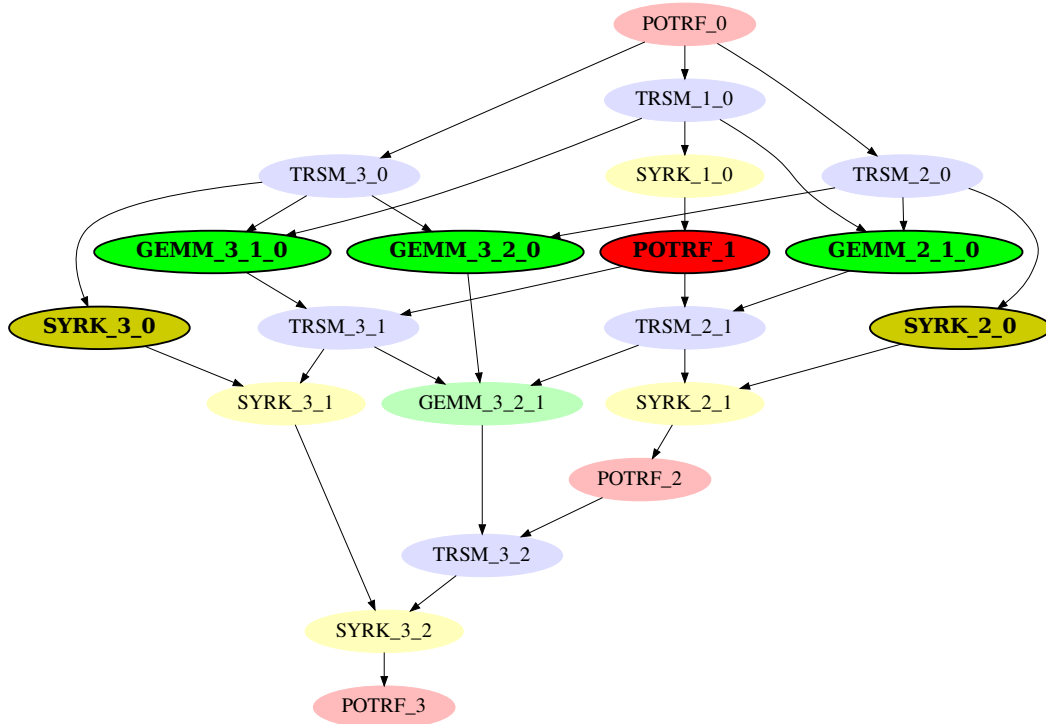


Figure 2.1: Cholesky factorization expressed as a task graph. Tasks with bright colors are independent of each other.

Since scheduling a subset can already lead to better performance, our *first step* to simplify our model is to only consider ready tasks, i.e., tasks with resolved dependencies.

2.1.2 Avoiding the conflicting goals of using multiple processing units

When working with multiple processing units, a task set must be both partitioned and scheduled on each processing unit. Partitioning tasks across multiple processing units raises a *load balancing* issue. Load balancing is the process of evenly distributing tasks across multiple processing units to minimize the amount of time they are idle. Load balancing and minimizing data transfers are two conflicting goals. A good task partitioning scheme will aim to distribute the workload fairly across the multiple processing units, regardless of data mapping and transfer issues. On the contrary, a schedule that is focused on minimizing IOs will process as many tasks as possible on a single processing unit to favor temporal locality. Therefore, a *second step* to simplify our optimization problems is to consider only a single processing unit.

2.1.3 Considering homogeneous processing time and data size

Finally, tasks may have different processing times and each data may vary in size. To properly balance the load, a scheduler must take into account the different processing times, which is more complex than balancing the number of tasks on each processing unit. With different data sizes, minimizing IOs leads to minimizing the sum of data sizes loaded from the RAM. The additional difficulty is to balance data reuse and data sizes: is it beneficial to load a large data multiple times if it is reused a lot? So, as a *third and final step* in simplifying our model, we first consider homogeneous processing time and data size.

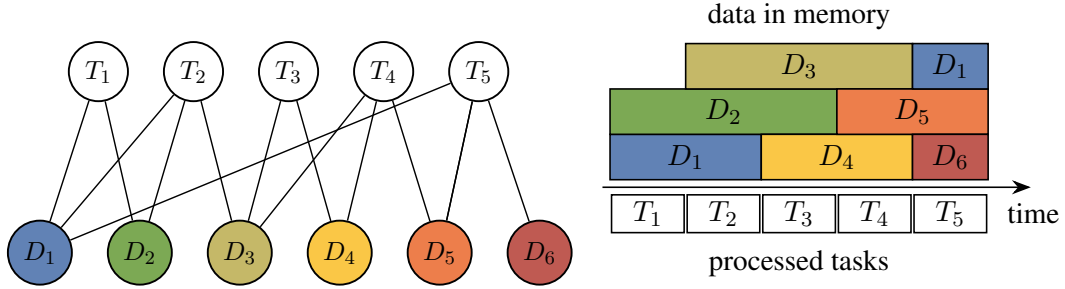


Figure 2.2: Example with 5 tasks and 6 data, with a memory holding at most $M = 3$ data. The graph of input data dependencies is shown on the left. The schedule on the right corresponds to processing the tasks in the natural order with the following eviction policy: $\mathcal{V}(1) = \mathcal{V}(2) = \emptyset$, $\mathcal{V}(3) = \{1\}$, $\mathcal{V}(4) = \{2\}$, $\mathcal{V}(5) = \{3, 4\}$. This results in 7 loads and only D_1 is loaded twice.

2.1.4 Making the model complex again

Now that we have seen how to simplify our model, we will describe it in Section 2.2. We will use it to provide optimality and complexity proofs. We will then gradually restore the complexity of our model.

We take successive steps to get back from our simple model to the more complex final model. Section 2.3 adds multiple processing units. Section 2.4 adds different processing times for the tasks and heterogeneous sizes for the data. Section 2.5 is the final and most complete model that accurately describes our problem.

2.2 Simplified model with an independent task set and a single processing unit

Here we formally define our simplified model, from which we formulate an optimization problem, which we prove to be NP-complete. We also describe an eviction policy and prove its optimality in the context of our model.

2.2.1 Expressing applications as a bipartite graphs

Tasks sharing their input data can be modeled as a bipartite graph $G = (\mathbb{T} \cup \mathbb{D}, E)$ [87]. The vertices of this graph are on one side the tasks $\mathbb{T} = \{T_1, \dots, T_m\}$ and on the other side the data $\mathbb{D} = \{D_1, \dots, D_n\}$. All m tasks must be processed. An edge connects a task T_i and a data D_j if task T_i requires D_j as input data. We denote by $\mathcal{D}(T_i) = \{D_j \text{ s.t. } (T_i, D_j) \in E\}$ the set of input data for task T_i . We assume that all data have the same size and all tasks have the same processing duration. These assumptions simplify the discussion and allow us to make optimality and complexity proofs more easily. Section 2.4 presents the more complex model with heterogeneous sizes.

The processing unit (PU) is equipped with a memory of limited size, which may contain at most M data simultaneously. PU is used to refer to both GPU and CPU. It allows us to make no distinction between the two, as our strategies target both architectures. During the processing of a task T_i , all its inputs $\mathcal{D}(T_i)$ must be in memory.

For simplicity, we do not consider the size of the written output. In the case of linear algebra for instance, the output data is most often smaller than the input data, and therefore can be transferred concurrently and in less time than it takes to transfer the input data. Data output is then not the driving constraint for efficient execution.

Our goal with this simplified model is to *minimize the amount of data movement*. To fulfill this objective, we will determine in *which order* to process each task and *when* each data must be *loaded* and *evicted*. More formally, we denote by σ the order in which tasks are processed, and by $\mathcal{V}(t)$ the set of data to be evicted from the memory before the processing of task $T_{\sigma(t)}$. A schedule is made of m steps, each step being composed of the following three stages (in this order):

1. All data in $\mathcal{V}(t)$ are evicted (unloaded) from the memory;
2. The input data in $\mathcal{D}(T_{\sigma(t)})$ that are not yet in memory are loaded;
3. Task $T_{\sigma(t)}$ is processed.

An example is shown in Figure 2.2. This example illustrates that input data are loaded in memory as late as possible: loading them earlier would possibly trigger more data movements. In real computing systems, a prefetch is usually designed to load data a bit earlier so as to avoid waiting for unavailable data. For the sake of simplicity, we do not consider this in our model: if needed, we may simply book part of our memory for the prefetch mechanism.

Using the previous definition, we define the *live data* $L(t)$ as the data in memory during the computation of $T_{\sigma(t)}$, which can be recursively defined:

$$L(t) = \begin{cases} \mathcal{D}(T_{\sigma(1)}) & \text{if } t = 1 \\ L(t) = (L(t-1) \setminus \mathcal{V}(t)) \cup \mathcal{D}(T_{\sigma(t)}) & \text{otherwise} \end{cases}$$

The memory limitation can then be expressed as $|L(t)| \leq M$ for each step $t = 1, \dots, m$. Our objective is to minimize the amount of data movement, i.e., to minimize the number of *load* operations: we consider that data are not modified so no *store* operation occurs when evicting a data from the memory. Assuming that no input data used at step t is evicted right before its processing ($\mathcal{V}(t) \cap \mathcal{D}(T_{\sigma(t)}) = \emptyset$), the number of loads can be computed as follows:

$$\#Loads(\sigma, \mathcal{V}) = \sum_t |\mathcal{D}(T_{\sigma(t)}) \setminus L(t)|$$

There is no reason for a scheduling policy to evict some data from memory if there is still room for new input data. We call *thrifty scheduler* such a strategy, formalized by the following constraints: if $\mathcal{V}(t) \neq \emptyset$, then $|L(t)| = M$. For this class of schedulers, the number of loads can be computed more easily: as soon as the memory is full, the number of loads is equal to the number of evictions. That is, for the regular case when not all data fit in memory ($n > M$), we have:

$$\#Loads(\sigma, \mathcal{V}) = M + \sum_t |\mathcal{V}(t)|$$

All the strategies presented in this thesis are *thrifty schedulers*.

2.2.2 Simplified optimization problem

Our simplified optimization problem is stated below:

Problem 2 (MIN-LOADS-FOR-TASKS-SHARING-DATA). *For a given set of tasks \mathbb{T} sharing data in \mathbb{D} according to \mathcal{D} , what is the task order σ and the eviction policy \mathcal{V} that minimizes the number of loads $\#Loads$?*

A solution to this optimization problem consists in two parts: the order σ of the tasks and the eviction policy \mathcal{V} . Note that when each task requests a single data, finding an efficient eviction policy corresponds to the classical cache management policy problem.

2.2.3 Optimal eviction policy proof

In order to reduce our objective to just finding the task order, we need to find an optimal eviction policy. When the full sequence of data requests is known, the optimal policy consists in evicting the data whose next use is the furthest in the future. This is the well-known Belady MIN replacement policy [26] (see proof in [98]). We prove in the following theorem that this rule can be extended to our problem, with tasks requiring multiple data.

Theorem 1. *We consider a task schedule σ for a MIN-LOADS-FOR-TASKS-SHARING-DATA problem. We denote by MIN the thrifty eviction policy that always evicts a data whose next use in σ is the latest (breaking ties arbitrarily). MIN reaches an optimal performance, i.e., for any eviction policy \mathcal{V} ,*

$$\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V}).$$

Proof. We consider a given task order σ . We transform our problem so that each task depends on a single data. We replace each task T_i depending on data $\mathcal{D}(T_i) = \{D_1, \dots, D_k\}$ by a series of $2k$ tasks: $T_i^{(1)}, T_i^{(2k)}$ such that $\mathcal{D}(T_i^{(j)}) = \mathcal{D}(T_i^{(j+k)}) = D_j$ for $j = 1, \dots, k$. This transformation is performed both in the task set \mathbb{T} (leading to \mathbb{T}') and in the task order σ (leading to σ').

Let \mathcal{V} be an optimal eviction policy for the original problem, i.e. for task set \mathbb{T} and task order σ . We now transform it into an eviction policy for \mathbb{T}' and σ' with the same number of loads and evictions. We group tasks by subsets of $2k$ tasks (as they were created above) and evict all data in $\mathcal{V}(t)$ before processing tasks $T_{\sigma(t)}^{(1)}, T_{\sigma(t)}^{(2k)}$ (and loading their missing inputs). We denote this strategy by \mathcal{V}' . Clearly, this is a valid strategy (we never exceed the memory if \mathcal{V} did not on the original problem) and it has the same number of loads as \mathcal{V} :

$$\#Loads(\sigma, \mathcal{V}) = \#Loads(\sigma', \mathcal{V}').$$

Symmetrically, we consider an optimal eviction policy for the transformed problem (\mathbb{T}' and σ') obtained with Belady's MIN replacement policy, denoted by MIN' : whenever some data must be evicted, it selects the one whose next use is the furthest in the future. We now prove that it can be transformed into an eviction policy MIN for the original problem with the same performance (loads and evictions), and that MIN also follows Belady's rule. We consider the subset of $2k$ tasks $T_i^{(1)}, T_i^{(2k)}$ coming from the expansion of task T_i scheduled at time t ($\sigma(t) = T_i$) and the set V_i of all data evicted by MIN' before some task $T_i^{(j)}$. By property of MIN' and as the memory is large enough for the inputs of task T_i ($M \geq k$), no input data of some $T_i^{(j)}$ belongs to V_i : during the first k tasks, their next occurrence is among the closest next tasks. Thus, there is no eviction during the processing of the last k of the $T_i^{(j)}$ tasks. Thus, we can adapt MIN' for the original problem by setting $MIN(t) = V_i$. It is easy to verify that MIN reaches the same performance as MIN' :

$$\#Loads(\sigma, MIN) = \#Loads(\sigma', MIN')$$

and that the data evicted at time t are (among the) ones whose next use is the furthest in the future.

As MIN' is known to be optimal for the transformed problem, we have $\#Loads(\sigma', MIN') \leq \#Loads(\sigma', \mathcal{V}')$ and we conclude that $\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V})$, which proves that MIN is optimal on the original problem. \square

For cache management, Belady's rule has little practical impact, as the stream of future requests is generally unknown; simple online policies such as LRU (Least Recently Used [52]) are generally used. However in our simplified model, the full set of tasks is available at the beginning. Hence, we can take advantage of this optimal offline eviction policy, as demonstrated in Chapter 3. Even with dynamic

scheduling, some tasks are scheduled for execution in advance, so the set of data to be used in the future is partially known to the scheduler. With this information, we want to adapt the intuition of Belady's rules to our dynamic scheduler in Chapter 4.

2.2.4 Complexity of finding an optimal task order

Thanks to the previous result, we can restrict our problem to finding the optimal task order σ . This problem is NP-complete.

Theorem 2. *Given a set of tasks \mathbb{T} sharing data in \mathbb{D} according to \mathcal{D} and an integer B , finding a task order σ such that $\#Loads(\sigma, MIN) \leq B$ is NP-complete.*

Proof. We first check that the problem is in NP. Given a schedule σ (and an eviction policy \mathcal{V} , which might be computed by MIN), it is easy to check in polynomial time that:

- The schedule is valid, that is, no more than M data are loaded in memory at any time step;
- The number of loads is not greater than the prescribed bound B .

The NP-completeness proof consists in a reduction from the cutwidth minimization problem (or CMP), proven NP-complete by Gavril in 1977 [66]. We denote by I_{CMP} an instance of CMP composed of a graph G , and by Δ the maximum degree of vertices in G . The question is to decide whether there exists a linear arrangement of the vertices such that the cutwidth is at most K . A linear arrangement α is an ordering of the vertices. The cutwidth $CUT_\alpha(v)$ of a vertex v under the linear arrangement α is the number of edges that connect vertices ordered before and after v in α , that is, the number of edges $(u, w) \in E$, such as $\alpha(u) < \alpha(v) < \alpha(w)$. The total cutwidth of G is the maximal cutwidth over all vertices : $CUT_\alpha(G) = \max_{v \in V} CUT_\alpha(v)$.

Given an instance I_{CMP} of CMP , we create an instance $I_{MinLoads}$ of our problem as follows. For each vertex $v_i \in I_{CMP}$, we create a task T_i , and for each edge $e_k = (v_i, v_j)$, we create a data D_k such that D_k is a shared input of T_i and T_j . In addition, each task T_i with degree δ_i has $\Delta - \delta_i$ specific input data, denoted by D_i^j for $j = 1, \dots, \Delta - \delta_i$. Then,

$$\mathcal{D}(T_i) = \{D_k, \text{ s.t. } e_k \text{ is adjacent to } v_i\} \cup \{D_i^j \text{ for } j = 1, \dots, \Delta - \delta_i\}.$$

Note that each task has exactly Δ input data. Finally, we set $M = K + \Delta$ and $B = |\mathbb{D}|$: we are looking for a solution where each data is loaded exactly once.

We now prove that if I_{CMP} has a solution, then $I_{MinLoads}$ has a solution. Let α be the linear arrangement solution of I_{CMP} . We consider the task order $\sigma = \alpha^{-1}$ (i.e., $\sigma(t) = i$ if $\alpha(i) = t$), which schedules tasks in the same order as in the linear arrangement. We also consider the optimal eviction policy MIN . We prove that:

- (i) A data is evicted only if it is not used anymore;
- (ii) Each data is loaded exactly once.

Note that (ii) is a direct consequence of (i). We consider a step t when some data D_j is evicted and some task T_i is processed. We consider the set S of data in memory before starting step t together with the inputs of $T_{\sigma(t)}$ that are loaded in memory during step t . If D_j is evicted, this means that $|S| > M$ (MIN is a thrifty policy). We consider S' , the subset of S containing the data that are used as inputs for a later step $t' > t$. By construction of $I_{MinLoads}$, each data $D_k \in S'$ corresponds to an edge $e_k = (v_a, v_b)$ in G such that $\sigma^{-1}(v_a) = \alpha(v_a) < t$ (the data was loaded for a task T_a scheduled before t) and $\sigma^{-1}(v_b) = \alpha(v_b) > t$ (the data is used for a task T_b scheduled after t). Hence, it corresponds to an edge counted in the cutwidth $CUT_\alpha(v_i)$. Since this cutwidth is bounded by K , there are at most K data in

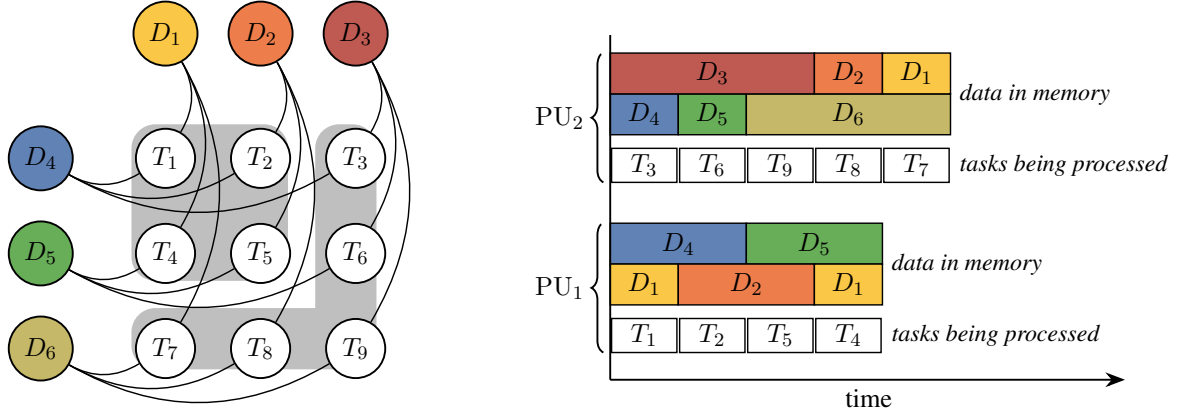


Figure 2.3: Example with 9 tasks with 2D grid dependencies. The graph of input data dependencies is shown on the left, together with the task partition among processing units. A possible schedule is described on the right. Each processing unit can hold 2 data in memory. PU_1 processes tasks T_1, T_2, T_5, T_4 (in this order), and data D_1 has to be loaded twice. PU_2 processes tasks T_3, T_6, T_9, T_8, T_7 in this order to avoid multiple loads of the same data. The total amount of data movement is 11.

S' . Together with the Δ input data of the current tasks, no more than $K + \Delta = M$ in memory. Thus, the evicted data D_j is not used later than t . Since all data are loaded exactly once, the number of loads is not larger than B .

We now prove that if $I_{MinLoads}$ has a solution, then I_{CMP} has a solution. Let σ the task order in the solution of $I_{MinLoads}$. We construct the solution of I_{CMP} such that $\alpha = \sigma^{-1}$. We now prove that its cutwidth is not larger than K . By construction, the cutwidth $CUT_\alpha(v_i)$ at some vertex v_i (corresponding to a task T_i scheduled at time t) is the number of data which are used both before t and after t . Given the constraint on the number of loads, each data is loaded once, so such a data must be in memory during the processing of T_i , in addition to the Δ inputs of T_i , and there are at most $M - \Delta = K$ such data. This proves that $CUT_\alpha(v_i) \leq K$. Hence α is a solution for I_{CMP} . \square

2.3 Making the model parallel

We now consider multiple processing units. We consider the problem of scheduling independent tasks on K processing units, denoted by PU_1, \dots, PU_K . Each of the K processing units is equipped with a memory of limited size M data. Our goal is to determine both *how to partition* the task set to the PUs and in *which order* to process them on each PU.

2.3.1 Adding the partitioning problem to the bipartite graph

We now denote by $\sigma(k, i)$ the i^{th} task processed on PU_k , and by $\mathcal{V}(k, i)$ the set of data to be evicted from the memory of PU_k before the processing of this i^{th} task. We also let nb_k be the number of tasks allocated to PU_k .

We can extend the model presented in Section 2.2.1 for k processing units. With $T_{\sigma(k, i)}$ the i -th task processed on PU_k , we can define the *live data* on PU_k as

$$L(k, i) = \begin{cases} \mathcal{D}(T_{\sigma(k, 1)}) & \text{if } i = 1 \\ \left(L(k, i-1) \setminus \mathcal{V}(k, i) \right) \cup \mathcal{D}(T_{\sigma(k, i)}) & \text{otherwise} \end{cases}$$

and the number of loads on PU_k can be computed as follows:

$$\#Loads_k = \sum_i \left| \mathcal{D}(T_{\sigma(k,i)}) \setminus L(k, i-1) \right|$$

An example of partitioning and ordering using tasks from a bipartite graph is shown in Figure 2.3.

2.3.2 Optimization problem in parallel

Our objective with multiple processing units is both to ensure a good load balancing and to minimize the amount of data movement.

Objective 1: Load Balancing We assume that all tasks have the same processing time on any processing unit. Thus, load-balancing the work on each PU amounts to minimizing the maximum number of tasks on any PU:

$$\text{Obj. 1 : } \textit{minimize} \max_k nb_k$$

Objective 2: Data Movement The second objective is to limit the amount of data movement, that is, to minimize the number of load operations from the main memory to the memory of the PUs:

$$\text{Obj. 2 : } \textit{minimize} \sum_k \#Loads_k$$

In Section 2.2.3, we proved that with a single processing unit and a schedule σ , it is possible to derive an optimal eviction policy \mathcal{V} by following Belady's rule. This rule can be extended to the multi-PU case: once tasks have been partitioned among PUs and ordered for computation, that is, once σ is set, we may compute the optimal eviction scheme for each PU by applying Belady's rule. Hence our objective is only to find a schedule σ of the tasks on the PUs. The decision version of the bi-objective problem is then expressed as follows.

Problem 3 (BI-OBJ-MULTI-PU-TASK-SCHEDULING). *Given a number K of processing units, m tasks sharing their inputs according to a bipartite graph G , and two bounds W and C , is there a schedule σ such that $\max_k nb_k \leq W$ and $\sum_k \#Loads_k \leq C$?*

We proved in Section 2.2.4 that ordering tasks to minimize data movements is NP-complete. Ordering task on a single processing unit is a sub-problem contained in the more parallel ordering problem presented here. This proves the complexity of our bi-objective problem.

2.4 Extension to heterogeneous task and data weights

The previous two models are extended to cope with *heterogeneous weights* where weight refers to both task processing times and data sizes. We outline here the difference from the previous models.

With heterogeneous weights, each task $T_i \in \mathbb{T}$ is associated with a computation time $C(T_i)$. Each data $D_i \in \mathbb{D}$ has a size $M(D_i)$. Instead of the number of loads, we now compute the sum of the data sizes that have been loaded:

$$\textit{Amount_Loads}(\sigma, \mathcal{V}) = \sum_t \left| M \left(\mathcal{D}(T_{\sigma(t)}) \setminus L(t) \right) \right|$$

The optimization problem for an independent task set processed on a single processing unit, MIN-LOADS-FOR-TASKS-SHARING-DATA, is now defined as MIN-LOADS-FOR-TASKS-SHARING-DATA-HETEROGENEOUS:

Problem 4 (MIN-LOADS-FOR-TASKS-SHARING-DATA-HETEROGENEOUS). For a given set of tasks \mathbb{T} sharing data in \mathbb{D} according to \mathcal{D} , what is the task order σ and the eviction policy \mathcal{V} that minimizes the sum of data loads sizes $Amount_Loads$?

The NP-completeness result naturally extends to this variant of the problem. However, the optimality of the *MIN* eviction policy does not hold anymore. Consider the following example. Data D_1 has size $M(D_1) = 2$ and will be used next at time t_{max} . Data D_2 has size $M(D_2) = 1$ and will be used at time $t_{max} - 1$. A space of size 1 must be freed from memory. According to *MIN*, the data to be evicted is D_1 . However, in this case, evicting D_1 will cause a data load of size 2 at time t_{max} , while evicting D_2 will only cause a data load of size 1. Nevertheless, by following the *MIN* principle, it is possible to construct promising (even if not optimal) eviction policies, as we will see in Section 4.3.4.

For the parallel model the following modifications are made. Again we compute the sum of data sizes that need to be transferred on PU_k as follows:

$$Amount_Loads_k = \sum_i \left| M \left(\mathcal{D}(T_{\sigma(k,i)}) \setminus L(k, i-1) \right) \right|$$

The two objectives associated are now described as:

Objective 1: Load Balancing Since tasks can have different processing times, load-balancing here consists of minimizing the sum of the processing times of tasks in (PU_k):

$$\text{Obj. 1 : } \textit{minimize} \max_k \sum_i^{nb_k} C(T_i)$$

Objective 2: Data Movement The second objective is to minimize the amount of data that are loaded from the main memory to the memory of the PUs:

$$\text{Obj. 2 : } \textit{minimize} \sum_k Amount_Loads_k$$

The previous BI-OBJ-MULTI-PU-TASK-SCHEDULING is translated to BI-OBJ-MULTI-PU-TASK-SCHEDULING-HETEROGENEOUS.

Problem 5 (BI-OBJ-MULTI-PU-TASK-SCHEDULING-HETEROGENEOUS). Given K processing units, m tasks sharing their inputs according to a bipartite graph G , and two bounds W and C , is there a schedule σ such that $\max_k \sum_i^{nb_k} C(T_i) \leq W$ and $\sum_k Amount_Loads_k \leq C$?

2.5 Adding dependencies to the model

Adding dependencies to our model is the final step in achieving our ultimate goal: building a generic scheduler that can be used with any task-based application. Solving the problem with dependencies does not add any complexity to the model because in practice, the scheduler only has the knowledge of tasks ready for computation, i.e, tasks that have satisfied all their dependencies. Thus, we can aim to solve the BI-OBJ-MULTI-PU-TASK-SCHEDULING-HETEROGENEOUS problem with dependent task sets.

However, dependencies introduce a new problem when building an efficient scheduler: the critical path. In a task graph, the critical path is the longest path from the starting task to the last one, taking into account task dependencies. In other words, it identifies the tasks that are critical to the overall

completion of the task graph. To help the scheduler, each task is assigned a priority that represents the distance to completion of the DAGs. A greedy approach focused on priorities would compute the highest priority tasks first, hoping to progress quickly enough along the critical path to avoid lacking parallelism. A more refined scheduler would be responsible for striking a balance between advancing along the critical path by computing high-priority tasks, and favoring data locality to improve execution efficiency, which can be two conflicting goals.

Dependencies also make the application dynamic. This means that \mathbb{T} and \mathbb{D} are not known to the scheduler in advance, but are updated as tasks are completed and release other tasks. This means two things for the scheduler. First, it must schedule an incomplete task set, which means that some optimal data-reuse patterns may be missed. Second, it must be able to run a schedule, start processing a task, and then schedule the new ready tasks that have become available, which can be hard to incorporate into the initial schedule. To solve this second point, one can either reschedule everything after each new ready task is released, or reschedule only after some time/number of tasks have been released. The first approach has the advantage of keeping the schedule up to date but with a large scheduling overhead, while the second approach minimizes the overhead but does not have a complete view of what is available. The same issue applies to eviction policies. The policies need to be aware of \mathbb{D} to decide if a data should be removed or if it will be used by a large number of subsequent tasks.

2.6 The STARPU Runtime System

We present here the STARPU runtime system that we used to integrate and evaluate our proposed schedulers. STARPU is a flexible, general-purpose runtime system that makes programmers' jobs easier. With STARPU, programmers do not have to worry about data transfer issues: STARPU automatically transfers data onto the processing unit when they are required and triggers data preloading to overlap communications with computations. In addition, STARPU makes it easy to incorporate new scheduling policies in a modular fashion. That is, it is possible to reuse existing components, such as resource-mapping components that can make scheduling decisions over a given subset of tasks, or worker components that handle the technical aspect of computing a task. This reduces the programmer's burden to a simple task ordering problem.

Computational libraries can be implemented on top of STARPU, giving it the ability to run on a wide variety of applications. For example, the Chameleon [8] dense linear algebra library uses such a feature. With Chameleon, STARPU has proven to be able to achieve very high performance for typical dense linear algebra applications such as Cholesky [6] and QR [5]. With PaStiX, another library ported on top of STARPU, it achieved high throughput for the sparse linear algebra case [91]. On the industrial side, STARPU supports the factorization of hierarchically-compressed dense matrices in out-of-core settings [61], which is now used by actors such as Airbus and the ArianeGroup.

STARPU supports simulation through SimGrid [35]. SimGrid is a framework for simulating the execution of an application on a distributed computing platform. Simulations are highly valuable when designing scheduling policies. They allow for a quick evaluation of performance without the need for a reservation on a highly demanded computing platform. They are flexible; many different use cases can be experimented with, such as changing memory limits, task durations, etc. Simulations are scalable, allowing experiments to be run on a very large number of nodes without slowing down a cluster for other users. Simulation also helps avoid unnecessary energy consumption, since the schedulers developed can be tested on real machines only at an advanced stage. Moreover, simulated results are highly reliable [36]. STARPU provides a tool for collecting performance models of real computing nodes. STARPU produces performance models with automatic calibration [17]: it measures the performance

of tasks during an actual execution. It also builds history-based performance models by storing the performance of tasks from previous executions. Such performance models are then used with SimgGrid to simulate the behavior of any node under different settings. Finally, with simulation, experimental results can be faithfully reproduced because two simulated executions with the same settings and performance models are strictly identical. It is therefore a great advantage for STARPU to be able to work with simulation.

Furthermore, in STARPU, programmers can rely on performance analysis tools such as traces using the FxT library ¹ or Gantt charts using ViTE ². In the following sections, we will give an overview of how STARPU works and describe in detail how a scheduler is integrated into the runtime.

2.6.1 Task and data

STARPU uses a task-based model, where each task is a function call with inputs and outputs. Tasks can have dependencies between each other, which are then used to construct a DAG. The built-in schedulers can then use this information to optimize the execution of such an application on heterogeneous platforms. A task is defined as a combination of three factors.

Kernels: STARPU supports hybrid architectures, tasks can thus be executed through a CPU or GPU kernel.

Handles: Data are abstracted as handles in the system, in order to efficiently manipulate them and keep information on their location and status. This abstraction notably allows data to be replicated to multiple different memories.

Access modes: They denote how the piece of data referred to by the handle should be accessed. It can be either through reading, writing, or both.

In this thesis, when we refer to a *data*, we refer to the handle whose access mode can be *read* or *read-write*, since this is the data that needs to be loaded and that can be shared among several tasks.

2.6.2 Tasks submission

Applications are typically transformed into a task graph in runtime systems. The task graph intuitively captures the idea that certain tasks must be completed before others can begin, forming a directed acyclic graph (DAG). Each node in the graph represents a task that is part of a larger process. The edges between the nodes signify which tasks depend on the completion of other tasks. STARPU constructs a DAG through task submissions. We use as an example Figure 2.4, which presents a series of task insertions corresponding to the Cholesky factorization. Figure 2.4 shows the four different components of the Cholesky factorization: POTRF, TRSM, SYRK and GEMM tasks.

Each call to `starpu_task_insert` is a task submission. It takes three types of parameters. First, the codelet. STARPU, supports hybrid architectures that require the implementation of different versions of a task. For example, a task may have both a CPU and a CUDA kernel implementation. The codelet is a data structure that contains all of these implementations. Second, the priority, corresponding to the value following `STARPU_PRIORITY`. This is an optional field. The priority is set to 0 by default. Third, the various data the task is accessing, in this example it is the tiles of A , with their access modes.

From these insertions we can understand how STARPU infers dependencies. For example, in Figure 2.4, the first submission is task POTRF_0 that accesses the data $A[k][k]$ in read-write mode. The next

¹<https://savannah.nongnu.org/projects/fkt/>

²<https://solverstack.gitlabpages.inria.fr/vite/>

```

1  [...]
2
3  starpu_data_handle_t A[N][N];
4
5  for (k = 0; k < N; k++) {
6      starpu_iteration_push(k);
7      starpu_task_insert(&cl_potrf ,
8                          STARPU_PRIORITY, 3*N - 3*k,
9                          STARPU_RW, A[k][k],
10                         0);
11
12     for (m = k+1; m < N; m++) {
13         starpu_task_insert(&cl_trsm ,
14                             STARPU_PRIORITY, 3*N - (2*k + m),
15                             STARPU_R, A[k][k],
16                             STARPU_RW, A[m][k],
17                             0);
18     }
19
20     for (n = k+1; n < N; n++) {
21         starpu_task_insert(&cl_syrk ,
22                             STARPU_PRIORITY, 3*N - (k + 2*n),
23                             STARPU_R, A[n][k],
24                             STARPU_RW, A[n][n],
25                             0);
26
27         for (m = n+1; m < N; m++) {
28             starpu_task_insert(&cl_gemm ,
29                                 STARPU_PRIORITY, 3*N - (k + n + m),
30                                 STARPU_R, A[m][k],
31                                 STARPU_R, A[n][k],
32                                 STARPU_RW, A[m][n],
33                                 0);
34         }
35     }
36     starpu_iteration_pop();
37 }
38 starpu_task_wait_for_all();

```

Figure 2.4: Task insertion of the Cholesky factorization within the STARPU runtime system.

submitted task is TRSM_0 accessing the data $A[k][k]$ in read-only mode. So, the last task that modified $A[k][k]$ (POTRF_0) will be connected with a dependency to this TRSM task. On a DAG, this is equivalent to putting an arc from POTRF_0 to TRSM_0. Similarly, the read-write access on data $A[m][k]$ by TRSM_0 creates a dependency with task GEMM_0 that is also using $A[m][k]$. The resulting DAG is similar to the one shown in Figure 2.1. Note that the loop submitting GEMM tasks (line 27) is nested inside the main loop of line 5. Also note that the GEMM loop iterates over m , making GEMM write to different data with $A[m][n]$, and $n > k$ so that no GEMM reads the result of another GEMM. Thus, for a given loop k , all GEMMs are independent of each other.

Tasks pass through an application thread and worker threads. The application thread submits the tasks one by one as shown in Figure 2.4. The worker threads resolve dependencies and send ready tasks to the scheduler, which must assign them to the workers. The application thread is asynchronous to the worker threads. With this asynchronicity, one does not have to wait for all tasks to be submitted to start computing the application. Independent tasks can be computed in parallel. No dependencies are added to an already submitted task, so, if a task has no unresolved dependencies at submission time, it is immediately ready. From Figure 2.4, we can see `starpu_task_wait_for_all`, which serves as a synchronization point after all tasks are completed.

2.6.3 Task flow

Figure 2.5 illustrates STARPU's management of tasks, from start to finish, and the role of the scheduler. This figure uses the distributed memory case with GPUs because it is the more general case, but the behavior is similar for shared memory with CPU cores. The pink boxes represent the common STARPU core. The blue elements represent the actions of the scheduler. A solid arrow is a task movement. Dashed arrows represent notifications to the system. Dotted arrows are reads. A solid red arrow indicates a data movement.

To begin with, from the application, STARPU infers a DAG through the *application thread* (1) mentioned earlier. Then, the *worker threads* read this DAG and push tasks that have all their dependencies satisfied into the *readyTasks* queue (2). The scheduler is notified that the *readyTasks* queue has been updated, and can thus start distributing the tasks into the *taskBuffer*. When PU_k is idling, it will pop the head of the *taskBuffer_k* queue of tasks. If that is empty, the scheduler will be notified and has the option to push a set of tasks into *taskBuffer_k* (3) using tasks from *readyTasks*. The schedulers have access to the performance model of each worker in order to estimate the processing time of a task. The schedulers can also look at the set of data loaded into a processing unit memory. This gives the scheduler information about what data is already loaded or in the process of being loaded. Tasks in *taskBuffer_k* are in the STARPU common core and their data are being prefetched (4), i.e. preloaded into memory to avoid waiting for unavailable data. PUs pop tasks one by one from *taskBuffer*. When a task is popped, all of its datas that are not yet in the PU memory must be loaded (5). If a PU memory is saturated and needs to load additional data, an eviction is performed (6) according to the an eviction policy. We specify the choice of such data in Section 2.6.4. Once all the data required by the popped task are in the PU memory, the task start to be computed (7). As soon as a task is completed (8), its results are written. In some cases, a task completion fulfills dependencies, which makes some tasks in the DAG ready. This is checked by STARPU (9) in order to update *readyTasks*.

2.6.4 New functionality to add custom eviction policies

The *Least Recently Used* eviction policy is the most common cache eviction algorithm. It removes the least recently used data from memory. The LRU policy keeps track of the most recent use of a data by

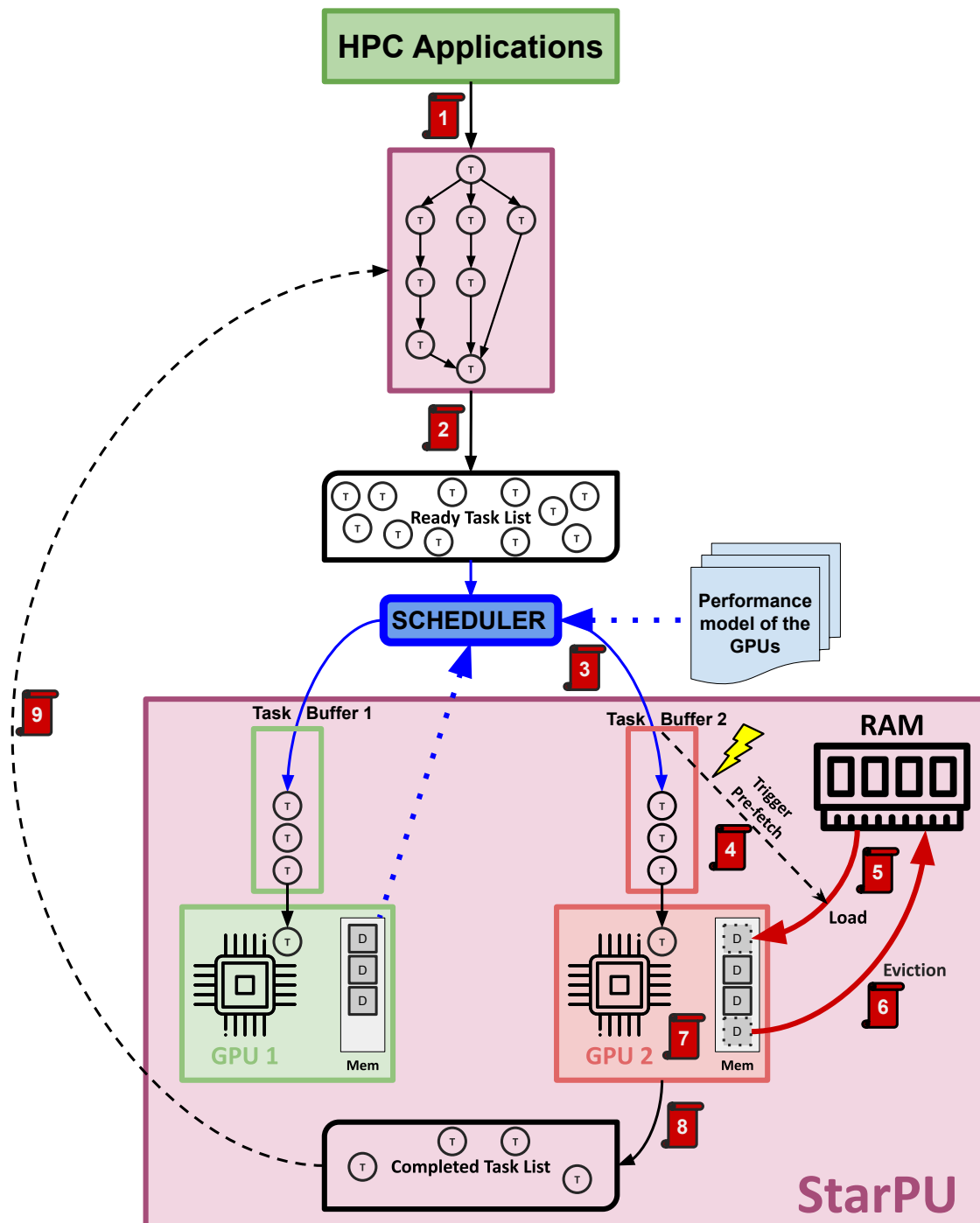


Figure 2.5: Task flow within the STARPU runtime. The pink boxes are the common STARPU core. The blue elements are the actions of the scheduler. A solid arrow is a task movement, a dashed one a notification and a dotted one a read. A red arrow indicates a data movement.


```

1 typedef starpu_data_handle_t starpu_data_victim_selector(
    starpu_data_handle_t toload, unsigned node, enum
    starpu_is_prefetch is_prefetch, void *scheduler_data);
2
3 typedef void starpu_data_victim_eviction_failed(starpu_data_handle_t
    victim, void *scheduler_data);
4
5 void starpu_data_register_victim_selector(
    starpu_data_victim_selector selector,
    starpu_data_victim_eviction_failed evicted, void *scheduler_data)
    ;
6
7 int starpu_data_can_evict(starpu_data_handle_t handle, unsigned node
    , enum starpu_is_prefetch is_prefetch);
8
9 #define STARPU_DATA_NO_VICTIM ((starpu_data_handle_t) -1)

```

Figure 2.6: Function definitions needed for custom eviction policy within the STARPU runtime.

placing the data used by the last started task at the top of a list. It can then simply pop the head of such a list to perform an eviction. LRU is a fast and usually satisfying policy to follow. However, when data movement becomes the key factor in achieving better performance, such as under memory constraints, LRU provokes pathological behavior, which we describe in the next chapter. To avoid this, we need to create custom eviction policies that match the scheduler’s strategy. For this reason, we added a new functionality in STARPU that allows a programmer to implement custom eviction policies. Functions presented in Figure 2.6 are the key components of our contribution and allow any new eviction policy to be added to STARPU.

First, `starpu_data_register_victim_selector` is called to select which selector eviction policy to use. By default, this is the LRU eviction policy. A programmer can now get his own eviction policy invoked by registering it. We also assign through `evicted` the policy that should be followed if a data returned by `selector` could not be removed from memory. `scheduler_data` contains all the personal parameters used by the current scheduling policy. It is the context of the scheduler. When StarPU needs to make room on a given memory node, the victim selector is called with `starpu_data_victim_selector`. This function can choose to return a data to evict from the memory node `node`. `toload` indicates the size and allocation pattern of the data we are trying to load, the cause of the eviction. `starpu_data_victim_selector` can use this information to evict a data with the same size and allocation pattern to avoid additional evictions and allocations, but it can also evict some other data, if possible that has at least the requested size. When selecting a data to evict, the eviction policy must check that the data is not being used by another worker. Otherwise the eviction will fail. If an eviction fails, `starpu_data_victim_selector` is called again. The PU has not processed any additional task since it needs an eviction. So the same data would be selected for eviction and rejected again, trapping the eviction policy in an infinite loop. Therefore, before returning any data to evict, `starpu_data_victim_selector` should call `starpu_data_can_evict` to check if the data `handle` can be evicted from `node`. Otherwise, it restarts its selection to choose another data to evict.

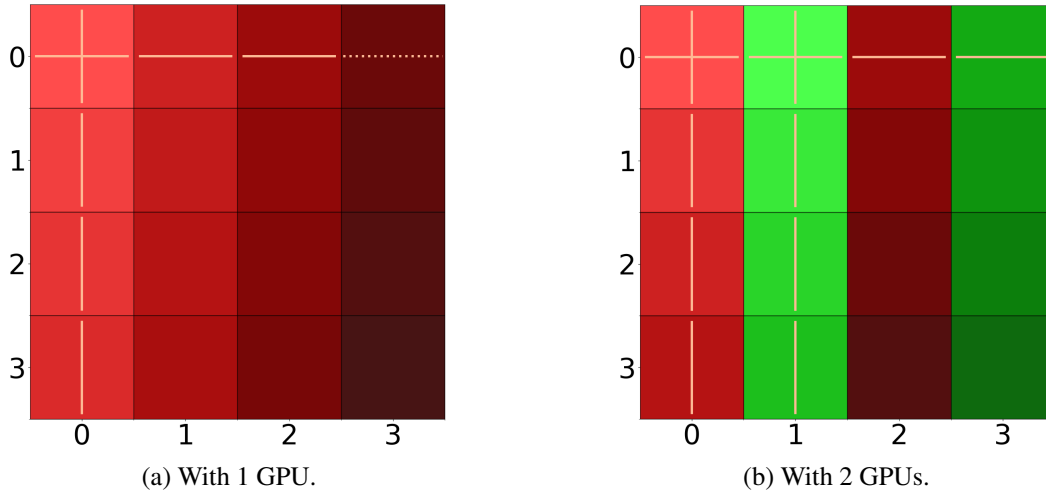


Figure 2.7: Visualization of the processing order on a 2D matrix multiplication. Side of the input matrices $N = 4$. The shading, from lighter to darker, represents the ordering. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches. With multiple GPUs, each color is a set of tasks assigned to a GPU.

An eviction can be requested by either the need for a fetch or a prefetch. A fetch is a data load that is required for the next task to be processed, a prefetch is a data preload for a task that is scheduled to be computed. Such information is provided to the victim selector with `is_prefetch`. In the case of a prefetch, it is less important to free space in the PU memory because it is not needed to resume task execution. It would also be detrimental to evict a data that is planned to be used for a close-future task in order to do a prefetch for a task further in the planned schedule. Thus, the victim selector can choose not to evict data immediately by returning `STARPU_DATA_NO_VICTIM`. Even if we use `starp_data_can_evict`, sometimes an eviction may fail. This is due to concurrency issues: the delay between the selection of a data to be evicted and its actual eviction may allow other workers to access it, making the eviction impossible. When this happens, to avoid invalid behavior, `starp_data_victim_eviction_failed` is called and the programmer must choose how to handle the situation. By default, when an eviction fails, we try to evict another data. An interesting perspective would be, for example, to be notified of the denied eviction in order to correct the eviction policy decision in the future.

2.6.5 New logging and visualization tool

To better understand the behavior of our schedulers, we introduced a new way of visualizing a task order and its impact on data transfer. We added a logging component to STARPU that automatically writes trace information about an execution. The information is written when a processing unit pops a task from `taskBuffer` to start an execution. Namely, we extract for each task: the used processing unit, the order in which it was computed relative to the total number of tasks, the number of required fetches or prefetches and the coordinates of the task (X, Y, and Z axis depending on the application data pattern).

To show the behavior of a scheduler from these logs, we developed a visualization tool in Python. See Figures 2.7 through 2.10 for small examples of these visualizations using experiments run on GPUs. Figure 2.7a shows the visualization obtained on an outer product ($C = A \times B$). Each tile is a task

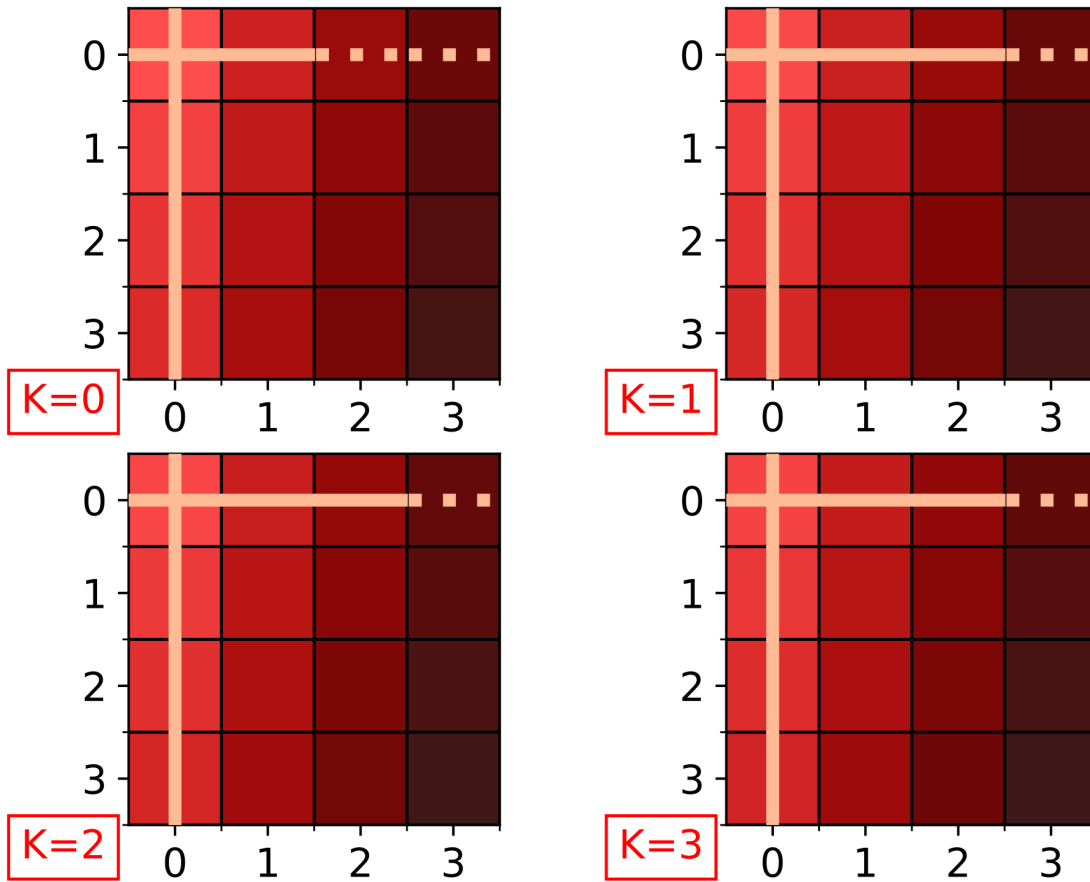


Figure 2.8: Visualization of the processing order on a 3D matrix multiplication with 1 GPU. Side of the input matrices $N = 4$. The shading, from lighter to darker, represents the ordering. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

and its position on the 2D grid represents its data requirement: a row of A and a column of B . As we can represent it with a 2D grid, we call the outer product a *2D matrix multiplication* in this thesis. The shading (from light to dark) represents the order in which the tasks were processed. We can see the first processed task of Figure 2.7a at coordinates (0,0). A horizontal beige line in a tile indicates that the column from B was not in memory when the task was processed and had to be loaded. Similarly, a vertical beige line means that a row from A was loaded before the tile was processed. Naturally, as the GPU's memory is empty at first, the first processed task requires two data loads. From the shading we learn that the tasks are computed column by column. Starting at tile (1,1), data start to be completely reused as we do not see any line in the tiles. Tile (0,3) has a dotted horizontal line. It means that the column from B was loaded during a prefetch for this tile. Figure 2.7b shows the same application but using two GPUs. For several GPUs, a color is assigned to each of them. Here, the red tiles were processed by GPU₁, while the green tiles were processed by GPU₂.

Figure 2.8 shows the processing order on a GEMM: A , B and C are square input matrices and the computation of each tile of C is decomposed into tasks each requiring one tile of A , B and C . A GEMM can be viewed as multiple layers of an outer product. We thus call this application a *3D matrix multiplication* in this manuscript. For $N = 4$ we can represent it with 4 layers K . Each layer represents

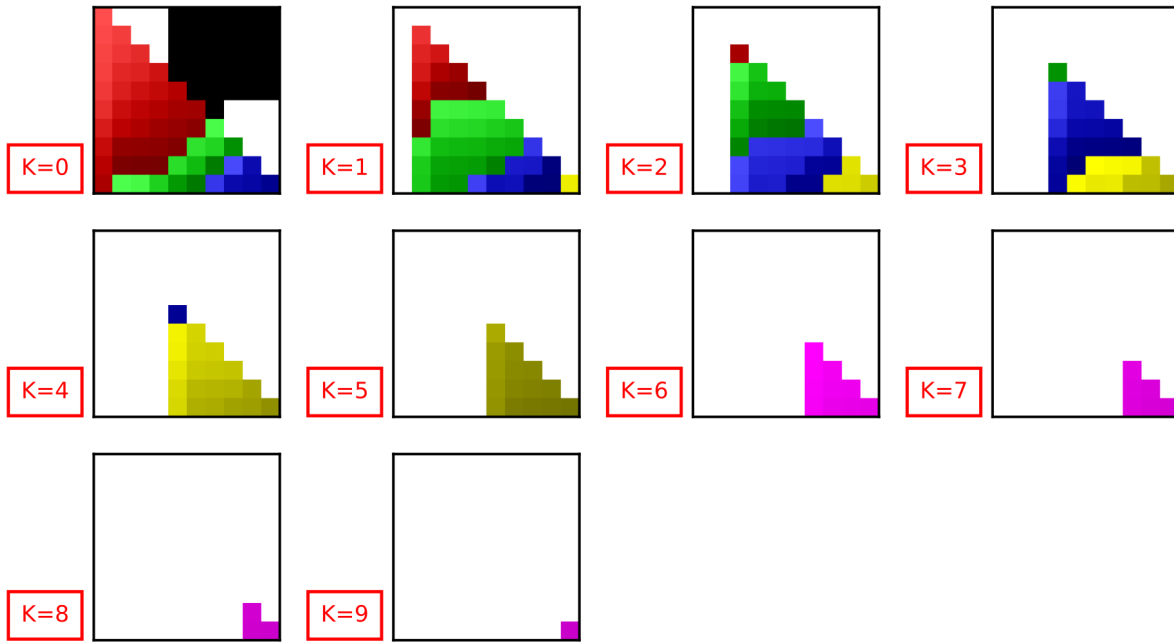


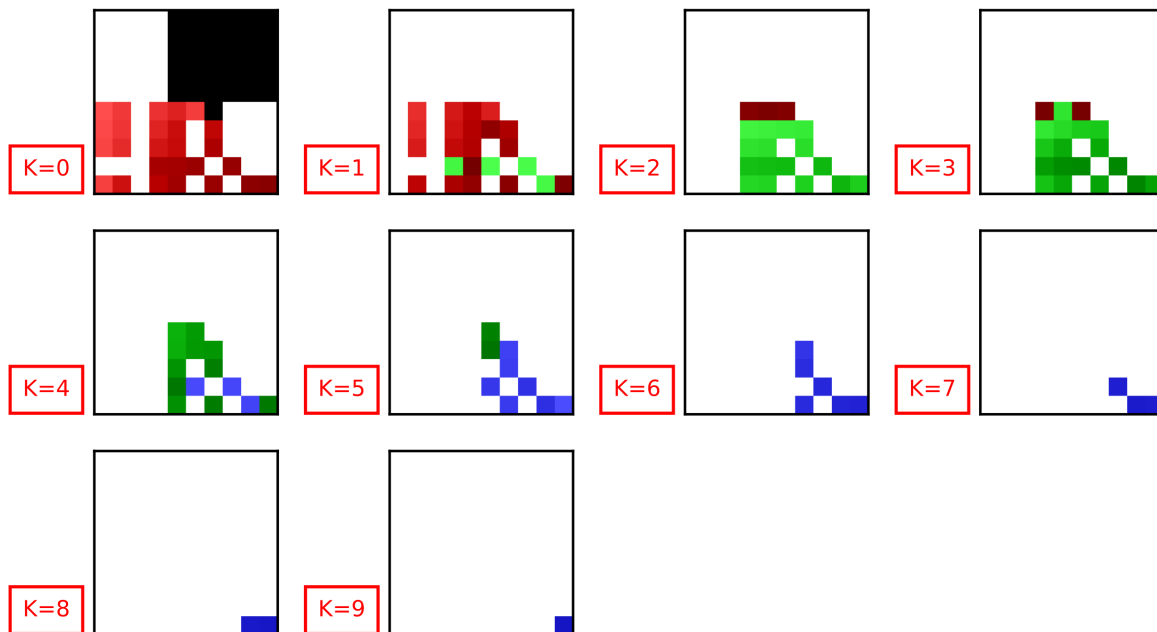
Figure 2.9: Visualization of the processing order on the Cholesky factorization with 1 GPU. Side of the input matrices $N = 10$. The first 50 tasks processed are in red, the next 50 in green, then blue, yellow, magenta and cyan. The shading, from lighter to darker, represents the ordering within each set of 50 tasks. The black area represents the amount of tiles that can be loaded in memory.

a different result matrix C . Similar to the 2D matrix multiplication visualizations, we use a shading and the lines in the tiles to represent the processing order and the amount of load required, respectively. C tiles load are not displayed. From the color shading, we can see that the first four processed tasks are the tasks at coordinates $(0,0)$ on each layer K . This is a pipelining on the different K iterations: tasks from a C tile are done in succession in order to reuse it.

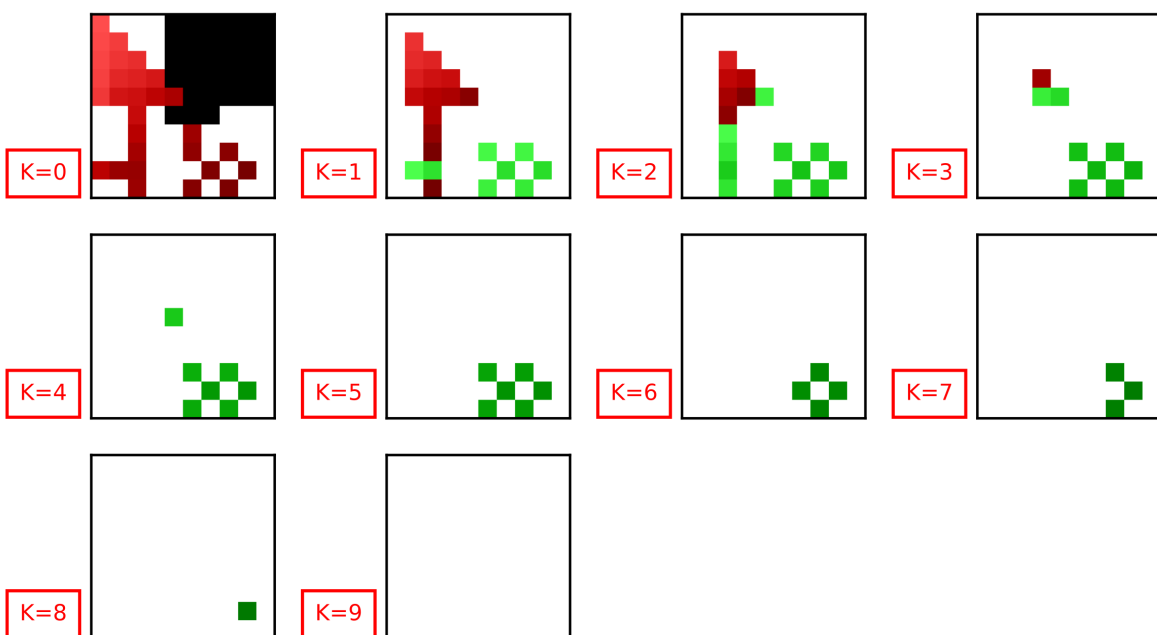
Figure 2.9 represents a scheduler's processing order on the Cholesky factorization ($A = LL^T$). The Cholesky factorization is here represented as a successions of iterations ($K = 0$ to $K = 9$) of the lower triangular matrix L . The black area represents the number of tiles that can be loaded in memory. For these representations the colors are used differently. The first 50 tasks processed are in red, the next 50 in green, then blue, yellow, magenta and cyan. The shading still represents the order within each set of 50 tasks. To avoid an overload of visual elements that would make it difficult to see the processing order, we do not plot the lines in the tiles. Thanks to the color division, we learn from such a figure that the first 50 tasks processed belong to iterations 0, 1, and 2. From the shading we learn that except for the first column of each iteration, the tasks are processed line by line.

Figure 2.10 shows the processing order with two GPUs. The colors serve the same purpose as in Figure 2.9, but each sub-figure represents the set of tasks assigned to the corresponding GPU. This format allows to more easily understand the division of the workload with a large number of tasks and GPUs.

Our logging tool is built into STARPU and can be applied to any of STARPU's schedulers. However, it is not specifically designed for STARPU. It could be integrated into any other task-based runtime system if we can plug it in when a processing unit pops a task for execution. The key requirements are to be able to extract the coordinates and data load needs from a task.



(a) Task processed by GPU 1.



(b) Task processed by GPU 2.

Figure 2.10: Visualization of the processing order on the Cholesky factorization with 2 GPUs. Side of input the matrices $N = 10$. The first 50 tasks processed on each GPU are in red, the next 50 in green, then in blue. The shading, from lighter to darker, represents the ordering within each set of 50 tasks. The black area represents the amount of tiles that can be loaded in memory.

2.7 Summary

Our goal is to build a scheduler capable of minimizing data transfers and maximizing performance by partitioning and ordering a set of tasks on one or multiple processing units. To identify the elements that make up a good strategy, it is necessary to reduce the number of observed parameters, in other words, to simplify our optimization problem. Therefore, using a task-based model, we divide our problem into four distinct steps, each more complex than its predecessor. Our first step is to schedule independent task sets on a single processing unit. We prove that this problem is NP-complete. This limits our approach to the use of heuristics. The second step is to add multiple processing units. This time the goal is twofold: partitioning and ordering. The third step is to add heterogeneous weights. Finally, we add dependencies. We present the STARPU task-based runtime system in which we have integrated our new algorithms. We extend it with two new features: a way to add custom eviction policies and a logging system to draw task order representations.

Chapter 3

Static Scheduling for a Single Processing Unit

Contents

3.1 Schedulers from the STARPU runtime system	44
3.1.1 A greedy baseline: EAGER	44
3.1.2 Deque Model Data Aware Ready (DMDAR)	44
3.2 Adapted strategies from the literature	45
3.2.1 Reverse-Cuthill-McKee (RCM)	45
3.2.2 Maximum Spanning Tree (MST)	47
3.3 Hierarchical Fair Packing (HFP)	48
3.3.1 Intuition	48
3.3.2 An NP-complete problem	48
3.3.3 Strategy	49
3.3.4 Complexity of HFP	51
3.3.5 Improving HFP with package flipping	52
3.3.6 Optimal eviction policy	53
3.3.7 Adaptation to heterogeneous data sizes	54
3.3.8 Improving the beginning of the schedule with the <i>Ready</i> re-ordering	54
3.4 Experimental settings	54
3.5 Experimental results and analysis	57
3.5.1 Results on the 2D matrix multiplication	57
3.5.2 Results on the 3D matrix multiplication	62
3.5.3 Results on the task set of the Cholesky factorization	65
3.5.4 Results on the 2D matrix multiplication with randomized task order	66
3.5.5 Results on the randomized pairs with 2D inputs	68
3.5.6 Results on the sparse 2D matrix multiplication	69
3.6 Conclusion on static scheduling for a single processing unit	70



THE focus in this chapter on *scheduling a set of tasks on one processing unit with limited memory, where tasks share some of their input data but are otherwise independent*. We use the simplified model presented in Section 2.2 and aim to solve the MIN-LOADS-FOR-TASKS-SHARING-DATA problem (see Problem 2).

The chapter is organized as follows:

- We present two existing heuristics from the STARPU runtime system in Section 3.1.
- We review and adapt two algorithms from the literature for this problem in Section 3.2.
- In Section 3.3, we describe our proposed scheduler, Hierarchical Fair Packing, a scheduler based on gathering tasks with similar data inputs into packages.
- After presenting the experimental settings used with STARPU in Section 3.4, we study in Section 3.5 the performance (amount of data transfers and total processing time) of all the schedulers mentioned above and obtained on various tasks sets coming from linear algebra operations.

3.1 Schedulers from the STARPU runtime system

STARPU provides multiple scheduling policies, including a simple baseline and a more advanced policy that considers expected completion times as a primary factor. By comparing our own scheduling approach to these two policies, we will be able to evaluate its effectiveness.

3.1.1 A greedy baseline: EAGER

EAGER serves throughout this thesis as our baseline. It is a greedy scheduler that lets processing units pick up tasks on demand from a shared queue that contains tasks in the submission order (e.g. row major for matrix multiplications). Like all other presented schedulers, it prefetches data of tasks to be computed soon.

3.1.2 Deque Model Data Aware Ready (DMDAR)

DMDA or “Deque Model Data Aware” (Algorithm 1) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the STARPU runtime [16] (also called *tmdp-pr*). DMDA is based on the HEFT scheduler [123], which is already known to be an efficient scheduler. Additionally, DMDA is a variant of DMDAS, the default state-of-the-art scheduler used by the Chameleon library [8]. These reasons make DMDA and its variants a good point of comparison. The variant used here, called DMDAR, ignores task priorities. As we consider independent tasks, such priorities are indeed not useful. DMDA computes the expected completion time $C(T_i)$ of the first task T_i in the queue of tasks, based on a prediction of the time for transferring the data to the PU (or communication time) $comm$ and of the task computation time $comp$:

$$C(T_i) = \sum_{\substack{D_j \in \mathcal{D}(T_i) \\ D_j \notin \text{InMem}(PU)}} comm(D_j) + comp(T_i) \quad (3.1)$$

Note that the data transfer time is counted only if the data is not already in memory. The task is then allocated to the processing unit which minimizes $C(T_i)$. Tasks are allocated to processing units with this rule, one by one in their order of submission. In the context of this chapter, with a single processing unit, DMDAR is reduced to selecting tasks in their submission order.

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

```

1:  $InMem \leftarrow \emptyset$ 
2: while all tasks have not been allocated do
3:    $T_i \leftarrow pop(\mathbb{T})$ 
4:   Compute  $C(T_i)$  using Eq. 3.1
5:   Allocate  $T_i$ 
6:   for each  $D_j \in \mathcal{D}(T_i)$  do
7:     Request data prefetch for  $D_j$ 
8:     Add  $D_j$  to  $InMem$ 

```

DMDAR includes an additional *Ready* strategy (Algorithm 2): tasks are reordered at runtime in order to favor tasks with the most input data already loaded into memory¹. DMDAR is a dynamic scheduler that relies on the actual state of the memory, it thus depends on the eviction policy, which is the LRU policy.

Algorithm 2 Ready reordering heuristic

Require: List L of tasks allocated

```

1: while  $L \neq \emptyset$  do
2:   Search first  $T \in L$  requiring the lowest amount of data transfers
3:   Wait for all data in  $\mathcal{D}(T)$  to be in PU's memory
4:   Start processing  $T$ 

```

DMDAR is already suited to heterogeneous data sizes, by taking them into account while computing $comm(D_j)$ and while selecting tasks in the *Ready* strategy.

3.2 Adapted strategies from the literature

We present here two heuristics adapted from the literature to solve the MIN-LOADS-FOR-TASKS-SHARING-DATA optimization problem.

3.2.1 Reverse-Cuthill-McKee (RCM)

Intuition We proved with Proof 2.2.4 the complexity of MIN-LOADS-FOR-TASKS-SHARING-DATA through a reduction from the cutwidth minimization problem. The proximity with the cutwidth minimization problem motivates the use of the Cuthill–McKee algorithm [46], which concentrates on a close metric: the bandwidth of a graph. It permutes a sparse matrix into a band matrix so that all elements are close to the diagonal. You can find an example of such re-ordering in Figure 3.1. If the resulting bandwidth is k , it means that vertices sharing an edge are not more than k edges away. We apply this algorithm on the graph of tasks $G^T = (\mathbb{T}, E^T, w^T)$ where there is an edge (T_i, T_j) if tasks T_i and T_j share some data, and where $w^T(T_i, T_j)$ is the number of such shared data. If the bandwidth of the graph is not larger than k , this means in our problem that any task T_i processed at time t has all its “neighbors” tasks (tasks sharing some data with T_i) processed in the time interval $[t - k; t + k]$. Hence, if k is low, this leads to a very good temporal data locality. Moreover, Cuthill–McKee has been used for Breadth-First

¹<https://files.inria.fr/starpu/testing/master/doc/html/Scheduling.html#DMTaskSchedulingPolicy>

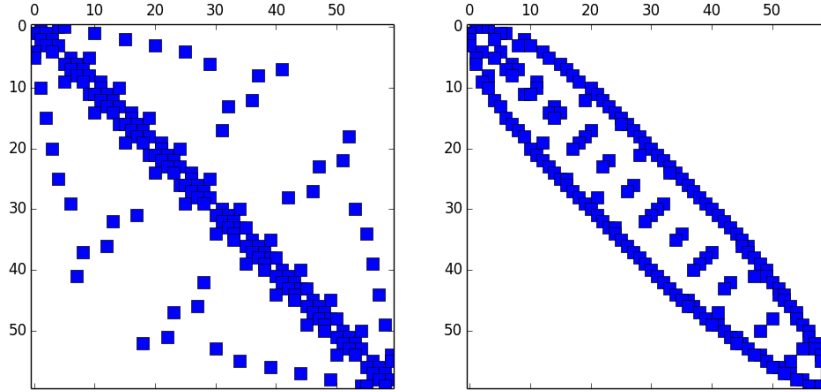


Figure 3.1: Reverse-Cuthill-McKee ordering on a symmetric sparse matrix. Image from [15].

Search in graph computing [80] or in task mapping algorithms for big scientific applications [117] in order to improve data locality and reduce communications, which motivates even more our use of it.

Reversing the obtained order is known to improve the performance of the Cuthill–McKee algorithm [96]. We prove in Proof 3.2.1 that reversing a schedule does not change the amount of data transfers. The adaption of the Reverse-Cuthill–McKee algorithm to our model is described in Algorithm 3.

Algorithm 3 Reverse-Cuthill-McKee heuristic

Build the graph G^T where vertices are tasks and edges are common data between tasks, weighted by the number of such data

$\sigma \leftarrow [v]$ where v is the vertex of G^T with the smallest weighted degree

$i \leftarrow 0$

while $|\sigma| < m$ **do**

 Let N be the set of vertices adjacent to $\sigma[i]$ in G^T not yet in σ

 Sort N by non-decreasing weighted degree

 Append N at the end of σ

$i \leftarrow i + 1$

Return σ in the reverse order

Differences between Cuthill-McKee (CM) and Reverse-Cuthill-McKee (RCM) We prove in the following theorem that both Cuthill-McKee and Reverse-Cuthill-McKee algorithms reach the same amount of data movement. More generally, reversing a schedule does not change the number of reads or evictions.

Theorem 3. For a given set of tasks \mathbb{T} sharing data in \mathbb{D} and a given task order $\sigma : \#Loads(\sigma, MIN) = \#Loads(\bar{\sigma}, MIN)$.

Proof. Given σ , an order of computation for \mathbb{T} , we know that data are used in the following order: $\mathcal{D}(T_{\sigma(1)}), \dots, \mathcal{D}(T_{\sigma(m)})$. Together with the knowledge of the *MIN* eviction policy, we can deduce the set of data that we need to load before computing task $T_{\sigma(t)}$, that we note S_t . It is the set of input data of T_i that were not in memory during the computation of the last task T_{i-1} . We reuse the *live data* definition $L(t)$ from Section 2.2.1 that represents the set of data in memory during the computation of a

task, to define S_t : $S_t = \mathcal{D}(T_{\sigma(t)}) \setminus L(t-1)$. We denote by \mathbb{S} the ordered list of datasets that we need to load before each task: $\mathbb{S} = [S_1, \dots, S_m]$. Similarly, we build \mathbb{V} , the ordered list of data that are evicted before each task: $\mathbb{V} = [\mathcal{V}(2), \dots, \mathcal{V}(m), \mathcal{V}(m+1)]$. Note that we start at task 2 (no data is evicted before the first task) and we denote by $\mathcal{V}(m+1)$ the operation needed to completely empty the memory at the end of the execution. \mathbb{S} and \mathbb{V} totally describe the memory operations for an execution, and can be used to count the number of loads:

$$\begin{aligned} \#Loads(\sigma, MIN) &= \#Loads_{ordered_list}(\mathbb{S}, \mathbb{V}) \\ &= \sum_{S_i \in \mathbb{S}} |S_i| = \sum_{\mathcal{V}(i) \in \mathbb{V}} |\mathcal{V}(i)| \end{aligned}$$

The last equality comes from the fact that each data is evicted exactly as many times as it is loaded, thanks to the last eviction that totally frees the memory.

We consider the reversed order of σ : $\bar{\sigma}$, and similarly the reversed list of loads ($\bar{\mathbb{S}}$) and evictions ($\bar{\mathbb{V}}$). We consider $\mathbb{S}' = \bar{\mathbb{V}}$ and $\mathbb{V}' = \bar{\mathbb{S}}$ and notice that the pair $(\mathbb{S}', \mathbb{V}')$ describes correct lists of loading sets and eviction sets for $\bar{\sigma}$: this is what happens if we reverse the task order, and consider that each eviction for σ is transformed into a load, and each load for σ is transformed into an eviction. Hence, the total memory used by $(\mathbb{S}', \mathbb{V}')$ for $\bar{\sigma}$ is the same as the one used by (\mathbb{S}, \mathbb{V}) for σ , and not larger than M . Because $(\mathbb{S}', \mathbb{V}')$ is a correct loading/eviction scheme, we have:

$$\#Loads_{ordered_list}(\mathbb{S}', \mathbb{V}') \leq \#Loads(\bar{\sigma}, MIN)$$

We also have:

$$\begin{aligned} \#Loads_{ordered_list}(\mathbb{S}', \mathbb{V}') &= \#Loads_{ordered_list}(\bar{\mathbb{V}}, \bar{\mathbb{S}}) \\ &= \sum_{S_i \in \mathbb{S}} |S_i| \\ &= \#Loads_{ordered_list}(\mathbb{S}, \mathbb{V}) \\ &= \#Loads(\sigma, MIN) \end{aligned}$$

Hence, we have $\#Loads(\bar{\sigma}, MIN) \leq \#Loads(\sigma, MIN)$. By reversing once again the schedule (as well as the list of loading sets and eviction sets), we obtain similarly that $\#Loads(\sigma, MIN) \leq \#Loads(\bar{\sigma}, MIN)$, proving the equality. \square

We experimentally tested CM and RCM and concluded that the performance reached by both variants are not similar. We observed that in practice, RCM is always slightly better than CM. Even if the total number of loads is the same, the distribution of loads in time is not equal: there is more overlap between data movements and computations in RCM than in CM, which allows RCM to reach better performance.

Adaptation to heterogeneous data sizes RCM can be adapted to heterogeneous data sizes by considering data weights instead of number of common data. We can thus modify line 1 of Algorithm 3 to: *Build the graph G^T where vertices are tasks and edges are common data between tasks, weighted by the weight of such data.*

3.2.2 Maximum Spanning Tree (MST)

Yoo et al. [128] proposed another heuristic to order tasks sharing data to improve data locality. They first build a Maximum Spanning Tree in the graph G^T using Prim's algorithm [43] and then order the

vertices according to their order of inclusion in the spanning tree. By selecting the incident edge with the largest weight, they increase the data reuse between the current scheduled tasks and the next one to process. The direct adaption of the Maximum Spanning Tree algorithm to our model is described in Algorithm 4.

Algorithm 4 Maximum Spanning Tree heuristic

```

for each vertex  $v_i$  do
  Set  $Key\_Value(v_i)$  to 0
 $Key\_Value(v_0) \leftarrow 1$ 
while  $|\sigma| \neq m$  do
  Choose  $v_i \in \mathbb{T} \setminus \sigma$  such that  $Key\_Value(v_i)$  is maximum
  Add  $v_i$  at the end of the list  $\sigma$ 
  for each couple  $(v_i, v_j)$  do
    Update  $Max\_Path\_Length(i, j)$ 
  for each  $v_j$  adjacent to  $v_i \cap \sigma$  do
    if  $Key\_Value(v_j) < Max\_Path\_Length(i, j)$  then
       $Key\_Value(v_j) \leftarrow Max\_Path\_Length(i, j)$ 
  Return  $\sigma$ 

```

3.3 Hierarchical Fair Packing (HFP)

We present here our main contribution of this chapter: the Hierarchical Fair Packing scheduler. As its name suggests, HFP creates *packages* of equal size M , each containing tasks sharing data. The packages are then ordered hierarchically. The intuition is to create a task order that reuses data within each package and from one package to another.

3.3.1 Intuition

HFP builds packages (denoted P_1, P_2, \dots) of tasks, which are stored as lists of tasks, forming a partition of \mathbb{T} . We denote by $\mathcal{D}(P_k)$ the set of inputs of all tasks in package P_k . We aim at building the smallest number of packages so that the inputs of all tasks in each package fit in memory: $|\mathcal{D}(P_k)| \leq M$. The intuition is that when looking for the smallest number of packages, the tasks that share the most input data will naturally be grouped together. For any k , once $\mathcal{D}(P_k)$ are loaded, all tasks in the P_k can be processed without any additional data movement. The MIN-NB-PACKAGES problem that arises from our intuition can be expressed as follows:

Problem 6. *For a given set of tasks \mathbb{T} sharing data in \mathbb{D} , is there a task partition into at most L packages P_1, \dots, P_L such that $|\mathcal{D}(P_i)| \leq M$ for each package P_i ?*

3.3.2 An NP-complete problem

We now prove that the MIN-NB-PACKAGES is NP-complete.

Proof. The certificate of our problem RP is the list of tasks grouped by package. We now verify our certificate. We need to verify that the number of packages is less than L and that the weight $W(P_i)$ of each package is less than B . To compute the weights, it is necessary, to add the weight of the data while

ignoring the duplicates. To detect duplicates it is necessary to browse $|D(P_i)| \times |D(P_i)|$ data. We are therefore in polynomial time. We now check that the size of the certificate is polynomial. We initially have $|\mathbb{T}|$ lists of data. Browsing the data is done in polynomial time ($\sum_{i=1}^{|\mathbb{T}|} |D(T_i)|$). We know that $L \leq |\mathbb{T}|$. Thus the size of the data is polynomial in $|\mathbb{T}|$. Hence the problem lies in NP.

We prove that the problem is NP-complete thanks to a reduction from the 3-partition problem: Given an integer B and $3n$ integers a_1, a_2, \dots, a_n , such that $\sum_{i=1}^{3n} a_i = nB$ the problem is to decide whether we can partition $3n$ into n triplet whose sum is B . We consider the restricted version of the problem where for all i , $B/4 < a_i < B/2$, which is still NP-complete [64]. In this variant, each subset of integers reaching B has exactly three elements.

We consider an instance I_{3P} of the 3-partition problem and build an instance I_{MinP} of the package minimization problem as follows. For each $a_i \in I_{3P}$, we create a task T_i and a_i input data $\mathcal{D}(T_i) = \{D_{i,1}, \dots, D_{i,a_i}\}$ (no input data is shared among two tasks). We set the size limit of a package to $M = B$ and the maximum number of packages to $L = n$. Thus, in instance I_{MinP} , we try to solve the following question: can we find at most n packages of input size at most M ?

$3P \Rightarrow MinP$: We prove here that if I_{3P} has a solution, then I_{MinP} has a solution. If I_{3P} has a solution, then we have n subsets of integers S_1, S_2, \dots, S_n which verify: $|S_i| = 3$ for all i . We group tasks in n packages P_1, P_2, \dots, P_n such that $P_j = \{T_i, a_i \in S_j\}$. As $L = n$, we have exactly L packages. The input size of each package is:

$$|\mathcal{D}(P_j)| = \sum_{T_i \in P_j} |\mathcal{D}(T_i)| = \sum_{a_i \in S_j} a_i = B = M.$$

Hence, this is a solution for I_{MinP} .

$MinP \Rightarrow 3P$: We now prove that if I_{MinP} has a solution, then I_{3P} has a solution. If I_{MinP} has a solution then there are at most L packages whose input size is at most M : $|\mathcal{D}(P_i)| \leq M$ for all i . We know that $\sum_{i=1}^n |S_i| = nM$, so $\sum_{i=1}^n |\mathcal{D}(P_i)| = nM$. We therefore have $L = n$ packages which must satisfy the following conditions:

$$\begin{cases} \sum_{i=1}^n |\mathcal{D}(P_i)| = nM \\ \forall i |\mathcal{D}(P_i)| \leq M \end{cases}$$

Any package with input size smaller than M would require that another package has a size larger than M , which is not possible. Therefore, we have $|\mathcal{D}(P_i)| = M$ for each package P_i . We denote by S_j the set of a_i corresponding to tasks T_i in P_j . Hence, $\sum_{a_i \in S_j} a_i = M$ for all S_j . We assume that $M/4 < a_i < M/2$, hence each S_j counts exactly three a_i , and the S_j are a solution to instance I_{3P} . \square

3.3.3 Strategy

Since building packages in an optimal way is NP-complete, we concentrate on a greedy heuristic to build them, as described in Algorithm 5. We start with packages containing a single task. Then we iteratively consider all packages with fewest tasks and try to merge each package with another package with which it shares the most input data. To do so, we first compute the number of common data between each of the smallest packages (identified in \mathbb{S}) and all other packages. Then, for each P_i of the smallest packages, we select a package P_j such that the number of data shared by P_i and P_j is maximal and merge these two packages. Then, we mark P_j as not available for a merge at this step (to avoid all small packages merging with the same package in a single step). We also enforce that only pairs of packages with a maximal number of shared data are merged to ensure good locality: if a small package cannot be merged

Algorithm 5 Hierarchical Fair Packing

```

1: Let  $P_i \leftarrow [T_i]$  for  $i = 1 \dots m$ ,  $\mathbb{P} = \{P_1, \dots, P_m\}$  and  $P_{\text{non\_connected}} \leftarrow \emptyset$ 
2:  $SizeLimit \leftarrow true$ ,  $MaxSizeReached \leftarrow false$ 
3: while  $|\mathbb{P}| > 1$  do
4:    $MinPackageSize \leftarrow \min_{P_i \in \mathbb{P}} |P_i|$ 
5:    $\mathbb{S} \leftarrow \{P_i \in \mathbb{P} \text{ with } |P_i| = MinPackageSize\}$   $\triangleright$  The smallest packages that can be merged at
   this iteration
6:   for all packages  $P_i \in \mathbb{S}$  and  $P_j \in \mathbb{P}$  do
7:      $SharedData[i][j] \leftarrow |\mathcal{D}(P_i) \cap \mathcal{D}(P_j)|$ 
8:    $MaxSharedData \leftarrow 0$ 
9:   if  $SizeLimit = true$  then  $\triangleright$  In case we are in the first phase
10:     $MaxSharedData \leftarrow \max_{P_i \in \mathbb{S}, P_j \in \mathbb{P}} SharedData[i][j]$  such as  $|\mathcal{D}(P_i \cup P_j)| \leq M$ 
11:    if  $MaxSharedData = 0$  then
12:       $SizeLimit \leftarrow false$   $\triangleright$  End of first phase: lift the size constraint for second phase
13:    if  $SizeLimit = false$  then  $\triangleright$  In case we are in the second phase
14:       $MaxSharedData \leftarrow \max_{P_i \in \mathbb{S}, P_j \in \mathbb{P}} SharedData[i][j]$ 
15:      if  $MaxSharedData = 0$  then  $\triangleright$  We have identified subsets of tasks without any common
      data with other tasks
16:      for all packages  $P_i \in \mathbb{S}$  do
17:        Merge  $P_i$  and  $P_{\text{non\_connected}}$   $\triangleright$  This package gathers all non-connected task subsets
18:        Remove  $P_i$  from  $\mathbb{P}$  and from  $\mathbb{S}$ 
19:       $\mathbb{Q} \leftarrow \emptyset$   $\triangleright$  Set of packages that are not anymore available to merge with
20:      for all  $P_i \in \mathbb{S}$  do
21:        Find  $j$  such that  $SharedData[i][j]$  is maximal and  $P_j \in \mathbb{P} \setminus \mathbb{Q}$ 
22:        if  $MaxSharedData = SharedData[i][j]$  and  $(|\mathcal{D}(P_i \cup P_j)| \leq M \text{ or } SizeLimit = false)$ 
then
23:          Merge  $P_i$  and  $P_j$ 
24:          Add  $P_i$  and  $P_j$  to  $\mathbb{Q}$ 
25: if  $|P_{\text{non\_connected\_tasks}}| > 0$  then
26:   Merge the only package in  $\mathbb{P}$  and  $P_{\text{non\_connected}}$   $\triangleright$  Retrieve all non-connected subsets
27: Return the only package in  $\mathbb{P}$ 

```

at some step, it will be more successful in a later step. If we did not prioritize small packages, a large package would always be the best one to merge at each step, as it contains more tasks and thus more data to share. This would result in creating a large package until it reaches the maximum size M , which can lead to poor scheduling decisions (for example, all data shares can be on a single column, leading to a schedule that would follow the submission order of task submission). Considering the smallest packages first allows for a fair growth of each package. Merging two packages P_1 and P_2 consists in appending the list of tasks of P_2 at the end of the list of tasks of P_1 : we never modify the order of tasks within an already built package, hence keeping a good data locality inside packages.

As presented above, we aim at building packages such that the size of the input data of a package is smaller than, or equal to M . This is done during the first phase. However, we do not stop here and we continue in a second phase to merge packages in the same way, without considering the bound on the size of input data anymore. The objective of this second phase is to create meta-packages that express the data affinity between the already built packages, in order to schedule packages sharing many common input data close to each other.

Note that we may have identified disconnected packages, that is, subsets of tasks that do not share any common data with other tasks. We then merge the packages containing these tasks in a dedicated package, denoted $\mathbb{P}_{\text{non_connected}}$, which is added to the last package in the very end of the algorithm. Eventually, the last remaining package after all merges provides the list of tasks of the final schedule.

3.3.4 Complexity of HFP

We now try to bound the complexity of HFP depending on the number m of tasks and on the maximal number of input data for any task: $\Delta = \max_i |\mathcal{D}(T_i)|$. We first remark that by keeping the input data of each package sorted, we can compute $SharedData[i][j]$ for any pair of packages P_i, P_j , with a complexity of $O(\mathcal{D}(P_i) + \mathcal{D}(P_j))$. Similarly, merging these packages can be done with the same linear complexity. The first part of an iteration of the while loop (Line 3 of Algorithm 5) consists in computing $SharedData[i][j]$ for any packages $P_i \in \mathbb{S}$ and $P_j \in \mathbb{P}$. The complexity of this phase is in $O(C)$ with

$$\begin{aligned} C &= \sum_{P_i \in \mathbb{S}} \sum_{P_j \in \mathbb{P}} \mathcal{D}(P_i) + \mathcal{D}(P_j) \\ &\leq \sum_{P_i \in \mathbb{P}} \sum_{P_j \in \mathbb{P}} \mathcal{D}(P_i) + \mathcal{D}(P_j) \\ &\leq 2|\mathbb{P}| \sum_{i \in \mathbb{P}} \mathcal{D}(P_i) \end{aligned}$$

Since all packages contain at least one task, we have $|\mathbb{P}| \leq m$. Moreover, we can bound the sum of all package inputs:

$$\sum_{i \in \mathbb{P}} \mathcal{D}(P_i) \leq \sum_{j=1}^m \mathcal{D}(T_j) \leq \Delta m$$

as each task has at most Δ inputs. Thus, the complexity of computing the shared data is in $O(\Delta m^2)$. Since there is at least one merge operation in each iteration of the while loop and there are at most m merge operations, the total complexity for computing shared data throughout the execution of the algorithm is in $O(\Delta m^3)$.

As outlined above, merging two packages P_i, P_j can be done in complexity $O(\mathcal{D}(P_i) + \mathcal{D}(P_j))$, which is bounded by $O(\Delta m)$. Since there are at most m merge operations, the total complexity for merging is in $O(\Delta m^2)$. Hence the total complexity of HFP is dominated by the computation of the shared data and is in $O(\Delta m^3)$.

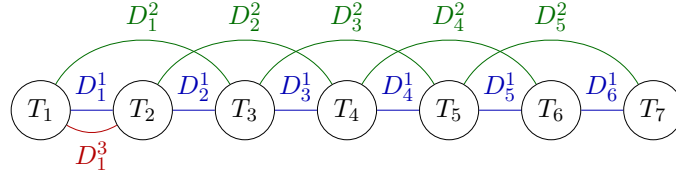


Figure 3.2: Data sharing among tasks that reach worst-case complexity for HFP (depicted for $m = 7$: an edge between two tasks represents a data shared by these tasks. Note that in addition to these data, tasks have private data so that each task exactly has Δ input data (except for T_m with $\Delta - 1$ data).

Note that this worst-case complexity can be reached on specific cases. We consider the following problem instance, depicted on Figure 3.2. Each task T_i , $3 \leq i \leq m$ has as input data D_{i-1}^1 , D_{i-2}^2 , and D_i^j for $j = 1, \dots, \Delta - 2$. Task T_1 has input data D_1^j for $j = 1, \dots, \Delta$; task T_2 has input data D_1^1, D_1^3 , and D_2^j for $j = 1, \dots, \Delta - 2$. In the very beginning, packages are made of single tasks, and they have at most one common data with another package, except for $\{T_1\}$ and $\{T_2\}$ which have two common data (D_1^1 and D_1^3), hence these packages are merged in the first step. In the second step, only package $\{T_3\}$ has two common data with the newly created package while all other packages have one or none common input data, hence $\{T_3\}$ is merge with the package created in the first step. Similarly, at step k , the package $\{T_{k+1}\}$ is merged with the package created at the previous step. At each step k , the shared input data must be recomputed. The first package contains k tasks, while the other $m - k$ packages contain a single task. Computing the shared data between the smallest packages and all other packages requires a complexity $O((m - k)^2 \Delta + m \Delta)$. Summing over all $m - 1$ steps, we reach a complexity $O((m - k)^3 \Delta)$, corresponding to the previous bound.

In contrast with the previous pessimistic scenario, the complexity may be largely reduced in some cases. In an optimistic scenario, we merge all the packages by pairs at each iteration of the while loop, resulting in $\log_2 m$ iterations of this loop. At iteration i , we have $\frac{m}{2^i}$ packages. At each iteration, the number of input data of a package at most doubles, thus this number is at most $2^i \Delta$ at iteration i . Since we need to compute many intersections of input datasets for a single merge, the cost of computing the intersections dominates the complexity. The complexity of this step for a single package at iteration i is thus $O(2^i \Delta)$. The total cost for iteration i is thus:

$$\left(\frac{m}{2^i}\right)^2 \times 2^i \Delta = \frac{m^2 \Delta}{2^i}$$

When summing over all iterations, we get:

$$\sum_{i=1}^{\log_2 m} \frac{m^2 \times \Delta}{2^i} = O(\Delta \times m^2)$$

which gives the complexity of HFP in this optimistic scenario. Note that in linear algebra operations, all tasks have a very similar data access pattern and the pattern of data sharing is regular. Hence, in practice, this optimistic complexity is often reached.

3.3.5 Improving HFP with package flipping

A concern appears in the second step of HFP (when we merge packages without taking care of the M bound): if P_i is merged with P_j , the merged package contains the tasks of P_i followed by the ones of P_j . However, the last tasks of P_i might have very little shared data with the first tasks of P_j , leading

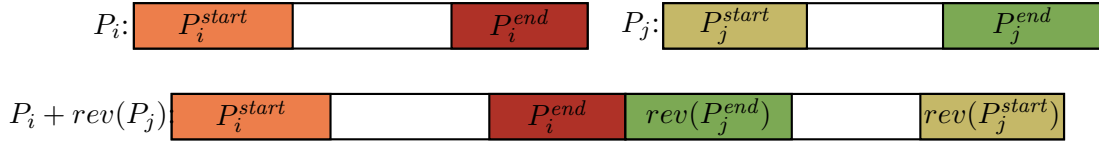


Figure 3.3: Flipping packages to improve HFP. Here we assume that the pair of subpackages (P_i^{end}, P_j^{end}) is the one with the most shared input data, so that only P_j is reversed before merging packages.

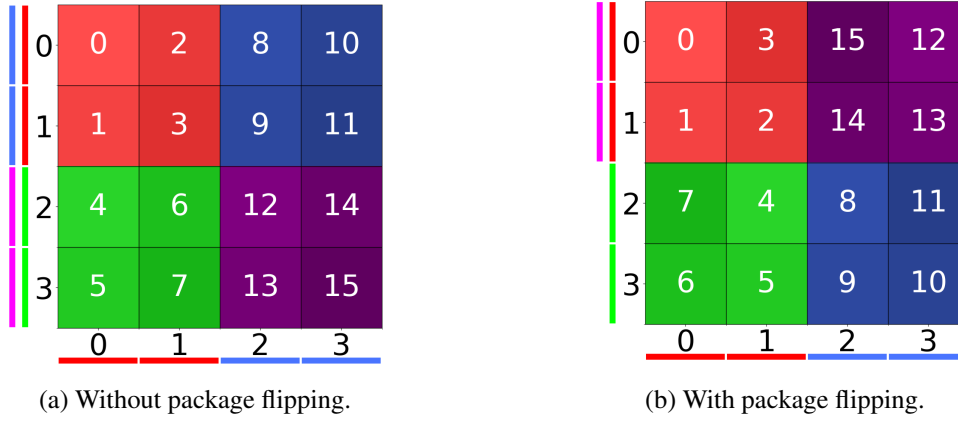


Figure 3.4: HFP's processing order on a 2D matrix multiplication with $M = 4$. The colored lines on the sides represent data loads of rows or columns for the tasks of the corresponding color.

to poor data reuse when starting P_j . Hence, for each package P_i , we consider two subpackages P_i^{start} and P_i^{end} containing the first and last tasks so that the number of their input data is smaller than M but their number of tasks is maximal, as illustrated on Figure 3.3. Then, we count the common input data of each pair: $(P_i^{start}, P_j^{start})$, (P_i^{start}, P_j^{end}) , (P_i^{end}, P_j^{start}) , (P_i^{end}, P_j^{end}) . We identify the pair with most common input data and selectively reverse the packages so that tasks in this pair of subpackages are scheduled consecutively in the resulting package.

Flipping packages requires to go through the set of tasks of two packages. In the worst case, both packages together contain all of \mathbb{T} , so the complexity is $O(m)$. This complexity can be neglected compared to the original complexity of HFP.

Figure 3.4a and 3.4b show an example of the task processing order of C on a 2D matrix multiplication with and without package flipping with $M = 4$ (4 rows or columns can fit in memory at the same time). On these figures the numbers correspond to HFP's processing order of each block of C . When going from a colored set of tasks to another, at least 2 rows or columns must be loaded. Without package flipping, to process the blue set of tasks, 2 rows and 2 columns must be loaded because the memory is filled with rows number 2, 3 and with columns number 0, 1, which gives us a total of 12 data loads. In contrast, package flipping HFP can reuse the same rows (number 2, 3) when going from the green to the blue set of tasks. We thus save 2 data loads for a total of 10 data loads. On a bigger scale, this "U-shape" formed by package flipping favors data reuse inside a package and thus reduces the total amount of data loads.

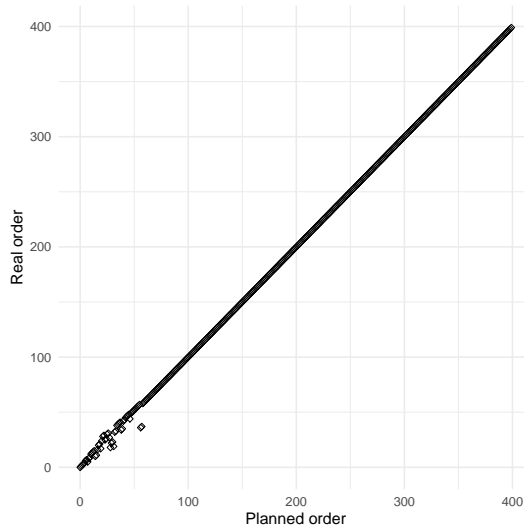


Figure 3.5: Difference between HFP (planned order) and HFP with *Ready* (real order) ordering on a 2D matrix multiplication with a single GPU. Each square is a task, and its position on the X or Y axis indicates its processing order. Deviation from linear progression means that planned and actual orders differ.

3.3.6 Optimal eviction policy

Lastly, we make another improvement to HFP: it is equipped with the optimal eviction policy adapted from Belady’s rule (see Theorem 1). To make it compatible with dynamic runtimes, such as the STARPU runtime used in our experiments, we use a dynamic version of the eviction policy: whenever the runtime needs to evict some data, we choose the one whose next usage is the latest.

3.3.7 Adaptation to heterogeneous data sizes

It is possible to extend the HFP algorithm to deal with input data that have heterogeneous sizes. We assume that $weight(D)$ gives the weight of the input data D , and that this function is extended to tasks and subsets of tasks to give the size of their input data. Line 7 of Algorithm 5 needs to be modified to compute the weight of the data shared by two tasks instead of their numbers:

$$7': SharedData[i][j] \leftarrow weight(\mathcal{D}(P_i) \cap \mathcal{D}(P_j))$$

Similarly, when testing if two packages P_i and P_j can be merged without exceeding the M bound, we need to replace the computation of the number of inputs data $|\mathcal{D}(P_i \cup P_j)|$ by the computation of the weight of these input data $weight(\mathcal{D}(P_i \cup P_j))$, on Lines 10 and 22 of Algorithm 5.

3.3.8 Improving the beginning of the schedule with the *Ready* re-ordering

We also apply the *Ready* strategy (Algorithm 2) to HFP. We can observe the effect of the reordering in Figure 3.5. Only the first 50 tasks are reordered. HFP groups tasks into packages of size M . For a data load of size M , a lot of data reuse occurs. However, this does not mean that the first few tasks in such a package will use similar input data. So, HFP’s scheduling does not allow for much data reuse at the very beginning of execution, but this is compensated for once a package’s data is fully loaded. The *ready* strategy helps refine the schedule by improving temporal locality at the beginning of the experiment.

3.4 Experimental settings

We now present the experimental settings used to compare the above schedulers to existing scheduling techniques in runtime systems². Experiments were conducted on a Tesla V100 GPU using cuBLAS 10.2 GPU kernels with single-precision floating-point numbers. Throughout this document, when we refer to GPUs, we are referring to the Tesla V100 GPUs from the Gemini-1 node on the Grid5000 computing platform³. The GPU memory is limited to $M = 500 MB$. This limitation allows us to distinguish the performance of different strategies even on small datasets. The V100 GPU has a memory of $M = 32 GB$, thus we would have to use applications with a dataset 64 times larger in order to have the same ratio between the dataset size and the GPU memory. The time and power consumption costs would be much higher to make the same observations. In order to test the performance of the strategies in different conditions, we have also used the ability to run a STARPU application with the SimGrid simulator [36].

The study in this chapter is limited to independent tasks, hence we concentrate on one of the major kernels in linear algebra made of independent tasks: matrix multiplication, as well as several variants. All our applications are based on linear algebra operations and we use matrices composed of tiles of 960×960 single-precision reals, a size that allows for a good tradeoff between performance and task density.

2D matrix multiplication. To compute $C = A \times B$, each task corresponds to the multiplication of one block-row of A per one block-column of B . Input data are thus the block-rows of A and block-columns of B . You can find an example of such data dependencies on Figure 3.6. Throughout this thesis, all matrices are square.

3D matrix multiplication. All matrices (A, B, C) are square and contains $N \times N$ tiles, and the computation of each tile of C is decomposed into multiple tasks, each of which requires one tile of A, B and C .

Task set of the Cholesky factorization. We consider the tasks of the tiled Cholesky decomposition [6] on a square $N \times N$ matrix, but remove all dependencies, as we are interested only in independent tasks. The given set of tasks is representative of what a scheduler might be exposed to at some point in a large execution with dependencies. Even if this does not compute the actual Cholesky decomposition, it allows to have data dependencies with a more complex regularity than the 2D or 3D matrix multiplication.

2D matrix multiplication with randomized task order. We consider the set of tasks from the 2D matrix multiplication, but with a randomized submission order.

Randomized pairs with 2D inputs. We consider the set of tasks and data from the 2D matrix multiplication, but with a random dependency pattern between tasks and data: each task requires one (random) block-row of A and one (random) block-column of B . This allows us to test our algorithms on an unstructured dependency graph.

Sparse 2D matrix multiplication. Starting from the 2D matrix multiplication scenario above, we randomly remove 90% of the tasks, thus largely increasing the communication-to-computation ratio.

²The code used to reproducibly obtain the results of this chapter is available at <https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/FGCS2021>

³<https://www.grid5000.fr/w/Lyon:Hardware#gemini>

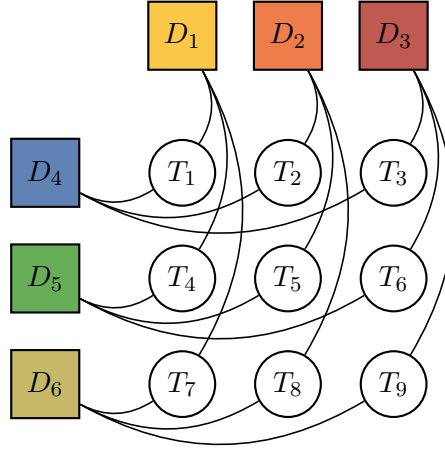


Figure 3.6: Data dependencies on a 2D matrix multiplication.

While we use the terms 2D and 3D matrix multiplication to make the difference clear, they refer respectively to the outer product and General Matrix Multiply.

We use the five scheduling heuristics presented above. DMDAR is considered as the state-of-the-art strategy as it is the one used in the Chameleon library. Each scheduling algorithm receives the whole set of tasks of an application in the submission order (row by row for a matrix multiplication for instance), and then outputs this same set of tasks in a new order, which is used in STARPU to process tasks on the GPU. We measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s, thus the higher, the better), as well as the total volume of data transferred between CPU and GPU (which we try to minimize) while varying the working set size. We define the working set size as the size in bytes of all input data. For example, for the 2D matrix multiplication, since we have 2 matrices of $N \times n$ tiles (where $n = 4$), each tile counting 960^2 elements of 4 bytes, we have:

$$working_set_size = 2 \times N \times n \times 960^2 \times 4$$

Each result is the average of the performance obtained over 10 iterations. For most of the results, the deviance is less than 2%, thus, we do not show error bars in the following graphs.

The plots in Chapters 3, 4 and 5 are all using the formalism shown on Figure 3.7. It shows the performance when varying the problem size. The dotted horizontal black line at the top represents the maximum throughput that the GPU can achieve when processing the selected application (without I/Os) and is thus our asymptotic goal. The red dotted vertical line denotes the situation when the GPU memory can fit exactly only one of the two input matrices, and the orange line denotes the situation when it can accommodate both input matrices.

Unless specified otherwise, for HFP we enable all three optimizations: *Ready* dynamic task reordering of DMDAR (see Algorithm 2), package flipping (called flip on the plots), and Belady's optimal eviction policy (called Belady on the plots). All schedulers use LRU's eviction policy except for HFP (unless otherwise stated). In some cases, we take into account the scheduling's overhead of HFP. Otherwise, it will not be counted when measuring performance. Our proposed heuristic is an offline scheduler, thus, if we know the size of the matrix as well as the size of the GPU memory, the processing order can be generated ahead of execution. For the same reasons, MST and RCM's scheduling time are not taken into consideration. On the contrary, DMDAR and EAGER are dynamic, so we cannot pre-compute their schedule and their scheduling time cannot be ignored. The overhead of the Belady's eviction policy (which is applied at runtime) is always taken into account in the results.

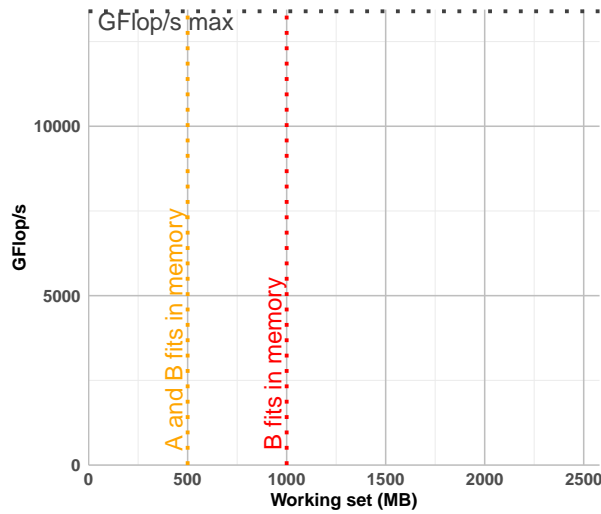


Figure 3.7: Conventions used in experimental evaluation figures.

3.5 Experimental results and analysis

We now present the results obtained by ordering the previously mentioned applications on a single GPU.

3.5.1 Results on the 2D matrix multiplication

General overview Figure 3.8a shows the performance of each scheduling heuristic when varying the size of the problem, while Figure 3.8b shows the amount of data transfers. Figure 3.9 shows the performance while varying the available memory. On Figure 3.8b, the black dotted curve represents the maximum number of transfers that can be done during the minimum time for computation (given by the bound on the throughput), thus the hard limitation induced by the PCI bus bandwidth: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal time for computation.

Communication lower bound for 2D matrix product The solid black line on Figure 3.8b represents the lower bound of the amount of data transfers required to complete the matrix product. The derivation of this lower bound follows the ideas introduced by Hong and Kung [77] and later used by many other studies. Note that we cannot use general formulas for the communication lower bound of the matrix multiplication (such as [116]) as it deals with 3D matrix multiplication, while we consider here the 2D matrix multiplication.

The considered matrices have the following size: matrix A is $N \times n$, matrix B is $n \times N$ and matrix C is $N \times N$. All these sizes are in number of blocks, and the memory size of a block is S . We denote by $m = M/nS$ the maximum number of block rows of A (or block columns of B) that fits in memory. We define a *phase* of the computation as a time window during which at most m block columns of A (or block columns of B) are read (corresponding to a volume of I/O of M). Together with the m block rows/columns that may originally reside in memory, $2m$ block rows/columns are available for the computation. We are looking for an upper bound on the number of computations that may be performed using these $2m$ blocs rows/columns of data. The most optimistic situation is when m blocks rows of A and m blocks rows of B are available, leading to m^2 block computations. Since in total, we need to

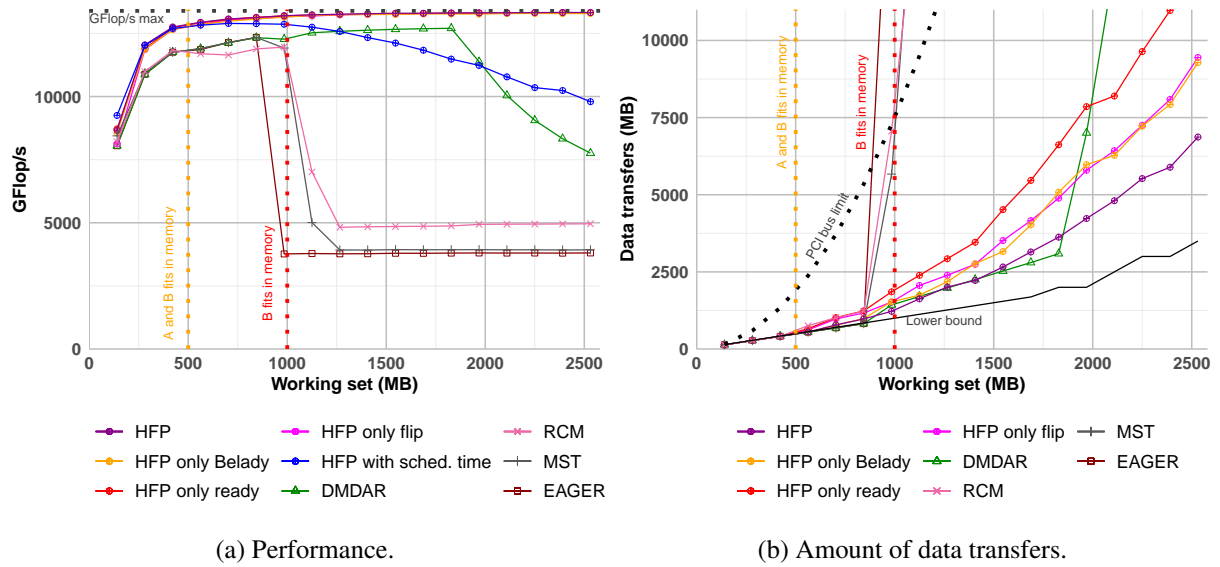


Figure 3.8: Results on the 2D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90. Memory limited to 500 MB.

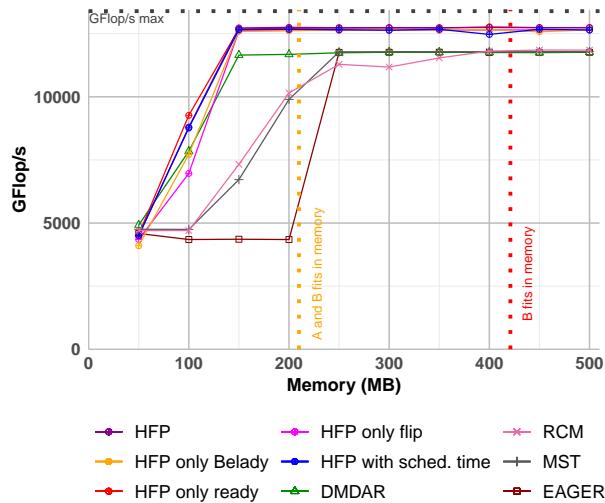


Figure 3.9: Performance on the 2D matrix multiplication in real execution with 1 Tesla V100 GPU while varying the memory size, N set to 15. Memory limited to 500 MB.

perform N^2 block computations, we have at least $\lfloor N^2/m^2 \rfloor$ full phases. Hence, a lower bound on the I/O is given by:

$$LB_{IO} = \left\lfloor \frac{N^2}{m^2} \right\rfloor M = \left\lfloor \frac{N^2 n^2 S^2}{M^2} \right\rfloor M = \left\lfloor \frac{\text{input_matrix_size}^2}{M^2} \right\rfloor M$$

where $\text{input_matrix_size} = nNS$ is the size of an input matrix (A or B). We may slightly refine this bound, by acknowledging the specificity of the first phase: no data is initially in memory when the computation starts. Hence, we define the first phase as the interval when the first $2m$ rows/columns are read (instead of m), which also leads to at most m^2 computations. We finally have two cases:

- Both input matrices fit in memory, leading to no full phases. In this case, the amount of I/O is $2 \times \text{input_matrix_size}$.
- Both input matrices do not fit in memory, leading to at least one full phase. In this case, the first phase leads to $2M$ I/Os.

This is summarized in the following formula:

$$LB_{IO} = \left\lfloor \frac{\text{input_matrix_size}^2}{M^2} \right\rfloor M + \min(M, 2 \times \text{input_matrix_size})$$

A pathological matrix size for EAGER, MST and RCM The EAGER, MST and RCM heuristics switch to pathological behavior at the red vertical line. We can both see the throughput plummeting (Figure 3.8a) and the data transfers increasing (Figure 3.8b) at the same working set size. These schedulers tend to process tasks along the rows of C . To explain the results, we need to understand LRU's behavior when multiplying matrix. We multiply A by B to get C . For small matrices, we can for example load all of B , a row of A and a piece of C to write the result in it. This results in few data transfers and thus good performance as we can see on Figure 3.8a before 1 000 MB. After 1 000 MB however, neither A nor B fits in memory. The scheduler is therefore forced to load a few columns of B , a row of A and a block of C . It computes the first row of C . Unfortunately it could not load all the columns from B , so when it wants to compute a block for which not all the data are in memory, it has to evict the first column from B (the one used least recently) in order to load the column of B it needs. But when it goes to the computation of the second row of C , it needs the first columns of B that it just evicted. It must therefore again evict the last columns of B . This generates many additional data transfers as we can see on Figure 3.8b. Consequently all the algorithms treating tasks row by row or column by column will suffer from this well-known pathological case of LRU. HFP aims to avoid this pathological case.

To help us understand the results, we can use the visualization tool that we developed. It represents the order produced by a scheduler as well as the resulting data loads by showing matrix C . Figure 3.10 shows the ordering of RCM on a matrix multiplication with 1 GPU. For better readability, our visualization uses a smaller memory ($M = 250 MB$) and matrix size ($N = 20$). However it leads to the same behavior as the eighth point ($N = 40$ and $M = 500 MB$) of Figure 3.8a. The shade and the numbering represent the processing order. A tile is fitted with a horizontal (resp. vertical) beige line if a column (resp. row) load was necessary before computing the tile. This line may be solid, representing a blocking fetch operation, or dotted for a prefetch done in the background of previous computations.

As we can see on Figure 3.10, by looking at the amount of solid horizontal lines in the squares, processing tasks rows by rows generates a lot of loads for the same columns of B . A similar phenomenon happens with EAGER and MST, and thus explains the amount of data transfers on Figure 3.8b for EAGER, MST and RCM.

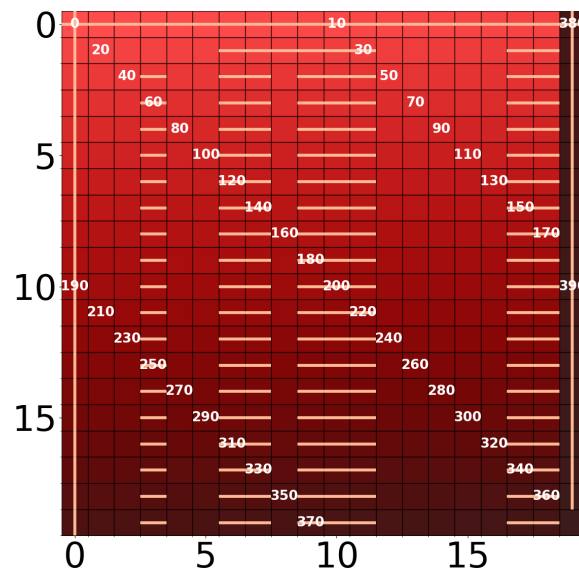


Figure 3.10: Visualization of RCM's processing order (represented by the shading from lighter to darker as well as the numbering) on the 2D matrix multiplication. $N = 20$ and $M = 250 MB$, which corresponds to the 8th point of Figure 3.8a. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

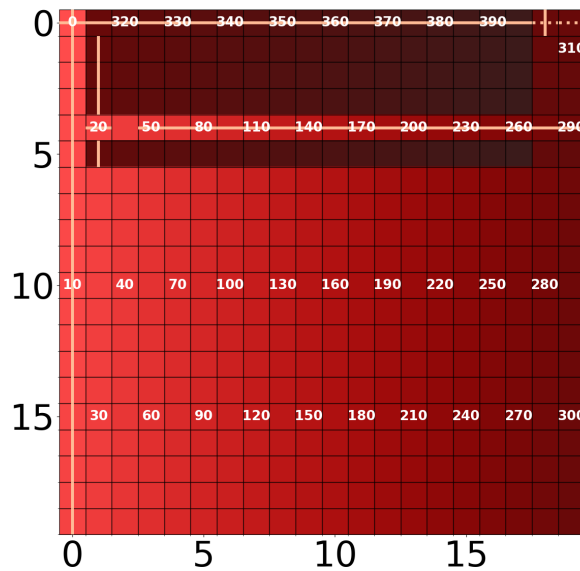


Figure 3.11: Visualization of DMDAR's processing order (represented by the shading from lighter to darker as well as the numbering) on the 2D matrix multiplication. $N = 20$ and $M = 250 MB$, which corresponds to the 8th point of Figure 3.8a. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

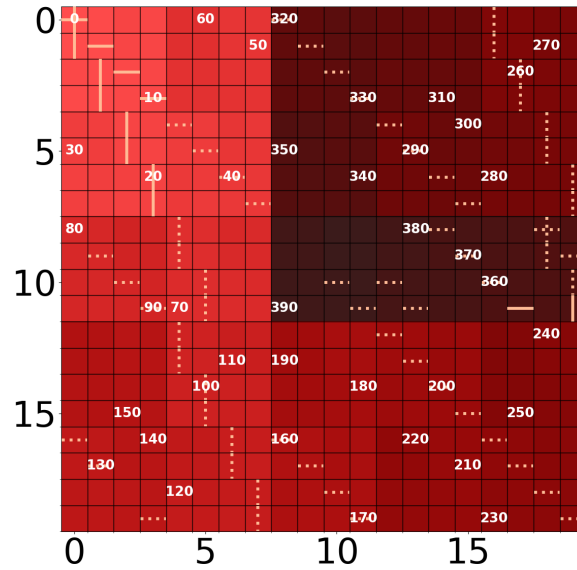


Figure 3.12: Visualization of HFP’s processing order (represented by the shading from lighter to darker as well as the numbering) on the 2D matrix multiplication. $N = 20$ and $M = 250 MB$, which corresponds to the 8th point of Figure 3.8a. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

DMDAR results As we can see on the visualization (Figure 3.11), DMDAR first processes the first column of C . Then instead of processing the first block from the second column of C , it will process a block from the second column on row number 4 because those data are already loaded in memory. Then it will continue with blocks from the second column. So, DMDAR does not suffer from LRU’s pathological case because its *Ready* strategy allows it to rather process tasks that need the block-row of A already in memory instead of reloading the whole matrix. DMDAR’s data transfers however start to rise for the last five working set sizes as we can see on Figure 3.8b. That corresponds to the performance drop on the last five points of Figure 3.8a. The reason is a conflict between data prefetching and eviction. Indeed, once the GPU is filled with data, it is not clear for DMDAR whether some data should be evicted in order to perform more prefetches. It will thus rather stop prefetching data as long as all the data currently in the GPU will be useful for the subsequent tasks to be executed there. Also, when some data is actually evicted, DMDAR does not reconsider the task ordering according to the new set of data loaded on the GPU. The basic problem of DMDAR here is that it does not have a global view of the whole set of data and tasks, and thus cannot find a balance between prefetching and eviction. HFP aims to solve this issue.

HFP results As we observe on Figure 3.8a, all HFP variants (excluding HFP with scheduling time) get performance very close to ideal. Indeed, it tends to gather tasks that compute a square part of C that requires parts of A and B , that can fit in memory size M . On Figure 3.12, thanks to the color shade (and the numbering), we can distinguish that the processing order forms rectangles of blocks of C . This allows HFP to execute a lot of tasks with very few data to load. Moreover, we can see on the visualization that most of these data loads are done during a prefetch (dotted beige lines in the tiles), thus allowing to overlap computations and data transfers. As we can see on Figure 3.8b, HFP with only *Belady*, *flip* or *Ready* induces more data transfers than DMDAR. However HFP performs better than

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	106.3 %	87.6 %	72.9 %	15.1 %

Table 3.1: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the 2D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90 (Figure 3.8a).

DMDAR, even with these variants, thanks to better prefetching. It means that HFP is able to better distribute data transfers over time, while DMDAR has to transfer a lot of data at once when computing a new row of C . It is also worth noting that HFP with all optimizations (Belady, flip and *Ready*) greatly reduces data transfers compared to its variants with only one optimization. As a consequence, HFP stays very close to the amount of data transfers required to complete the matrix product with at most two times more transfers than the lower bound. This does not impact performance in 2D, but we will see in the following applications that combining all the optimizations is crucial to achieve peak performance. Table 3.1 offers us a summary of the performance of HFP with respect to the other algorithms.

The blue curve in Figure 3.8a represents the performance of HFP when the scheduling time is not ignored and with a *fast start*, that is, with a reduced complexity for the first iteration of HFP. Indeed, during the first iteration of HFP, the search for the pair of tasks sharing the most data leads to a huge complexity (m^2 intersections to compute). For the 2D and 3D matrix products, the tasks are all identical, so we can predict the first merge of tasks of HFP. Thus, we merge together the tasks sharing at least one data without looking for the maximum of intersections. This leads to a reduced complexity without impacting the scheduling quality. HFP with scheduling time + *fast start* is outperformed by DMDAR only on working set sizes between 1 300 and 1 900 MB. The scheduling time of HFP greatly reduces its performance when the number of tasks becomes important, which may cancel out the benefit of a better locality. However, as the full set of task is available at the start of the computation, the whole schedule can be pre-processed offline, thus suppressing the scheduling cost. For the sake of readability, we do not show HFP with scheduling time on the next applications.

Figure 3.9 shows the dual view of Figure 3.8a: The working set is now set to 422 MB and we vary the amount of available GPU memory. The measurements at 500 MB on Figure 3.9 (its last point) are the same as the measurements at 422 MB on Figure 3.8a (its third point). We can observe the same results as on Figure 3.8a but reversed: when the available memory is smaller than the working set, heuristics get pathological behavior. Since we strongly reduce the amount of available memory, we get a more restrictive situation, and the *Ready* task selection provides a large improvement on the second point. HFP with scheduling time is also always better than DMDAR. This graph confirms that HFP can achieve better performance than DMDAR, even under very constrained memory.

3.5.2 Results on the 3D matrix multiplication

General overview On Figure 3.13a, we plot the performance for all heuristics on the 3D matrix multiplication in real execution. On this set of tasks, matrix C now plays a role in affinities since its tiles accumulate contributions. Figure 3.13b shows the amount of data transfer from the RAM to the GPU in these experiments. Remember that we do not count results written back to main memory in these data transfers. The green dotted vertical line indicates the working set size that allows all three input matrices to be loaded into the GPU’s memory. Figure 3.14 shows the same experiment but in simulation with a performance model from the same GPU.

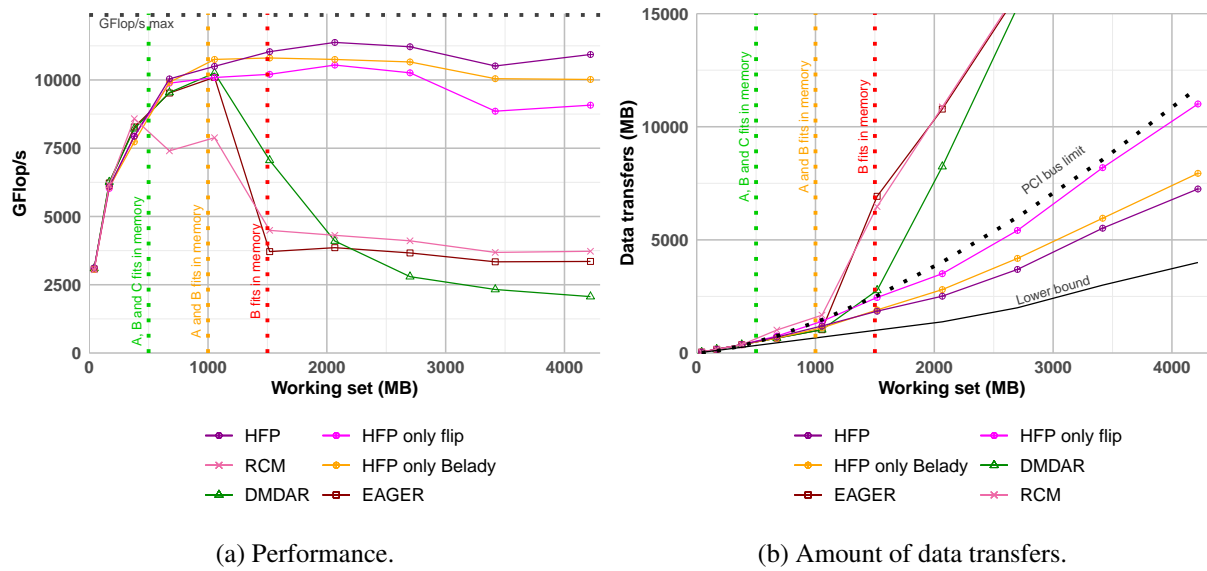


Figure 3.13: Results on the 3D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 2 to 20. Memory limited to 500 MB.

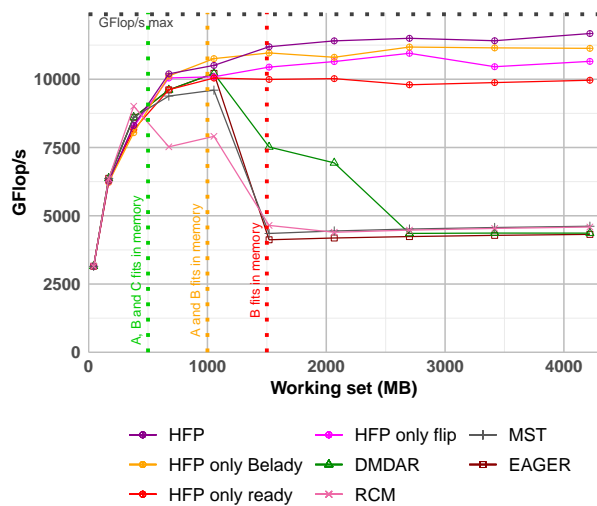


Figure 3.14: Performance on the 3D matrix multiplication in simulation with the performance model of 1 Tesla V100 GPU, N ranging from 2 to 20. Memory limited to 500 MB.

Communication lower bound for the 3D matrix product As for the 2D matrix product, we plot in Figure 3.13b with a solid black line the lower bound of the amount of data that need to be loaded from the main memory to the GPU memory (we do not consider the communication of the results back to the main memory). We can indeed apply the lower bound on data reads from [116]:

$$LB_{IO}^{3D} = \frac{2nN^2}{\sqrt{M/S}}S - 2M$$

This bound can be slightly tighten by computing the number of full phases with a floor function:

$$LB_{IO}^{3D} = 2M \left\lfloor \frac{nN^2S}{M\sqrt{M/S}} \right\rfloor$$

As previously for the 2D matrix product, we take the maximum between this quantity and the size of A and B as both input matrices should be loaded at most once:

$$LB_{IO}^{3D} = \max \left(2M \left\lfloor \frac{nN^2S}{M\sqrt{M/S}} \right\rfloor, 2nNS \right)$$

EAGER and RCM results EAGER and RCM still compute tasks row by row, but one layer of C at a time. Thus, they still get pathological performance, but this time, when the memory cannot accommodate matrix A and B , as we can see on Figure 3.13a. Indeed, their numbers of loads (on Figure 3.13b) get dramatically high after the orange dotted line, from 1100MB on the fifth point to 7600 MB on the sixth point. We observe the same results in simulation on Figure 3.14.

DMDAR results DMDAR suffers here from the same issue of balance between prefetching and eviction, mentioned in Section 3.5.1. It loads the full set of data from C with prefetches. Thus, when matrices A and B cannot fit in memory, we are in a situation where the memory is filled with data from C , but we still need to evict and load data from A and B . In this case, DMDAR will load a full column of B and then evict and load data from the rows of A , in order to avoid the eviction of prefetched data. This generates a lot of transfers as we can see on Figure 3.13b. This affects the throughput: DMDAR achieves good performance for the first five points but, once the memory is constrained, the performance is diminished. On Figure 3.13a, DMDAR has worse performance than EAGER for the last three points, because of the scheduling cost induced by looking at the full set of tasks in order to find a task that can load a minimum number of data. With a large number of tasks, as it is the case with the 3D matrix, this causes a large scheduling cost that overcomes the benefit of DMDAR's scheduling. As we can see on Figure 3.14, DMDAR achieves more sustained performance for the last few points when the scheduling time is ignored, as it is the case in simulation.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	68.3 %	64.0 %	73.7 %	66.5 %

Table 3.2: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the 3D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 2 to 20 (from Figure 3.13a).

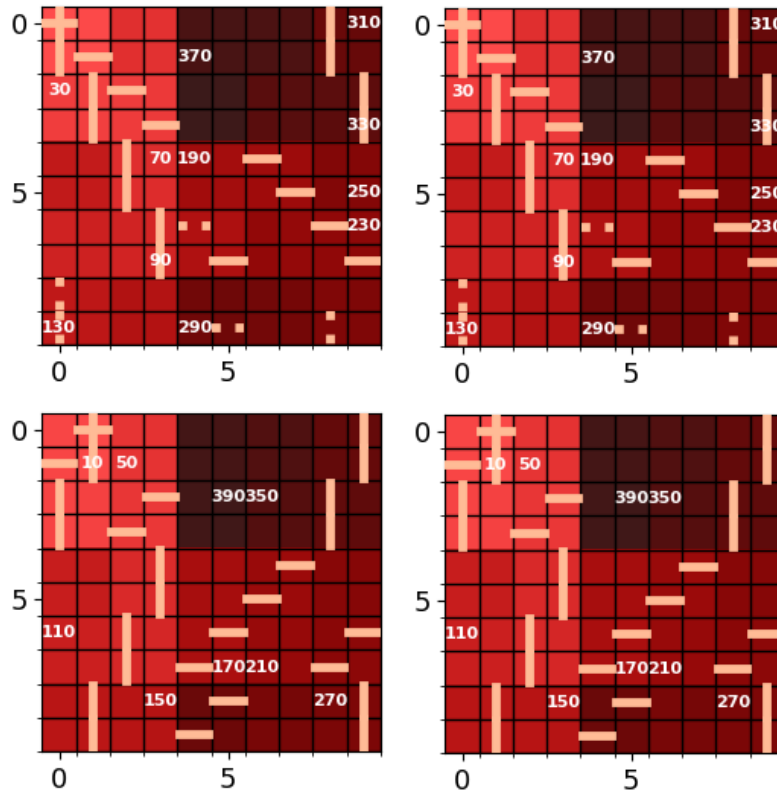


Figure 3.15: Visualization of HFP's processing order (represented by the shading from lighter to darker as well as the numbering) on the 3D matrix multiplication with only 4 layers. $N = 10$ and $M = 250$. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches. C tiles load are not displayed.

HFP's results For the sake of readability, Figure 3.15 shows a visualization of a smaller version of the 3D matrix multiplication, when the size of the inner loop is limited to 4 (matrix A is $N \times 4$ while B is $4 \times N$). The whole subset of tiles products (of size $N \times 4 \times N$) is plotted in 4 panels: the same tile on all four panels uses the same tile of C . HFP keeps gathering tasks forming rectangular blocks with all four layers of C , which provides a good locality between input matrices A , B and C , as well as some data loads during a prefetch (as one can see with the beige dotted lines). It allows HFP to generate at most two times more transfers than the lower bound of data transfers. Table 3.2 shows us the percentages of average improvement of HFP over the other heuristics. HFP is thus still better on average. It is worth noting that HFP without the Belady rule gets higher performance than RCM and DMDAR. HFP's ordering is able to reduce data transfers even without the optimal eviction policy. The

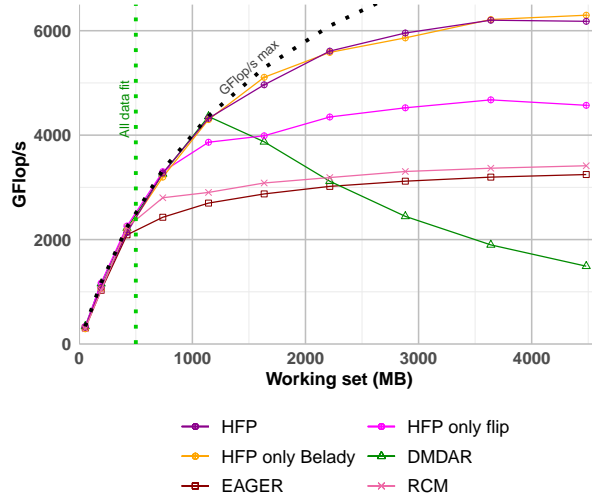


Figure 3.16: Performance on the task set of the Cholesky factorization in real execution with 1 Tesla V100 GPU, N ranging from 5 to 50. Memory limited to 500 MB.

Belady rule further reduces the quantity of data transfers, which explains the better performance of HFP with this eviction policy.

As we can observe on Figure 3.14, HFP still outperforms the other heuristics when the scheduling time is not taken into consideration. The percentage of average improvement in simulation can be seen on Table 3.3.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	61.8 %	60.4 %	68.9 %	46.0 %

Table 3.3: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the 3D matrix multiplication in simulation with 1 Tesla V100 GPU, N ranging from 2 to 20 (from Figure 3.14).

3.5.3 Results on the task set of the Cholesky factorization

Figure 3.16 shows the performance of each scheduling heuristic on the task set of the Cholesky factorization (with dependencies being ignored). The green dotted vertical line denotes the working set size allowing to load all the data needed for the computation on memory.

EAGER and RCM’s results We notice that the EAGER and RCM heuristics get pathological performance as soon as the whole matrix cannot fit the memory. They indeed do not manage to reuse more than one tile between consecutive tasks, thus entailing a lot of tile reloads.

DMDAR and HFP’s results DMDAR has similar results with HFP for a working set inferior to 2.5 times the memory. DMDAR indeed takes advantage of the actual task submission order of the Cholesky algorithm, which starts with tasks which require few input data (POTRF and TRSM kernels). Meanwhile, it can load data for the subsequent tasks with more input dependencies (GEMM kernel). HFP, on the contrary, does not pay attention to the task submission order, and aims for data sharing

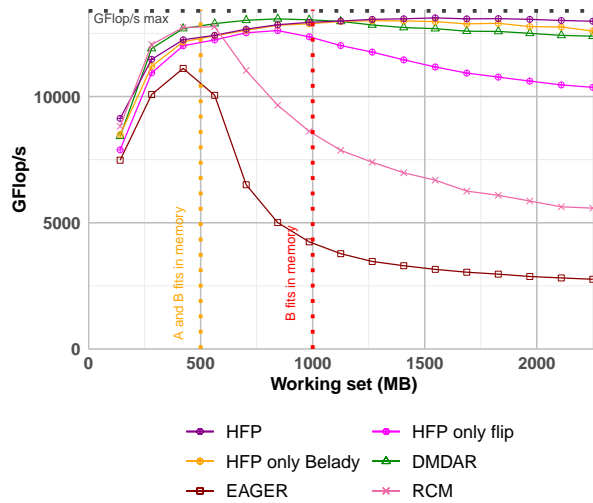


Figure 3.17: Performance on the random task order from 2D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80. Memory limited to 500 MB.

as much as possible. It will thus introduce a lot of GEMM tasks at the beginning of the execution, and is therefore producing more data transfers. As the working set increases, however, HFP achieves better performance than DMDAR. Belady’s eviction policy is crucial in this case, indeed HFP with only Belady greatly increases the performance and allows us to stay close to our asymptotic goal. The average improvement of HFP over the other heuristics with Cholesky are available on Table 3.4.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	66.8 %	57.5 %	56.3 %	65.8 %

Table 3.4: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the task set of the Cholesky factorization in real execution with 1 Tesla V100 GPU, N ranging from 5 to 50 (from Figure 3.16).

3.5.4 Results on the 2D matrix multiplication with randomized task order

The results on the 2D matrix multiplication with randomized task order, shown on Figure 3.17, have an interesting impact on the results previously discussed from Figure 3.8a.

EAGER and RCM’s results Due to the task order randomization, EAGER and RCM cannot take advantage of the natural order of submission of tasks, and their performance is greatly affected. Indeed for EAGER, once A or B cannot fit in memory, processing tasks in this setting results in a random order, which cannot bring data reuse as it is very unlikely that two tasks on the same row or column would be computed consecutively. Now that RCM cannot follow the order of submission, its ability to minimize bandwidth really contributes to its performance. Tasks are ordered to find at least one data share with their neighbors. Although it does not find affinity for a large number of consecutive tasks, it still brings enough improvements to beat EAGER.

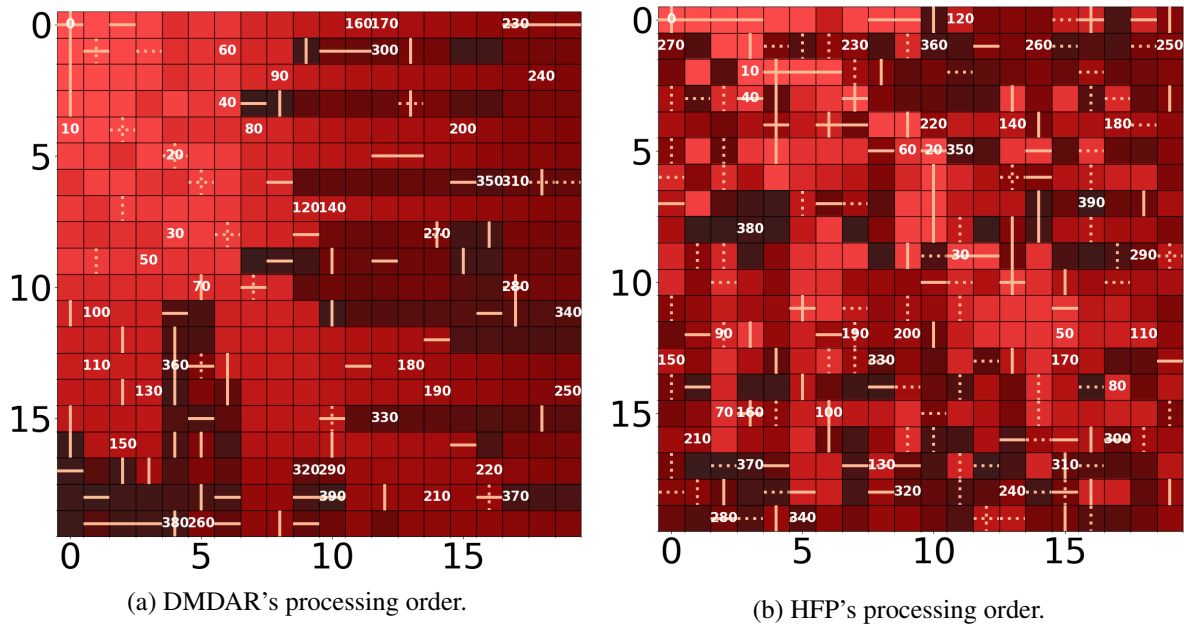


Figure 3.18: Visualization of the processing order (represented by the shading from lighter to darker as well as the numbering) on the random task order from 2D matrix multiplication. $N = 20$ and $M = 250 MB$, which corresponds to the 8th point of Figure 3.17. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

DMDAR's results The randomized task submission positively affects the performance of DMDAR. With the randomized submission order, DMDAR can no longer process tasks column by column, as it was the case in Section 3.5.1 and illustrated by Figure 3.11. Indeed with this random order, when DMDAR is planning the next task to process, it will naturally process blocks of C forming squares. It is very unlikely that random blocks from the same columns would be chosen. Moreover, once three blocks of C forming a square have been processed, DMDAR will necessarily find the fourth block, completing the square thanks to *Ready*, as it is looking for a data sharing the most data with what is loaded in memory. This is clearly visible on Figure 3.18a. The light red blocks form a perfect square representing the situation when both matrices fit in memory ($N = 8$ with $M = 250 MB$ which corresponds to the third point on Figure 3.17). For the rest of the visualization we can still distinguish rectangles of tasks computed one after another. It therefore completely avoids the LRU's pathological case mentioned earlier. Thus the data transfers of DMDAR are much smoother with the random submission order, which improves its performance compared to Figure 3.8a.

HFP's results For the first seven points, HFP obtains poorer performance than DMDAR, which was not the case on Figure 3.8a. As explained above, DMDAR is better with the random submission order. However, HFP suffers from an issue in this situation, which is illustrated by the visualization on Figure 3.18b. For the first 100 tasks (corresponding to the very light red squares), we observe that the tasks share the same rows and columns of data. However, if we want to merge this package with another one, the data sharing will be much more limited than in the classic case of the 2D matrix, where we could merge packages using strictly the same rows or columns. Thus, the randomization forces us to

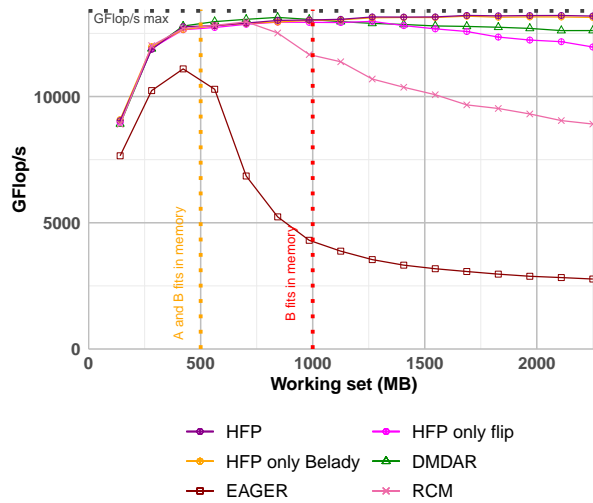


Figure 3.19: Performance on the randomized pairs with 2D inputs in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80. Memory limited to 500 MB.

form and merge packages corresponding to sparse matrices, which limits data reuse. This explains our performance on the first few points.

However, HFP is still able to form squares blocks of C that allows to compute a large number of tasks with a limited number of transfers. So even if we observe a large number of transfers on the visualization of Figure 3.18b, these are overlapped with the computation. Indeed there are a lot of transfers in beige dotted line on the visualization, meaning they are done in prefetch. This explains the good performance of HFP for the last points. In addition, we observe on Figure 3.17 that HFP without Belady does not achieve optimal GFlop/s. It is therefore crucial to use Belady, which allows us to avoid evicting, and thus having to load again a data that will be used again in the future. Indeed, it is even more important to manage eviction when the submission order is random because we no longer rely on the submission order. Likewise, we can observe that HFP with all the improvements obtains better performance compared to HFP with only Belady. This shows that the combination of HFP’s improvements is effective. The average improvement of HFP over the other heuristics can be seen on Table 3.5.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	143.5 %	109.4 %	50.1 %	1.2%

Table 3.5: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the 2D matrix multiplication with randomized task order in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80 (from Figure 3.17).

3.5.5 Results on the randomized pairs with 2D inputs

The experiments on randomized pairs obtained from the 2D matrix multiplication, shown on Figure 3.19, lead to results very similar to the previous experiment. The performance of EAGER decreases once a matrix no longer fits in memory. Processing tasks along the rows of C is equivalent to a random processing order with a randomized matrix operation. RCM gets more sustained performance: the

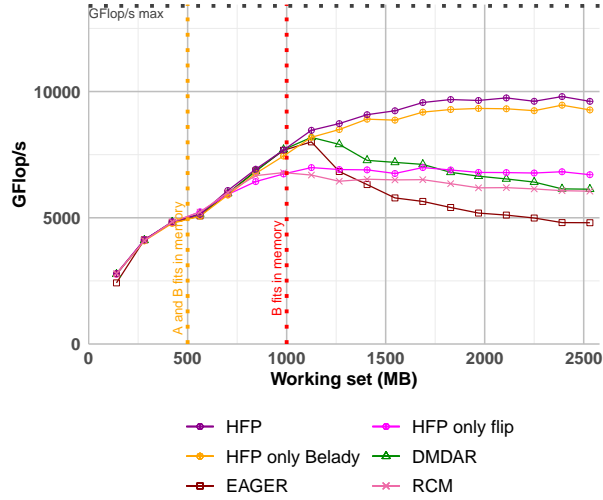


Figure 3.20: Performance on the sparse 2D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90. Memory limited to 500 MB.

randomization of data dependencies actually decreases the effect of the classical LRU pathological case, since it does not tend to execute tasks rows by rows any more. For the same reasons as explained in Section 3.5.4, DMDAR gets performance close to ideal. HFP outperforms DMDAR from the eighth working set size. The randomization of data dependencies does not allow HFP to form perfect squares of tasks. Thus it can only beat DMDAR when the working set is very restrictive compared to the GPU’s memory. The percentages of improvement of HFP over the other heuristics averaged on the sixteen points is shown on Table 3.6.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	142.5 %	66.4 %	18.2 %	1.6%

Table 3.6: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the randomized pairs with 2D inputs in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80 (from Figure 3.19).

3.5.6 Results on the sparse 2D matrix multiplication

We can observe on Figure 3.20 the performance obtained on a sparse matrix multiplication with only 10% of the tasks of the corresponding dense multiplication.

EAGER, RCM and DMDAR’s results Like in the previous experiments, following the submission order does not carry enough locality to deal with the memory limitation. Both RCM and DMDAR manage to find affinities on the rows or columns, but the sparsity increases the amount of data transfers compared to the classical 2D matrix multiplication.

HFP’s results The way HFP groups packages does not depend on the ratio between the number of tasks and the number of different data. Even in this more data-intensive case, HFP is able to form clusters of data-sharing tasks. As we can see with *HFP only flip* in Figure 3.20, the package flipping

optimization has no impact in this situation because we cannot form optimal orders in a sparse set of tasks. However, the packing heuristic coupled with Belady’s eviction policy produces a good amount of data reuse, which explains why a good performance is kept even when memory is a constraint. The percentage of improvements of HFP are given on Table 3.7.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	65.9 %	55.6 %	46.6 %	41.6 %

Table 3.7: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the sparse 2D matrix multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90 (from Figure 3.20).

3.6 Conclusion on static scheduling for a single processing unit

Considering data movement is crucial to get the best performance out of a GPU. We focused in this chapter on the ordering of tasks sharing some of their input data on a single GPU with limited memory. We adapted two heuristics from the literature (MST and RCM). We proved that a task order and its reverse produce the same number of data load. We designed a new algorithm gathering tasks with similar input data into packages of increasing size, called HFP and demonstrated its computational complexity. All three ordering strategies have been implemented using the STARPU runtime and tested together with a baseline and a state-of-the-art scheduler for various sets of tasks on a Tesla V100 GPU.

On 2D and 3D matrix products, the HFP algorithm outperforms its competitors: it is able to group together tasks of C forming rectangular blocks, as well as nicely ordering tasks among those blocks. Hence, it both reduces the data transfers and allows a smooth overlap of communications with computations. In the case of the 3D matrix product, the Belady eviction policy further helps to improve the performance. On more heterogeneous applications such as tasks from the Cholesky factorization, HFP with all its optimizations (package flipping, *Ready* and Belady’s eviction policy) is also able to reduce the amount of transfers and always reaches the best performance. On the randomized variants of the 2D matrix multiplication, HFP still manages to group tasks sharing data in order to obtain peak performance. Finally, on the sparse application, HFP is also able to form packages of tasks, increasing data reuse and thus leading to the best performance once the memory is a constraint. The Belady rule reduces drastically the number of data transfers. Without this rule, HFP may entail much more data transfers than other heuristics, but achieves better performance, which shows that HFP is also good at distributing data transfer over time to increase transfer/computations overlap. This experimental evaluation allowed us to show that HFP is generic with regard to the memory size: it can perform both with and without memory limitations. The DMDAR scheduler, currently the best choice among available scheduling policies in STARPU, obtains good performance for 2D matrix multiplication but is unable to cope both with very large number of tasks and with more intricate data sharing patterns such as the ones from 3D matrix multiplication. With the randomized variants, DMDAR avoids LRU’s pathological case and get more sustained performance.

Our final objective is to build a generic scheduler that can deal with multiple processing units and any task-based applications. HFP is very well suited to be extended to multiple GPUs as will be demonstrated in the next chapter.

However, to work on any task-based application, a scheduler must be dynamic, since there are task sets with dependencies. HFP could become dynamic in two ways, but both have significant drawbacks.

The first way would be to schedule the set of *readyTasks* and reschedule each time a new task becomes available. This would mean that the schedule would be computed multiple times. We have already observed in this chapter that the complexity of HFP is a problem for achieving good performance unless the scheduling overhead is ignored, which is the case with dynamic schedulers. Computing the schedule multiple times would introduce even more scheduling overhead and negate the benefits of HFP's strategy. The second way is to reschedule after an arbitrary amount of time or after a certain number of dependencies have been released. This would lead to poor scheduling decisions, and finding such arbitrary values is not guaranteed to yield good results on any task-based application.

These two limitations motivate the creation of a new dynamic scheduler that does not rely on a huge complexity to solve the scheduling problem. We introduce such a new dynamic strategy in the next chapter.

Chapter 4

Harnessing the Power of Multiple GPUs

Contents

4.1 State-of-the-art schedulers	74
4.1.1 Leveraging expected communication time with DMDAR	74
4.1.2 Using (hyper-)graph partitioning	76
4.2 Hierarchical Fair Packing adaptation to multiple processing units (mHFP) . . .	77
4.2.1 Strategy	77
4.2.2 Additional unused solutions explored for mHFP	78
4.3 A dynamic data-aware scheduler: DARTS	81
4.3.1 Intuition	81
4.3.2 STARPU's task flow with DARTS	81
4.3.3 Strategy	81
4.3.4 Eviction policy	83
4.3.5 Dealing with more input data per task	84
4.3.6 Reducing the scheduling overhead	84
4.3.7 Faster code with fewer mutex	84
4.4 Experimental evaluation with multiple processing units	86
4.4.1 Settings	86
4.4.2 Results on the 2D matrix multiplication with a single GPU	86
4.4.3 Results on the 2D matrix multiplication with multiple GPUs	87
4.4.4 Result on the 2D matrix multiplication with randomized task order and 2 GPUs	93
4.4.5 Result on the 3D matrix multiplication with 4 GPUs	96
4.4.6 Result on the task set of the Cholesky factorization with 4 GPUs	96
4.4.7 Results on the sparse 2D matrix multiplication with 4 GPUs	97
4.5 Conclusion on scheduling for multiple processing units	97



PC APPLICATIONS are often executed on nodes containing multiple processing units, be they CPUs or GPUs. This is particularly relevant for task-based systems, as one can control the granularity with which the application should be partitioned to fine-tune the distribution of tasks across workers. In the previous chapter, we learned how to reduce data transfers on a single processing unit: through temporal locality (i.e., processing tasks with common inputs consecutively). In this chapter, we tackle the issue of maintaining good temporal locality, but on multiple processing units. This is stated in Problem 3 as the BI-OBJ-MULTI-PU-TASK-SCHEDULING problem, introduced in the model from Section 2.3. This model brings a new challenge, since one must both *partition* the tasks (and consequently the data associated with it) and *order* the tasks for each processing unit.

In this chapter, we present several algorithmic solutions to the partitioning and scheduling problem on a set of *independent tasks*. Some of these methods solve the partitioning problem first, and then schedule the tasks on each processing unit while others tackle both simultaneously.

- We first present strategies from the state-of-the-art in Section 4.1.
- In Section 4.2 we present an extension to HFP to manage multiple processing units.
- In Section 4.3 we describe a new dynamic scheduler.
- The performance achieved by the different strategies is presented in Section 4.4.

4.1 State-of-the-art schedulers

DMDAR and EAGER are reused from the previous chapter. The EAGER strategy does not differ from the one presented in Section 3.1.1. However, DMDAR offers more advantages with multiple processing units (or PUs).

4.1.1 Leveraging expected communication time with DMDAR

DMDAR is a dynamic scheduler presented in the single processing unit case in Section 3.1.2. DMDAR computes the expected completion time $C_k(T_i)$ of the first task T_i in the queue on each PU_k , based on a prediction of the time for transferring the data to the PU (or communication time) $comm$ and of the task computation time $comp$. With a single processing unit, this only processes task in the submission order and the *Ready* reordering allows to prioritize task whose input data are already (or partially) loaded onto the processing units memory. With multiple processing units, DMDAR can now take advantage of two key benefits:

1. $comm_k(D_j)$ is the expected duration of transferring D_j to PU_k . But with multiple processing units, this transfer can be achieved in various ways. To understand how STARPU predicts the transfer time of D_j to PU_k we need to look at the topology of the node we use for our experimental evaluations: the Gemini node. Indeed, it is the topology of the node (Figure 4.1) that dictates the speed at which data can be transferred. Note that all bandwidth values used in the following paragraph have been measured experimentally with actual data transfers, they may not reflect the theoretical bus bandwidths. There are three different paths that data can take in order to be loaded onto a GPU:
 - (a) Gemini consists of 8 GPUs (the numbered cores). They are coupled two by two through a PCI switch (the black rectangles closest to the GPUs) and then four by four through a second PCI switch. The PCI Express buses (black lines) connecting them to the switches

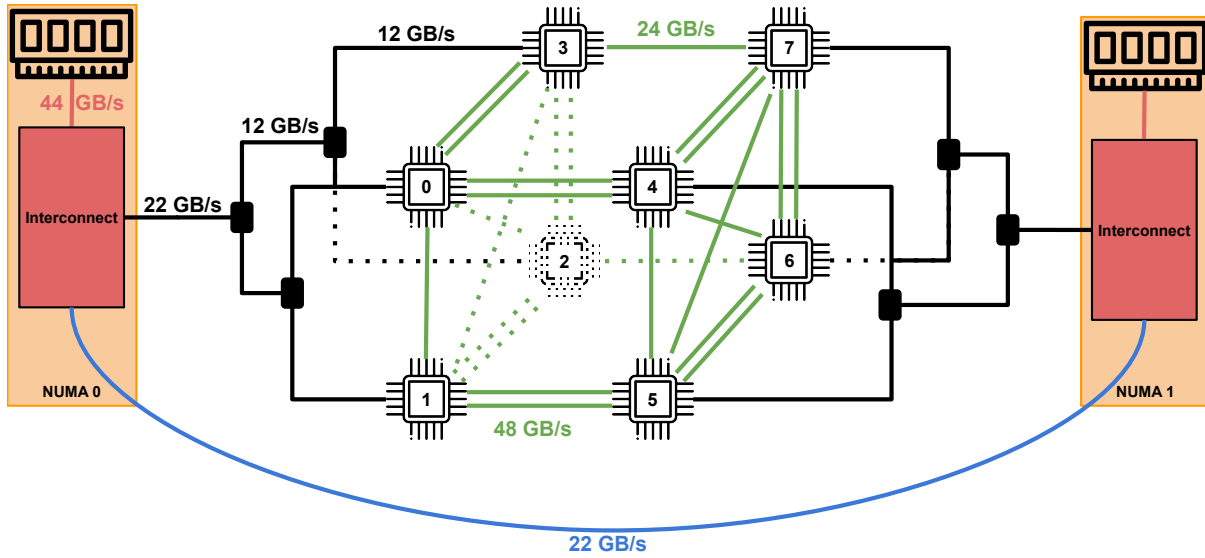


Figure 4.1: Topology of the Gemini node. Dotted and solid lines of the same color represent the same cables. The dots are used for perspective purposes only.

has a bandwidth of 12 GB/s. There are two NUMA nodes (orange rectangles in the figure), each connected to 4 GPUs via a 22 GB/s PCI Express bus using the outer PCI switch. The NUMA nodes consist of a CPU interconnect system that connects to the RAM at a bandwidth of 44 GB/s (red lines). The first method to load data for a GPU is to pass through the PCI switches and reach the RAM of the NUMA node. A first contention for bandwidth happens on the inner PCI switches as two 12 GB/s buses connect the outer switch with a 12 GB/s bandwidth as well.

- (b) NVlinks are short-range communication links between two GPUs or CPUs. In our case, using the Gemini node, they connect GPUs with a bandwidth of 24 GB/s (single green lines). An NVLink can be thought of as a cable, and a GPU can have multiple NVLink slots. Multiple cables can be linked together to increase the bandwidth when connecting the same GPUs (double green lines are two NVLinks connected to the same two GPUs, doubling the bandwidth). If there is no NVLink available to connect two GPUs directly, data can be routed between the GPUs. So, the second way to get data to a GPU is to get it from another GPU. This is the fastest way to get access to data, but it requires another GPU to have a valid copy of the data, which is not always the case.
- (c) If the data is not available on a GPU or would be too slow to transfer, the GPUs must rely on the RAM of its NUMA node. If the needed data is not on the GPU's associated NUMA node, it must travel through a link between the two NUMA nodes using a 22 GB/s interconnect link (blue line in Figure 4.1). This is the third and slowest way to access data. Note, however, that the bandwidth between the interconnect and the RAM allows for simultaneous transfers from both the PCI switch and the other NUMA node.

Considering the transfer duration with $comm_k(D_j)$ and multiple PUs allows DMDAR to favor the fastest path between a GPU and a data. This is especially powerful with NVLinks. In other words, DMDAR is able to exploit *spatial locality*.

2. $comp_k(T_i)$ is the expected completion time of T_i on PU_k . On a node with multiple PUs, it is not surprising to see disparities in computation speed due to temperature, power supply, or uneven wear. If load balancing is not required, DMDAR can choose the fastest processing unit. This is especially helpful for small task sets or toward the end of an execution: with few tasks remaining, the benefit of choosing the fastest processing unit is much more visible.

The equation with multiple resources is:

$$C_k(T_i) = \sum_{\substack{D_j \in \mathcal{D}(T_i) \\ D_j \notin InMem(k)}} comm_k(D_j) + comp_k(T_i) \quad (4.1)$$

From Algorithm 1, we highlight the modifications to select the processing unit that can complete each T_i task the fastest in red:

Algorithm 6 Deque Model Data Aware heuristic (DMDA)

- 1: **For each** PU_k , $InMem(PU_k) \leftarrow \emptyset$
 - 2: **while** all tasks have not been allocated **do**
 - 3: $T_i \leftarrow pop(\mathbb{T})$
 - 4: **For each** PU_k , **compute** $C_k(T_i)$ **using** Eq. 4.1
 - 5: **Select** k **such that** $C_k(T_i)$ **is minimal**
 - 6: **Allocate** T_i **to** PU_k
 - 7: **for each** $D_j \in \mathcal{D}(T_i)$ **do**
 - 8: Request data prefetch for D_j **in** PU_k
 - 9: Add D_j to $InMem(PU_k)$
-

4.1.2 Using (hyper-)graph partitioning

A graph partitioner is a very suitable tool to decompose the set of tasks sharing input data into several subsets of similar size while minimizing the number of common data among subsets. It has been extensively outlined by Yoo et al. [128]. Each subset is then allocated to a PU, and tasks within the subset are scheduled to further increase data locality. The fact that subsets have similar sizes ensures the load balancing among PUs, while the minimization of common data reduces the amount of data that must be sent to several PUs.

Yoo et al. [128] model data reuse through a graph whose vertices are tasks, and edges between two tasks are weighted by the amount of shared input data between these two tasks. Then, they use the METIS [84] graph partitioner to get a balanced partition of tasks while maximizing the number of data shares (edges) within a subset. A limitation of this method appears when data is shared by 3 (or more) tasks. Consider for example that some input data D_i is required by tasks T_a , T_b and T_c . In the modeled graph, this shared data leads to three edges (T_a, T_b) , (T_a, T_c) and (T_b, T_c) , leading METIS to count three times its weight. It is thus more reasonable to use a hypergraph to model data reuse, as proposed in [86]. A hyperedge is created for each data D which includes all the vertices corresponding to tasks using D as input. In the previous example, D_i is modeled with a single hyperedge $\{T_a, T_b, T_c\}$. We can then use a hypergraph partitioner, namely hMETIS [85] to split the tasks into subsets of similar size with few hyperedges between subsets, that is, few shared data. We mostly used default parameters when calling hMETIS [85], except for the default imbalance between partition (UBfactor), which is set to 1 in order to have almost perfectly balanced partitions, and V-cycle, set to 2, as it is advised by the authors if time

is an issue (which is the case). The number of bisections (Nruns) is set to 20 as it is advised by the authors.

Also in [128], the authors propose a technique based on constructing a minimum spanning tree on the task graph to schedule tasks within the different subsets. It was shown in Chapter 3 (and notably Figures 3.8a, 3.13a and 3.16) that this technique does not adapt well to the problem of scheduling tasks for a PU. We thus do not apply MST re-ordering here. Instead, to ensure good temporal data locality within each subset of tasks assigned to a single PU, we decided to use the *Ready* strategy of DMDAR (see Algorithm 2). Additionally, even if the partition produced by METIS is well-balanced in terms of number of tasks, data transfers make some PUs process tasks faster than others, leading to load imbalance. Thereby, we also implement dynamic load balancing using task stealing: when a PU has terminated its allocated tasks and some other PU still has work to do, the idle PU steals half of the remaining tasks from the PU with the most unprocessed tasks (starting at the tail of the list). We call this hMETIS+R, detailed in Algorithm 7.

Algorithm 7 hMETIS heuristic with ready (hMETIS+R)

- 1: For $j = 1, \dots, m$, $h_j \leftarrow \{T_i, \text{ s.t. } D_j \in \mathcal{D}(T_i)\}$
 - 2: Build hypergraph $H = (\{T_1, \dots, T_n\}, \{h_1, \dots, h_m\})$
 - 3: Apply hMETIS on H to produce a task partition P_1, \dots, P_K
 - 4: Allocate tasks of P_k on PU_k
 - 5: If at some point PU_k has no more tasks to process, steal half of the remaining tasks from the most loaded PU and allocate them to PU_k
 - 6: Reorder tasks using Ready at runtime
-

Adaptation to heterogeneous task durations To handle heterogeneous task durations, hMETIS+R requires two modifications. First, when building the hypergraph at Line 2 of Algorithm 7, it needs to add weights to the vertices by using the expected task durations. With this modification, hMETIS is then able to produce task partitions (Line 3) that have balanced expected completion times. Second, Line 5 needs to be modified to balance the total expected completion time of a partition instead of the number of tasks of a partition: PU_k steals a subset of tasks equal to half of the expected completion time of the most loaded PU.

4.2 Hierarchical Fair Packing adaptation to multiple processing units (mHFP)

We adapt here the HFP algorithm presented in Section 3.3. As a reminder, it consists in gathering tasks sharing many common input data into packages of maximum size the memory bound M . Packages are merged that way as long as they do not exceed M . In a second step, resulting packages are merged again in order to bind together packages with high data affinity, so they can be scheduled one after the other.

4.2.1 Strategy

We adapted HFP for the multi-PU case as follows. When scheduling tasks for K PUs, we merge packages until we reach K of them. It is unlikely that all these packages represent the same load (computed as the number of tasks if tasks have the same duration, or else the total duration of tasks for heterogeneous tasks). To achieve load-balancing, we first compute the average load L_{avg} of a PU. Unlike DMDAR,

HFP cannot compute the computation time of a task on a particular GPU as this is done before assigning the task to it. To avoid computing the duration of a task on each GPU, we use the performance model of the first GPU with $comp_0(T_i)$ to compute the expected completion time of each task and derive L_{avg} . The GPUs used in our work are homogeneous. Due to the negligible speed differences between these GPUs, our load balancing approach is not significantly affected by considering only the performance model of the first GPU. It is worth noting that in a heterogeneous environment we would adopt a different strategy: consider the average completion time on each processing unit. We then move the last tasks of the package P_{max} with highest load to the package P_{min} with smallest load in order to balance the load without exceeding L_{avg} . This process is repeated until the load is L_{avg} on all PUs. The additional tasks are placed at the end of a package, as we noticed that there is usually more slack for communication near the end of the computation.

The static process above is unable to provide a completely accurate load-balancing, as it is hard to predict the duration of communications on a shared bus as well as their overlap with computations. Thereby, we also implement for HFP the dynamic load balancing strategy using task-stealing introduced for hMETIS+R (see Section 4.1.2). Similarly, to adapt to heterogeneous weights, we would use the same technique as described for hMETIS+R.

Finally, it uses the *Ready* reordering strategy from DMDAR to favor tasks with better data availability. The resulting strategy is called mHFP (for multi-PU extension of HFP) and described in Algorithm 8.

Algorithm 8 multi-PU Hierarchical Fair Packing heuristic

- 1: Use HFP (Algorithm 5) to create K packages P_1, \dots, P_k
 - 2: $TotalLoad \leftarrow comp_0(T_i)$ for $i = 1 \dots m$
 - 3: $L_{avg} \leftarrow TotalLoad/K$
 - 4: **while** There exists P_i with $|P_i| > L_{avg}$ **do**
 - 5: Let P_{max} be the largest package
 - 6: Let P_{min} be the smallest package
 - 7: Remove $\min(|P_{max}| - L_{avg}, L_{avg} - |P_{min}|)$ tasks from the tail of P_{max} and append them to P_{min}
 - 8: Allocate tasks of P_k on PU_k
 - 9: If at some point PU_k has no more tasks to process, steal half of the remaining tasks from the most loaded PU and allocate them to PU_k
 - 10: Reorder tasks using Ready at runtime
-

4.2.2 Additional unused solutions explored for mHFP

We present here other interesting solutions that have been explored to adapt HFP to the multi-PU case. We explain how they could be useful and why we decided not to apply them.

mHFP² With mHFP, we merge packages until we reach K of them. The first unused solution was to apply HFP to each package K just before assigning them to the PUs. It consists of adding the line: *Use HFP on each P_k* after Line 7 of Algorithm 8 and outside of the *while loop*. We call this mHFP². Each P_k is already a subpackage of HFP's ordering of \mathbb{T} . As we demonstrated in the previous chapter, the transition from one package to another also brings locality thanks to the *package flipping* strategy introduced in Section 3.3.5. Some of these data reuse patterns might have been lost when assigning a package to each processing unit. The intuition with mHFP² is to form a task order that is coherent with

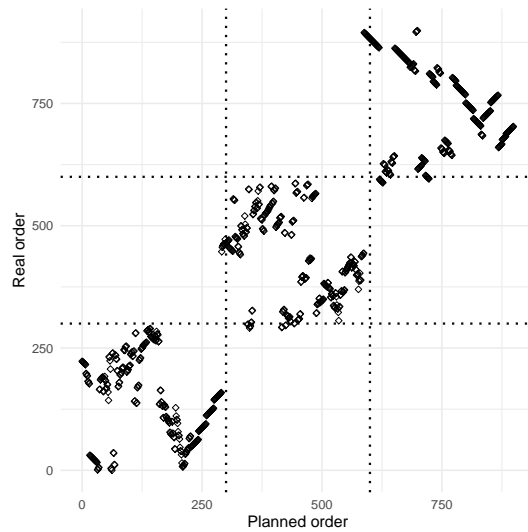


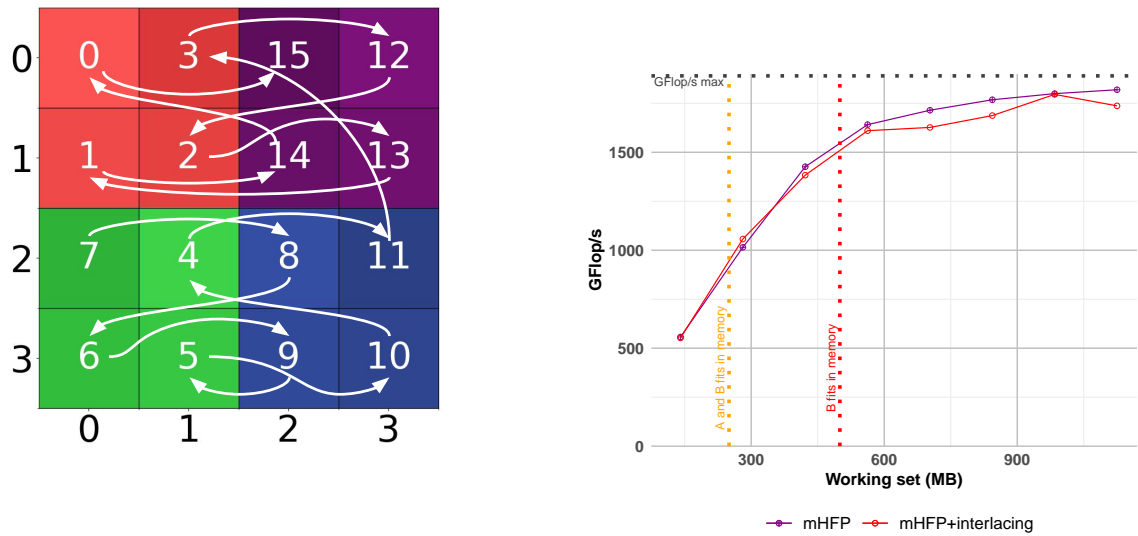
Figure 4.2: Difference between mHFP (planned order) and mHFP² on the 2D matrix multiplication with 3 GPUs. Each square is a task, and its position on the X or Y axis indicates its processing order. Deviation from linear progression means that planned and actual orders differ. The dotted lines delimit the 3 GPUs.

respect to the tasks in each P_k , and to maintain good data reuse between the tasks of P_k . It is therefore reasonable to reapply HFP to each of the K packages.

After experimental evaluations on the 2D and 3D matrix multiplication, we observed a small (about 5%) performance penalty when using mHFP². We can see the difference between the processing order of mHFP and mHFP² on Figure 4.2. For each GPU, all tasks have been reordered with mHFP². We can still find on the figure, lines of tasks that most likely share data and therefore, mHFP² kept them in a consecutive order but executed them earlier or later compared to mHFP. However, many tasks are isolated. mHFP²'s second use of HFP is based on a subset of HFP's complete scheduling. So, it can suffer from an edge effect: some data reuse would have been unlocked on the next merge and does not appear in each subpackage P_k . Ordering a small set of tasks is harder than ordering a full task set, which can lead to such isolated tasks. In addition, we saw in a previous plot (Figure 3.8a) that the scheduling time of HFP is important. Applying HFP again to each subpackage would again increase the overall scheduling overhead. For these reasons, we do not use mHFP².

However, such a strategy would be interesting on a larger scale: executing a very large set of tasks on multiple nodes. Indeed, with multiple nodes each containing multiple PUs, one could use mHFP² to first divide the application into as many packages as there are nodes, and then use HFP again on each node to divide the subpackages into as many packages as there are processing units. The scheduling overhead would be less important because with larger applications and more PUs, it is less significant in the overall computation time. Also, the schedule would still be good after applying HFP a second time because a large number of tasks would diminish the edge effect mentioned earlier.

Interlacing The second unused idea was to use interlacing after reaching K packages. The intuition behind this is that when packages are formed, most data reuse occurs in the middle of the packages, not at the extremities. In fact, subpackages at the extremities suffer from having only one adjacent package to share common data with. However, at the beginning of the execution, when the memory is empty, the loads cannot be overlapped by the computations, so we need to reuse the data as much as possible



(a) HFP's processing order on a 2D matrix multiplication. The numbers is the order produced by HFP. The white arrows indicate the HFP+interlacing order, starting at tile 7.

(b) Performance on the 2D matrix multiplication of HFP with and without interlacing in simulation. Memory limited to 500 MB.

Figure 4.3: Using interlacing with HFP.

at the start of the schedule. So, the idea is to cut each package K into two subpackages and interlace the tasks from the two subpackages, starting with the end of the first subpackage and the beginning of the second subpackage. This way, we first alternate with tasks sharing data (the middle of the original package), and end with tasks sharing little data (the extremities of the original package). It puts the most data reuse at the beginning of execution, when we need it most. We call this mHFP+interlacing.

An example is shown in Figure 4.3a. The numbers are HFP's order and the white arrows show the reordering with interlacing. Starting with tile 7, the order is then 8, 6, 9, 5, 10, 4, 11, 3, 12 etc. An experiment with interlacing is shown in Figure 4.3b: the red curve is mHFP using interlacing while the magenta curve is mHFP. The non-interlaced variant achieves better performance than mHFP+interlacing. This is because we were missing a key element: if data reuse is important at the start of the execution, it is even more important when evictions begin to occur. In a very memory-constrained situation, it makes sense to first load a lot of data at the beginning of the execution, and then to favor data reuse with the data already loaded, as mHFP does. This way, only a minimal amount of data needs to be evicted. mHFP+interlacing reduces the load at the beginning of execution, then struggles with loading and eviction towards the end of it. If memory is limited enough compared to the PU memory, eviction and the resulting data reloads overcome the benefits of better overlap at the beginning of execution. We can clearly see this effect in Figure 4.3b: the interlacing strategy is as good as or better than mHFP on the first two points, before the memory constraint, and fall behind only after the two input matrices cannot fit in memory and evictions become necessary. Thus, increasing locality at the beginning of the schedule, when memory is empty, will not bring any improvement under memory constraint. For this reason, we are not going to use this method.

However, the idea is interesting for situations where access to RAM is contentious, for example when multiple GPUs want to access RAM and the sum of the bandwidths between the GPUs and the PCI switch is greater than the bandwidth between the switch and the RAM. Such an example can

be seen in Figure 4.1. Interlacing could be used to smooth the data load over time. After applying mHFP+interlacing and starting to execute the tasks, we observe the bandwidth usage. If we observe that the bandwidth is saturated, processing units should pop tasks from the beginning of their attributed package. The beginning is the side that increases data reuse, which will reduce their need to transfer data. When the bandwidth is no longer constrained, processing units pop tasks from the end of their package and consequently switch to the ordering with low data reuse. In doing so, transfers could be more spread out over time, resulting in a less contentious access to the PCI buses and more overlap of communications and computations.

4.3 A dynamic data-aware scheduler: DARTS

This section is dedicated to introducing our new scheduler: DARTS, which stands for Data-Aware Reactive Task Scheduling.

4.3.1 Intuition

The previous strategies that partition the tasks in order to increase data locality, namely hMETIS+R and mHFP, are static algorithms: they require a preliminary phase where the partition of tasks is computed, and their complexity might be prohibitive for large numbers of tasks or data. We propose here a dynamic strategy, called DARTS (for Data-Aware Reactive Task Scheduling), adapted from previous algorithms specifically designed for linear algebra operations, such as outer products and matrix products [25]. The main idea of these algorithms is to perform as many tasks as possible with the data at hand. When a new data is loaded on a processing unit memory, we allocate to this processing unit all the tasks that depend on the new data and on data previously loaded on this processing unit. New data are chosen at random to make sure different processing units have little chance to compete for the same tasks.

4.3.2 STARPU's task flow with DARTS

In order to grasp DARTS' strategy, one must understand the flow of tasks within the STARPU runtime system when associated with our scheduler. STARPU's task flow was presented in Figure 2.6.3 of Chapter 2. Figure 4.4 is a simplified version of that figure with the addition of DARTS. The blue elements are specific to the DARTS scheduler. As with any scheduler, when a PU_k is idling, it will pull the head of the $taskBuffer_k$ queue of tasks. With DARTS, if that is empty, it will pull tasks from $plannedTasks_k$. More precisely, it will pull as many tasks as it can fit into $taskBuffer_k$. $taskBuffer_k$ has a limited size of 30 tasks (in order to pipeline a reasonable amount of prefetches), while $plannedTasks_k$ is not limited. If $taskBuffer_k$ is also empty, DARTS will be called to fill $plannedTasks_k$. DARTS cannot access tasks in $taskBuffer_k$ (as those are in the STARPU common core), however, tasks in $plannedTasks_k$ can still be removed by DARTS. This additional layer of $plannedTasks_k$ allows to maintain a schedule in accordance with the evictions, as we will see in Section 4.3.4. DARTS uses a custom eviction policy to determine what data should be removed from a PU's memory as needed.

4.3.3 Strategy

The main idea of our new scheduler, detailed in Algorithm 9, is to first consider data movement before task allocation. Whenever some PU_k requests some new task, we first look in the set $dataNotInMem_k$ (which initially contains all data) for the data D_{opt} that, if moved into the memory of PU_k , would maximize the number of new "free" tasks, i.e., tasks that can be allocated and processed on PU_k without

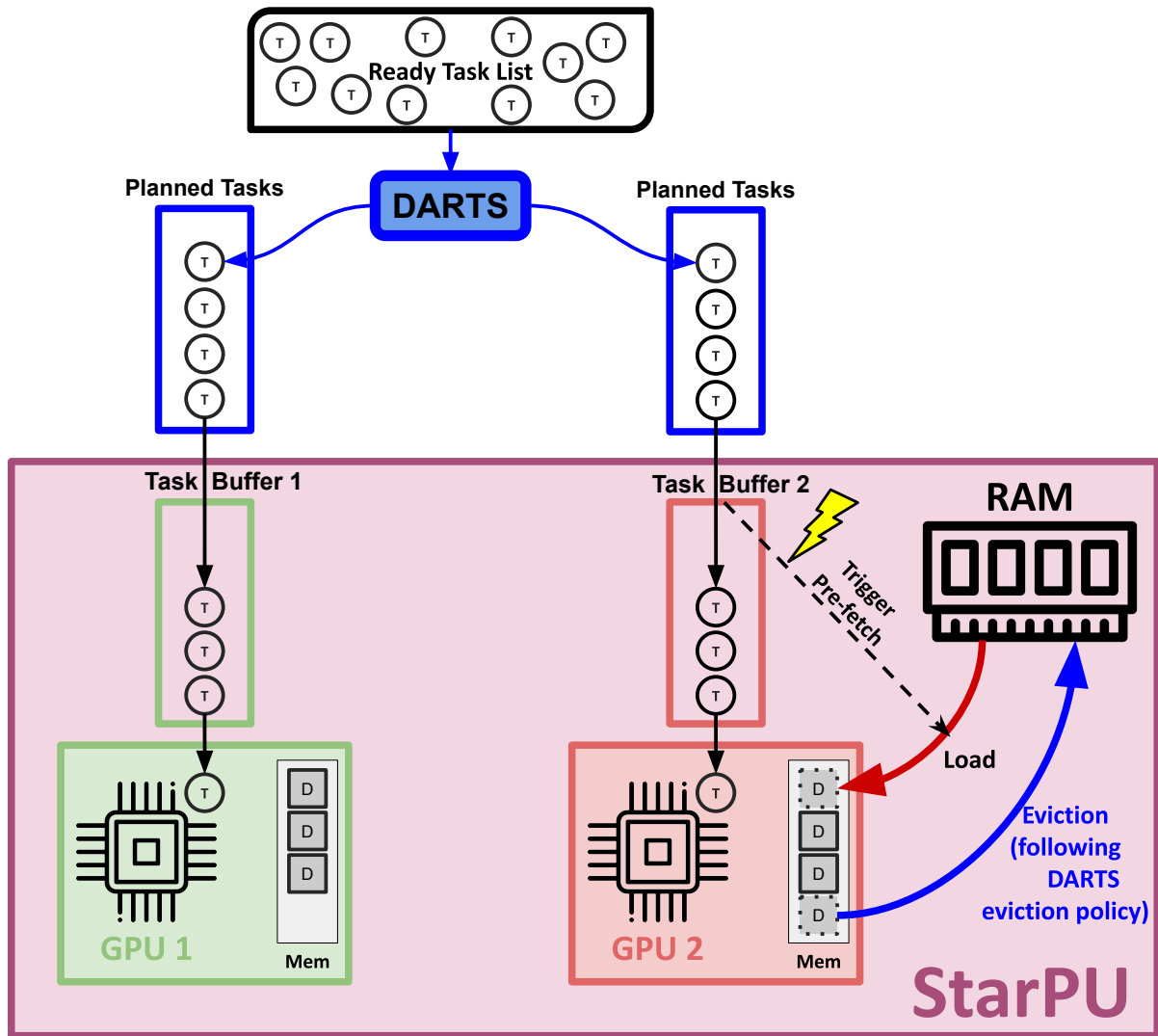


Figure 4.4: Simplified task flow within the STARPU runtime when using DARTS. The pink boxes are the common STARPU core. The blue elements are the actions of DARTS. A solid arrow is a task movement and a dashed one is a notification. A large red or blue arrow indicates a data movement.

any additional data movement. Once such a data is found, all these free tasks are allocated on PU_k . More precisely, they are put into $plannedTasks_k$. The process is started again when $plannedTasks_k$ is empty. It may happen that we do not find any data that enables some free tasks. It occurs for example at the very beginning of the computation when all tasks depend on two or more data: at least two data must be loaded in order to produce some free task. In this case, some random unprocessed task is allocated to PU_k . On the contrary, when there exists several candidate data which may produce the maximum number of free tasks, we select a data among the candidates that is useful for the highest number of tasks (free or not). When a tie occurs (either in selecting a task or a data), we randomly pick some elements. This is important to make sure that different PUs have little chance to load the same data and compete for the same tasks.

We describe here the values required to select the optimal data D_{opt} (see Algorithm 9):

- $S_0(D)$: the set of tasks that depend only on D and some data already loaded in the memory of PU_k .
- $task_left(D)$: the sum of the durations of the tasks in $readyTasks$ that use D as an input.

Algorithm 9 DARTS scheduler on PU_k

```

When  $PU_k$  requests a new task
1: if  $plannedTasks_k = \emptyset$  then                                ▷ We need to fill  $plannedTasks_k$ 
2:   for each data  $D \in dataNotInMem_k$  do
3:     Compute  $S_0(D)$  and  $task\_left(D)$ 
4:   Choose the data  $D_{opt}$  with the highest  $|S_0(D)|$ , tiebreak with  $task\_left(D)$  and then randomly
5:   if  $|S_0(D_{opt})| > 0$  then
6:     Append  $S_0(D_{opt})$  to  $plannedTasks_k$ 
7:     Remove  $D_{opt}$  from  $dataNotInMem_k$ 
8:   else
9:     Select a random unprocessed task  $T$ 
10:    Append  $T$  to  $plannedTasks_k$ 
11:    Remove the inputs of  $T$  from  $dataNotInMem_k$ 
12: Return head of  $plannedTasks_k$ 

```

Note that DARTS is not explicitly adapted to heterogeneous weights, but this is not a problem. Because, unlike hMETIS+R and mHFP, DARTS is a dynamic scheduler: it only assigns work to PUs when they are idle. As a result, load balancing is automatic, even with heterogeneous task durations.

4.3.4 Eviction policy

In order to improve the performance of our dynamic scheduler, we also designed a custom eviction policy: since we plan ahead which tasks will be allocated to a PU (through the use of $plannedTasks$), we can take this information into account when we have to remove some data from the memory. This strategy is named Least Used in the Future (LUF) and is detailed in Algorithm 10. We consider all data currently in memory and check if they are used as input for a future task. There are two types of such future tasks: tasks in $plannedTasks_k$ that have been reserved for a later allocation on the PU as mentioned above, but also tasks in $taskBuffer_k$, which are the tasks that have been popped from $plannedTasks_k$ for execution on PU_k , and thus whose PU placement cannot be changed any more (the required data for these tasks may already have been prefetched). We first try to evict a data which is

not useful for any task in $taskBuffer_k$, and which is an input of few tasks in $plannedTasks_k$. If this is not possible, we apply Belady’s rule [26] on tasks already allocated: we select the input data whose next usage in $taskBuffer_k$ is the furthest in the future, which is known to minimize data movement. In practice, this last rule is rarely used as we usually succeed finding a data not useful for any task in $taskBuffer_k$.

Algorithm 10 Eviction procedure LUF for DARTS on PU_k

- 1: **for each** data D in the memory of PU_k **do**
 - 2: $n_b(D) \leftarrow$ number of tasks using D in $taskBuffer_k$
 - 3: $n_p(D) \leftarrow$ number of tasks using D in $plannedTasks_k$
 - 4: **if** the minimum value of $n_b(D)$ on any data D is 0 **then**
 - 5: Select V such that $n_b(V) = 0$ and $n_p(V)$ is minimum
 - 6: **else**
 - 7: Select V the data whose next use in $taskBuffer_k$ is the furthest in the future
 - 8: Remove tasks depending on V from $plannedTasks_k$
 - 9: Evict V from memory, push it to $dataNotInMem_k$
-

4.3.5 Dealing with more input data per task

We introduce here the *3inputs* variant of DARTS (Algorithm 11). The interest of this variant comes into play when no data allows to compute a task without additional loads, i.e., in the “else” case on Line 8 of Algorithm 9. Instead of loading a random data, we first look for a data which enables as many tasks as possible to be processed with a single additional data load. Namely, we look for a data D such that the number of tasks depending on D , on another unloaded data D' and on some data already in memory, is maximal. If we find such data D , we return a random selection of such task T , otherwise we return a random task.

We introduce the $S_1(D)$ notation:

- $S_1(D)$: the set of tasks that depend only on D , some data already loaded in the memory of PU_k , and 1 additional data.

4.3.6 Reducing the scheduling overhead

With a large number of data available at once, as it might be the case with an independent task set, the complexity of finding D_{opt} can induce a sizable scheduling overhead. We thus enhance DARTS with the additional *OPTI* strategy to reduce scheduling time: instead of looking for the data that enables the most tasks, we stop the search as soon as we find a data allowing to compute at least one task. In order to do this, we add, in Algorithm 9, after Line 3 and inside the *for each* loop: *If* ($|S_0(D)| > 0$) $\{Break\}$

4.3.7 Faster code with fewer mutex

The main optimization of the DARTS implementation is a *refined mutex policy*. Mutual exclusion (or mutex) protects shared resources from being accessed by concurrent threads during overlapping time intervals. As mentioned above, the main complexity of DARTS comes from finding D_{opt} . Once D_{opt} is found, the tasks are moved from the *readyTasks* list to the *plannedTasks* list. This needs to be protected by a mutex. However, to find D_{opt} , the scheduler does not need to access a shared resource, apart from

Algorithm 11 DARTS 3inputs variants on PU_k

When PU_k requests a new task

- 1: **if** $plannedTasks_k = \emptyset$ **then** ▷ We need to fill $plannedTasks_k$
- 2: **for each** data $D \in dataNotInMem_k$ **do**
- 3: Compute $S_0(D)$ and $task_left(D)$
- 4: Choose the data D_{opt} with the highest $|S_0(D)|$ tiebreak with $task_left(D)$ then randomly
- 5: **if** $|S_0(D_{opt})| > 0$ **then**
- 6: Append $S_0(D_{opt})$ to $plannedTasks_k$
- 7: Remove D_{opt} from $dataNotInMem_k$
- 8: **else**
- 9: **for each** data $D \in dataNotInMem_k$ **do**
- 10: Compute $S_1(D)$
- 11: **if** $|S_1(D_{opt})| > 0$ **then**
- 12: Task $T \leftarrow$ head of $S_1(D_{opt})$
- 13: Append T to $plannedTasks_k$
- 14: Remove the inputs of T from $dataNotInMem_k$
- 15: **else**
- 16: Select a random unprocessed task T
- 17: Append T to $plannedTasks_k$
- 18: Remove the inputs of T from $dataNotInMem_k$
- 19: **Return** head of $plannedTasks_k$

reading $readyTasks$. Therefore, we introduce a refined mutex policy that lets concurrent PUs select D_{opt} during overlapping time intervals. The mutex is then used to protect task movement from $readyTasks$ to $plannedTasks$.

With our refined mutex policy, multiple PUs may select the same data D_{opt} during the same time interval. We call this a *conflict*. We use the following example to explain how we handle a conflict: PU_1 selects data D_1 between time t_1 and t_3 and PU_2 selects data D_1 between time t_2 and t_3 . PU_1 found D_1 earlier. We let it fill $plannedTasks_1$ with the tasks using D_1 and other data in PU_1 's memory. PU_2 can then fill $plannedTasks_2$ with the tasks using D_1 and its own data in memory. The first downside of such a refined mutex policy is that a conflict may lower the scheduling quality on PU_2 , since some of the tasks it planned to add to $plannedTasks_2$ have been picked by PU_1 .

More importantly, in some cases, PU_2 might not be able to add any tasks to $plannedTasks_2$ because they would all have been taken by PU_1 . We call this a *critical conflict*. In such a case, PU_2 must restart the process of finding D_{opt} . If such critical conflicts occur frequently, the overhead of restarting the search for D_{opt} could lead to a large reduction in performance. We observe on Figure 4.5a the performance obtained with (orange line) and without (blue line) our refined mutex policy and ensure that the number of conflicts is not significant with Figure 4.5b. We observe a large performance improvement with the refined mutex policy. It is not accompanied by a large number of conflicts, as with 57 600 tasks ($N = 240$) and 11 iterations, only 3 conflicts occurred. And only 2 were critical over the 12 matrix sizes tested. For this reason, DARTS uses the refined mutex policy for the remainder of this thesis.

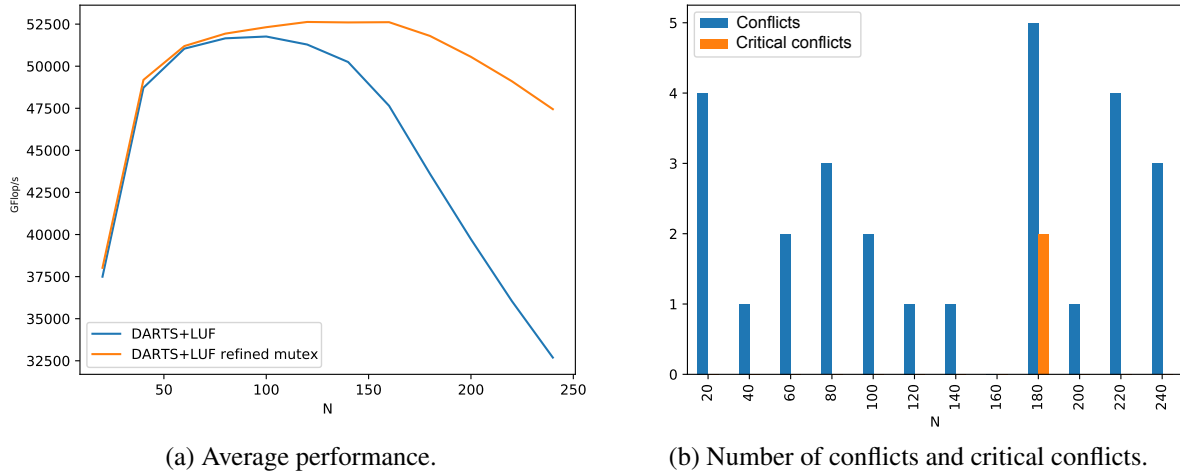


Figure 4.5: Results of DARTS mutex policies on the 2D matrix multiplication with 4 Tesla V100 GPUs over 11 iterations while varying the side N of the input matrices A and B . Memory limited to 500 MB per GPU.

4.4 Experimental evaluation with multiple processing units

We present below the experimental evaluation conducted to compare the strategies presented above¹.

4.4.1 Settings

We use the same GPUs, settings, and set of applications as in Chapter 3 and presented in Section 3.4. The main difference is that we are using multiple GPUs of the Gemini node, connected as shown in Figure 4.1. We have most often limited the GPU memory to 500 MB, to better differentiate the performance of the schedulers on smaller datasets, but Section 4.4.7 also presents results with no memory limitation.

We plot the performance obtained with various problem sizes, the number of tasks ranging from 5×5 to 300×300 (which corresponds to working set sizes from 140 MB to 8 400 MB) for the 2D matrix and up to 50 000 MB for the 3D matrix, in order to test all strategies on various memory conditions.

4.4.2 Results on the 2D matrix multiplication with a single GPU

To validate the performance of our new DARTS scheduler, and to verify that we have not altered the HFP strategy with mHFP, we first test our strategies on a single GPU. Figure 4.6 shows the results obtained by the various algorithms with a single GPU.

EAGER and DMDAR results As explained in Section 3.5.1, with a single GPU, EAGER switches to pathological behavior at the red vertical line. We can both see the throughput plummeting (Figure 4.6a) and the data transfers increasing (Figure 4.6b) at the same working set size. Similar to the case in Section 3.5.1, DMDAR suffers from a conflict between data prefetching and data eviction, which prevents it from staying close to the asymptotic goal.

¹The code used to reproducibly obtain the results of this chapter is available at: <https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/IPDPS2021>

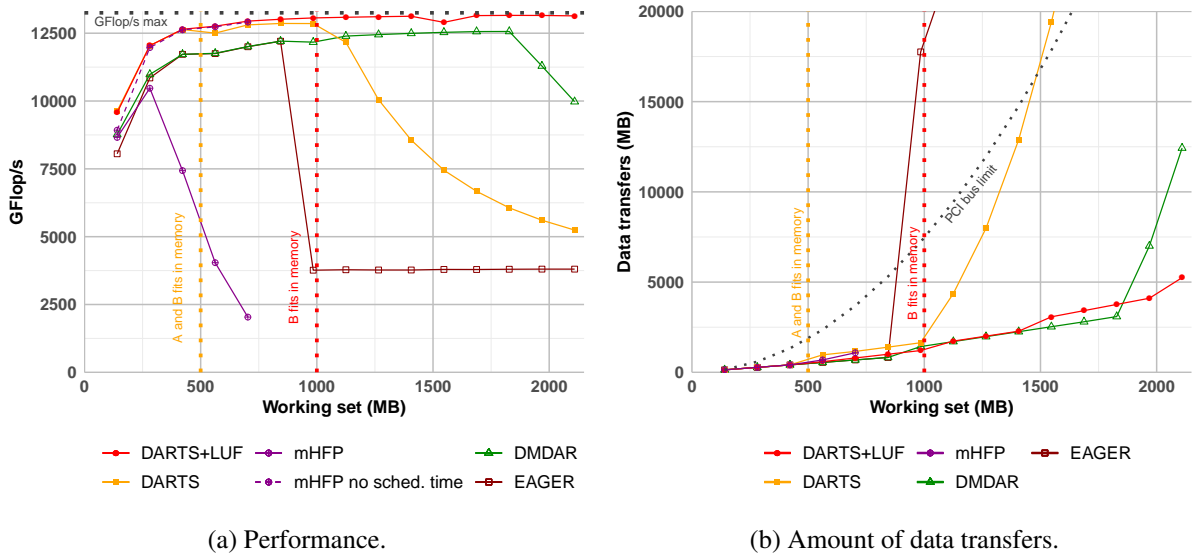


Figure 4.6: Results on the 2D matrix multiplication in real with 1 Tesla V100 GPU. Memory limited to 500 MB.

mHFP results We show two variants of mHFP for a few working set sizes on Figure 4.6a. The dashed line represents the performance when we ignore the scheduling time, that is, when excluding the first phase in which the static task mapping is computed. We notice that it achieves very good performance. The continuous line represents the performance obtained while taking into account the scheduling time (like we do for every other heuristics). Unfortunately, the scheduling time of mHFP is very long for large working sets (1 minute for a 1 300 MB working set) and rapidly grows. Thus, the overhead induced by the scheduling time overcomes its benefits.

DARTS results We can see on the first seven points of Figure 4.6a, that DARTS and DARTS+LUF achieve near perfect performance. Indeed, loading a single data, which enables multiple tasks to be computed without any additional data load, reduces data transfers and increases overlap between transfers and computations. However, after the red-dotted line, when the memory is constrained, DARTS has to load and evict data following LRU's policy. As a result, the tasks in *taskBuffer* that are supposed to be ready for computation, have to load more data. Indeed, the previous tasks caused evictions, and the data evicted might be needed by the tasks in *taskBuffer*. This causes a domino effect where each new task requires a new data load. On the contrary DARTS+LUF sustains on average 8.5% more GFlop/s than DMDAR. When an eviction is needed, it avoids as much as possible evicting a data that is used by the few tasks already planned for computation. This way DARTS+LUF avoids the pathological case of DARTS, and achieves a better balance between prefetching and eviction since DARTS+LUF maintains in *plannedTasks* and *taskBuffer* an accurate overview of the tasks to be computed, even when eviction removes some data from a GPU. In the end, it achieves near-optimal performance and the least amount of data transfers on the last two points.

4.4.3 Results on the 2D matrix multiplication with multiple GPUs

We now move to the multi-GPU case. Figure 4.7 shows the results obtained using simulation (for 2 GPUs), thus not taking the scheduling costs of all heuristics and the partitioning costs of hMETIS into

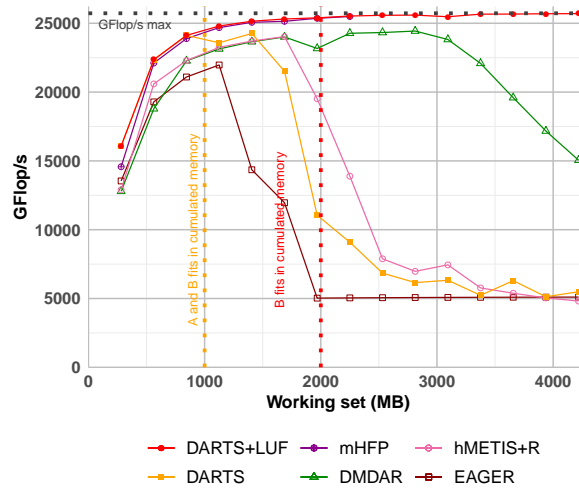


Figure 4.7: Performance on the 2D matrix multiplication in simulation with the performance models of 2 Tesla V100 GPUs.

account. Figures 4.8 and 4.13 show the results obtained with real executions (for 2 or 4 GPUs). On these last two figures we added two versions of hMETIS+R, one with partitioning time and one without (hMETIS+R no part. time) to show its impact on performance. Now, the vertical lines depict the thresholds when one or both input matrices fit in the *cumulated* memory, that is, can be distributed over the memory of all GPUs.

mHFP results with multiple GPUs in simulation As we can see in Figure 4.7, mHFP achieves near-perfect performance, almost hugging the maximum GFlop/s upper bound, when scheduling time is not taken into consideration. It shows that mHFP’s strategy of ordering, load balancing, and task stealing achieves good results in theory. To understand the reason behind these results, we visualize mHFP’s ordering with two GPUs on Figure 4.9. HFP’s order adapts nicely to the multi-PU case, as they are all assigned rectangles of tasks, which contains a lot of data reuse. For example, the red package starts on the tile with coordinates 8:8, which of course requires two fetches (indicated by the two lines inside the tile). Then the light red tiles around it (the 8 by 8 square starting at coordinates 8:8) are processed, requiring only a few data transfers thanks to package flipping. Going from this 8 by 8 subpackage to the next also requires no more transfers than necessary. The same phenomenon happens with the green package. In total, only 30 fetches are required here, whereas DMDAR requires 69 fetches in the same experiment, as we will see in Figure 4.11.

For mHFP, however, the scheduling time increases significantly with the working set size. For example, for a working set of 2000 MB, mHFP takes more than 8 minutes to schedule, while DARTS finishes in just 2 seconds. As seen in Figure 4.6a, mHFP’s results would be very poor if scheduling time were taken into account. Therefore, for the following graphs in this document, we do not show mHFP on the plots.

Nevertheless, mHFP would achieve near perfect performance if we consider that the scheduling is done before the application starts. We can see in Figure 4.10 that even with 8 GPUs, mHFP can create packages (each different color is a set of tasks assigned to a GPU) that have a lot of data reuse. Again, we find in the processing order these *U-shapes* generated by HFP’s package flipping and packing,

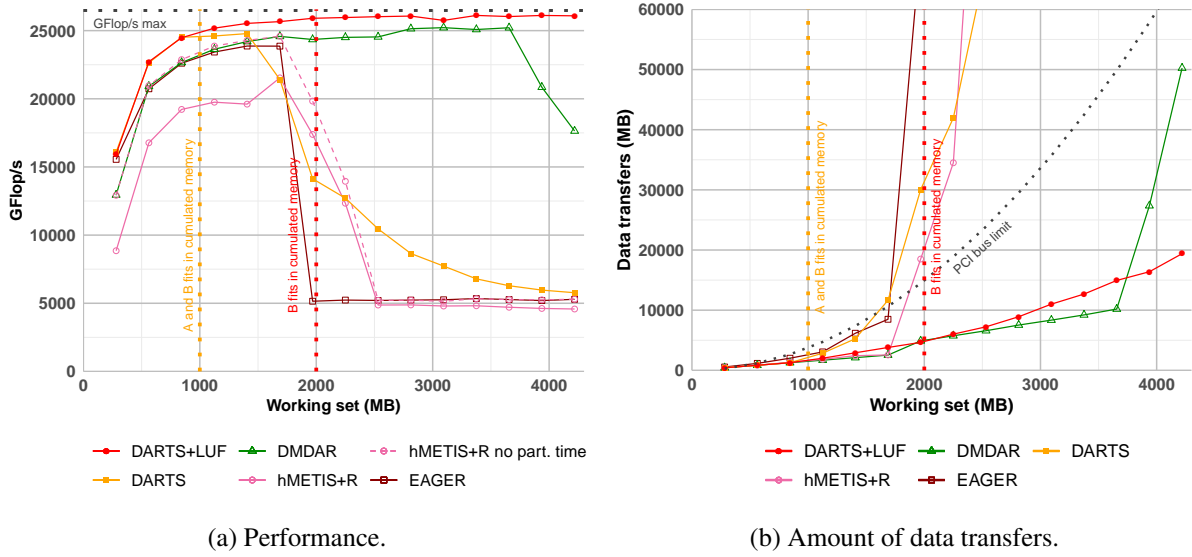


Figure 4.8: Results on the 2D matrix multiplication in real with 2 Tesla V100 GPUs. Memory limited to 500 MB per GPU.

that minimizes data transfers. If we look at the red package, we can find these *U-shapes* at any size granularity of a package.

To summarize, mHFP is an algorithm that achieves its goal of reducing transfers to achieve peak performance, be it on 1 or 8 GPUs. However, its scheduling overhead makes it unusable for online scheduling, but it would be a highly efficient offline scheduler.

EAGER, hMETIS+R and DARTS results Similarly to the single-GPU case, we observe in Figures 4.7 and 4.8a that EAGER, hMETIS+R and DARTS show lower performance under memory constraint. hMETIS+R gives a partitioning based on the data-sharing graph. In constrained situations, the lack of task ordering inside a partition, does not allow for good data reuse. The *Ready* heuristic can only reorder a limited number of tasks ahead of the computation and cannot improve performance by a significant margin. EAGER and DARTS both suffer from the same pathological case induced by the LRU strategy. By observing the two curves of hMETIS+R in Figure 4.8a, we notice that the partitioning time of hMETIS+R has a significant impact on performance, and that this impact increases with the number of GPUs.

DMDAR results The DMDAR results on two GPUs are also very similar to the single-GPU case. DMDAR achieves a good load balance between the two GPUs and favors locality, but at some point it cannot properly manage both prefetching and LRU eviction like in the single-GPU case as can be seen on the last two points of Figures 4.7 and 4.8a. However, DMDAR is also slightly slower than DARTS+LUF and mHFP before these last two points. Figure 4.11 shows DMDAR's behavior at the 8th point of these figures. We can see that most data transfers are not done with a prefetch. This is due to DMDAR's ordering, which still mainly orders tasks row by row. Going from one row to the next causes a burst of data loads that do not allow all the inputs to overlap with computations. We also notice that some rows are loaded multiple times, showing that DMDAR does not reuse each data optimally. This explains why the performance is slightly lower compared to DARTS+LUF in real and compared to mHFP in simulation.

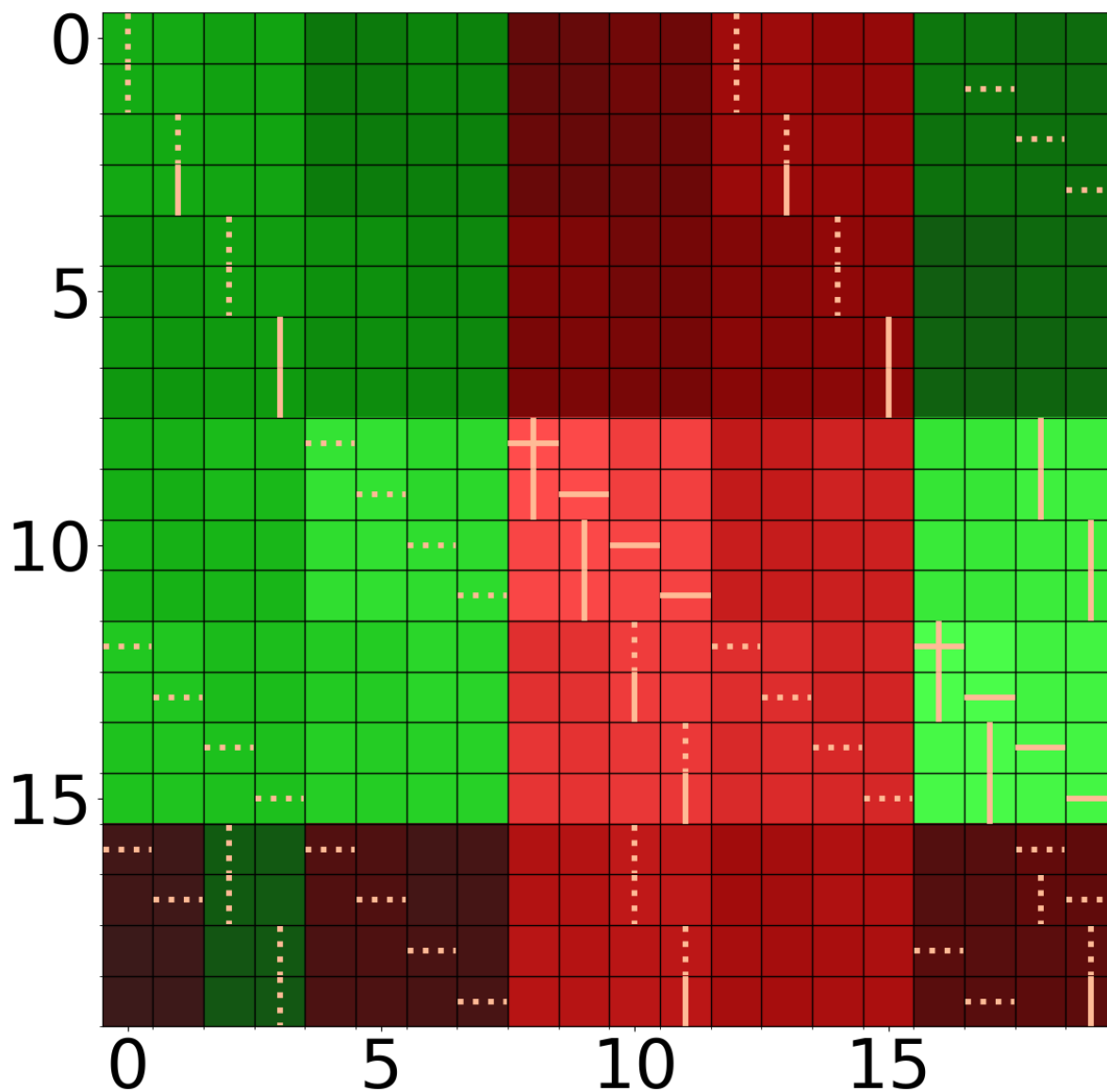


Figure 4.9: mHFP's ordering (represented by the shading from lighter to darker) on the 2D matrix multiplication with 2 Tesla V100 GPUs. $N = 20$ and the GPUs memory is limited to 250 MB, corresponding to the 8th point of Figure 4.7. Tiles in green are assigned to GPU₁ and those in red to GPU₂. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

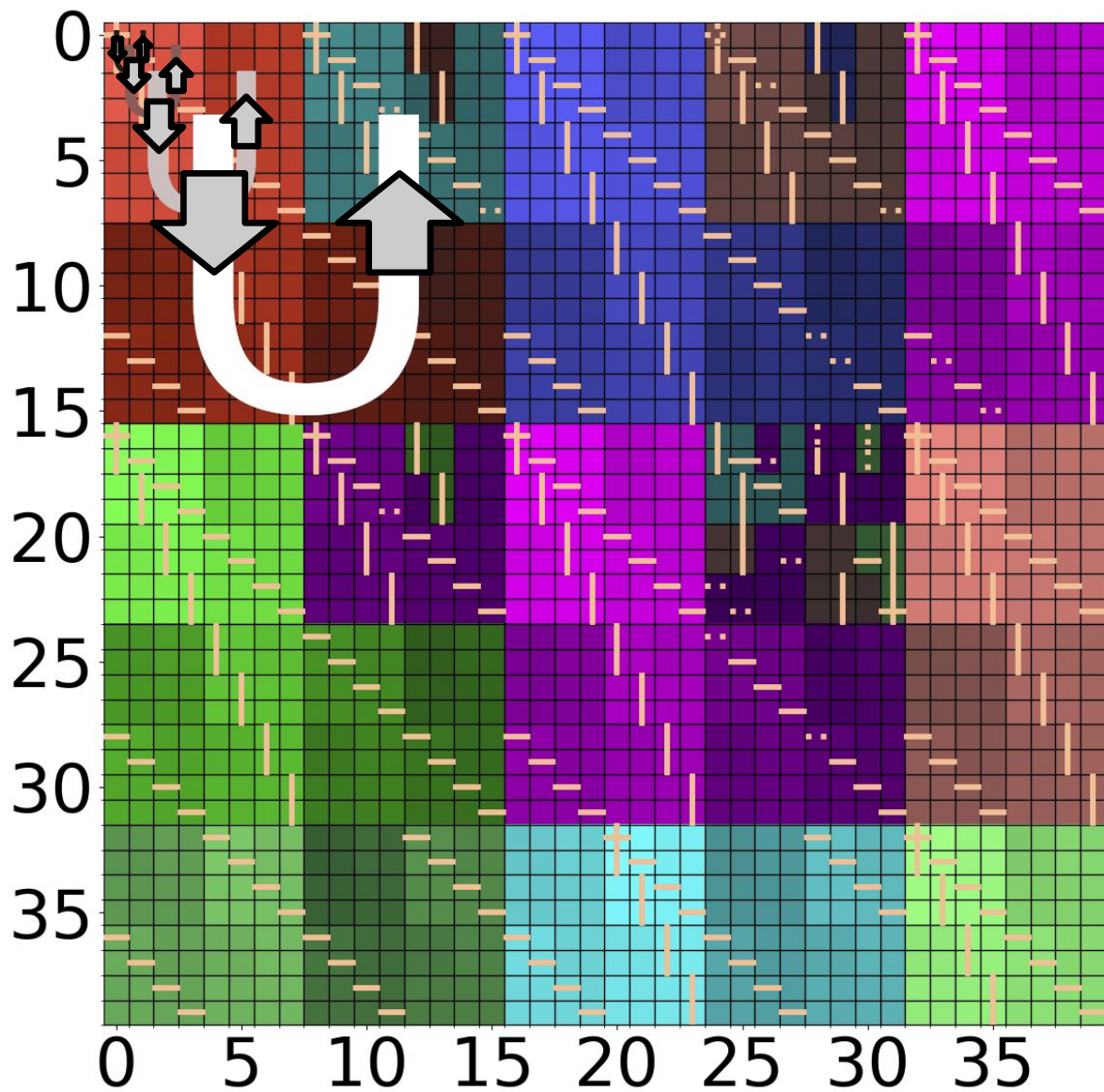


Figure 4.10: mHFP's ordering (represented by the shading from lighter to darker) on the 2D matrix multiplication with 8 Tesla V100 GPUs. $N = 40$ and the GPUs memory is limited to 250 MB. Each color is a set of tasks assigned to a GPU. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches. The different "U" and the associated arrows show how mHFP's processing order brings data reuse at different packages granularities.

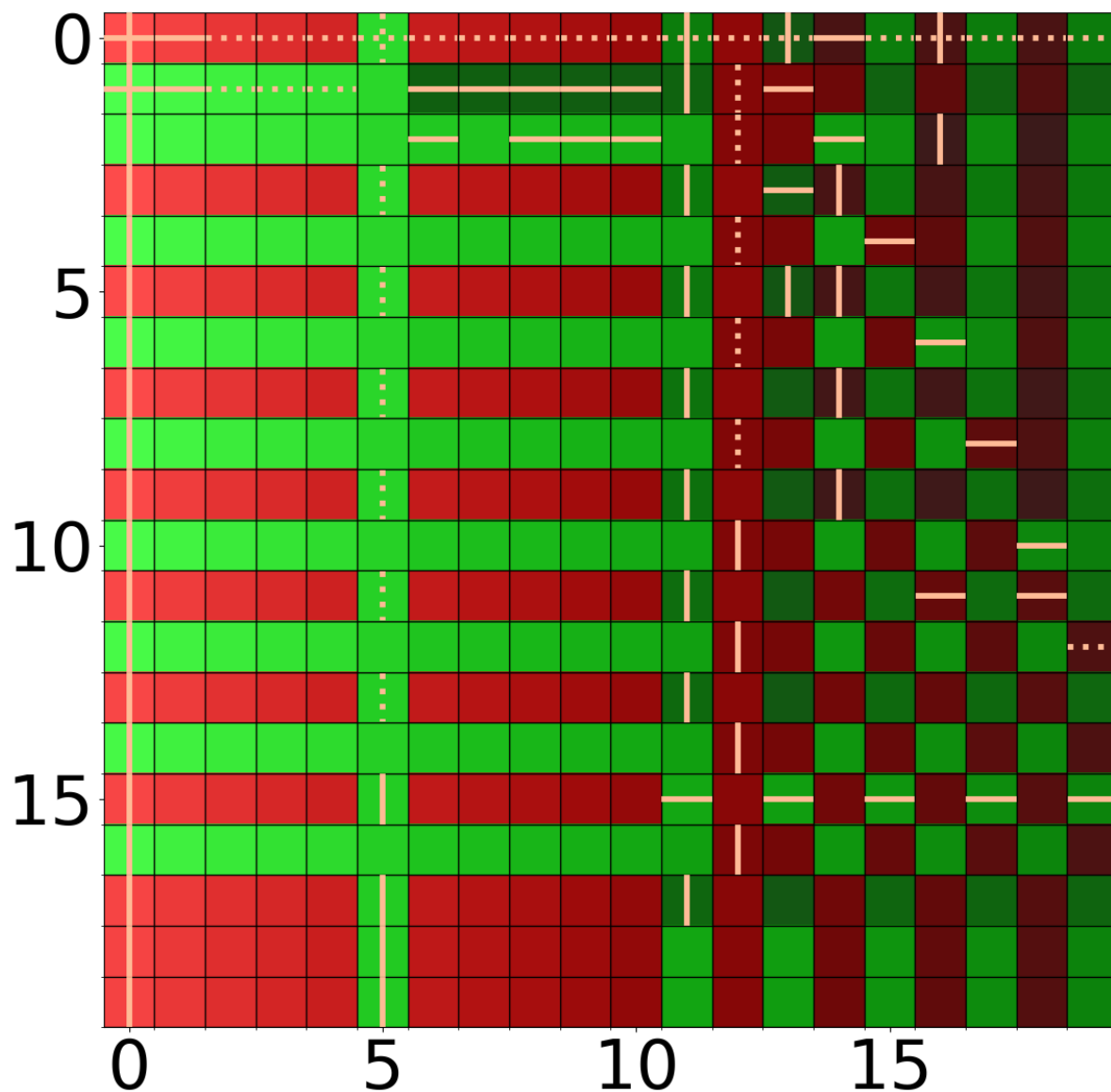


Figure 4.11: DMDAR's ordering (represented by the shading from lighter to darker) on the 2D matrix multiplication with 2 Tesla V100 GPUs. $N = 20$ and the GPUs memory is limited to 250 MB, corresponding to the 8th point of Figures 4.7 and 4.8a. Tiles in green are assigned to GPU₁ and those in red to GPU₂. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches. Although we observe that the GPUs compute tasks one out of every two rows, this does not mean that there is no data reuse from one row to another: by renumbering, we can create squares of tasks that more visually show data reuse.

DARTS+LUF results DARTS+LUF gets performance close to ideal, with a 9.4% improvement over DMDAR for two GPUs in real, while maintaining a very low complexity. In multi-GPU, DARTS assigns to each GPU its own set of data $dataNotInMem_k$ to pick from. However, all GPUs share the same set of tasks. When a GPU is assigned a task, it is removed from the common set of available tasks. Thus, our scheduler will naturally assign to the other GPUs data from a row or column that has not been used for tasks yet. This will evenly distribute tasks among GPUs and mostly separate data usage between GPUs. LUF’s eviction policy allows us to keep the expected data loading order by evicting data that will be used the least for future tasks. Thus, the scheduling can still be effective despite the memory constraint. It is also important to note that we observe in Figure 4.8b that DARTS+LUF has more data transfers than DMDAR between 2500 and 3500 MB. However, its throughput is always higher. We can understand this result if we take a look at the Figure 4.12. Most tiles that require a data load are able to use a prefetch. In fact, splitting data loads between several executions of tasks from $taskBuffer$ induces a better distribution of transfers. On the contrary, DMDAR tends to load a large number of data at once as we mentioned above. In the example of Figure 4.12, DARTS requires 55 fetches, which is more than mHFP (30 fetches) but less than DMDAR (69 fetches). Despite this, DARTS and mHFP results are the same in simulation. We can explain this again by looking at Figure 4.12. If we renumber the rows, we could form squares of green and red tiles, similar to the packing of mHFP. This shows that there is a lot of data reuse both from the rows of A and the column of B within each GPU. This is a great result for DARTS because it can achieve the same performance as an offline scheduler with a much smaller scheduling time. Thus, DARTS can achieve high throughput in both simulated and real experiments. Therefore, DARTS will thus be our main focus in the rest of the manuscript.

Trends with more GPUs Using 4 GPUs (as in Figure 4.13) mainly impacts the performance of DARTS. EAGER, hMETIS+R, DARTS, and DMDAR have similar results as on Figure 4.8a with 2 GPUs. The only difference is that the working set must be larger to observe the same results, indeed the sum of memory is 2x larger, so the working set is pushed 2x further. At around 4000 MB we observe that DARTS+LUF’s performance starts decreasing. As we use larger task sets, the scheduling time required to find the optimal data to load begins to degrade the global performance of the strategy. To reduce the impact of the scheduling overhead, we have added a threshold on the number of data we can pick from when filling $plannedTasks$ for working sets larger than 3500 MB only (in line 3 of Algorithm 9). This reduces the quality of the scheduling for these working set sizes, but lets DARTS partially compensate for the performance drop (as we can see on the plots with the red dashed line), and to surpass DMDAR on the last two points. However, it is difficult to come up with an optimal threshold that limits scheduling time without impacting too much the schedule quality.

4.4.4 Result on the 2D matrix multiplication with randomized task order and 2 GPUs

In order to test our heuristics in more irregular cases, we randomize the task submission order. It will also highlight the link between performance and tasks’ submission order. Figure 4.14 shows that EAGER, DMDAR and hMETIS+R are highly impacted by the randomized order of submission as soon as the memory cannot contain both input matrices. This shows that DMDAR actually relies on the tasks submission order to get good performance in previous graphs. Note that the performance is much worse than when we randomized the task order with a single GPU in Figure 3.17; with a single GPU, data reuse can happen more often even with a randomized set of tasks. With multiple GPUs, the randomization causes the GPUs to replicate data loads, greatly reducing performance. DARTS manages to maintain

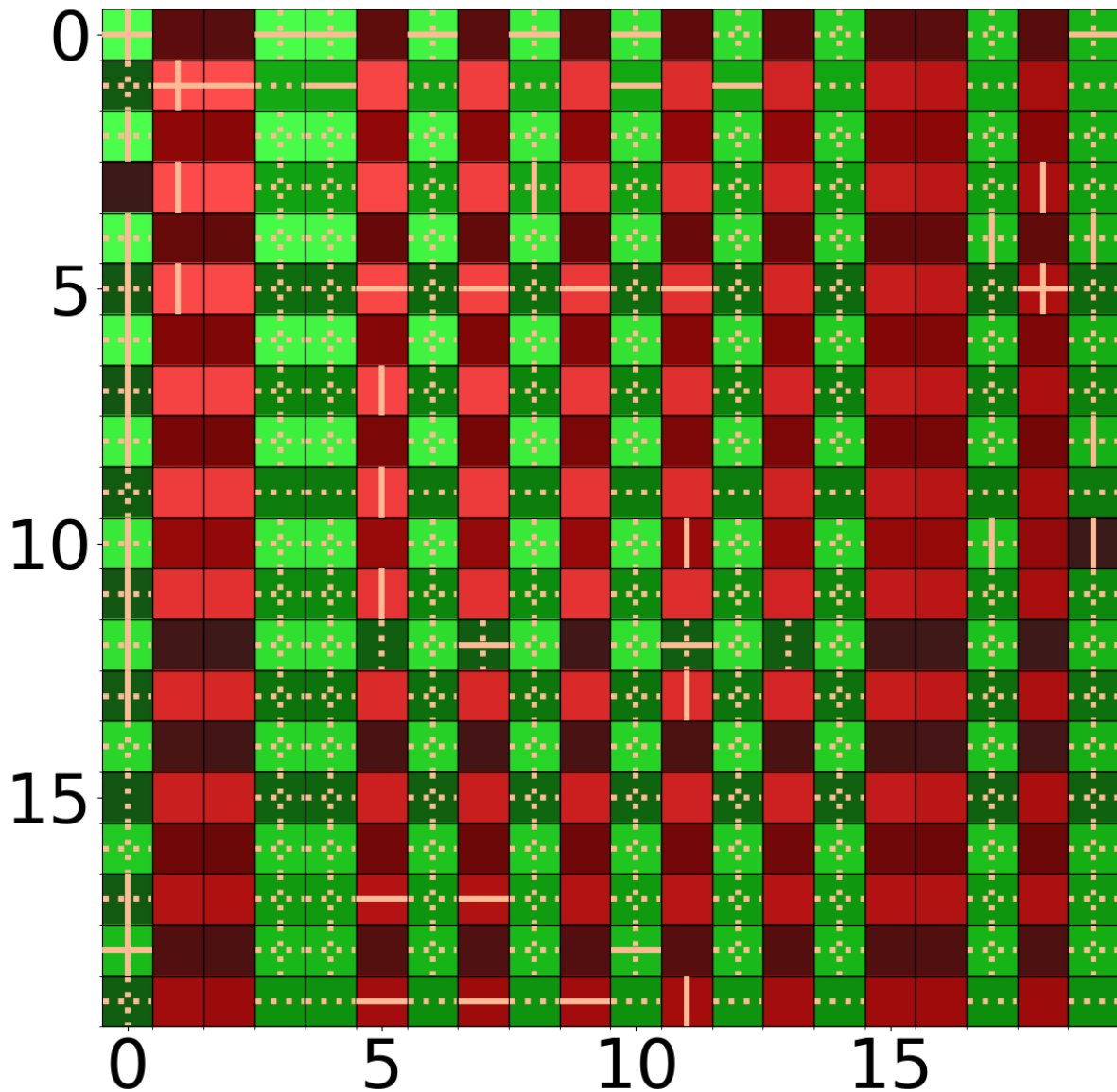


Figure 4.12: DARTS' ordering (represented by the shading from lighter to darker) on the 2D matrix multiplication with 2 Tesla V100 GPUs. $N = 20$ and the GPUs memory is limited to 250 MB, corresponding to the 8th point of Figures 4.7 and 4.8a. Tiles in green are assigned to GPU₁ and those in red to GPU₂. A beige vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. Solid lines are fetches while dotted lines are prefetches.

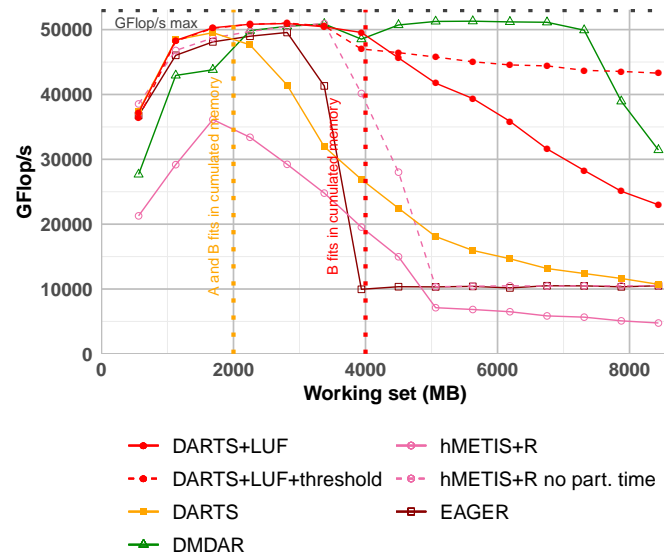


Figure 4.13: Performance on the 2D matrix multiplication in real with 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.

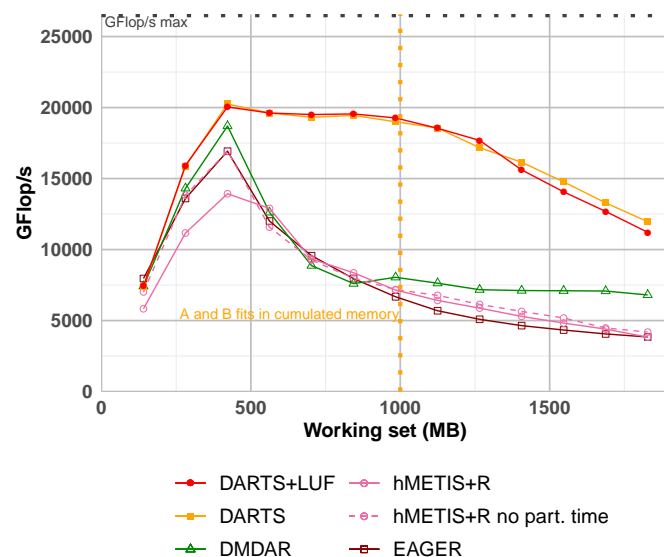


Figure 4.14: Performance on the 2D matrix multiplication with randomized task order in real with 2 Tesla V100 GPUs. Memory limited to 500 MB per GPU.

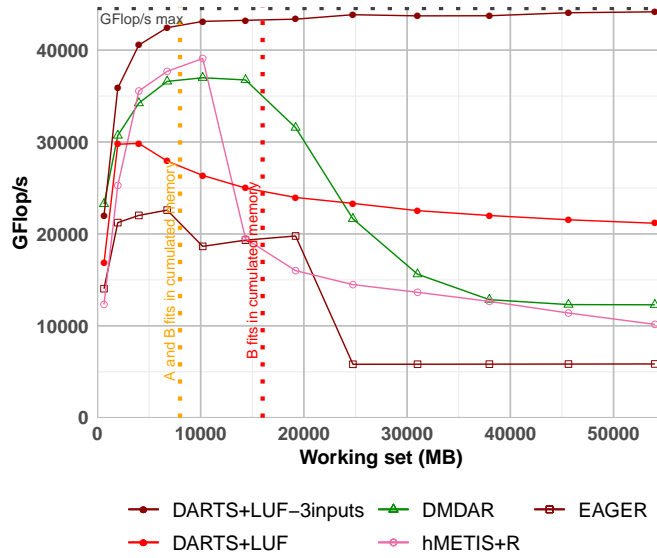


Figure 4.15: Performance on the 3D matrix multiplication in simulation with the performance models of 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.

high throughput until the memory size is less than an input matrix size. On this graph, DARTS+LUF averages 75% more GFlop/s than DMDAR on all points.

4.4.5 Result on the 3D matrix multiplication with 4 GPUs

Figure 4.15 shows the results on the 3D matrix multiplication in simulation with 4 GPUs. We add here the *3inputs* variant named "DARTS+LUF-3inputs" and presented in Section 4.3.5. We observe in Figure 4.15 that this variant leads to a better schedule. DARTS+LUF-3inputs reaches a throughput about 61% larger than the one of DMDAR. It is important to note that from the second working set size, DARTS+LUF-3inputs is better than its competitors, which shows that even without memory limitation, the order of processing of our variant allows for a better overlap of tasks and data transfers.

In real, with multiple GPUs, DARTS performs very poorly because of its scheduling time. Finding the optimal D_{opt} is too costly to get any results comparable to the other strategies.

4.4.6 Result on the task set of the Cholesky factorization with 4 GPUs

Figure 4.16 shows the results on the Cholesky factorization (without dependencies) with 4 GPUs in real. Here, the green vertical line marks the working set size where all of input data fit into memory. DARTS is unable to achieve good performance, even with the *3inputs* variant. This is explained by the huge number of tasks and the resulting scheduling time. The *OPTI* extension (see Section 4.3.6) allows to maintain performance close to the optimal up to a 3000 MB working set. DARTS with *OPTI* gets on average 49% more GFlop/s compared to hMETIS+R no part. time. Note that DMDAR also suffers from a large scheduling time induced by looking at all the tasks in order to choose the one allowing to avoid data loads. As a conclusion, DARTS can easily be extended to scenarios with more than 2 inputs per tasks, using the *3inputs* variants, as well as scenario with a very large number of tasks, using *OPTI*.

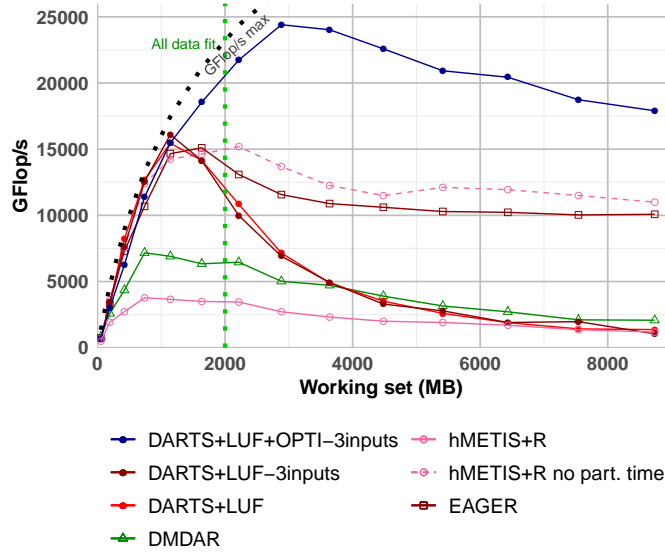


Figure 4.16: Performance on the task set of the Cholesky factorization in real with 4 Tesla V100 GPUs. Memory limited to 500 MB per GPU.

4.4.7 Results on the sparse 2D matrix multiplication with 4 GPUs

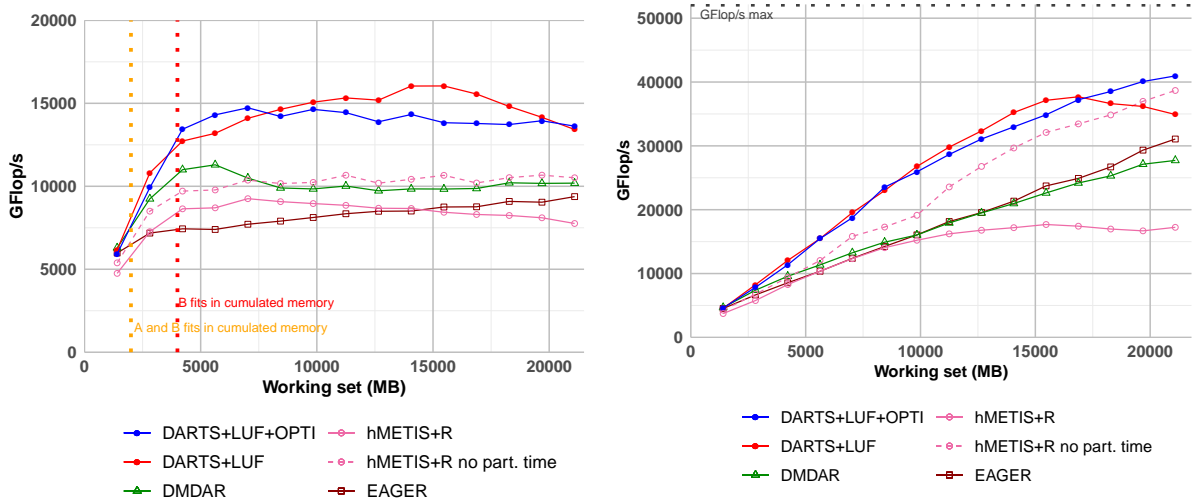
Figure 4.17 shows the results on the sparse 2D matrix multiplication scenario, in which much less tasks can be computed with the same number of data. On Figure 4.17a, DARTS manages to navigate between sparse tasks without generating too many transfers, which is not the case for other schedulers. On this application we observe that DARTS+LUF obtains 40% more GFlop/s than DMDAR. As the total number of tasks is smaller than before, the *OPTI* variant is not needed, but we also see that it does not negatively impact performance.

Figure 4.17b shows the same application but without memory limitation. In this case, DARTS+OPTI obtains the best performance. This shows the ability of DARTS to produce a processing order that best distributes transfers over time. We also note that hMETIS+R suffers from an important partitioning cost; this largely decreases its performance that would otherwise be only slightly lower than DARTS.

4.5 Conclusion on scheduling for multiple processing units

Maximizing the performance of multiple GPUs on a single node, all sharing a few communication buses with limited bandwidth, is challenging. In this chapter, we have proposed several alternatives for scheduling tasks that share input data on such multi-GPU platforms and implemented them in the STARPU runtime. After detailing the node architecture, we proposed to implement a strategy based on a graph partitioner and extended it with task stealing. We extended a previous strategy to manage multiple processing units and called it mHFP. We proposed a new dynamic strategy, DARTS, which considers data movement before task allocation. We extensively evaluated our proposed schedulers on applications with different data access patterns.

For the 2D matrix multiplication, with one or two GPUs, mHFP gives almost optimal results as an offline scheduler. As an online scheduler, its performance is not competitive because of its scheduling overhead. DARTS can achieve very good performance with both a single and multiple GPUs and on all



(a) With memory limitation (500 MB per GPU).

(b) Without memory limitation (hardware limitation at 32 GB per GPU).

Figure 4.17: Performance on the sparse 2D matrix multiplication in real with 4 Tesla V100 GPUs.

the matrix multiplication variants as well as tasks from the Cholesky factorization without dependencies. However, when a large number of tasks are ready at the same time (e.g., the 3D matrix multiplication), the results are mixed.

Our goal is to create a generic scheduler: it must therefore be able to deal with dependencies. mHFP is not incremental, therefore, with a dynamic task set, mHFP would have to be applied each time a new task is made available. We have seen in this chapter that mHFP has an important scheduling overhead, so it cannot not be applied to a dynamic task set. Therefore, in the next chapter, focused on scheduling dependent task sets, we aim at improving DARTS.

Chapter 5

Dynamic Scheduling for Task Graphs

Contents

5.1 Existing runtime schedulers	100
5.1.1 A work stealing policy: LWS	100
5.1.2 A priority-based scheduler from the PaRSEC runtime: AP	101
5.2 Improving the DARTS scheduler	101
5.2.1 Intuition	101
5.2.2 Strategy	101
5.2.3 Eviction policy	103
5.3 Experimental settings	103
5.4 Cholesky factorization with GPUs	104
5.4.1 Overview	104
5.4.2 Optimal data access pattern	105
5.4.3 Single GPU case	105
5.4.4 With multiple GPUs	107
5.4.5 With multiple GPUs and no memory limitation	110
5.5 LU factorization with GPUs	111
5.5.1 Results on 4 GPUs	111
5.5.2 Results on a single GPU and no memory limitation	112
5.6 3D matrix multiplication with GPUs	112
5.7 LU factorization on a multi-core CPU	114
5.8 Conclusion on dynamic scheduling of task sets with dependencies	116



CHAPTER 3 focused on grouping task sharing data in a static schedule to reduce data transfers. Chapter 4 introduced a new strategy, called DARTS, that embraces this idea of data reuse while being dynamic, reducing complexity, and managing multiple processing units. In this chapter, we tackle our main goal, using the final model presented in Section 2.5: building a generic scheduler capable of reducing data transfers and increasing performance by partitioning and scheduling a set of tasks (with and without dependencies) sharing data on one or more processing units with limited distributed or shared memory.

The remaining complexity that we need to add in order to reach our final model is the handling of task graphs. Task graphs introduce task dependencies: some tasks must be processed before others and therefore have a higher priority. A scheduler must then both progress along the critical path (the longest path from the starting task to the last task) to avoid lack of parallelism, and order tasks to reduce data transfers. These are two conflicting goals that raise the following question: *How to balance locality and priority?* Because DARTS is dynamic, we favor it over mHFP, and focus in this chapter on rethinking the DARTS strategy to answer this question. Our goal is for the DARTS scheduler to be able to override priority in favor of locality when it can provide benefits.

We first present in Section 5.1 other strategies from runtime systems that deal with dependent task sets. Then, in Section 5.2, we describe how DARTS was redesigned to reach its most optimized version. Finally, we show a series of experimental evaluations. We first present an IO optimal scheduler and use its behavior to explain our results on the Cholesky factorization (Section 5.4). Then, we use the LU factorization with both distributed memory (Section 5.5) and shared memory (Section 5.7) settings. Lastly, in Section 5.6 we present results on the 3D matrix multiplication.

5.1 Existing runtime schedulers

In this section, we present various algorithmic solutions that already exist in the literature to solve the partitioning and scheduling problem with dependent task sets. As in previous chapters, we use the greedy EAGER scheduler. We also use DMDAS, a variant of DMDAR that sorts tasks in the queue by priority order (as provided by the application). It is the default state-of-the-art scheduler used by the Chameleon library [8] when managing dependent task sets. DMDAS also includes the *Ready* strategy (see Algorithm 2 in Chapter 3). We introduce two new competitors that are relevant in priority-based workloads: a work stealing policy to deal with load balancing and locality (Section 5.1.1), and a strategy from another runtime that uses priority as the main focus (Section 5.1.2).

5.1.1 A work stealing policy: LWS

Work stealing policies exist on several runtimes systems like XKaapi [65], or libraries like TBB [108]. Work stealing has proven to be efficient in situations where partitioning and reducing communication is critical. We present here Locality Work Stealing (or LWS), a scheduler that combines work stealing for load balancing, and locality to reduce communication. LWS is similar to the HWS scheduler introduced by Quintin et al. [107].

Each worker has a queue of tasks planned for future execution on the worker. LWS can deal with locality in two ways. First, when a task is released, it is queued on the worker that released it. Thus, by default a task and its descendants are all scheduled on the same worker. In most cases, the descendants of a task all share at least one input data, so scheduling them on the same worker favors data reuse. Secondly, when a worker becomes idle, it steals a set of tasks from neighboring workers. It steals tasks from the end of their queue. There is a high chance that the inputs of these tasks have not been prefetched

yet. This encourages workers to work on different input data. It thus favors distribution of different data to different workers, which increases the locality within each worker’s task queue.

5.1.2 A priority-based scheduler from the PaRSEC runtime: AP

DPLASMA [29] is a well-known implementation of dense linear algebra operations for heterogeneous architectures. It uses PaRSEC [31], a dynamic runtime for architecture-aware scheduling of tasks on heterogeneous architectures. PaRSEC comes with different schedulers that all share two common aspects. First, they use the theoretical performance of CPUs and GPUs to balance the load assigned to each processing unit. Second, they use the knowledge of the DAG [101] to evaluate the cost of a task relative to its input, i.e. if multiple tasks use the same input data, the cost of a task will only be its computation time (without counting the data loading time) as its input data can be reused. Thus, multiple tasks that share input data have a higher probability of being computed on the same processing unit.

In our experiments, we use the Absolute Priority scheduler (AP) because of its affinities with priorities, an important feature for the applications we will be using. With AP, all processing units share a task-waiting queue that is sorted by priority, and each time a processing unit becomes available, it takes for execution a task from the head of the waiting queue. We also evaluated the default scheduler of PaRSEC: LFQ. We found that AP always gets slightly better performance on our applications. For this reason, we only show AP in the following experiments.

5.2 Improving the DARTS scheduler

A preliminary version of DARTS limited to tasks without dependencies has already been proposed in Section 4.3. We present here the improvements added to the DARTS scheduler. We only detail the new features.

5.2.1 Intuition

Strategies mentioned in the previous section were favoring priority over locality: DMDAS and AP sort tasks by priority, favoring progress on the task graph critical path over data reuse. On the contrary, DARTS can contradict priority when useful to favor data reuse. Trading priority for more locality is interesting when (i) a large number of tasks are available for computation at a given time, and (ii) computation of tasks from the critical path is not urgent because many tasks must be computed before the critical path becomes really critical. In such a case, there is enough parallelism available to occupy all the processing units, and favoring data locality can thus bring more benefits. DARTS aims to take advantage of this fact by considering as a primary factor the data that would bring the most data reuse if loaded, and as a secondary factor, in case of a tie between two data to be loaded, the one associated with the highest priority task.

5.2.2 Strategy

The fundamental principle of DARTS is, when requested for tasks to put in $plannedTasks_k$ for PU_k , to first look for the best new data to load, that is, a data that will maximize the work that can be performed on PU_k . Our first improvement over the previous DARTS design comes from the selection of the best data to load D_{opt} . Whenever some PU_k requests for some task to execute, we first look in the $dataNotInMem_k$ set (which initially contains all data) for the optimal data D_{opt} that, if moved into

the memory of PU_k , would minimize the ratio between the transfer duration of D_{opt} and the computation time of tasks that become “free”, i.e., tasks that can be assigned and processed on PU_k without any additional data movement. Considering the transfer duration favors direct communication between GPUs (using NVLinks), which is much faster than using the PCI Express bus. In case of equality between several optimal data, we choose the data used by the task with the highest priority and associated with the most remaining work. Once such data is found, all the corresponding free tasks are assigned to $plannedTasks_k$. If we cannot find a data which enables free tasks, we look for the data D_{opt} that enables the computation of the highest-priority task T with one additional data load. In such a case, T depends on D_{opt} , on another data D not in memory, and possibly other data already in memory. It may happen that we do not find any such data, for example if all ready tasks depend on at least three data not in memory. In this case, the highest-priority task from $readyTasks$ is assigned to PU_k .

The *3inputs* variant from the last chapter is now obsolete. It was used as a secondary decision tree when there was no data to compute a free task and thus accommodate tasks with three inputs. We now use a single decision tree that can handle tasks with such input patterns. The *OPTI* variant that was presented for the last implementation of DARTS is also now useless, since this new version of the code has a large code optimization that partially solves our complexity problem.

We describe here the values required to select the optimal data D_{opt} (see Algorithm 12). Some values have already been presented when DARTS was first introduced in Algorithm 9. For the sake of clarity, we introduce them again here and outline the novelties in red.

- $S_0(D)$: the set of tasks that depend only on D and some data already loaded in the memory of PU_k .
- $S_1(D)$: the set of tasks that depend only on D , some data already loaded in the memory of PU_k , and 1 additional data.
- $max_prio(D)$: the highest priority of a task in $S_0(D)$ if $|S_0(D)| > 0$, otherwise the highest priority of a task in $S_1(D)$.
- $computation_time(D)$: the sum of the durations of the tasks in $S_0(D)$.
- $task_left(D)$: the sum of the durations of the tasks in $readyTasks$ that use D as an input.
- $transfer_duration(D)$: time required to load D to PU_k .

The previous DARTS design only supported independent tasks. The support of task graphs with dependencies requires supporting the dynamic addition of tasks in $readyTasks$ when their dependencies are resolved. When such addition occurs, we dynamically update each $dataNotInMem_k$, to include data used by this new ready task but which was not used by any other ready task, and not loaded or planned for load on PU_k . The $dataNotInMem_k$ sets thus always contain exactly the set of data used by tasks that can be started (either in $readyTasks$, in some $plannedTasks_k$, or in some $taskBuffer_k$), which are not yet loaded on PU_k . As a reminder from the last chapter, $taskBuffer_k$ is the set of tasks that have been popped from $plannedTasks_k$ for execution on PU_k .

Besides, for some newly-released ready tasks, we can bypass $readyTasks$ and directly push them to $plannedTasks_k$ when they are already “free”. Since we know which tasks have been queued to each $taskBuffer_k$ and $plannedTasks_k$, we know the next data loading operations performed on PU_k . When a new task becomes ready, we can check if it will be free on PU_k , i.e., if it can be already be processed by some PU_k without any additional data load (because its inputs are already loaded or queued to be loaded). In such a case, we directly assign the new ready task to the corresponding $plannedTasks_k$. If several PUs qualify, we first consider the one with the fewest-queued tasks, to balance the load.

Algorithm 12 DARTS scheduler on PU_k

When PU_k requests a new task

- 1: **if** $plannedTasks_k = \emptyset$ **then** ▷ We need to fill $plannedTasks_k$
- 2: **for each** data $D \in dataNotInMem_k$ **do**
- 3: Compute $S_0(D), S_1(D), max_prio(D),$
- 4: $computation_time(D), task_left(D)$ and $transfer_duration(D)$
- 5: Compute $ratio = \frac{transfer_duration(D)}{computation_time(D)}$
- 6: Choose the data D_{opt} with the smallest $ratio$, tiebreak in this order with $|S_0(D)|, max_prio(D),$
 $|S_1(D)|$ and $task_left(D)$
- 7: **if** $|S_0(D_{opt})| > 0$ **then**
- 8: Append $S_0(D_{opt})$ to $plannedTasks_k$
- 9: Remove D_{opt} from $dataNotInMem_k$
- 10: **else if** $|S_1(D_{opt})| > 0$ **then**
- 11: Choose task T with highest priority from $S_1(D_{opt})$
- 12: Append T to $plannedTasks_k$
- 13: Remove the inputs of T from $dataNotInMem_k$
- 14: **else**
- 15: Choose task T with highest priority from $readyTasks$
- 16: Append T to $plannedTasks_k$
- 17: Remove the inputs of T from $dataNotInMem_k$
- 18: Return head of $plannedTasks_k$

5.2.3 Eviction policy

The eviction policy is the same as presented in Section 4.3.4. We have updated this eviction policy to update $dataNotInMem_k$ for each PU_k after the eviction. The goal is to keep $dataNotInMem_k$ up to date with the data that is currently required by ready tasks. This is essential for task sets with dependencies, as the dataset must be dynamically managed to reduce the computational complexity of finding D_{opt} . If the evicted data is not used by any task in $readyTasks$, we remove it from all $dataNotInMem_k$. It will be added there again when a new ready task with this input becomes available. Otherwise, if any task of $readyTasks$ uses the evicted data, we add it to $dataNotInMem_k$ for each PU_k that does not hold it in memory (and is not scheduled to load it).

5.3 Experimental settings

We present below the experimental evaluation conducted to compare the strategies presented above with the improved DARTS scheduler¹. The schedulers have been tested on three linear algebra applications: the Cholesky factorization ($A = L \times L^T$), the LU factorization ($A = L \times U$) without pivoting and the 3D matrix multiplication ($C = A \times B$ and the computation of each tile of C is decomposed into multiple tasks, each requiring one tile of A, B and C). Cholesky and LU are composed of tiled matrix multiplications (GEMM, requiring 3 input data), symmetric rank-k update for Cholesky only (SYRK, requiring 3 input data), triangular matrix equation (TRSM, requiring 2 input data), and Cholesky decomposition (POTRF, requiring 1 input data) or LU decomposition (GETRF, requiring 1 input data).

¹The code to reproduce the results of this chapter, including DARTS and the applications we used, is available at: <https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/ICPP2023>

When the scheduler requires task priorities (which is the case for DMDAS, LWS and DARTS), they are computed as the bottom-level of the task in the task graph, which is the minimum time needed from the start of the task to the completion of the whole graph, assuming unbounded resources [14]. The PaRSEC AP scheduler uses its own set of priorities (although we noticed no difference when using bottom-level priorities). Although we are running tests on Cholesky and LU factorizations, we made sure that our new DARTS performs as well as the previous version on the applications from Chapter 4.

As in the previous chapters, we performed experiments on Tesla V100 GPUs (using cuBLAS 10.2 single precision GPU kernels) equipped with a 12 GB/s PCI bus. We use tiles of size 1920 as it allows to achieve best peak performance on GPUs. Although being separate implementations for flexibility, our applications are identical to those of the Chameleon linear algebra library [7]. In the case of the PaRSEC-AP scheduler, we use the DPLASMA implementation of the Cholesky decomposition. To facilitate a clear distinction of our various strategies, we have imposed a memory limitation of 2000 MB on the processing units. This limitation allows us to evaluate the different strategies with smaller datasets. We also provide results without any memory limitation. The shared memory setting with CPU is described in the section presenting results with CPU cores: Section 5.7.

5.4 Cholesky factorization with GPUs

We present here a series of evaluations performed on GPUs using the Cholesky factorization. We first describe how a communication-optimal algorithm behaves on a Cholesky factorization, before evaluating our scheduler on 1 and 8 GPUs under memory constraints. We also present results without memory constraints on 8 GPUs.

5.4.1 Overview

Figure 5.2 shows the results obtained by the various algorithms using one GPU. Again, we find on Figure 5.2a the maximum GFlop/s achievable, and the hard memory limitation with a black and a green dotted line, respectively. We can also see on Figure 5.2b, with the solid gray line, the PCI bus limit: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal time for computation.

Figure 5.2b also depicts, with a solid black line, the lower bound on the communication volume required by the Cholesky factorization of an $N \times N$ symmetric matrix. In a sequential out-of-core setting, with a single RAM of infinite size and processing units of memory size S , Beaumont et al. [24] prove that the minimal amount of IOs is computed as:

$$\frac{N^3}{3 \times \sqrt{2 \times S}}$$

To be more precise, S is the number of 32-bit scalars that can be loaded in memory. If we simplify, we can say that the proof of [24] states that the application can be divided into blocks of compute operations that maximize the number of compute operations performed with a minimal amount of scalar loads. In the proof, a single processing unit computes all the blocks of compute operations. With multiple processing units, all starting with an empty memory, one can partition these same blocks of compute operations to the GPUs, making them work on different data and therefore do not replicate data loads. Thus, this result can be extended to multiple processing units on a single node.

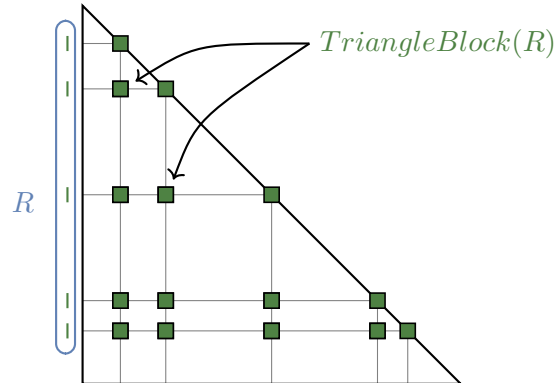


Figure 5.1: Representation of the tiles computed by a triangle block in an iteration of the Cholesky factorization. R is a set of TRSM results that have already been computed. $\text{TriangleBlock}(R)$ is the corresponding set of tasks to be processed to minimize IOs. Figure courtesy of Lionel Eyraud-Dubois.

5.4.2 Optimal data access pattern

The authors of [24] prove (i) that it is not possible to have fewer IOs than the previous lower bound, and (ii) that there exists a sequential algorithm that matches such a result. We briefly introduce how the optimal schedule would be adapted to multiple processing units on a single node. The proof mainly focuses on the communication volume when processing SYRKs and GEMMs. Therefore, we only focus on this part.

Figure 5.1 is an example of how the optimal schedule processes tasks on a given processing unit. The intuition is to partition the results of the TRSM tasks across multiple processing units and then compute as many SYRK and GEMM tasks as possible with such results. In Figure 5.1, we see on the left side, in blue, the results of TRSM operations that have already been computed on this processing unit. A node that has such results loaded into memory can compute the tiles in green (which are SYRKs or GEMMs) with a minimal amount of IOs. The communication-optimal algorithm would compute such *triangle blocks* associated with TRSM results already loaded on the GPU. Each GPU would compute tiles associated with different TRSM results, thus not replicating the data. Finally, such triangle blocks must be computed over multiple iterations. A scheduler that demonstrates a data access pattern in a triangular block that effectively reuses TRSM results across multiple iterations has the potential to significantly reduce IOs.

5.4.3 Single GPU case

From Figure 5.2a, we can see that LWS and EAGER greatly suffer from the memory constraint, as their performance plummet after the green line. With only 1 GPU, they both process tasks in their natural order of arrival, resulting in poor progress on the critical path for EAGER and a large number of data loads for LWS, as shown in Figure 5.2b.

DMDAS and AP results DMDAS has more sustained performance but is far from the asymptotic goal on large working set sizes. It suffers from the same issue as DMDAR: DMDAS does not reassign tasks according to the new data loaded on the GPU because it does not have a global view of the set of data and tasks, and thus cannot strike a balance between prefetching and eviction. However, we have seen in previous chapters that such a problem causes only small performance losses, not as dramatic as the one shown here. To better understand those results we can have a look at Figure 5.3. It corresponds

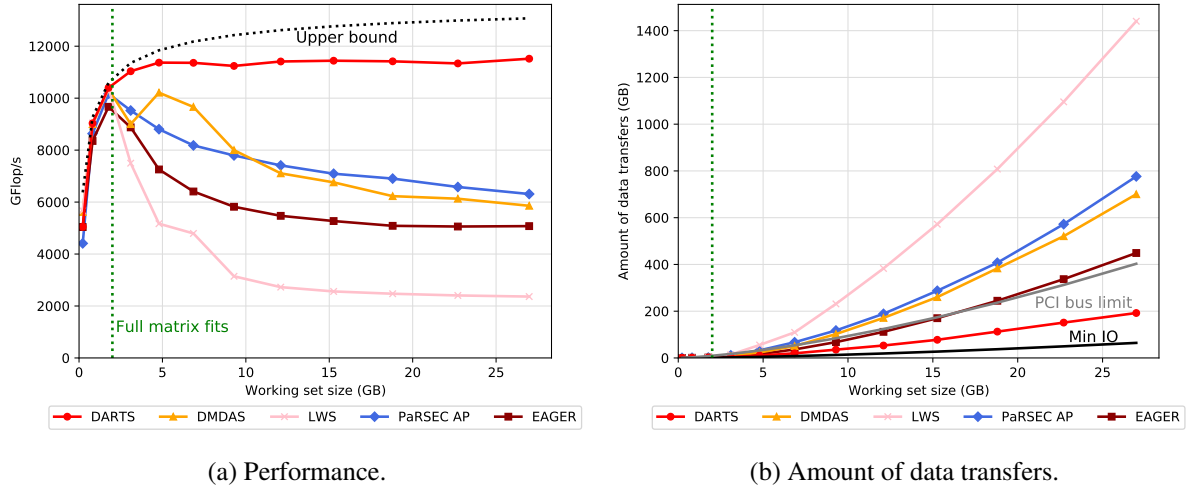


Figure 5.2: Results on the Cholesky factorization with 1 Tesla V100 GPU. Memory limited to 2000 MB.

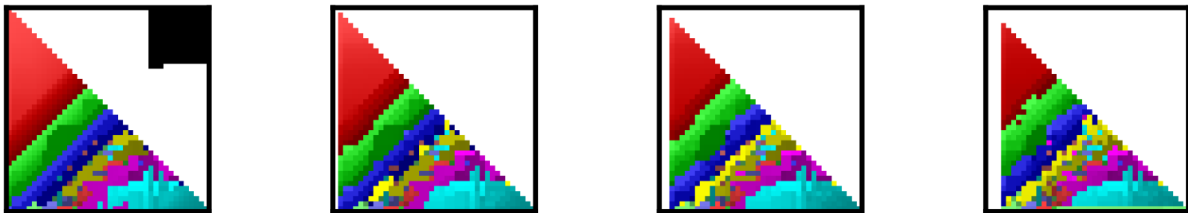


Figure 5.3: DMDAS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU. $N = 40$ and the GPU's memory is limited to 2000 MB. The first 1000 processed tasks are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. The shading within each color represents the processing order. The black area represents the amount of tiles that can be loaded in memory. Corresponds to the 8th point of Figure 5.2a.

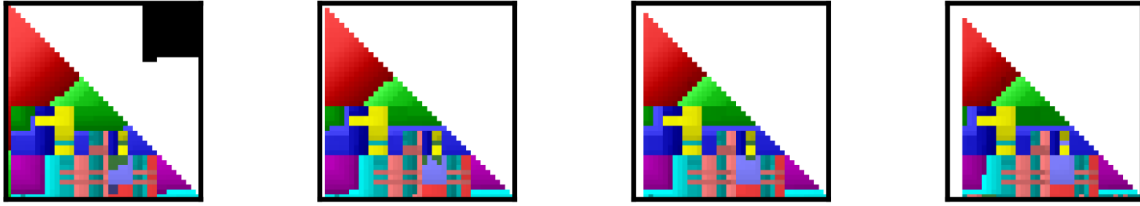


Figure 5.4: DARTS’s ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU. $N = 40$ and the GPU’s memory is limited to 2000 MB. The first 1000 processed tasks are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. The shading within each color represents the processing order. The black area represents the amount of tiles that can be loaded in memory. Corresponds to the 8th point of Figure 5.2a.

to the 8th point of Figure 5.2a. It shows the processing order of each task on the first four iterations of the Cholesky factorization. Each small square is a task. Here the first 1000 processed tasks are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. Again, within each color, the shading (from light to dark) represents the processing order. The black area represents the amount of tiles the GPU can load in its memory. We can see that DMDAS processes tasks following anti diagonals (because of task priorities). If we look at the 1000 blue tasks for DMDAS, we can see that these diagonals do not allow much data reuse (there are only data shares between the k iterations), and if the memory cannot hold more tasks, the affinity on the rows and columns cannot be found, resulting in multiple loads of the same rows or columns. This results in more data being transferred, as it can be seen in the Figure 5.2b. AP behaves similarly to DMDAS. It uses the expected completion time of tasks and sorts them by priority, resulting in similar results.

DARTS results DARTS maintains good performance with an increasing working set. DARTS also processes tasks in a diagonal while it can fit in memory (tasks in red on Figure 5.4) but then switches to what we call *DARTS triangle blocks*. We can notice such triangle blocks on the blue and magenta tasks (a triangle on the main diagonal preceded by a square of tasks of the same color on the first few columns). Note that DARTS triangle blocks are different from the optimal triangle blocks presented earlier because they do not necessarily group tasks from different zones, thus not forming a complete triangle. However, they have the benefit of reusing TRSM results to compute a maximum amount of tasks. Those triangles and the associated squares fit exactly in memory and share a lot of common data and are replicated over multiple iterations (up to $k = 8$) in order to progress on the critical path, while maximizing data reuse. As explained earlier, accessing data with such a pattern allows to reduce communications with the RAM as shown in Figure 5.2b: DARTS has the lowest amount of data transfers and is the only strategy under the bus limit. It means that theoretically, all transfers can be overlapped by a computation, which explains our good performance.

5.4.4 With multiple GPUs

Figure 5.5a shows the performance with 8 GPUs and Figure 5.5b shows the amount of data transfers. We first notice that compared to the performance with a single GPU, all strategies are much further from the upper bound. All strategies generate more transfers than the PCI bus limit (see Figure 5.5b). With multiple GPUs, some data has to be replicated on various GPUs memories, which greatly increases the total amount of data transfers but it is necessary to increase parallelization. This is normally not

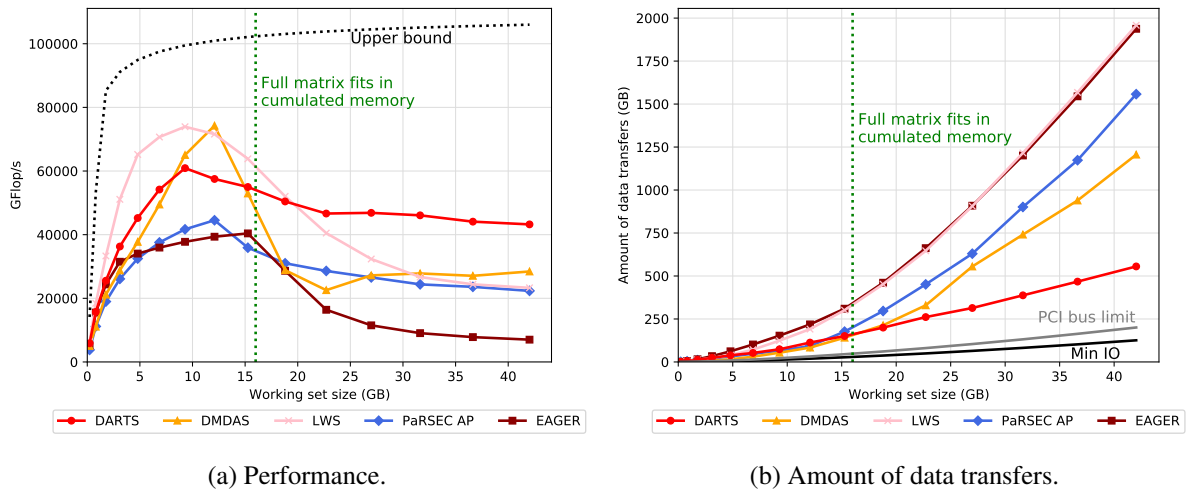


Figure 5.5: Results on the Cholesky factorization with 8 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.

an issue, but with a memory constraint this replication greatly reduces performance because each GPU will suffer the same performance loss as seen with 1 GPU. For matrices smaller than the limit size (green dotted line), LWS achieves the best performance. By stealing work from neighbor GPUs, LWS is able to largely increase transfers using NVLinks, which are much faster than transfers with the CPU memory. With 8 GPUs, the opportunity for such transfers are much more important which explains those performance. This explains why on Figure 5.5b, even with more transfers than DARTS, LWS achieves a better throughput on the left of the green line. For larger matrices, LWS and EAGER have much more data transfers and thus much smaller throughput. They do not consider memory limitation and thus do not favor data reuse. Similarly for AP and DMDAS, the low throughput is associated with high transfer rates on Figure 5.5b, as in the case with one GPU. However, before the constraint, as they consider the performance model to schedule, they are able to distribute tasks in a way that reduces the completion time.

DARTS results DARTS has the best performance once the memory becomes a constraint. Figure 5.6 represents the DARTS scheduling with 8 GPUs and $N = 40$ on the first two iterations of the outer-loop of the Cholesky algorithm. It corresponds to the 8th point of Figure 5.5. Note that the behavior of DARTS is similar at larger sizes, we show a visualization with $N = 40$ for the sake of clarity.

In Figure 5.6a, we can find a column of TRSM (in red) being processed ahead of a triangular block of GEMMs and SYRK. They are on the same rows of the TRSM results which means they can reuse their data. As we can see, this is also done in the second iteration and even up to iteration 10 (not shown here). Our triangle blocks are not complete, so we are not optimal in terms of IOs, but finding such a structure explains our reduction in data transfers. We can also find such triangular structures with a TRSM column associated on Figures 5.6d and 5.6h.

Sometimes, the TRSM column is not associated with the triangle blocks, as we can see for example in Figure 5.6e. This is not an issue as such data can be transferred from other GPUs: from GPUs 1 and 8 with our current example. TRSM tasks must be computed before GEMM and SYRK tasks and their input data are used by a large number of subsequent tasks. So, there is a good chance that their input data are still in memory and can be quickly transferred using NVLinks. After doing so, the triangle block in Figure 5.6e can be computed with few data transfers. DARTS triangle blocks with a missing TRSM

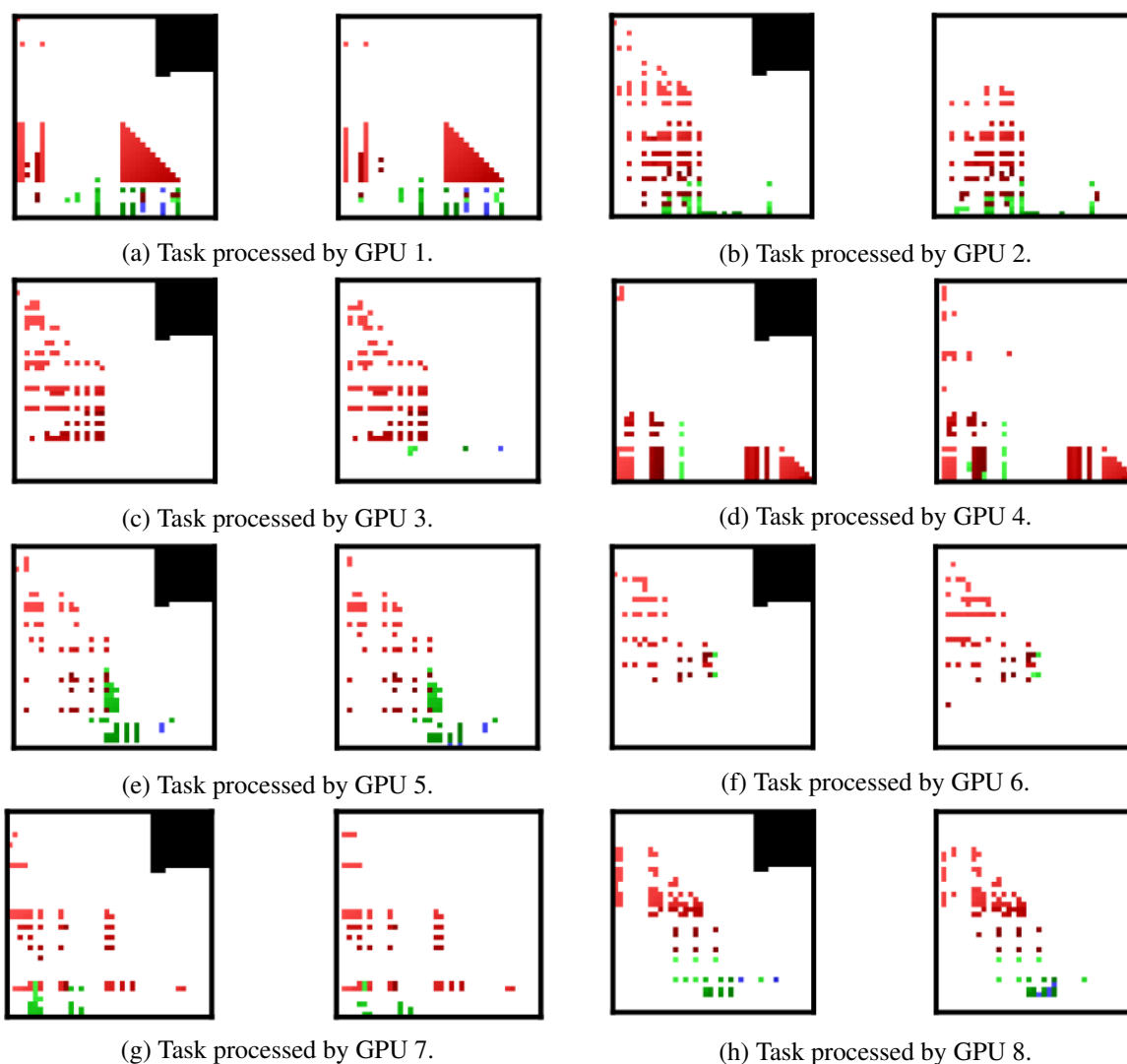


Figure 5.6: DARTS' ordering on the first two iterations of the Cholesky factorization with 8 Tesla V100 GPUs. $N = 40$ and the GPU's memory are limited to $2000 MB$. The first 1000 tasks processed on each GPU are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. The shading within each color represents the processing order. The black area represents the amount of tiles that can be loaded in memory. Corresponds to the 8th point of Figure 5.5.

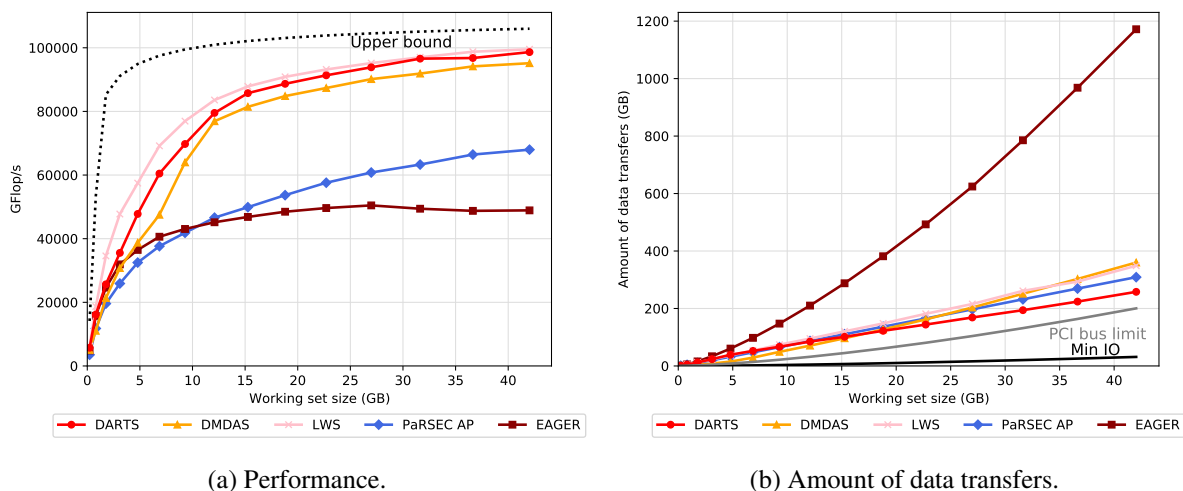


Figure 5.7: Results on the Cholesky factorization with 8 Tesla V100 GPUs. Hardware limitation of each GPU memory at 32 GB.

column are also seen on Figures 5.6b, 5.6c, 5.6f and 5.6g. In addition, for all the figures mentioned, if we renumber the columns, we can again find triangles of task, especially for the red tiles, that visually show data sharing.

Once the memory becomes a constraint, such a strategy allows DARTS to greatly reduce data transfers. DARTS triangle blocks are the results of our ratio ($ratio = \frac{transfer_duration(D)}{computation_time(D)}$) used to select the best data to load D_{opt} . $transfer_duration(D)$ favors loading TRSM results from other GPUs which is the first requirement to compute tasks in a triangular block. $computation_time(D)$ favors tasks that can be computed with only one additional load, which is the case for GEMM and SYRK tasks once the TRSM's results are in memory: this is the second step to form DARTS triangle blocks.

Moreover, there is a good distribution of the data load on the 8 GPUs. The ratio we just mentioned favors the selection of a data associated with as many unprocessed tasks as possible. The total work associated with a data is reduced after some of its task is scheduled on another processing unit. Thus, two distinct processing units have a low probability of selecting the same data. This encourages GPUs to work on different datasets, further reducing the total amount of data to load by minimizing the replication of data on multiple GPUs.

However, we can observe a performance loss on the first few points of Figure 5.5a, when only a few tasks are computed. With a lot of GPUs and few tasks, it is sometimes more efficient to assign many tasks sharing data to a single GPU, even if another one is idling, as it can lead to a smaller completion time overall. DARTS always assigns a task to idle GPUs, leading to this result on small matrices. Once the workload is large enough, it is not beneficial to keep some GPUs idle, which allows us to avoid the performance loss on the first few points. The sustained performance of DARTS after the memory constraint with both 1 and 8 GPUs shows that it is generic enough to adapt to various numbers of processing units.

5.4.5 With multiple GPUs and no memory limitation

Figures 5.7a and 5.7b show the results obtained with 8 GPUs and without imposing a memory limit (each GPU is equipped with 32 GB of RAM). On Figure 5.7b, we observe that DARTS manages to reduce the amount of transfers on large matrices. This reduction is not significant enough to gain performance

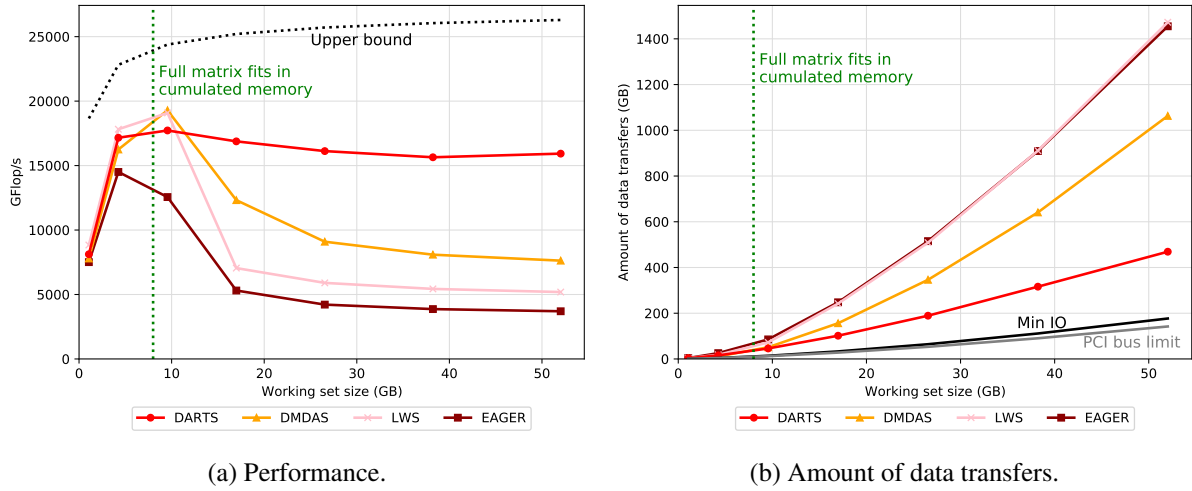


Figure 5.8: Results on the LU factorization with 4 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.

because, DARTS is unable to keep some GPU idle, which leads to slightly worse results than LWS on the first few points. Apart from this, DARTS performs very similarly to LWS, which is the best strategy here. Our scheduling strategy can thus also be used in situations where memory is not a constraint, making DARTS generic enough to adapt to various memory sizes.

5.5 LU factorization with GPUs

We now move to the LU factorization. Figures 5.8b, 5.9b, 5.12b and 5.12d plot the minimal amount of IO required by the LU factorization. Table 2 from a study by Olivry et al. [104] show that the minimal amount of data transfers that can be achieved is:

$$(2 \times N^3)/(3 \times \sqrt{S})$$

Table 1 from the same paper shows that such a limit is achievable.

5.5.1 Results on 4 GPUs

Figure 5.8a presents performance using the LU factorization with 4 GPUs. This figure does not present results for ParSEC-AP as the DPLASMA implementation of LU does not make use of the cuSolver library to solve GETRF kernels of the LU factorization. Hence, AP uses a much slower version of this kernel, which makes it impossible for us to make a fair comparison with AP. LWS and EAGER process tasks in their submission order (sorted by priorities for LWS), which makes it impossible to reuse data on consecutive tasks when memory is limited. Hence, these two schedulers end up with 3 times more data transfers than DARTS (see Figure 5.8b). DARTS has more sustained performance, and similarly to the Cholesky case, is able to reduce data transfers by distributing the data on multiple GPUs and reusing data for consecutive tasks.

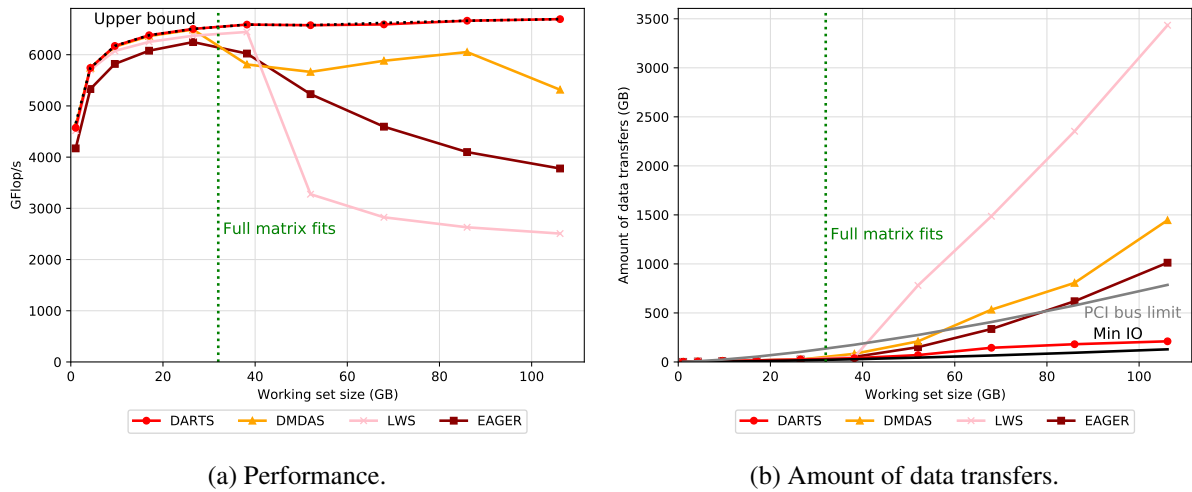


Figure 5.9: Results on the LU factorization with 1 Tesla V100 GPU. Hardware memory limitation at 32 GB.

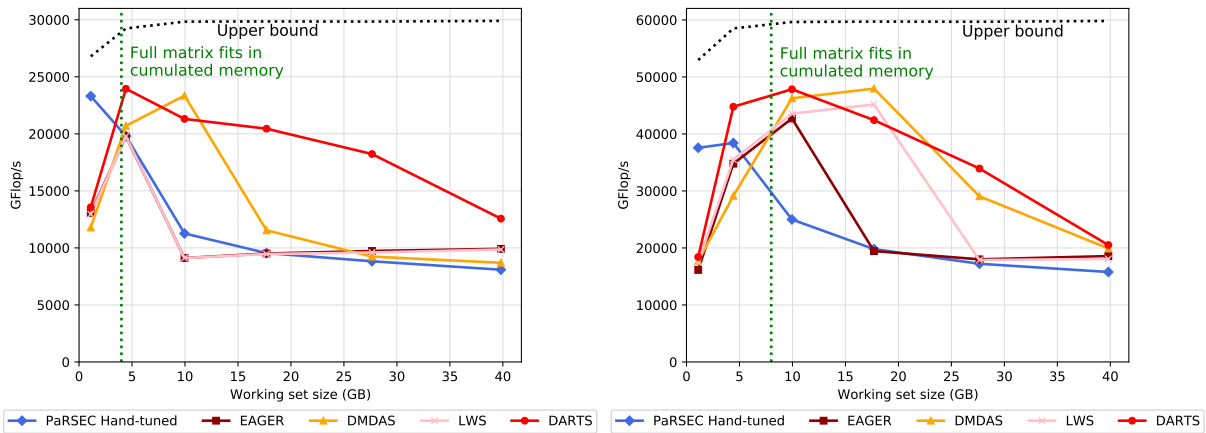
5.5.2 Results on a single GPU and no memory limitation

Figure 5.9a shows the results for the LU factorization on a single GPU without limiting the memory. The GPU embeds 32 GB of RAM, we can see this value on the plot with the green line. DMDAS has more sustained performance than LWS and EAGER. The *Ready* strategy (see Algorithm 2 in Chapter 3) allows DMDAS to re-order tasks so as to first compute tasks for which the input data is already in memory, thus reducing data transfers. DARTS stays very close to the maximum obtainable GFlop/s for the whole range of matrix sizes.

LU and Cholesky are very closely related applications: LU can be considered a "duplex Cholesky". Because of the symmetry of the matrix, LU has twice as much data as Cholesky, but also twice as many tasks to compute. So, they have the same communications to computations ratio. The same data reuse patterns can be found for Cholesky and LU: building square blocks of GEMMs increases data reuse. For Cholesky, however, one must exploit the symmetry of the matrix to increase data reuse, which is a more complicated solution. This solution is much harder for a dynamic scheduler to reproduce. Since DARTS strategy is similar for LU and for Cholesky, it is easier to reduce data transfers with LU. This way, DARTS can get closer to the minimum IOs with LU, as demonstrated in Figure 5.9b: DARTS has 1.6 times more data transfers than the optimum and has 3.7 times less transfers than the PCI bus limit. As a result, it can completely overlap communications with computations, resulting in near-optimal performance with a single GPU.

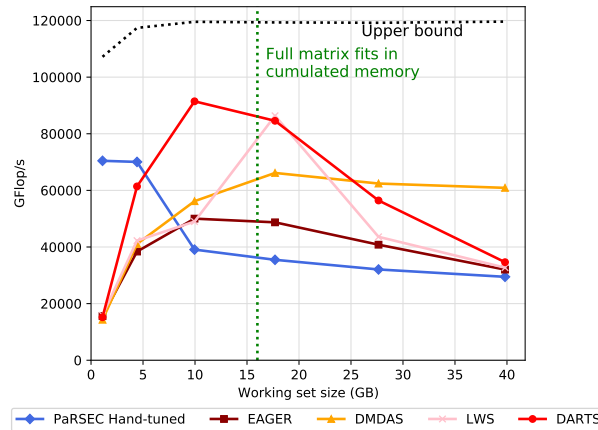
5.6 3D matrix multiplication with GPUs

As mentioned in the last chapter, DARTS was unable to achieve good performance on 3D matrix multiplication due to its scheduling overhead (see section 4.4.5). After many complexity improvements through code optimizations, our final version of DARTS is now much more scalable. To test it on a task set with a large number of tasks that are ready at the same time, we compare ourselves to a hand-tuned version of the 3D matrix multiplication [75] from the PaRSEC runtime. It implements a control flow to avoid critical overflows of GPU memory. We are aware that such a hand-tuned version was aimed at large distributed systems, used with specific tile sizes and without limiting the memory of the GPUs.



(a) With 2 GPUs.

(b) With 4 GPUs.



(c) With 8 GPUs.

Figure 5.10: Performance on the 3D matrix multiplication with Tesla V100 GPUs. Memory limited to 2000 MB.

The current context does not allow a fair comparison. However, it is interesting to see how it behaves when we plug it directly into our memory-constrained application case.

Figure 5.10 presents experiments on the 3D matrix multiplication with 2, 4 or 8 GPUs. We learn from these figures that the hand-tuned version achieves the highest performances on very small matrix sizes. If memory was not constrained, such results would be unbeatable. However, under strong memory constraints, performance are degraded. We also learn from these figures that the complexity of DARTS causes issues only on very large working set sizes. Also, the more GPUs we add, the more DARTS is affected by the scheduling overhead because the scheduling is done every time a GPU is idle, which happens more frequently with more GPUs. Finally, we can observe in all figures the point where DARTS performance is maximal compared to the other strategies: just before or after the hard memory limitation. Indeed, when only the input matrices A or B can fit into the cumulated memories, as is the case just before the green line, the DARTS eviction strategy comes in very handy, as it avoids evicting data that would be useful for the subsequent tasks. This is something the other schedulers do not deal with, while DARTS updates the task in its *plannedTasks* buffer for each GPU to keep it up to date with the data in memory.

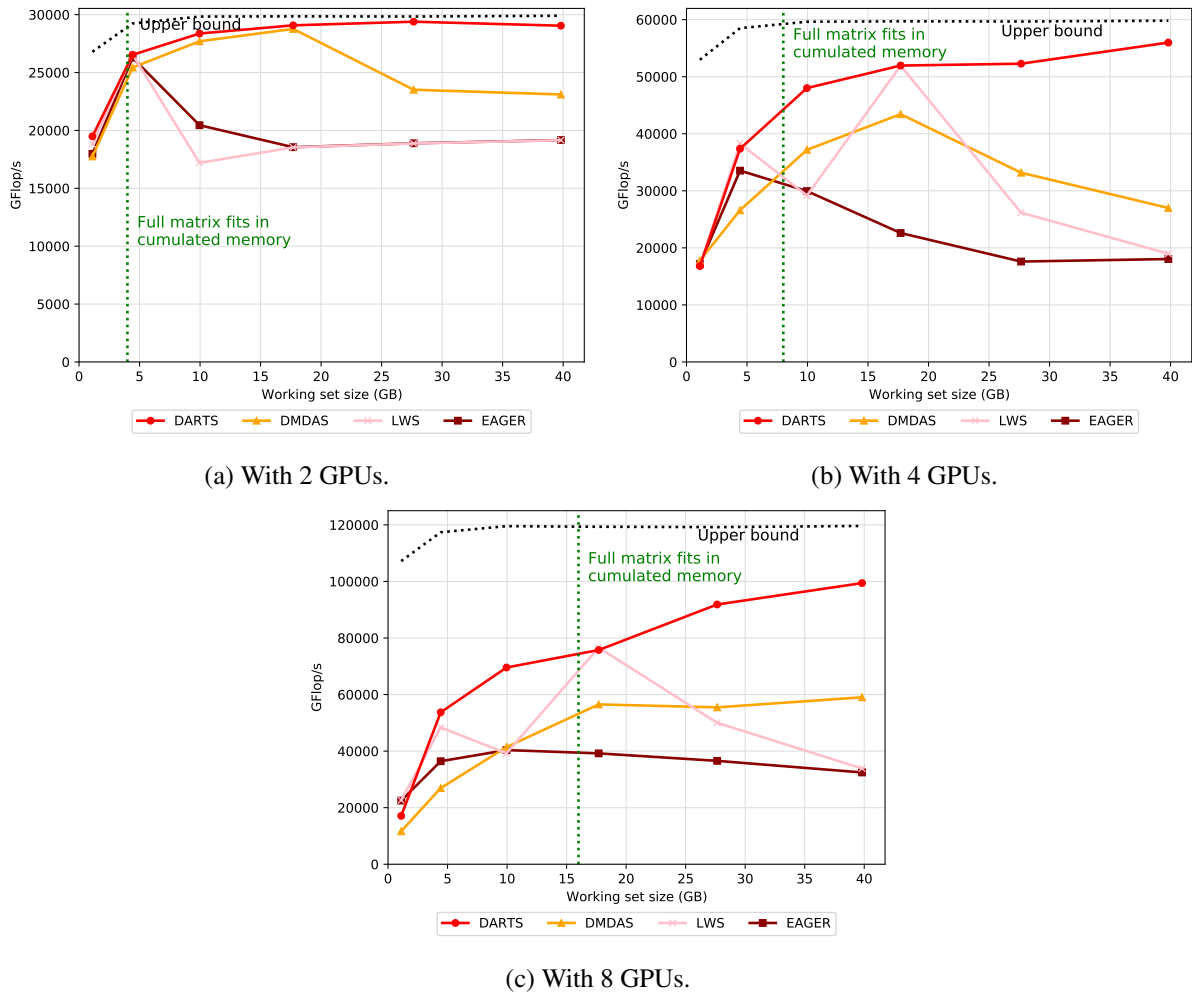


Figure 5.11: Performance on the 3D matrix multiplication in simulation with the performance models of Tesla V100 GPUs. Memory limited to 2000 MB.

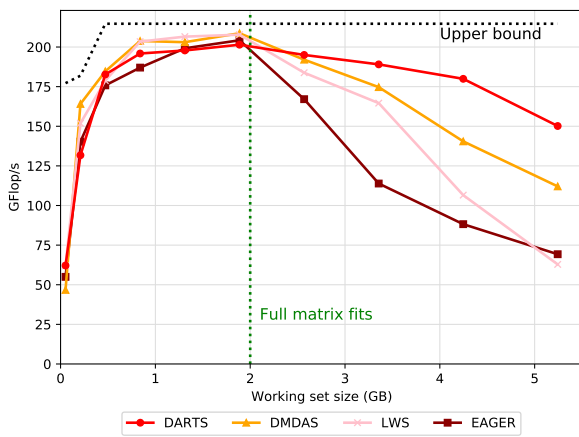
Figure 5.11 is the same set of experiments as presented above but in simulation. Since the scheduling time is not taken into account when calculating the GFlop/s, all strategies have a higher throughput. We do not have results from the hand-tuned version in simulation since PaRSEC does not support it. From these three figures, we can verify that theoretically, our new DARTS can achieve the best performance on the 3D matrix multiplication.

5.7 LU factorization on a multi-core CPU

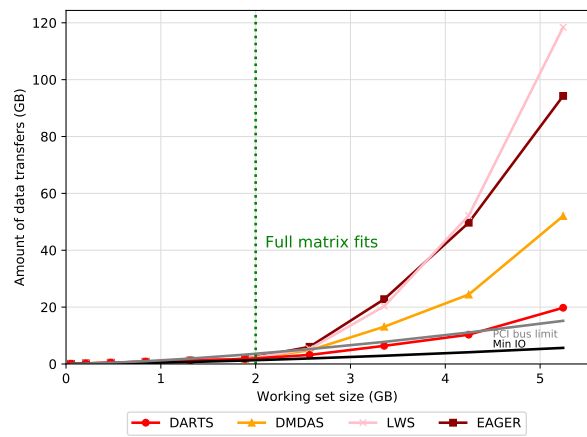
Our scheduler is able to consider any memory-limited system, and STARPU manages disk-CPU transfers exactly like CPU-GPU transfers. As a result, we are able to perform experiments on a multi-core CPU.

Figure 5.12 reports experiments on an AMD EPYC 7642 CPU² with 16 or 48 cores, 2 GB of shared memory, and disks that sustain a 250 MB/s bandwidth. The application scenario here is only the LU

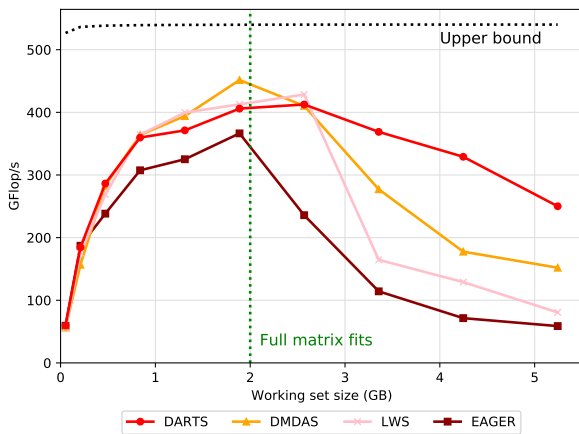
²From the neowise-1 node on the Grid5000 computing platform: <https://www.grid5000.fr/w/Lyon:Hardware#neowise>



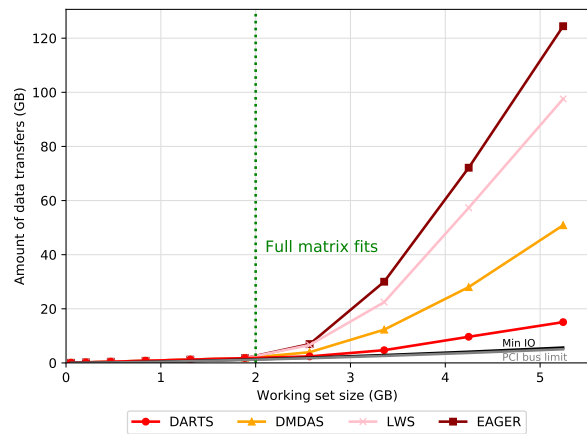
(a) Performance with 16 CPU cores.



(b) Amount of data transfers with 16 CPU cores.



(c) Performance with 48 CPU cores.



(d) Amount of data transfers with 48 CPU cores.

Figure 5.12: Results on the LU factorization with an AMD EPYC 7642 CPU. Memory limited to 2000 MB.

factorization. We use tiles of size 320, which is a size optimized for CPU cores. In this shared-memory setting, DARTS behaves in a similar way as it does with one GPU: it uses a single *plannedTasks* queue as well as a single *taskBuffer*. Before the memory limitation, DARTS is at worst 5% slower than DMDAS or LWS. This can be explained by two factors. (i) Like with GPUs, DARTS does not favor the fastest CPU cores. CPU cores are supposed to have identical performance, but in practice they have small differences. Choosing the fastest cores brings improvements when few tasks are available. This is the case for the first few points. (ii) So far, we have tested DARTS with at most 8 workers (GPUs). With 2 and 6 times more processing units, workers are requesting tasks much more often from the scheduler. The DARTS scheduling overhead is slightly higher than those of LWS and DMDAS, which can cause such performance loss.

After the memory limitation, DARTS is the best strategy overall. With 16 cores, DARTS is able to stay close to the disk bandwidth limit, symbolized by the gray curve in Figure 5.12b. Theoretically, DARTS could overlap most communications and computations. As we just mentioned in (ii), scheduling overhead is an issue here, which explains why DARTS performance is not at the asymptote. With 48 cores, the performance difference between DARTS and the other schedulers is greater. We can explain this by looking at the topology. The bandwidth between the disk and the 48 CPU cores is around 250 MB/s. Each core may need to transfer data from disk at the same time, placing a high demand on a bandwidth that cannot be expanded. With three times more cores, this problem becomes more severe, which explains why all strategies are further from the asymptote with 48 cores. With 48 cores, DARTS stays very close to the minimum number of IOs required, as shown in Figure 5.12d. Because DARTS reduce data transfers, it partially avoids this severe problem and is able to widen the performance gap with 48 cores compared to 16.

5.8 Conclusion on dynamic scheduling of task sets with dependencies

Adding dependencies to the partitioning and scheduling problem brought two additional challenges. (i) Tasks are now becoming ready gradually, meaning that a scheduler must produce data reuse patterns without having a full view of the task set. (ii) Priorities are assigned to tasks to progress on the critical path, however, following such path can be suboptimal if one wants to minimize data transfers. After detailing schedulers we want to use as references, we focused in this chapter on improving our DARTS scheduler. It now answers the two challenges we just mentioned, respectively by dynamically updating its list of tasks and data used to make scheduling decisions, and adding priority as a secondary factor in its schedule.

We have performed experiments with three classical linear algebra operations: Cholesky and LU factorizations and the 3D matrix multiplication. Thanks to its modularity, DARTS reaches very good performance in a large variety of situations, from multi-core CPU with shared memory to multiple GPUs with distributed memory. Among available schedulers, DARTS is the only strategy to reach good performance in memory-limited scenarios, and it also ranks among the best ones in all settings. In particular, we explain that its strategy for choosing which data to load next enables it to replicate a near-optimal schedule to minimize data transfers. On top of that, we ensured that our new DARTS gets similar results compared to the previous DARTS on the 2D matrix multiplication, making DARTS able to deal with a large scope of linear algebra applications.

DARTS answers our first stated goal. Without knowledge of the full task graph, it is able to lower data transfers and increase performance on task sets with or without dependencies, on both GPUs and CPU cores, and under any memory setting.

Chapter 6

Leveraging Locality for Batch Schedulers

Contents

6.1	Motivation	118
6.2	Related work	119
6.2.1	Scheduling jobs on large clusters	119
6.2.2	Using distributed file systems to deal with data-intensive workloads	119
6.2.3	Using schedulers to deal with data-intensive workloads	120
6.2.4	Reducing I/O contention	120
6.3	Framework	120
6.4	Schedulers	122
6.4.1	Two schedulers from the state of the art: FCFS and EFT	123
6.4.2	Data-locality-based schedulers	123
6.4.3	Adding backfilling to all strategies	125
6.5	Experimental settings	126
6.5.1	Platform description	126
6.5.2	Workloads description	126
6.5.3	Usage of real cluster logs	127
6.5.4	Simulator description	128
6.6	Experimental evaluation and analysis	129
6.6.1	Results on an underutilized cluster	129
6.6.2	Results on a saturated cluster	131
6.6.3	Complete results	132
6.7	Conclusion on locality-aware batch scheduling	135



We have studied in Chapters 3, 4, and 5 how to partition and order tasks across multiple processing units in order to solve linear algebra applications faster, even when the memory is a scarce resource. To do this, we have seen that data locality is crucial: reusing similar data for successive tasks leads to fewer data transfers and thus better performance. In this chapter, we address a problem that requires similar solutions to more efficiently compute scientific applications, but in a completely different setting.

We aim to schedule workloads on a high performance computing cluster. This is called *batch scheduling*, and it plays a critical role in the efficient management of a supercomputer by orchestrating the execution of computational *jobs*. In the area of batch scheduling, we seek to schedule data-intensive workloads, i.e., job sets with large *input files*. We want to apply locality techniques to reduce the amount of file load and thus minimize the completion time of a job set.

The chapter is organized as follows. After presenting our motivation in Section 6.1, we detail existing work related to batch scheduling and how they relate to our issue with Section 6.2. Then, in Section 6.3, we formalize our model of scheduling data-intensive jobs sharing input files on a cluster. Section 6.4 presents three schedulers focusing on re-using input files while minimizing evictions and avoiding starvation. Lastly, we explain how we conducted our experimental evaluation in Section 6.5 before showing the achieved results in Section 6.6.

6.1 Motivation

In the previous chapters, we explored how to reduce the execution time of scientific applications on a node composed of multiple GPUs or CPU cores. High-performance computing platforms typically consist of a large number of computation nodes, each equipped with computational resources. To further explore the topic of reducing the execution time of scientific applications, one can also take a step back and look at the cluster as a whole. Scientists typically submit their computation jobs to a scheduler, which decides the ordering and mapping of the jobs on the platform. This needs to be performed with particular care to balance resource utilization and user satisfaction, so as to leverage the computation resources as efficiently as possible, while avoiding adverse pathological cases that could particularly impact some users rather than others.

Computation jobs need data input which, more often than not, can be very large, notably for many subfields of life science with highly data-dependent workflows like taxonomic identification of DNA fragments, genome alignments or ancestral reconstructions. It is a frequent use pattern for users of such communities to submit a large batch of jobs using the same input files. Moreover, workloads from these specific subfields require the data to be loaded ahead of the computation. Loading such data input from the storage nodes may consume a significant part of the job duration. This load penalty can however be avoided altogether when the data was actually already used by the previous job running on the computation node, and thus still immediately available on the node. Therefore, by scheduling jobs that use the same input data one after another on the same node, it is possible to reduce job completion time, leading to better platform utilization efficiency. We only consider applications with very small output data compared to the large databases used as inputs. This simplification lets us to avoid adding an additional layer of complexity, as we want to focus primarily on locality-aware scheduling.

Unfortunately, classical job schedulers mostly do not take data input into account, and thus do not benefit from such data reuse ; most jobs must always re-load their data input. From this observation, and inspired by what we have learned about locality-aware scheduling in the previous chapters, we want to extract the benefits of reusing input data between successive jobs. We thus present in this chapter three new algorithms that add such data reuse to the scheduling equation. By tracking what data is loaded on

which node for the scheduled jobs, they are able to significantly reduce data loads, thereby improving both resource utilization and user satisfaction.

We evaluated these algorithms thanks to traces of actual jobs submissions observed on a large cluster platform. This allows to assess the effectiveness of our heuristics over a variety of realistic working sets. This revealed that while our heuristics get slightly worse results over some working set samples (those which exhibit ample cluster underuse), most working set samples largely benefit from our heuristics.

6.2 Related work

We now describe the literature specific to the management of batch systems. We begin by describing classical scheduling policies and identify those that are most commonly used. We then describe different ways to manage data-intensive workloads: through distributed file systems, data-aware scheduling, or by reducing I/O contention.

6.2.1 Scheduling jobs on large clusters

Workloads managers are the main component of High-Performance Computing clusters. They profile, distribute and schedule jobs on all the nodes. Workload managers like SLURM [127], OAR [34], TORQUE [119], LoadLeveler [83], Portable Batch System [74], SunGrid [67] or Load Sharing Facility [129] all offer various scheduling strategies.

The First-Come-First-Served (FCFS) algorithm is the prevalent default scheduler on most of these solutions [56]. Moreover, SLURM is used on most of the TOP500 supercomputers and its default strategy is FCFS [115] as well. We can then safely assume that comparing ourselves to FCFS will bring significant insights on what improvements can be achieved on data-intensive workloads.

A backfilling strategy is known to improve the use of supercomputer resources [78] [94]. The most commonly-used backfilling strategy is called conservative backfilling [118] [95]. It follows a simple paradigm: "a job can only be backfilled if it does not delay a job of higher priority". However, backfilling strategies can lead to issues like unfair advantages for small jobs (either by length of amount of requested cores/nodes). We will thus compare our strategies to FCFS with or without backfilling.

Other scheduling strategies exist. Maui [79], Gang scheduling [60], RASA [105] that use the advantages of both Min-min and Max-min algorithms, RSDC [48] that divides large jobs in sub jobs for a more refined scheduling, or PJSC [68] and PSP+AC [47] that are priority-based schedulers; however these heuristics do not consider the impact that input re-use could have on data-intensive workloads. We aim at resolving this issue in this chapter.

6.2.2 Using distributed file systems to deal with data-intensive workloads

Distributed file systems are a solution to ease the access to shared input files. They facilitate the execution of I/O-intensive batch jobs by selecting appropriate storage policies. HDFS [28] (Hadoop Distributed File System) is the most commonly used and incorporates storage-aware scheduling. It migrates a computation closer to where the data is located, rather than moving the data to where the application is running, in order to reduce communication. These solutions are mainly storage systems that use a history of file locations to serve as a backup. In our scenario, we copy the data from an already-redundant system (an online database for example) and store it locally on the node in an ephemeral way. Thus, in the event of a crash, we do not manage the data which is already redundant, it simply results in an aborted job. Secondly, the scheduling can cause issues (notably MapReduce, used in HDFS), as described by

Weets et al. [125]. By not using HDFS or any distributed file system, we avoid these problems altogether. Lastly, file systems are particularly efficient when the input data used are identical over time. In our case, between users, the inputs will be largely different, making distributed file systems less efficient.

6.2.3 Using schedulers to deal with data-intensive workloads

Some schedulers tackle the issue of data-intensive workloads. A solution can be to minimize network contention by allocating nodes to even out node and switch contention [100]. In our model, we are not studying the network topology and consider independent nodes. This is reasonable, since our main concern is the cross-section bandwidth to a shared storage solution.

Nikolopoulos et al. [103] focus on a better utilization of idling memory together with thrashing limitation. Our focus will be to control data loads in order to limit eviction and will thus naturally limit thrashing.

Agrawal et al. [4] propose to schedule jobs not sharing a file first and to use a stochastic model of job arrivals for each input file to maximize re-use. That work focuses the Map-Reduce model and predicts future jobs arrivals, two prerequisites that we do not consider.

Selvarani et al. propose an improved activity-based costing scheduler [112] where the scheduler adapts to different application types (CPU intensive, high memory usage, high I/O cost) in order to make the right decision. Our approach is more focused on maximizing data re-use on a set of identical nodes.

An interesting solution proposed specifically for the Jacobi-Davidson method [99] addresses both load balancing and memory constraints. It estimates how long it will take the fastest processor to complete a set of jobs. It can use this information to balance the load. To deal with the memory constraint, the strategy used is to suspend the execution of jobs that exceed the memory limit. Our model does not include jobs that can be stopped and restarted, so we cannot apply this method.

6.2.4 Reducing I/O contention

Reducing contention for the bandwidth used to transfer files is another solution to handle data-intensive workloads. Herbein et al. propose to avoid assigning jobs to nodes that would cause I/O contention [76]. It is also possible to try to predict I/O contention in order to avoid it [120]. Reducing I/O contention is an important but orthogonal task to the one we study here. We are only interested in where to allocate jobs to reuse data, but an additional step can be taken to better organize unavoidable I/Os. However, considering both steps simultaneously would most likely make the model too complex.

6.3 Framework

We consider the problem of scheduling a set of \mathcal{J} independent jobs, denoted $\mathbb{J} = \{J_1, J_2, \dots, J_{\mathcal{J}}\}$ on a set of \mathcal{N} nodes: $\mathbb{N} = \{\text{Node}_1, \text{Node}_2, \dots, \text{Node}_{\mathcal{N}}\}$. Each node Node_i is equipped with m cores noted: c_1^i, \dots, c_m^i sharing a memory of size M . Each job J_i depends on an input file noted $\text{File}(J_i)$, which is initially stored in the main shared file system.

During the processing of a job J_i on Node_k , $\text{File}(J_i)$ must be in the memory of Node_k . If this is not the case before starting computation of job J_i , then $\text{File}(J_i)$ is loaded into the memory. We denote by $\mathbb{F} = \{F_1, F_2, \dots, F_{\mathcal{F}}\}$ the set of distinct input files, whose size is denoted by $\text{Size}(F_i)$. Each job runs on a single node, but they can make use of a different number of cores. Single node jobs largely prevail in our studied dataset and adding the complexity of multi-node jobs is not necessary in the context of studying data reuse.

Each job J_i has the following attributes:

- Resource requirement: job J_i requests $Cores(J_i)$ cores, such that $1 \leq Cores(J_i) \leq m$;
- Input file: $File(J_i) \in \mathbb{F}$;
- Submission date: $SubTime(J_i)$;
- Requested running time (or walltime): $WallTime(J_i)$. If not finished after this duration, job J_i is killed by the scheduler;
- Actual running time: $Duration(J_i)$ (unknown to the scheduler before the job completion).

We do not consider the data output of jobs, as they are negligible in our workloads as mentioned above. Each of the \mathcal{J} jobs must be processed on one of the \mathcal{N} nodes. As stated earlier, the shared file system initially contains all files in \mathbb{F} . Each node is connected to the file system with a link of limited bandwidth, denoted by $Bandwidth$: transferring a data of size S from the shared file system to some node memory takes a time $S/Bandwidth$. The limited bandwidth as well as the large file sizes are the reasons why we aim at restricting the amount of data movement.

We consider that the memory of a node, denoted by M , is only used by the jobs input files, since all other data are negligible compared with the input files. We assume that jobs are devoted a fraction of the memory proportional to the number of requested cores, so that jobs willing to process large input files must request large number of cores. This way, we make sure that the memory of a node is large enough to accommodate all input files of running jobs. A file stored in the memory of a node can be shared by two jobs J_i and J_j only in either of the following situations:

1. J_i and J_j are computed in parallel on the same node.
2. J_i and J_j are computed on the same node consecutively (i.e., no job is started on this node between the completion of J_i and the start of J_j).

This can hold true if the file data is accessed through I/O (traditional or memory-mapped), allowing the same page cache to serve multiple processes from different jobs. Otherwise we consider that memory operations of jobs scheduled between J_i and J_j will cause the file to be evicted.

For each job J_i , the scheduler is in charge of deciding which node will process J_i , and more precisely which cores of this node are allocated to the job, as well as a starting time t_k for J_i . More precisely, J_i is allocated a time window from t_k to $t_k + WallTime(J_i)$ devoted to (i) possibly loading the input file $File(J_i)$ in the memory (if it is not already present at time t_k) and (ii) executing job $J(i)$. If the job is not completed at time $t_k + WallTime(J_i)$, it is killed by the scheduler to ensure that later jobs are not delayed. The scheduler must also make sure that two jobs are not executed simultaneously on the same cores.

We can see a summary of a job lifecycle in Figure 6.1. When a job is submitted by a user, it is added to \mathbb{J} . From jobs in \mathbb{J} , the job scheduler is responsible for assigning jobs to nodes. It also creates a planned schedule for each node, that can be modified at any time. Each node has a limited amount of memory and can compute multiple jobs simultaneously. The distributed file system allows the nodes to load the files that they are missing for a computation. It is important to note that the file transfer is done before the computation and cannot be overlapped. Jobs are non-preemptible: when started, a job is executed throughout its completion.

Our objective is to reach an efficient usage of the platform and to limit job waiting times. Each user submitting jobs is interested in obtaining the result of jobs as soon as possible. Hence we focus on the

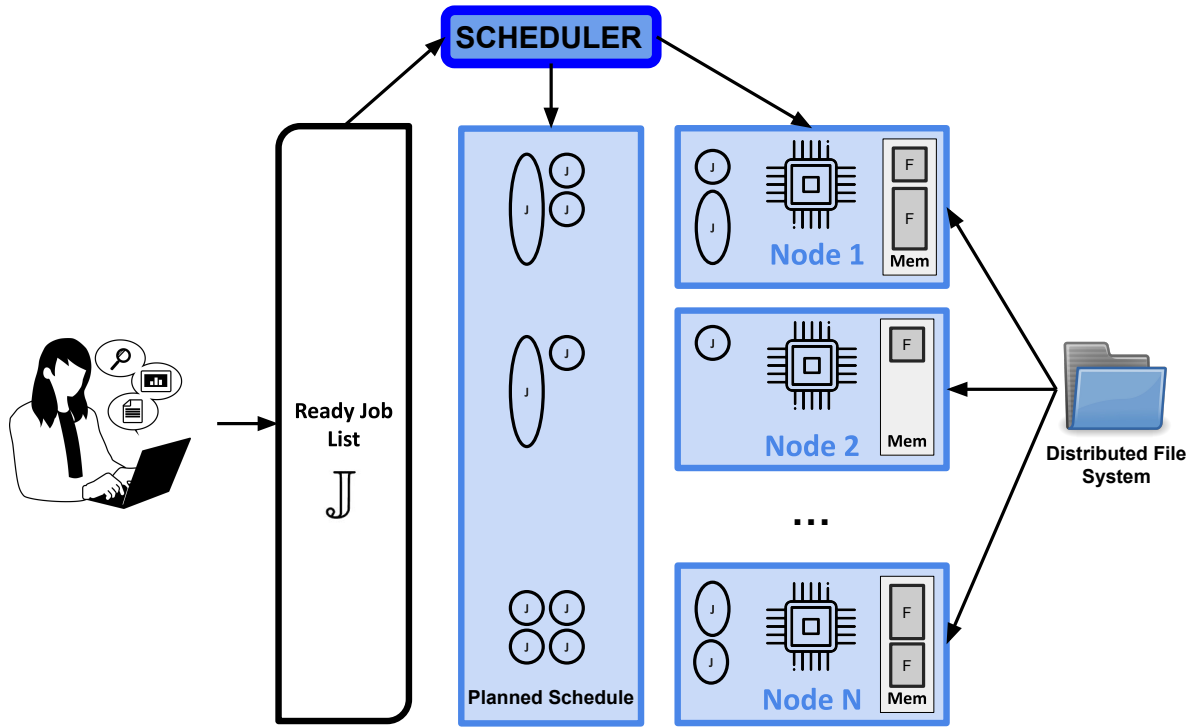


Figure 6.1: Platform representation.

time spent in the system for each job, also called the flow time (or flow) of the job:

$$Flow(J_i) = CompletionTime(J_i) - SubTime(J_i)$$

In the following, we want to consider aggregated performance metrics on job flows, such as average flow. However, the duration of a job significantly impacts its flow time. Jobs with the same flow but very different durations do not experience the same quality of service. To avoid this, the *stretch* metric has been introduced that compares the actual flow of a job to the one it would experience on an empty cluster:

$$ReferenceFlow(J_i) = \frac{Size(File(J_i))}{Bandwidth} + Duration(J_i)$$

$$stretch(J_i) = \frac{Flow(J_i)}{ReferenceFlow(J_i)}$$

The stretch represents the slow-down of a job due to sharing the platform with other jobs and is a commonly used metric in the evaluation of batch systems [37, 38, 40]. Considering the stretch allows to better aggregate performance from small and large jobs.

6.4 Schedulers

Here, we present various schedulers used to allocate jobs to computing resources. We start with two reference schedulers (FCFS and EFT) and then move to our contribution: three locality-aware job schedulers (named LEA, LEO and LEM). Each of these five schedulers can be used with or without

backfilling. We first present the simpler version, without backfilling, before detailing the modifications needed to include backfilling.

The role of the scheduler is to allocate a set of cores from a node to each job submitted until now. Some jobs may be started right away, while other jobs may be delayed and scheduled later: resource reservations are made for these jobs. Jobs are presented to the scheduler in the form of a global queue, sorted by job submission time. The scheduling policies are online algorithms which are called each time a job completes (making cores available) or upon the submission of a new job. Note that in accordance to our framework, each job is allocated to one or several cores of a single node.

Some of these methods use start or completion time as a way to schedule each job (Algorithms 13 and 14) while another compute a score to choose the best node (Algorithm 15) and others are opting for a mixed strategy between locality and earliest possible start time (Algorithms 16 and 17).

6.4.1 Two schedulers from the state of the art: FCFS and EFT

A widely-used job scheduler that is typically considered to be efficient is First-Come-First-Serve (FCFS), detailed in Algorithm 13. Implementing this scheduler requires to remember the time of next availability for each core. Then, for each job, we look for the first time when a sufficient number of cores is available, and we allocate the job to those cores.

Algorithm 13 First-Come-First-Serve (FCFS)

- 1: **for each** $J_i \in$ the jobs queue **do**
 - 2: **for each** $\text{Node}_k \in \mathbb{N}$ **do**
 - 3: Find smallest time t_k such that $\text{Cores}(J_i)$ cores are available on Node_k
 - 4: Select Node_k with the smallest t_k
 - 5: Schedule J_i on $\text{Cores}(J_i)$ cores of Node_k that are available starting from t_k
 - 6: Mark these cores busy until time $t_k + \text{WallTime}(J_i)$
-

FCFS is a standard baseline comparison for job scheduling. However, it is not aware of the capability of the system to keep a large data file in the memory of a node between the execution of two consecutive jobs. A first step towards a locality-aware scheduler is to select a node for each job not only based on the cores availability time, but also using the file availability time, based on the file transfer time. This is the purpose of the Earliest-Finish-Time (or EFT) scheduler, described in Algorithm 14: by selecting the node that can effectively start the job at the earliest time, it minimizes the job completion time. There are three scenarios to compute the time t'_k at which the input file $\text{File}(J_i)$ of job J_i is available on Node_k :

1. $\text{File}(J_i)$ is already in memory, then $t'_k = t_k$;
2. $\text{File}(J_i)$ is not in memory, then $t'_k = t_k + \frac{\text{Size}(\text{File}(J_i))}{\text{Bandwidth}}$;
3. $\text{File}(J_i)$ is partially loaded on Node_k : this happens when some job J_j , using the same input file, has been scheduled on other cores of the same node at time $\text{StartTime}(J_j) < t_k$ but the file transfer has not been completed at time t_k . Then the file will be available at time: $t'_k = \text{StartTime}(J_j) + \frac{\text{Size}(\text{File}(J_i))}{\text{Bandwidth}}$.

6.4.2 Data-locality-based schedulers

The previous strategies focus on starting (FCFS) or finishing (EFT) a job as soon as possible, respectively. Those are good methods to avoid node starvation and reduce queue times. However, they may

Algorithm 14 Earliest-Finish-Time (EFT)

```

1: for each  $J_i \in$  the jobs queue do
2:   for each  $\text{Node}_k \in \mathbb{N}$  do
3:     Find smallest time  $t_k$  such that  $\text{Cores}(J_i)$  cores are available  $\text{Node}_k$ 
4:     Find time  $t'_k \geq t_k$  at which  $\text{File}(J_i)$  is available on  $\text{Node}_k$ 
5:     Select  $\text{Node}_k$  with the smallest  $t'_k$ 
6:     Schedule  $J_i$  on  $\text{Cores}(J_i)$  cores of  $\text{Node}_k$  that are available starting from  $t_k$ 
7:     Mark these cores busy until time  $t_k + \text{WallTime}(J_i)$ 

```

lead to loading the same input file on a large number of nodes in the platform, only to minimize immediate queue times. Time is thus spent loading the input file multiple times. This can affect the global performance of the system by delaying subsequent jobs. We present three strategies that attempt to take data locality into account in a better way to reduce queuing times in the long run by increasing data reuse.

The first proposed algorithm, called Locality and Eviction Aware (LEA) and detailed in Algorithm 15, aims at a good balance between node availability and data locality. We consider three quantities to rank nodes:

- The availability time for computation t_k ;
- The time needed to complete loading the input file for the job on Node_k ($t'_k - t_k$);
- The time required to reload files that need to be evicted before loading the input file; this time is computed using all files in memory and considering that a fraction of these files need to be evicted, corresponding to the fraction of the memory used by the job.

The intuition for the third criterion is that if loading a large file in memory requires the eviction of many other files, these files will not be available for later jobs and may have to be reloaded. In the LEA strategy, we put a strong emphasis on data loading, in order to really favor data locality. Hence, when computing the score for each node Node_k , we sum the previous three quantities, with a weight W for the second one (loading time). In our experiments, based on empirical evaluation, we set this value to $W = 500$, incidentally roughly equivalent to the number of nodes. Note that the other two quantities usually have very different values: the availability time is usually much larger than the time for reloading evicted data. Hence this last criterion is mostly used as a tie-break in case of equality of the first two criteria.

The LEA strategy puts a dramatic importance on data loads. Hence, it is very useful when the platform is fully loaded and some jobs can safely be delayed to favor data reuse and avoid unnecessary loads. However, when the platform is not fully loaded, delaying jobs can be detrimental, as it can increase the response time for some jobs, without any benefit for other jobs. Our second proposed strategy, named Locality and Eviction Opportunistic (LEO) and described in Algorithm 16, tries to adapt based on the current cluster load: if we find some nodes that can process the job right away, we select the one that will minimize the completion time (as in the EFT strategy). Otherwise, we assume that the platform is fully loaded and we apply the previous LEA strategy, to favor data reuse.

We present a third strategy called Locality and Eviction Mixed (LEM) and described in Algorithm 17 that takes a similar approach to LEO but performs a simple mix between the EFT and the LEA strategies: when the platform is saturated (each node is running at least one job), the LEA strategy is applied, otherwise the EFT strategy is used.

Algorithm 15 Locality and Eviction Aware (LEA)

```

1: for each  $J_i \in$  the jobs queue do
2:   for each  $\text{Node}_k \in \mathbb{N}$  do
3:     Find smallest time  $t_k$  such that  $\text{Cores}(J_i)$  cores are available
4:     Find time  $t'_k \geq t_k$  at which  $\text{File}(J_i)$  is available on  $\text{Node}_k$ 
5:      $\text{LoadOverhead} \leftarrow t'_k - t_k$ 
6:     Let  $\mathfrak{F}$  be the set of files in the memory of node  $\text{Node}_k$  at time  $t_k$ 
7:      $\text{EvictionPenalty} \leftarrow (\sum_{F_j \in \mathfrak{F}} \text{Size}(F_j) \times \text{Size}(\text{File}(J_i)) / M) / \text{Bandwidth}$ 
8:      $\text{score}_{\text{Node}_k} \leftarrow t_k + W \times \text{LoadOverhead} + \text{EvictionPenalty}$ 
9:   Select  $\text{Node}_k$  with the smallest  $\text{score}_{\text{Node}_k}$ 
10:  Schedule  $J_i$  on  $\text{Cores}(J_i)$  cores of  $\text{Node}_k$  that are available starting from  $t_k$ 
11:  Mark these cores busy until time  $t_k + \text{WallTime}(J_i)$ 

```

Algorithm 16 Locality and Eviction Opportunistic (LEO)

```

1: for each  $J_i \in$  the jobs queue do
2:   for each  $\text{Node}_k \in \mathbb{N}$  do
3:     Find smallest time  $t_k$  such that  $\text{Cores}(J_i)$  cores are available
4:     Find time  $t'_k \geq t_k$  at which  $\text{File}(J_i)$  is available on  $\text{Node}_k$ 
5:     if  $t_k = \text{current\_time}$  then
6:        $\text{score}_{\text{Node}_k} \leftarrow t'_k$ 
7:     else
8:        $\text{LoadOverhead} \leftarrow t'_k - t_k$ 
9:       Let  $\mathfrak{F}$  be the set of files in the memory of node  $\text{Node}_k$  at time  $t_k$ 
10:       $\text{EvictionPenalty} \leftarrow (\sum_{F_j \in \mathfrak{F}} \text{Size}(F_j) \times \text{Size}(\text{File}(J_i)) / M) / \text{Bandwidth}$ 
11:       $\text{score}_{\text{Node}_k} \leftarrow t_k + W \times \text{LoadOverhead} + \text{EvictionPenalty}$ 
12:   Select  $\text{Node}_k$  with the smallest  $\text{score}_{\text{Node}_k}$ 
13:   Schedule  $J_i$  on  $\text{Cores}(J_i)$  cores of  $\text{Node}_k$  that are available starting from  $t_k$ 
14:   Mark these cores busy until time  $t_k + \text{WallTime}(J_i)$ 

```

Algorithm 17 Locality and Eviction Mixed (LEM)

```

1: for each  $J_i \in$  the jobs queue do
2:   if each node is running at least one job at  $\text{current\_time}$  then
3:      $\text{LEA}(J_i, \mathbb{N})$ 
4:   else
5:      $\text{EFT}(J_i, \mathbb{N})$ 

```

6.4.3 Adding backfilling to all strategies

As mentioned above, backfilling has been proposed to increase the performance of production cluster job schedulers, by allowing jobs with lower priority to be scheduled before jobs with higher priority. In our setting, the priority is directly linked to the submission order: if J_i is submitted before J_j , then J_i has a higher priority than J_j . In order to avoid jobs being perpetually delayed, bounds have to be set on how already-scheduled jobs can be affected by backfilling. As discussed above, conservative backfilling is the most restrictive version and one of the most commonly-used strategies to improve cluster utilization. It forbids any modification on the resource reservations of high-priority jobs: a job may be scheduled before other jobs that appear earlier in the queue, provided that it does not impact the starting time of these jobs.

For each of the previous scheduling strategies, we consider a variant using conservative backfilling (suffixed by -BF). To add backfilling, Algorithms 13, 14, 15 and 16 have to be modified: we change the choice of the earliest time when resources are available for a job (Line 3). Instead of considering the time at which cores are (indefinitely) available, we look for an availability time window starting at t_k that is long enough to hold the job. Specifically, we change Line 3 into:

3': Find smallest time t_k such that $Cores(J_i)$ cores are available from t_k until $t_k + WallTime(J_i)$ on $Node_k$

Note that this requires the schedulers to store the whole occupation profile of each core (with availability and unavailability times), whereas the version of each scheduler without backfilling simply requires the time of last job completion on each core.

6.5 Experimental settings

In order to perform our experimental evaluation, we ran simulations using information from a real platform, coupled with the same platform logs of jobs that have been historically submitted. Changing the scheduling strategy of a production cluster for the purpose of scheduling research would be disruptive to the community that uses this cluster. For this reason, we choose to run simulations based on historical logs from the cluster. The combination of historical logs and platform information provides a high-fidelity representation of real-world user behavior. In this section, we describe our platform (Section 6.5.1), the workloads (Section 6.5.2), how we used historical logs (Section 6.5.3) and our simulator (Section 6.5.4).

6.5.1 Platform description

UPPMAX (for Uppsala Multidisciplinary Center for Advanced Computational Science) is Uppsala University's resource of high-performance computers. It includes Rackham¹, the platform we used as a reference. Rackham is a university cluster shared by several research labs. It contains 9720 cores spread over 486 nodes. Each node has two 10-core Intel Xeon V4 CPU at 2.20 GHz/core. 450 nodes have 128 GB of RAM, 32 have 256 and 4 have 1024 GB of RAM. To avoid an additional constraint, while maintaining a model close to the real cluster, we consider that the platform is made of 486 homogeneous nodes of size $M = 128 GB$. Based on past job submissions, we studied the characteristics of data-intensive jobs.

¹<https://www.uppmx.uu.se/resources/systems/the-rackham-cluster/>

6.5.2 Workloads description

Actual workloads on the Rackahm cluster and other HPC resources shared by a large number of users with diverse needs include projects that start a burst of jobs on the same file just a few thousand core hours in length, then sit idle for a long time processing the results, and then start another such burst. These files have sizes that can range from a few megabytes to hundreds of gigabytes. We have observed such behavior on our cluster. For example, computational tasks in bioinformatics often require large input files to be loaded onto a cluster. A good example comes from members of the Department of Organismal Biology at the Evolutionary Biology Center at Uppsala University. They are recovering and analyzing ancient human genomes to determine migration routes [73]. DNA samples from ancient humans decay over time. Thus, computationally expensive methods must be used to reconstruct these damaged genomes in order to reconstruct potential migration routes². These DNA sequences are large and must be loaded onto the cluster to perform such analysis. The cluster has also been used for taxonomic identification of DNA fragments. To perform such identification, the Kraken2 tool³ is often used together with an input database⁴, whose size vary from a few GB up to 800 GB. We also noticed the use of genome alignment tools that require to store large text files⁵, and it is likely that they need to load multiple of them, resulting in high I/O demands. Now that we have identified the presence of I/O intensive jobs in our workloads, we can use historical logs to replicate this behavior.

6.5.3 Usage of real cluster logs

The logs we use contain historical data on jobs, namely their exact submission time, their requested walltime, their actual duration, the number of cores they required and the corresponding user's name. Since explicit data dependencies are not encoded in SLURM job specifications, we do not have access to the actual input files of these jobs. We thus create an artificial data dependency pattern that replicates user behaviors. Each job uses exactly one input file. As an example, let us consider two jobs, J_i and J_j . These jobs share their input file if they match the three following requirements:

1. $Cores(J_i) = Cores(J_j)$, i.e., they request the same number of cores;
2. J_i and J_j are submitted by the same user, which means that they are from the same experimental campaign;
3. J_i and J_j are submitted within an 800 seconds time frame. We consider this timeframe to be a reasonable amount of time for a user to submit all of their jobs using the same input file.

Otherwise, we consider that J_i and J_j are using distinct input files. In theory, two users could share the same file. However, because they are using subsets of different databases, it is very unlikely that two users would work on the same project using the same databases, therefore we ignore this possibility.

We consider that these jobs are dedicated to processing their input file. Hence, the more cores the job requests, the larger its input file. This allows to estimate the size of files as follows:

$$Size(File(i)) = \frac{Cores(J_i)}{20} \times 128 GB$$

²<https://www.uppmx.uu.se/projects-and-collaborations/research-at-uppmx/arielle-munters/>

³<https://github.com/DerrickWood/kraken2/tree/master/src>

⁴<https://benlangmead.github.io/aws-indexes/k2>

⁵For example for primates: https://ftp.ensembl.org/pub/release-107/maf/ensembl-compara/multiple_alignments/10_primates.epo/

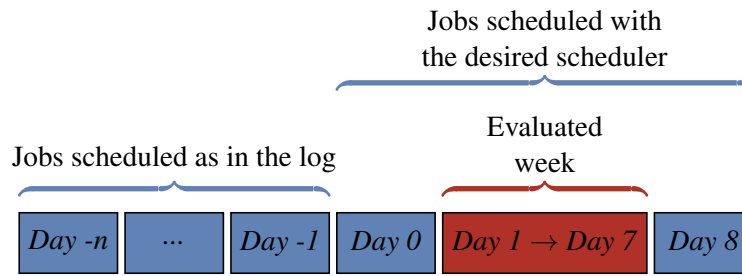


Figure 6.2: Methodology followed to schedule and evaluate jobs from a specific week while avoiding edge effects.

Indeed, if a user needs 128 GB of memory but would request only 1 of the 20 available cores, that user would block the node for all other users, for lack of remaining memory. Consequently, a user who needs the whole memory of a node will reserve all its cores. Similarly, if a user needs a fraction of the memory, the user can reserve a fraction of the cores on the machine.

The utilization levels in the log of the platform are typically high ($> 90\%$), but not fully consistently. The vast majority of jobs on these resources are single-node jobs and thus fit in our framework. The few multi-nodes jobs are not representative of the typical usage of the platform, and are replaced by as many single-node jobs as necessary to represent the same workload. We notice that jobs durations extend up to 10 days, while some jobs only last a few minutes. Even if the workload is not homogeneous, it is representative of the real usage from an actual user community including, but not exclusively consisting of, many subfields of the life sciences with highly data-dependent workflows.

In our experiments we evaluate our schedulers week by week. Our workload is constituted of 51 weeks, ranging from January the 3rd 2022 to December the 25th 2022. To target different scenarios but avoid scheduling the whole year, we randomly selected 12 weeks and extracted the jobs submitted within these weeks from the logs. The most important for a user is that all the jobs that have been submitted at once are finished as soon as possible. Thus we introduce the notion of *user session*. The stretch (defined in Section 6.3) of a *user session* is the sum of the stretch of jobs submitted by a user in a 5-minute window. Our evaluation is based on three metrics: the stretch of each *user session*, the total time spent waiting for a file to become available, as well as the total core time. We evaluate the proposed schedulers on these metrics and compare them to FCFS and EFT, with and without backfilling. Over these 12 weeks of workloads, we scheduled 1 986 496 jobs and evaluated 1 493 151 of them. There have been 1083 distinct users and 136 404 *user sessions*.

To simulate these jobs in a realistic, steady-state operation of the platform, we consider both jobs submitted before and after the week under consideration. We proceed as illustrated in Figure 6.2. We choose a week we want to evaluate (in red on the figure). To avoid edge effects from starting and ending a schedule on an empty cluster, we add jobs submitted the day before (*Day 0*) and after (*Day 8*). The desired scheduler is used at the start of *Day 0*. Thus, the evaluated jobs do not suffer from edge effects. On top of that, we schedule jobs submitted before *Day 0* on the actual node that was used for this job from the logs information. This replicates the exact state of the cluster as it was that day. n is chosen depending on the maximum observed duration of a job on the cluster. In our case, users typically do not submit jobs that last longer than a week. So we choose $n = 7$, which covers all jobs submitted before the start of *Day 0*.

6.5.4 Simulator description

We used simulations to avoid disrupting users on a production cluster, but our strategies could be implemented on the SLURM workload by asking the users to flag the input files they are using. All strategies as well as the two baselines (FCFS and EFT) have been implemented on a simulator that we developed⁶. The simulator is a simple event-based simulator focusing on data locality. It is made in Python and follows these principles:

- As in real systems, the schedule is re-computed entirely at each unpredicted event (job submission or job termination before the walltime);
- The scheduler is only aware of jobs that have been submitted before the current time;
- When a job is submitted, the scheduler knows its submission time, requested walltime, number of cores required and input file name;
- Scheduling a job consists in assigning to the job a start time, a node, and as many cores on that node as requested.

In the context of our specific needs, we chose to develop our own batch simulator instead of relying on an existing solution such as Batsim [54]. One of the primary reasons was the lack of coverage for our specific use cases in existing solutions: data reuse and data loading prior to computation. By developing our own simulator, we were able to tailor it specifically to add load time before a job starts and allow a job to bypass this time by reusing an existing file. Moreover, we wanted to explore and compare various scheduling and backfilling algorithms to identify the most efficient approach for our workload. By developing our own simulator, we had complete control over the design and could easily experiment with different strategies. Finally, given the time constraints of the three-month internship in Sweden, developing our own batch simulator was a practical decision that allowed us to focus on testing the schedulers. Future integration with existing simulators remains a possibility, although time consuming, which would provide more reliable results.

6.6 Experimental evaluation and analysis

We evaluated our schedulers on 12 distinct randomly selected weeks. These weeks have different workloads, with different levels of cluster saturation. We start with a low utilization case.

6.6.1 Results on an underutilized cluster

Figure 6.3a depicts the distribution of the improvement of each *user session* stretch, compared to FCFS, obtained by each strategy on week 40⁷. In other words we represent $Stretch_{FCFS}/Stretch_{Scheduler}$ for each *user session* and each scheduler. In this figure, the horizontal black dotted line corresponds to no improvements from FCFS, which is the situation where the sum of the queue time and transfer time of the *user session* is the same as with FCFS. The percentages next to the scheduler names denotes the fraction of *user sessions* where the improvement was at least 1% different from 1: only these *user sessions* are depicted on this figure, for the sake of readability. The solid lines are the median and the triangles are the average improvements. Because outliers bring a huge improvement, the averages

⁶Code, anonymized logs and the methodology followed to randomly select our evaluated weeks are available at: <https://github.com/userdoubleblind/Locality-aware-batch-scheduling>

⁷The 40th week of 2022: jobs submitted between the 3rd and the 9th of October 2022.

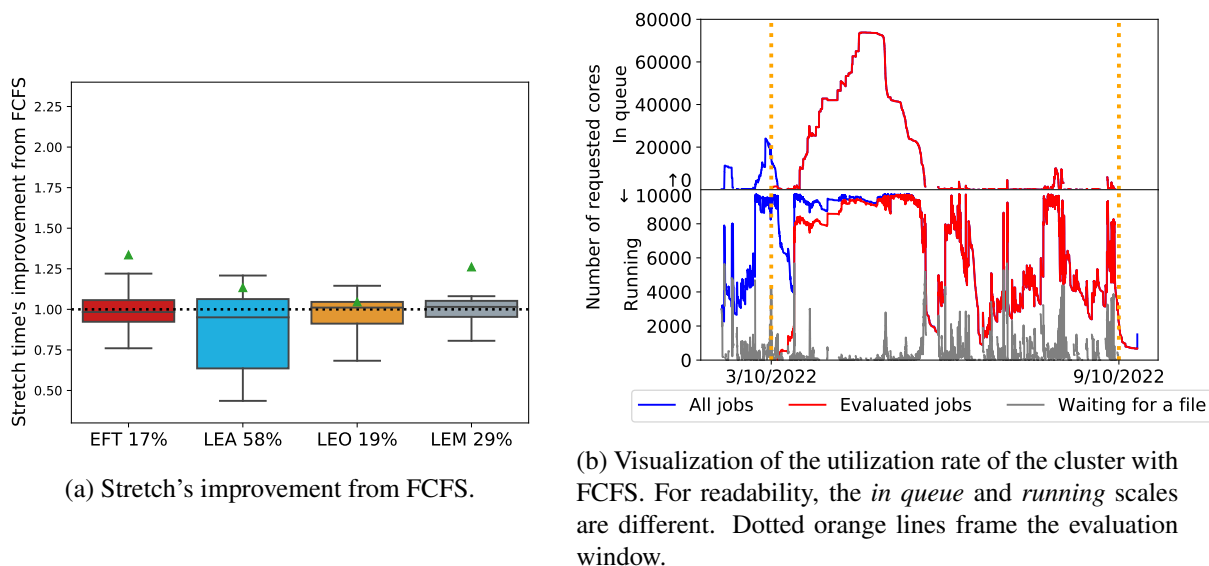


Figure 6.3: Results on the workload of week 40.

are usually higher than the medians. Because of such extreme outliers, we are more interested in median values. An improvement above 1 is a speed-up. An improvement at 0.5 means that the sum of the queue times and transfer times for this *user session* was two times larger with the scheduler than with FCFS. For all boxplots shown in this chapter, the box contains results within the [25%,75%] range, while whiskers are drawn at 12.5% and 87.5%.

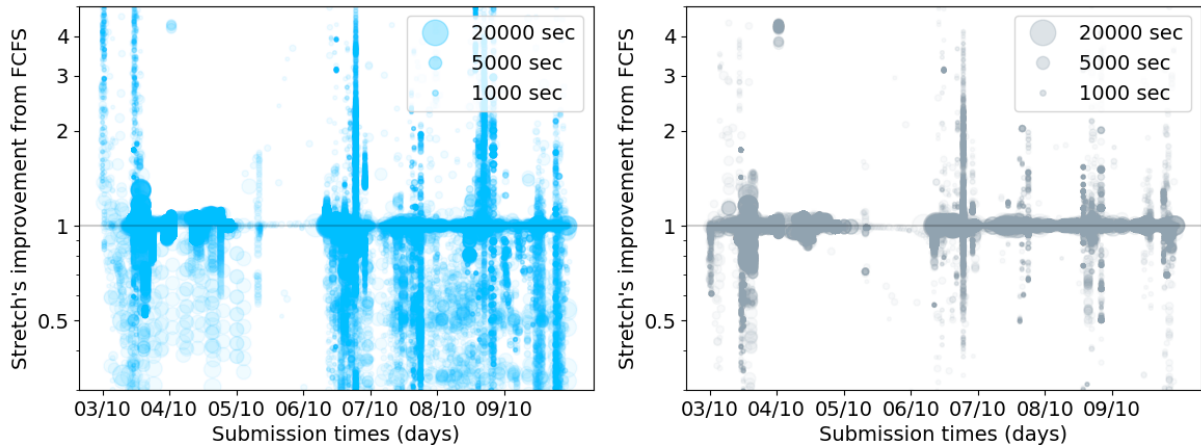
Figure 6.3a shows that on this workload, all schedulers result in almost no improvement, nor degradation of the *user session* stretch, apart from LEA that degrades performance for significantly more than 50% of *user sessions*.

To understand the issues LEA encounters on this workload, we need to study the cluster's usage over time. Figure 6.3b shows the cluster usage when using FCFS. The vertical axis represents the number of requested cores either running on a node (lower half, the maximum is the total number of cores on our cluster: 9720) or in the queue of jobs waiting to be executed (top half, where the number of requested cores can be much higher than the cluster's capability). The blue line shows the number of cores (used or requested) for all jobs. The red lines show the number of cores used by the evaluated jobs, i.e., those jobs that have their submission time within our evaluation window. Thus, this forms a subset of the set indicated by the blue line. If a line is present in the top half, it means that some jobs could not be scheduled and are thus waiting in the queue of available jobs. A node may have some available cores but not enough to accommodate some jobs with large requirements. This explains why the waiting job queue may not be empty even if the lower half does not reach the maximum. The gray line represents the number of cores currently loading a file. Lastly, the orange lines delimits the submission times of our evaluated jobs, in other words it is our evaluation window.

By looking at the top half of Figure 6.3b we understand that the job queue is empty for more than half of the evaluation window. In this situation, FCFS and EFT are very efficient. The earliest available node is in most cases a node that can start the job immediately, explaining the mean stretch close to 1 in Figure 6.3a. LEO is a strategy that uses the earliest available time t_k of a node to decide if it should compute a score like LEA, that puts a large weight on transfer time or weigh equally t_k , the transfer and eviction duration. Thus, on underused clusters, LEO has a behavior close to EFT, while trying to minimize evictions. Similarly, LEM switches between EFT and LEA depending on the cluster's usage.

Schedulers	EFT	LEA	LEO	LEM
Reduction from FCFS	0.0%	14.8%	0.8%	3.8%

Table 6.1: Percentage reduction in time spent waiting for a file to be ready before starting the computation, relative to the total wait time of FCFS.

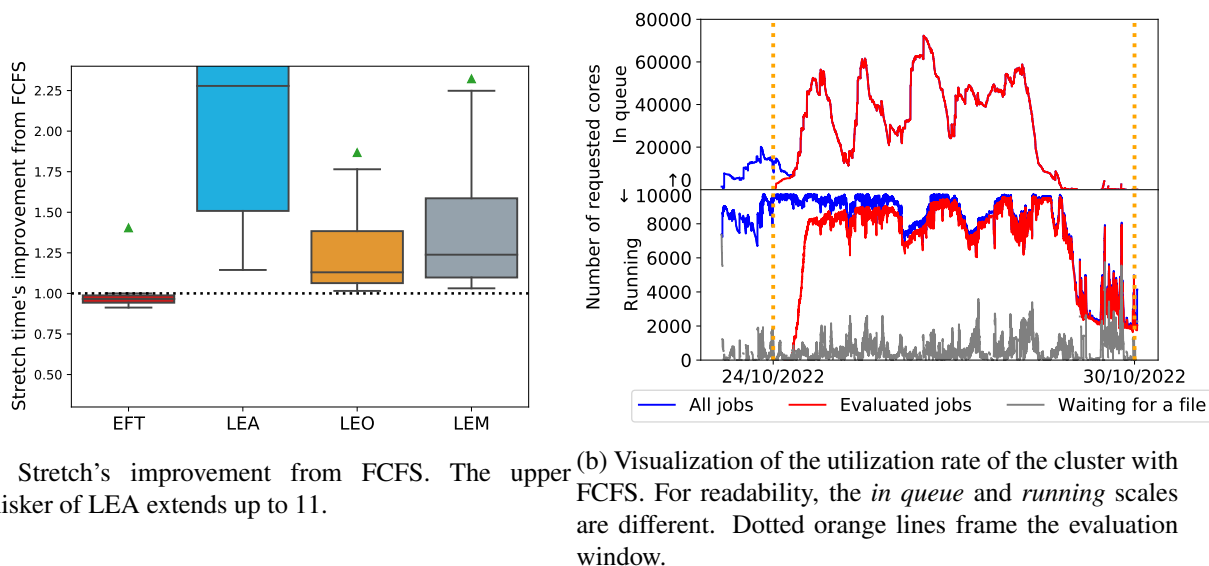


(a) With LEA. The max and min values are 543 and 0.003. (b) With LEM. The max and min values are 543 and 0.007.

Figure 6.4: Stretch times of each user session from week 40 compared to FCFS. Cropped at 0.3 and 5.

On this particular workload the cluster’s usage is under 100% more than half the time, so LEM behaves similarly to EFT. On the contrary, LEA favors data re-use over an early start time for a job.

Table 6.1 shows the reduction of the amount of time spent waiting for a file to be ready before starting the computation, relative to the total waiting time of FCFS. In this case, LEA is the only strategy that decreases this waiting time by re-using input files. It will create a queue of jobs that already have a valid copy of their file loaded on a node, waiting to be able to re-use their input. On an underutilized cluster, this creates situations where some nodes are idle while submitted jobs wait in a queue, resulting in increased queue times. The consequences for the stretch are immediately distinguishable on Figure 6.4a. This plot shows for each *user session*, the ratio of its stretch with LEA over its stretch with FCFS (hence a value above 1 means LEA improves the stretch). The size of a circle is proportional to the *user session*’s duration. On the workload of week 40, we observe several “columns” of jobs submitted at the same time (hence sharing the same file with large probability) with a ratio lower than 1, hence a worst performance with LEA. This means LEA is waiting to re-use the files before starting the jobs, whereas FCFS paid the cost of loading the file on other nodes, but started the jobs earlier than LEA, leading to shorter completion times. From Figure 6.4b concerning LEM, we see that most jobs have an improvement close to 1, showing that LEM does not fall into LEA’s pathological case. To summarize, on an underutilized cluster, LEA’s focus on locality does not allow optimal utilization of the cluster, while LEO and LEM, thanks to their flexibility, achieve performance close to EFT.



(a) Stretch's improvement from FCFS. The upper whisker of LEA extends up to 11. (b) Visualization of the utilization rate of the cluster with FCFS. For readability, the *in queue* and *running* scales are different. Dotted orange lines frame the evaluation window.

Figure 6.5: Results on the workload of week 43.

Schedulers	EFT	LEA	LEO	LEM
Reduction from FCFS	0.4%	20.8%	1.0%	11.4%

Table 6.2: Percentage of reduction of the amount of time spent waiting for a file to be ready before starting the computation, relative to the total waiting time of FCFS on week 43.

6.6.2 Results on a saturated cluster

We concentrate here on week 43, which we identify as a workload saturating the cluster. Indeed, Figure 6.5b shows a queue of several thousands requested cores for the whole duration of the evaluated week. In this situation, re-using files has a significant impact on the queue times. As you can see on Table 6.2, LEA and LEM greatly reduces the transfer time. This is confirmed by Figure 6.5a: more data re-use is associated with better improvement for LEA and LEM. On a saturated cluster, filling all cores with the first jobs of the queue, like FCFS does, is not crucial. It is more beneficial to group jobs using the same file. The first few jobs have a longer queue than with FCFS, but, over time, re-using files causes a snowball effect that reduces the queue times of all subsequent jobs. Moreover, the queue contains enough jobs to fill all the nodes even when grouping them by input file. We thus avoid the pathological case outlined in Section 6.6.1. In this case, the strategy of LEA, also found in LEM, allows to greatly reduce the stretch of each *user session*. This is confirmed by Figure 6.6: very few jobs have a worse stretch than FCFS and a large amount of jobs display an improvement above 2.

6.6.3 Complete results

We report here the results of the 5 schedulers on the 12 evaluated weeks. Figures 9 to 12 represent the aggregated results of the ratio of each 136 404 *user session* stretches over 12 weeks and 1 986 496 jobs relative to FCFS.

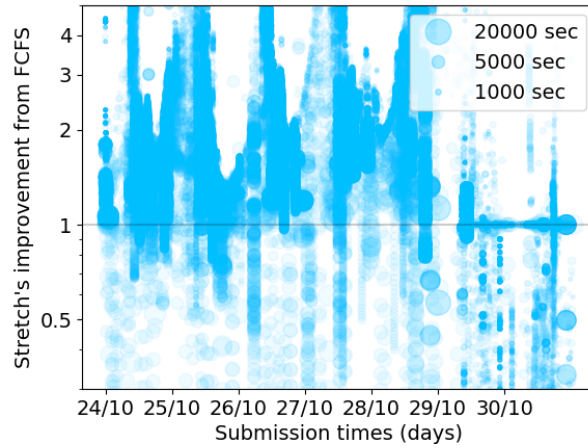
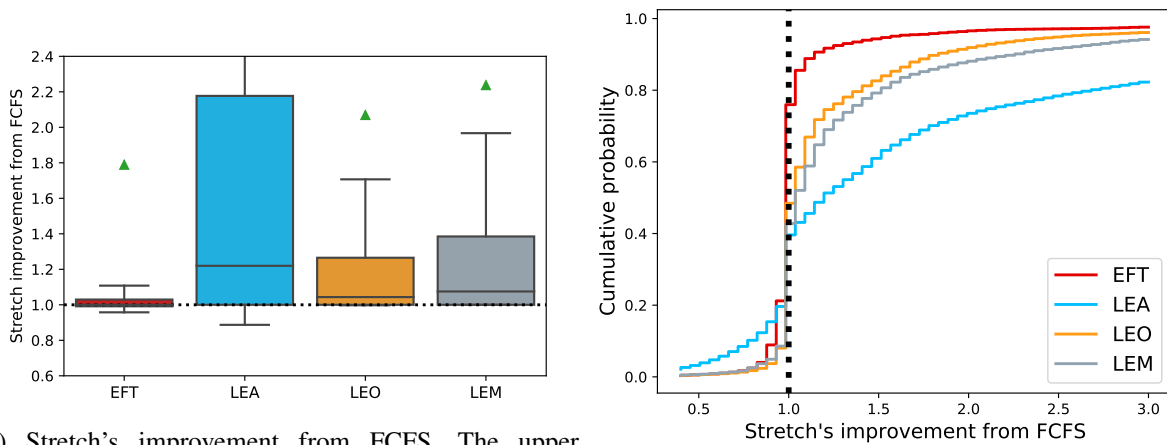


Figure 6.6: Stretch times of each user session from week 43 using LEA compared to FCFS. Cropped at 0.3 and 5. The max and min values are 314 and 0.006.



(a) Stretch's improvement from FCFS. The upper whisker of LEA extends up to 5.

(b) Empirical distribution function of the stretch's improvement from FCFS.

Figure 6.7: Results without backfilling on all evaluated weeks.

Schedulers	EFT	LEA	LEO	LEM
Reduction from FCFS	0.9%	17.1%	0.9%	7.1%

Table 6.3: Percentage of reduction of the amount of time spent waiting for a file to be ready before starting the computation, relative to the total waiting time of FCFS on all workloads.

Schedulers	EFT	LEA	LEO	LEM
Reduction from FCFS	0.01%	0.37%	0.02%	0.10%

Table 6.4: Percentage of reduction of the total core time used on all workloads relative to FCFS.

Results without backfilling As in the previous results, we observe on Figure 6.7a that EFT does not bring any real improvement compared to FCFS. EFT takes into account file transfers when scheduling jobs but is a less aggressive strategy than LEA or LEM, and the savings in data transfers are only 0.9% (see Table 6.3). EFT is not able to see that a large number of jobs using the same file should be scheduled on the same node: even if it would generate more queue time, the overall execution time would be lowered thanks to data re-use. On the contrary, LEA has the largest median improvement.

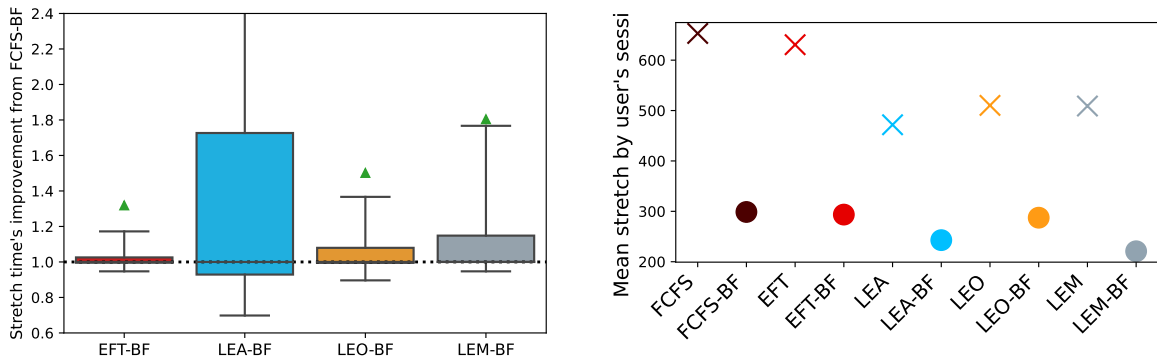
We can explain the larger median value for LEA from the good performance on heavily saturated clusters (see Section 6.6.2). Re-using the same files is not detrimental to the filling of all the nodes because there are enough jobs to cover all nodes. A large decrease (see Table 6.3) in the time spent waiting for a file greatly reduces the stretch of each job. LEM has a lower median, however, as can be seen on Figure 6.7a, at least 87.5% of its results are above 1, i.e., an improvement, whereas for LEA, only approximately 75% of the results are above 1. LEM is a more versatile strategy and offers higher sustained performance on non-saturated cluster at the cost of fewer extreme improvements on heavily saturated clusters.

Table 6.4 shows the reduction in terms of total core time (i.e. the core time used on each core of each node over the 12 weeks). LEA reduces the total core time used by 49 000 core hours over the 12 weeks (approximately 5h of the whole platform). Although it is a small percentage reduction, it can be of interest on large scale clusters that are highly demanding in terms of electrical resources.

From the same data shown on Figure 6.7a, we plot an empirical distribution function on Figure 6.7b. EFT's low variance is clearly visible in the sudden jump in probability around an improvement of 1. It is interesting to note that for LEA, 20% of the results are above an improvement of 300%. In addition, thanks to its change of scheduling strategy between EFT and LEA, LEM clearly reduces performance losses on the left of the black line.

Thus, LEA is the strategy that leads to more significant improvements: it is able to compute jobs between 1 and 2 times faster in 50% of the cases and between 2 and 6 times faster in 25%. It is slower for only 25% of the jobs, and 12.5% of those are within a 0.15 slow-down. LEO is a more sustained strategy with 87.5% of its results with an improvement compared to FCFS, which shows that our opportunistic strategy is much more consistent, while still having great improvements in some cases. LEM is more versatile. It leads to an improvement in 87.5% of the results, with a speed-up of at least 7.5% in 50% of the results.

Results with backfilling Figure 6.8a shows the results with the backfilling version of our schedulers and compared to FCFS with backfilling (FCFS-BF) on all workloads. We notice that our schedulers have smaller improvements with backfilling, because FCFS-BF already performs much better than FCFS.



(a) Stretch's improvement from FCFS. The upper whisker of LEA extends up to 4. (b) Average *user session* stretch with (circles) and without (crosses) backfilling. Cropped at 200.

Figure 6.8: Results with backfilling on all evaluated weeks.

Figure 6.8b shows that the difference of performance with (circles) or without (crosses) backfilling is much higher for FCFS-BF and EFT-BF. Our proposed strategies do not benefit from backfilling as much as FCFS-BF does for two reasons. Firstly, even if we consider data locality when backfilling, trying to fill a node as much as possible and optimizing data re-use are two contrary goals. Backfilling a job can compromise a re-use pattern that was planned by our locality-aware strategy, thus reducing the total amount of re-used files. Secondly, our strategies are already able to nicely fill the nodes without needing backfilling. Grouping jobs by input file implies that similar jobs end up on the same nodes. Jobs having the same duration and number of requested cores can much more easily fill a node to its fullest than a completely heterogeneous set of jobs. Consequently, FCFS-BF and EFT-BF already naturally benefit from increased data locality thanks to backfilling, leading to a reduced benefit in using LEA-BF, LEO-BF or LEM-BF. Compared to FCFS-BF, our strategies still reduce the total queue time with backfilling but the difference is less significant.

Figure 6.8a shows that the improvement of EFT-BF compared to FCFS-BF is not significant. Out of our four heuristics, LEM-BF is the best compromise here. It is better than FCFS-BF in more than 75% of the cases, with 12.5% of those results above an improvement of 1.8. Among the slow-downs, only 12.5% are worse than 0.95.

6.7 Conclusion on locality-aware batch scheduling

Research on topics such as ancestry reconstruction from DNA samples requires high computational power coupled with large input data. When using computing clusters, users in such area commonly submit dozens of jobs using the same multi-GB input file.


Batch schedulers are key components of computing clusters and are designed to improve resource utilization and reduce job response time. Classical job schedulers are unaware of data locality and thus fail to re-use data on nodes in data intensive workloads. We have studied how one may improve their performance by taking job input file into account: we have proposed three new locality-aware strategies, named LEA, LEO and LEM, capable of increasing data locality by grouping together jobs sharing inputs.

The first one has a major focus on data locality, while the other two target a balance between data locality and load balancing. We have performed simulations on logs of an actual cluster. Our results show that LEA significantly improves the mean waiting time of a job, especially when the cluster is under a high computing demand. Without backfilling, LEA is better than our baseline in 75% of the

cases (50% of the cases with backfilling). Our strategy called LEM is the best compromise. LEM is better than the baseline in more than 75% of the cases with or without backfilling, with a median improvement of 7.5% compared to our baseline without backfilling.

Conclusion and Perspectives

Summary

 HIS WORK is motivated by a well-known pathology of modern supercomputers: limited memory. Indeed, modern processing units can achieve unprecedented throughput, but their nearest memory is always limited in size due to space, cost, or access speed constraints. This would not be a problem if the main memory containing all the data was easily accessible. It is not; the bandwidth between a CPU and a disk is low, and the bandwidth between a GPU and a CPU is too slow compared to the GPU's computational speed. The execution of large scientific applications on a supercomputer is then reduced to waiting for data transfers for a large fraction of the total execution time. The goal of this thesis is to reduce the time required to process scientific applications on modern supercomputers under memory constraints. To unlock the full potential of supercomputers, we have been working on two different hardware levels: *runtime* and *batch* systems.

Runtime schedulers For runtime systems, our objective was to build a generic scheduler capable of reducing data transfers and increasing performance by partitioning and scheduling a set of tasks (with and without dependencies) sharing data on one or more processing units with limited distributed or shared memory. Since this is a goal with multiple parameters that require their own optimizations, we first start with a simple model. Our first model expresses the application in terms of a bipartite graph of independent tasks computed by a single processing unit. Thanks to this simplification, we provided proofs of an optimal eviction policy and the complexity of finding an optimal task order. Our full model includes multiple processing units, heterogeneous weights, and dependencies.

From the simplified model, we built our first scheduler: Hierarchical Fair Packing (HFP). HFP begins with an initial packing phase to create packages of tasks that share a lot of data and whose required data fits exactly into the memory of the processing units used. A package is an aggregation of subpackages that have been carefully merged to achieve data reuse across successive tasks. After this initial packing phase, HFP continues packing until there are as many packages as processing units, even if it means exceeding the memory limit. A package is then assigned to each processing unit. In addition to this task partitioning, the tasks within each package are carefully ordered thanks to the way the packages were created. Because it schedules the entire set of tasks in advance, we were able to endow HFP with an optimal eviction policy. We tested HFP on 2D matrix multiplication and its variants with sparsity or randomization, on 3D matrix multiplication and tasks from the Cholesky factorization, with one or several GPUs. On the 2D matrix multiplication, HFP achieves near-optimal performance, with one or two GPUs. This is explained by what the packages produce: a lot of data reuse and distributed transfers

over time, allowing a high overlap between communications and computations. For the other applications, HFP offers the best results compared to our competitors: adapted strategies from the literature and a state-of-the-art runtime scheduler. However, this sophisticated scheduling does not come without a cost: complexity. The complexity comes from the packing phase, which requires looking at each task input data. With a lot of parallelism and a large number of tasks available at once, the cost of packing becomes non-negligible. We have tested HFP on 2D and 3D matrix multiplication, applications with a large number of tasks and a high degree of parallelism. It thus exacerbates the complexity of HFP. However, not all applications have a large number of tasks ready at once. A progressive task submission is common for many applications [9]. With progressive task submission, the number of tasks available at a given time is less important, thus reducing the computational cost of HFP packing. However, we would still need to make HFP incremental, i.e., able to dynamically add new ready tasks to an existing schedule, in order to adapt to an application with a progressive task submission. We outline other methods to mitigate the complexity of HFP in the perspectives. Thus, HFP is a great offline scheduler and partially meets our target, since it can perform well under memory constraints. However, it cannot become a dynamic scheduler, which is a requirement of our main goal, since it is needed to tackle linear algebra applications with dependencies.

With the complete model, we built a scheduler that would overcome this complexity constraint and manage task sets with dependencies: Dynamic Data Aware Reactive Task Scheduling (DARTS). DARTS considers data movement before anything else. The strategy considers, for each processing unit, the state of its memory and uses this information to select the data that would be most beneficial to load. For DARTS, the most beneficial data to load enables many tasks to be computed without additional data movement, but it also considers fast data transfers, progress along the critical path, and allowing processing units to work on different sets of data. Tasks associated with that data are then queued for execution. DARTS can contradict priorities to favor locality when it brings benefits, but also recognize situations where following the critical path is more important. In addition, DARTS has an eviction policy that works in synergy with its schedule. DARTS has been tested on 2D, 3D and sparse matrix multiplication as well as Cholesky and LU factorization. Similar to HFP, DARTS achieves near perfect performance when scheduling time is not considered. When it is taken into account, DARTS is the best overall solution for all of the cited applications. However, a lack of performance has been observed when computing small workloads with many GPUs. DARTS distributes the workload across the different GPUs as much as possible. However, for small workloads, it is preferable to increase data reuse, even if it means leaving some GPUs idle. DARTS fulfills our initial target of reducing the execution time of task-based applications under memory constraints, as it generally exhibits higher throughput than its competitors. DARTS also satisfies our generic requirement since it is generic enough to perform well with a single or multiple GPUs, with CPU cores, with any memory constraint, on a wide range of linear algebra applications, and works without full knowledge of the task graph.

Four other steps were required to achieve the results presented above. First, we added a support for custom eviction policies in STARPU. Second, we developed a visualization tool for 2D, 3D matrix multiplication and Cholesky factorization. Third, we gained deeper insight into the performance of our competitors. For example, we discovered that a random task submission positively affects the performance of DMDAR and highlighted that the default LRU eviction policy causes a pathological case under memory constraint. Fourth, we have built schedulers inspired by theoretical studies that are aware of the peculiarities and constraints of practical implementation in a runtime system.

Batch schedulers Batch systems also share a similar motivation: to improve the execution time of scientific applications on supercomputers. Batch schedulers are important parts of computing clusters, and aim to improve resource utilization by carefully assigning jobs to compute nodes. We identified

a use case where common batch schedulers lack locality techniques: data-intensive workloads. Such workloads include, for example, DNA reconstruction, where scientists typically submit hundreds of jobs using the same input files, sometimes several gigabytes in size. We have developed three strategies called LEA, LEO, and LEM (for Locality and Eviction Aware, Opportunistic, or Mixed) which are designed to increase data reuse by applying locality techniques.

LEA strongly favors data locality. LEO and LEM balance locality and load balancing. To evaluate them, we developed a batch simulator and used logs of real jobs submitted to a cluster. Our experimental evaluation over two million jobs shows that LEM is the best compromise, as it reduces the average wait time of a job in 75% of cases, with or without backfilling.

Future works

We now review future work for both our runtime and batch schedulers.

Runtime schedulers

Handle multiple MPI nodes HFP and DARTS currently handle only a single MPI node. The complexity of HFP would not be usable with a large number of nodes that require a fast response time from the scheduler to avoid starvation. Here we suggest ways to use our DARTS scheduler on multiple MPI nodes by applying it on top of the nodes, inside each node, or both. DARTS should be used differently at a small (less than ten nodes) or large scale (hundreds of nodes). At a small scale, an instance of DARTS would be used at each memory level: one would distribute the tasks and data to the various nodes, and each node would be equipped with an instance of DARTS to distribute and order the task on each processing unit. This would work well with nodes equipped with GPUs, as they are generally not too numerous in a single node. However, within nodes mainly composed of CPUs, DARTS would not be able to assign the tasks to the many individual cores without getting hit by starvation issues. Thus, in such a case, we aim to couple DARTS with other schedulers, such as work-stealing algorithms, to take advantage of all their features. Since CPUs operate with a shared memory, DARTS does not distinguish two CPU cores sharing the same memory. So, the goal would be to use DARTS to partition the task set across the different shared memory NUMA nodes, since they are not too numerous within a node, and then a locality-aware work-stealing policy would use its load balancing capability to carefully order the tasks onto the different individual cores. At a large scale, with possibly hundreds of nodes, the complexity of DARTS could become an issue because the response time would not be fast enough to serve all the workers. One possible solution is to use a static workload partitioning, which would assign a subset of tasks to each node, and then an instance of DARTS on each node would manage the memory constraint of each processing unit.

Manage the complexity of HFP As we mentioned, HFP's complexity makes it difficult to use on highly parallel independent task sets. However, it has high performance if we ignore its complexity. Several solutions can help to deal with the complexity of HFP. For example, HFP can be used offline to determine a schedule for a large matrix multiplication of fixed size. If the matrix size and the processing units used are known in advance, HFP can generate a task order that increases performance. The complexity of HFP can also be handled on a heterogeneous node. Since GPUs are much faster than CPUs, they are usually used for the vast majority of tasks in a linear algebra application, leaving the CPUs idle most of the time. HFP could take advantage of this by quickly distributing a small set of tasks to the GPUs while the idling CPUs are used to compute the rest of the schedule. Hopefully, before the GPUs finish their initial sets of tasks, the CPUs will have computed the full schedule and can then assign the

packages to the GPUs. However, such a solution would be difficult to implement on very large systems with dozens of GPUs, as they would be too fast for HFP's schedule. Finally, the complexity of HFP is not an issue with large tile sizes. On a linear algebra application with very large tiles, they will take an enormous amount of time to compute on a processing unit. This results in a much more favorable ratio of scheduling time to computation time for HFP. The applications are also composed of fewer tasks, making HFP packing much faster, which can negate the scheduling overhead.

Add more data reuse at the beginning of the execution At the very beginning of execution, memory is empty, all processing units are idle, so it is crucial to schedule tasks with few input data first, as they maximize the ratio between computation and required communication. This is something our schedulers do not take into account today. Knowing the number of processing units and the duration of a task, it would be advantageous to compute a schedule for each of them that favors data reuse for a certain amount of time before switching to our initial strategies.

Adapt Belady's rule to *Ready* To make full use of Belady's eviction policy, it needs to know the set of processed tasks in advance to infer the order in which data will be used. However, the *Ready* reordering, which has been shown to be powerful, reorders the set of tasks assigned to a processing unit. It thus messes with the planned order of data use that Belady knows. This does not significantly affect performance on the experiments we ran. However, it could be detrimental for very large matrix sizes. A future optimization is to inform Belady's eviction policy of any task movement from the *Ready* dynamic task reordering, making it much more efficient.

Analyze communications and computations overlap For both HFP and DARTS, we found that they can have as many data transfers as DMDAR, but more GFlop/s. This means that they are good at distributing data transfers over time. A future goal is to quantitatively evaluate the amount of overlapping data transfers during an execution. With this evaluation, it is possible to identify the decisions that maximize overlap and find ways to tune our algorithms in this direction.

Accept to let GPUs be idle We have found that DARTS is not able to achieve high performance with many GPUs and few tasks because it is not able to let a GPU idle when another can provide benefits through data reuse. To solve this problem, when DARTS has multiple idle GPUs, it needs to use the expected completion time of a set of tasks on one GPU and evaluate whether it is more beneficial to distribute the tasks across all GPUs or put it all on one GPU.

Apply to end-user cases An important future work is the use of DARTS in real-world applications. These applications include the EXAGEOSTAT project mentioned in the context. It relies for some applications on the resolution of large Cholesky factorizations and supports STARPU, which means that little or no adjustment would be required to use DARTS.

Improve our visualization tool Lastly, our visualization tool can be improved. It can easily be extended to support LU factorization, as it can be plotted as two Cholesky triangles. In a similar way, QR can also be easily represented as squares of task. In the long run, extending our tool to any task-based application is feasible. By grouping all data shares in an adjacency matrix and sorting them by iterations, it is possible to find a 2D pattern to represent the application. Automating this process is a non-negligible development effort, but would be beneficial for future performance analysis.

Batch schedulers

Integration in real computing platforms The most impactful perspective is to integrate LEA, LEO, and LEM into real cluster schedulers to test their robustness in real-world situations. However, it is difficult to deploy a new scheduling strategy on a real platform without disrupting the research work of hundreds of scientists. A reasonable solution would be to use the modularity of existing workload managers to slowly add locality techniques. For example, since we have seen that LEO is the more stable algorithm, it is conceivable to add LEO's intuition to an existing batch scheduler on a real platform.

Gradual LEM Speaking of our batch schedulers: LEM could be improved. One simple upgrade is to tune LEM to more finely adapt to the cluster level of utilization. Switching between a locality-first and a load-balancing-first strategy could be done gradually.

More complex job modeling Lastly, the model we used for batch scheduling contained many simplifications. For example, we only consider single-node jobs, as they are widely prevalent in the studied dataset. However, our scheduler could be extended to multi-node jobs, although it is a more complex task since one needs to find several nodes that all hold the corresponding data in order to reuse them. Other simplifications include the overlooking of jobs outputs and jobs that require load during computation and not only at the beginning. Finally, our model does not consider IO contention. We are interested in where to allocate jobs to reduce IOs, an additional step can be taken to better organize unavoidable IOs and minimize contention. Considering both steps simultaneously would most likely yield large improvements, but is a much more complex task.

Afterthoughts

In the long run, many topics would benefit from more thorough study. I discuss here the points that seem to me to be of greatest importance.

Hierarchical task scheduling Hierarchical tasks (also called recursive tasks) are a new way to manage heterogeneous nodes [58]. A hierarchical task is a large divisible task that can accommodate the heterogeneity of available computational resources by providing different task sizes to fit different processing units. Additionally, it may reduce the cost of task graph submission. I believe that using locality-aware scheduling policy with hierarchical tasks can bring newfound improvements. In addition to dealing with the memory constraint, the scheduler also has the responsibility of deciding whether or not to divide a task: it can decide to use the optimal tile size, or to recursively break tasks into the smallest possible size to run on individual cores. With this additional control, a scheduler can manage granularity to increase parallelism when memory is not constrained, and schedule tasks with small granularity when a refined schedule is needed to avoid pathological cases. This control can also be used to occupy nodes with large tasks while a complex scheduler plans a set of small tasks for later execution. Scheduling large tasks at the beginning of execution, when all processing units are idle, reduces the scheduling overhead. Scheduling small tasks at the end of execution, when all processing units are busy, is more important to reduce data transfer and eviction, and increase overlap of computations and communications. By dynamically controlling task granularity, the scheduler itself can control the overlap of computations and scheduling overhead, which is an exciting future direction for research in scheduling.

Implementation of theoretical studies Theoretical studies of communication-optimal algorithms such as [24] and many others provide schedulers that minimize communication for sequential out-of-core settings: a single processor with a distant slow but large memory and a nearby fast but small memory. Execution time is not the focus in such a setting. Execution time is more interesting to optimize in the case we studied, with many processing units and distributed memory. If one wants to use theoretical studies to try to reduce the execution time of an application, a main question arises: Is it better to focus on reducing communication above all else, or to use non-communication-optimal algorithms to add more parallelism? To answer this question, I think we should implement static communication-optimal schedulers in runtime systems and adapt them to multiple processing units. Observing the associated performance compared to a dynamic scheduler would bring a lot of knowledge and perhaps enable the creation of new hybrid strategies that would select ideas from both theoretical and practical scheduling to achieve peak performance.

Improve the efficiency of existing supercomputers We have seen in this thesis how to increase the performance of supercomputers on linear algebra applications. But what about applications that do not rely on linear algebra? For such applications, scientists try to get as much parallelism as possible and use as many nodes as possible. Beyond the common goal of getting as much parallelism as possible, they also rely on using more GPUs and memory to meet their memory and compute speed needs. This is especially true in AI, where large amounts of training data need to be processed. This encourages the construction of larger and larger supercomputers with more and more nodes. However, I believe that this is not sustainable, neither ecologically nor economically. Instead of larger systems, I advocate better utilization of existing clusters. To this end, I believe that more work on locality-aware scheduling will enable applications to require fewer nodes and less memory while still being completed in a reasonable amount of time.

Bibliography

- [1] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. “ExaGeoStat: A High Performance Unified Software for Geostatistics on Manycore Systems.” In: *IEEE Transactions on Parallel and Distributed Systems* 29.12 (2018). DOI: [10.1109/TPDS.2018.2850749](https://doi.org/10.1109/TPDS.2018.2850749).
- [2] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. “Geostatistical Modeling and Prediction Using Mixed Precision Tile Cholesky Factorization.” In: *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2019. DOI: [10.1109/HiPC.2019.00028](https://doi.org/10.1109/HiPC.2019.00028).
- [3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. “The Data Locality of Work Stealing.” In: *Theory Comput. Syst.* 35.3 (2002).
- [4] P. Agrawal, D. Kifer, and C. Olston. “Scheduling Shared Scans of Large Data Files.” In: *Proc. VLDB Endow.* 1.1 (Aug. 2008). DOI: [10.14778/1453856.1453960](https://doi.org/10.14778/1453856.1453960).
- [5] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. “QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators.” In: *2011 IEEE International Parallel & Distributed Processing Symposium*. 2011. DOI: [10.1109/IPDPS.2011.90](https://doi.org/10.1109/IPDPS.2011.90).
- [6] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, J. Roman, S. Thibault, and S. Tomov. “Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators.” In: *Symposium on Application Accelerators in High Performance Computing*. July 2010.
- [7] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. “Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs.” In: *GPU Computing Gems*. Vol. 2. Morgan Kaufmann, 2010. URL: <https://hal.inria.fr/inria-00547847>.
- [8] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault. “Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model.” In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: [10.1109/TPDS.2017.2766064](https://doi.org/10.1109/TPDS.2017.2766064).
- [9] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault. “Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model.” In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: [10.1109/TPDS.2017.2766064](https://doi.org/10.1109/TPDS.2017.2766064).

- [10] E. Agullo, M. Felšöci, and G. Sylvand. *A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems*. Research Report RR-9412. Inria Bordeaux Sud-Ouest, June 2021. URL: <https://inria.hal.science/hal-03263603>.
- [11] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. "A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory." In: *Parallel Computing* 34.6 (2008). DOI: [10.1016/j.parco.2008.03.007](https://doi.org/10.1016/j.parco.2008.03.007).
- [12] S. Albers. "New Results on Web Caching with Request Reordering." In: *Algorithmica* 58.2 (2010).
- [13] S. Albers. "On the Influence of Lookahead in Competitive Paging Algorithms." In: *Algorithmica* 18.3 (1997).
- [14] C. Alias, S. Thibault, and L. Gonnord. *A Compiler Algorithm to Guide Runtime Scheduling*. Research Report RR-9315. INRIA Grenoble ; INRIA Bordeaux - Sud-Ouest, Dec. 2019. URL: <https://hal.inria.fr/hal-02421327>.
- [15] Anonymous. *Reverse Cuthill-McKee Ordering in Python & Cython*. 2014. URL: <http://science-and-samgaetang.blogspot.com/2014/01/reverse-cuthill-mckee-ordering-in.html>.
- [16] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. "Data-Aware Task Scheduling on Multi-Accelerator based Platforms." In: *Int. Conf. on Parallel and Distributed Systems*. 2010.
- [17] C. Augonnet, S. Thibault, and R. Namyst. "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures." In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-14122-5.
- [18] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures." In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 2011). DOI: [10.1002/cpe.1631](https://doi.org/10.1002/cpe.1631).
- [19] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. "Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication." In: *SIAM Journal on Scientific Computing* 38.6 (2016). DOI: [10.1137/15M104253X](https://doi.org/10.1137/15M104253X).
- [20] N. Balin, G. Sylvand, and J. Robert. "Fast Methods applied to BEM Solvers for Acoustic Propagation Problems." In: *22nd AIAA/CEAS Aeroacoustics Conference*. 2016. DOI: [10.2514/6.2016-2712](https://doi.org/10.2514/6.2016-2712).
- [21] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication." In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2012. DOI: [10.1145/2312005.2312044](https://doi.org/10.1145/2312005.2312044).
- [22] M. Bauer. "Legion: programming distributed heterogeneous architectures with logical regions." PhD thesis. 2014. URL: https://stacks.stanford.edu/file/druid:kk063hx7516/bauer_thesis-augmented.pdf.

-
- [23] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. “Legion: Expressing locality and independence with logical regions.” In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012. DOI: [10.1109/SC.2012.71](https://doi.org/10.1109/SC.2012.71).
- [24] O. Beaumont, L. Eyraud-Dubois, M. V erit e, and J. Langou. “I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels.” In: *ACM Symposium on Parallelism in Algorithms and Architectures*. 2022. URL: <https://hal.inria.fr/hal-03580531>.
- [25] O. Beaumont and L. Marchal. “Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms.” In: *The 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, 2014. DOI: [10.1145/2600212.2600223](https://doi.org/10.1145/2600212.2600223).
- [26] L. A. Belady. “A study of replacement algorithms for a virtual-storage computer.” In: *IBM Systems Journal* 5.2 (1966). ISSN: 0018-8670.
- [27] P. Bogiatzis, M. Ishii, and T. A. Davis. “Towards using direct methods in seismic tomography: computation of the full resolution matrix using high-performance computing and sparse QR factorization.” In: *Geophysical Journal International* 205.2 (Feb. 2016). DOI: [10.1093/gji/ggw052](https://doi.org/10.1093/gji/ggw052).
- [28] D. Borthakur et al. “HDFS architecture guide.” In: *Hadoop apache project* (2008).
- [29] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczyk, A. YarKhan, and J. Dongarra. “Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA.” In: *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011.
- [30] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemariner, and J. Dongarra. “DAGuE: A generic distributed DAG engine for High Performance Computing.” In: *Parallel Computing* 38.1 (2012). Extensions for Next-Generation Parallel Programming Models. DOI: [10.1016/j.parco.2011.10.003](https://doi.org/10.1016/j.parco.2011.10.003).
- [31] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. H erault, and J. Dongarra. “ParSEC: A programming paradigm exploiting heterogeneity for enhancing scalability.” In: *Computing in Science and Engineering* 15.6 (2013).
- [32] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. “Productive programming of GPU clusters with OmpSs.” In: *International Parallel and Distributed Processing Symposium*. 2012.
- [33] D. Callahan, B. Chamberlain, and H. Zima. “The cascade high productivity language.” In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings*. 2004. DOI: [10.1109/HIPS.2004.1299190](https://doi.org/10.1109/HIPS.2004.1299190).
- [34] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. “A batch scheduler with high level components.” In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005*. Vol. 2. 2005. DOI: [10.1109/CCGRID.2005.1558641](https://doi.org/10.1109/CCGRID.2005.1558641).
- [35] H. Casanova. “Simgrid: a toolkit for the simulation of application scheduling.” In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2001. DOI: [10.1109/CCGRID.2001.923223](https://doi.org/10.1109/CCGRID.2001.923223).

- [36] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms.” In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014).
- [37] H.-L. Chan, T.-W. Lam, and K.-S. Liu. “Extra Unit-Speed Machines Are Almost as Powerful as Speedy Machines for Flow Time Scheduling.” In: *SIAM Journal on Computing* 37.5 (2008). DOI: [10.1137/060653445](https://doi.org/10.1137/060653445).
- [38] W.-T. Chan, T.-W. Lam, K.-S. Liu, and P. W. Wong. “New resource augmentation analysis of the total stretch of SRPT and SJF in multiprocessor scheduling.” In: *Theoretical Computer Science* 359.1 (2006). DOI: [10.1016/j.tcs.2006.06.003](https://doi.org/10.1016/j.tcs.2006.06.003).
- [39] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing.” In: *SIGPLAN Not.* 40.10 (Oct. 2005). ISSN: 0362-1340.
- [40] C. Chekuri, A. Goel, S. Khanna, and A. Kumar. “Multi-Processor Scheduling to Minimize Flow Time with e Resource Augmentation.” In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: Association for Computing Machinery, 2004. ISBN: 1581138520. DOI: [10.1145/1007352.1007411](https://doi.org/10.1145/1007352.1007411).
- [41] S. S. Chen, J. J. Dongarra, and C. C. Hsiung. “Multiprocessing linear algebra algorithms on the CRAY X-MP-2: Experiences with small granularity.” In: *Journal of Parallel and Distributed Computing* 1.1 (1984). DOI: [10.1016/0743-7315\(84\)90009-1](https://doi.org/10.1016/0743-7315(84)90009-1).
- [42] Q. Cheng, M. Glick, and K. Bergman. “Chapter 20 - Optical interconnection networks for high-performance systems.” In: *Optical Fiber Telecommunications VII*. Ed. by A. E. Willner. Academic Press, 2020. DOI: [10.1016/B978-0-12-816502-7.00020-8](https://doi.org/10.1016/B978-0-12-816502-7.00020-8).
- [43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8.
- [44] M. Cosnard and M. Loi. “Automatic task graph generation techniques.” In: *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. Vol. 2. 1995. DOI: [10.1109/HICSS.1995.375471](https://doi.org/10.1109/HICSS.1995.375471).
- [45] N. R. Council et al. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.
- [46] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices.” In: *Proceedings of the 1969 24th National Conference*. ACM '69. Association for Computing Machinery. DOI: [10.1145/800195.805928](https://doi.org/10.1145/800195.805928).
- [47] D. M. Dakshayini and D. H. Guruprasad. “An optimal model for priority based service scheduling policy for cloud computing environment.” In: *International journal of computer applications* 32.9 (2011).
- [48] A. G. Delavar, M. Javanmard, M. B. Shabestari, and M. K. Talebi. “RSDC (reliable scheduling distributed in cloud computing).” In: *International Journal of Computer Science, Engineering and Applications* 2.3 (2012).
- [49] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. “Communication-optimal Parallel and Sequential QR and LU Factorizations.” In: *SIAM Journal on Scientific Computing* 34.1 (2012). DOI: [10.1137/080731992](https://doi.org/10.1137/080731992).

-
- [50] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. *Communication-optimal parallel and sequential QR and LU factorizations*. Tech. rep. UCB/EECS-2008-89. Current version available in the ArXiv at <http://arxiv.org/pdf/0809.0101> Replaces EECS-2008-89 and EECS-2008-74. EECS Department, University of California, Berkeley, Aug. 2008. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-89.html>.
- [51] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. “Avoiding communication in sparse matrix computations.” In: *IEEE International Symposium on Parallel and Distributed Processing*. 2008. DOI: [10.1109/IPDPS.2008.4536305](https://doi.org/10.1109/IPDPS.2008.4536305).
- [52] P. J. Denning. “The working set model for program behavior.” In: *Communications of the ACM* 11.5 (1968).
- [53] J. Dongarra and A. Geist. *Report on the Oak Ridge National Laboratory’s Frontier System*. Tech Report No. ICL-UT-22-05. University of Tennessee, May 2022. URL: <https://icl.utk.edu/files/publications/2022/icl-utk-1570-2022.pdf>.
- [54] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard. “Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator.” In: *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-61756-5.
- [55] J. P. Eckert. “Univac-Larc, the next Step in Computer Design.” In: *Papers and Discussions Presented at the December 10-12, 1956, Eastern Joint Computer Conference: New Developments in Computers*. AIEEE-IRE ’56 (Eastern). New York, New York: Association for Computing Machinery, 1956. URL: [10.1145/1455533.1455539](https://doi.org/10.1145/1455533.1455539).
- [56] Y. Etsion and D. Tsafirir. “A short survey of commercial cluster batch schedulers.” In: *School of Computer Science and Engineering, The Hebrew University of Jerusalem* 44221 (2005).
- [57] Y. Fan. *Job Scheduling in High Performance Computing*. 2021. arXiv: [2109.09269](https://arxiv.org/abs/2109.09269) [cs.DC].
- [58] M. Faverge, N. Furmento, A. Guermouche, G. Lucas, R. Namyst, S. Thibault, and P.-A. Wacrenier. “Programming heterogeneous architectures using hierarchical tasks.” In: *Concurrency and Computation: Practice and Experience* (). DOI: [10.1002/cpe.7811](https://doi.org/10.1002/cpe.7811).
- [59] T. Feder, R. Motwani, R. Panigrahy, S. Seiden, R. van Stee, and A. Zhu. “Combining request scheduling with web caching.” In: *Theoretical Computer Science* 324.2 (2004). URL: <http://www.sciencedirect.com/science/article/pii/S0304397504003792>.
- [60] D. G. Feitelson and M. A. Jette. “Improved utilization and responsiveness with gang scheduling.” In: *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-69599-8.
- [61] M. Felšöci. “Solveurs rapides pour l’aéroacoustique haute fréquence.” Theses. Université de Bordeaux, Feb. 2023. URL: <https://theses.hal.science/tel-04077474>.
- [62] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard. “Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures.” In: *Parallel Computing* 44 (2015).
- [63] Z. Fu, Z. Tang, L. Yang, and C. Liu. “An Optimal Locality-Aware Task Scheduling Algorithm Based on Bipartite Graph Modelling for Spark Applications.” In: *IEEE Transactions on Parallel and Distributed Systems* 31.10 (2020). DOI: [10.1109/TPDS.2020.2992073](https://doi.org/10.1109/TPDS.2020.2992073).
- [64] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. London (UK): W.H. Freeman and Co, 1979.

- [65] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin. “Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures.” In: *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*. Berlin, Germany, Jan. 2013. URL: <https://inria.hal.science/hal-00780890>.
- [66] Gavril. “Some NP-complete problems on graphs.” In: *Proceedings of the 11th conference on Information Sciences and Systems*. 1977.
- [67] W. Gentsch. “Sun Grid Engine: towards creating a compute power grid.” In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2001. DOI: [10.1109/CCGRID.2001.923173](https://doi.org/10.1109/CCGRID.2001.923173).
- [68] S. Ghanbari and M. Othman. “A priority based job scheduling algorithm in cloud computing.” In: *Procedia Engineering* 50.0 (2012).
- [69] A. Giersch, Y. Robert, and F. Vivien. “Scheduling tasks sharing files on heterogeneous clusters.” In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting (EuroPVM/MPI)*. Lecture Notes in Computer Science. Springer. 2003.
- [70] H. H. Goldstine and A. Goldstine. “The Electronic Numerical Integrator and Computer (ENIAC).” In: *Mathematical Tables and Other Aids to Computation* 2.15 (1946). ISSN: 08916837. URL: <http://www.jstor.org/stable/2002620> (visited on 06/15/2023).
- [71] L. Grigori, J. W. Demmel, and H. Xiang. “CALU: A Communication Optimal LU Factorization Algorithm.” In: *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011). DOI: [10.1137/100788926](https://doi.org/10.1137/100788926).
- [72] R. Gu, Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang. “Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms.” In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017).
- [73] T. Günther, H. Malmström, E. M. Svensson, A. Omrak, F. Sánchez-Quinto, G. M. Kılınc, M. Krzewińska, G. Eriksson, M. Fraser, H. Edlund, A. R. Munters, A. Coutinho, L. G. Simões, M. Vicente, A. Sjölander, B. Jansen Sellevold, R. Jørgensen, P. Claes, M. D. Shriver, C. Valdiosera, M. G. Netea, J. Apel, K. Lidén, B. Skar, J. Storå, A. Götherström, and M. Jakobsson. “Population genomics of Mesolithic Scandinavia: Investigating early postglacial migration routes and high-latitude adaptation.” In: *PLOS Biology* 16.1 (Jan. 2018). DOI: [10.1371/journal.pbio.2003703](https://doi.org/10.1371/journal.pbio.2003703).
- [74] R. L. Henderson. “Job scheduling under the portable batch system.” In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995.
- [75] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra. “Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC.” In: *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (Scala)*. 2019. DOI: [10.1109/Scala49573.2019.00010](https://doi.org/10.1109/Scala49573.2019.00010).
- [76] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer. “Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters.” In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’16. Kyoto, Japan: Association for Computing Machinery, 2016. DOI: [10.1145/2907294.2907316](https://doi.org/10.1145/2907294.2907316).
- [77] J.-W. Hong and H. Kung. “I/O complexity: The red-blue pebble game.” In: *STOC’81: Proceedings of the 13th ACM symposium on Theory of Computing*. ACM Press, 1981.

-
- [78] D. Jackson, Q. Snell, and M. Clement. “Core Algorithms of the Maui Scheduler.” In: *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-45540-0.
- [79] D. Jackson, Q. Snell, and M. Clement. “Core Algorithms of the Maui Scheduler.” In: *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-45540-0.
- [80] Z. Jiang, T. Liu, S. Zhang, Z. Guan, M. Yuan, and H. You. *Fast and Efficient Parallel Breadth-First Search with Power-law Graph Transformation*. 2020.
- [81] G. Jin, T. Endo, and S. Matsuoka. “A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of GPUs.” In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013. DOI: [10.1109/CLUSTER.2013.6702633](https://doi.org/10.1109/CLUSTER.2013.6702633).
- [82] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. “HPX: A Task Based Programming Model in a Global Address Space.” In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. Eugene, OR, USA: Association for Computing Machinery, 2014. ISBN: 9781450332477.
- [83] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira. “Workload management with loadleveler.” In: *IBM Redbooks 2.2* (2001).
- [84] G. Karypis and V. Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs.” In: *SIAM Journal on scientific Computing* 20.1 (1998).
- [85] G. Karypis and V. Kumar. *hMETIS 1.5 : A hypergraph partitioning package*. Available at <http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download>.
- [86] K. Kaya and C. Aykanat. “Iterative-Improvement-Based Heuristics for Adaptive Scheduling of Tasks Sharing Files on Heterogeneous Master-Slave Environments.” In: *Trans. Parallel Distributed Syst.* 17.8 (2006).
- [87] K. Kaya, B. Uçar, and C. Aykanat. “Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories.” In: *J. Parallel Distributed Comput.* 67.3 (2007). DOI: [10.1016/j.jpdc.2006.11.004](https://doi.org/10.1016/j.jpdc.2006.11.004).
- [88] E. Kayraklioglu, E. Ronaghan, M. P. Ferguson, and B. L. Chamberlain. “Locality-Based Optimizations in the Chapel Compiler.” In: *Languages and Compilers for Parallel Computing*. Ed. by X. Li and S. Chandrasekaran. Cham: Springer International Publishing, 2022. ISBN: 978-3-030-99372-6.
- [89] G. Keramidas, P. Petoumenos, and S. Kaxiras. “Cache replacement based on reuse-distance prediction.” In: *2007 25th International Conference on Computer Design*. 2007. DOI: [10.1109/ICCD.2007.4601909](https://doi.org/10.1109/ICCD.2007.4601909).
- [90] G. Kwasniewski, M. Kabic, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefer. “Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication.” In: *Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC 2019*. 2019. DOI: [10.1145/3295500.3356181](https://doi.org/10.1145/3295500.3356181).
- [91] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. “Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes.” In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 2014. DOI: [10.1109/IPDPSW.2014.9](https://doi.org/10.1109/IPDPSW.2014.9).

- [92] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler, M. Burtscher, M. Gligoric, et al. “Parla: A Python Orchestration System for Heterogeneous Architectures.” In: *SC '22: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2022. URL: <https://userweb.cs.txstate.edu/~burtscher/papers/sc22.pdf>.
- [93] J. Lee, H. Kang, H.-j. Yeom, S. Cheon, J. Park, and D. Kim. “Out-of-core GPU 2D-shift-FFT algorithm for ultra-high-resolution hologram generation.” In: *Opt. Express* 29.12 (June 2021).
- [94] S. Leonenkov and S. Zhumatiy. “Introducing New Backfill-based Scheduler for SLURM Resource Manager.” In: *Procedia Computer Science* 66 (2015). 4th International Young Scientist Conference on Computational Science. DOI: [10.1016/j.procs.2015.11.075](https://doi.org/10.1016/j.procs.2015.11.075).
- [95] S. Leonenkov and S. Zhumatiy. “Introducing New Backfill-based Scheduler for SLURM Resource Manager.” In: *Procedia Computer Science* 66 (2015). 4th International Young Scientist Conference on Computational Science. DOI: [10.1016/j.procs.2015.11.075](https://doi.org/10.1016/j.procs.2015.11.075).
- [96] W.-H. Liu and A. H. Sherman. “Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices.” In: *SIAM Journal on Numerical Analysis* 13.2 (1976).
- [97] L. Marchal, S. McCauley, B. Simon, and F. Vivien. “Minimizing I/Os in Out-of-Core Task Tree Scheduling.” In: *International Journal of Foundations of Computer Science* 34.01 (2023). DOI: [10.1142/S0129054122500186](https://doi.org/10.1142/S0129054122500186).
- [98] P. Michaud. *(Yet another) proof of optimality for MIN replacement*. Oct. 2007.
- [99] R. T. Mills, A. Stathopoulos, and E. Smirni. “Algorithmic Modifications to the Jacobi-Davidson Parallel Eigensolver to Dynamically Balance External CPU and Memory Load.” In: *Proceedings of the 15th International Conference on Supercomputing, ICS '01*. Sorrento, Italy: ACM, 2001. DOI: [10.1145/377792.377903](https://doi.org/10.1145/377792.377903).
- [100] P. Mishra, T. Agrawal, and P. Malakar. “Communication-aware Job Scheduling using SLURM.” In: *49th International Conference on Parallel Processing-ICPP: Workshops*. 2020.
- [101] S. Moustafa, M. Faverge, L. Plagne, and P. Ramet. “3D cartesian transport sweep for massively parallel architectures with PARSEC.” In: *IEEE International Parallel and Distributed Processing Symposium*. 2015.
- [102] C. R. Nigro. “Evaluation of the PlayStation 2 as a cluster computing node.” In: (2004).
- [103] D. S. Nikolopoulos and C. D. Polychronopoulos. “Adaptive scheduling under memory constraints on non-dedicated computational farms.” In: *Future Gener. Comput. Syst.* 19 (2003).
- [104] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, and F. Rastello. *Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs*. 2019. arXiv: [1911.06664](https://arxiv.org/abs/1911.06664) [cs.CC].
- [105] S. Parsa and R. Entezari-Maleki. “RASA: a new grid task scheduling algorithm.” In: *JDCTA* 3 (Jan. 2009). DOI: [10.4156/jdcta.vol13.issue4.10](https://doi.org/10.4156/jdcta.vol13.issue4.10).
- [106] B. Peccerillo and S. Bartolini. “PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores.” In: *IEEE Transactions on Parallel and Distributed Systems* 30.1 (2019).
- [107] J.-N. Quintin and F. Wagner. “Hierarchical Work-Stealing.” In: *Euro-Par 2010 - Parallel Processing*. 2010. ISBN: 978-3-642-15277-1.

-
- [108] A. Robison, M. Voss, and A. Kukanov. “Optimization via Reflection on Work Stealing in TBB.” In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008. DOI: [10.1109/IPDPS.2008.4536188](https://doi.org/10.1109/IPDPS.2008.4536188).
- [109] K. Rupp. *50 Years of Microprocessor Trend Data*. 2021. URL: <https://github.com/karlrupp/microprocessor-trend-data>.
- [110] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W.-m. W. Hwu. “Program optimization carving for GPU computing.” In: *Journal of Parallel and Distributed Computing* 68.10 (2008). General-Purpose Processing using Graphics Processing Units. DOI: [10.1016/j.jpdc.2008.05.011](https://doi.org/10.1016/j.jpdc.2008.05.011).
- [111] E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, and Ü. V. Çatalyürek. “An out-of-core task-based middleware for data-intensive scientific computing.” In: *Handbook on Data Centers* (2015).
- [112] S. Selvarani and G. S. Sadhasivam. “Improved cost-based algorithm for task scheduling in cloud computing.” In: *2010 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE, 2010.
- [113] H. Senger, F. A. Silva, and W. M. Nascimento. “Hierarchical scheduling of independent tasks with shared files.” In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID’06)*. 2006. DOI: [10.1109/CCGRID.2006.1630942](https://doi.org/10.1109/CCGRID.2006.1630942).
- [114] K. Shirahata, H. Sato, and S. Matsuoka. “Out-of-core GPU memory management for MapReduce-based large-scale graph processing.” In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 2014. DOI: [10.1109/CLUSTER.2014.6968748](https://doi.org/10.1109/CLUSTER.2014.6968748).
- [115] *Slurm Workload manager*. https://slurm.schedmd.com/sched_config.html. Accessed: 2022-12-06.
- [116] T. M. Smith, B. Lowery, J. Langou, and R. A. van de Geijn. *A Tight I/O Lower Bound for Matrix Multiplication*. 2019. arXiv: [1702.02017](https://arxiv.org/abs/1702.02017) [cs.CC].
- [117] S. Sreepathi, E. D’Azevedo, B. Philip, and P. Worley. “Communication Characterization and Optimization of Applications Using Topology-Aware Task Mapping on Large Supercomputers.” In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. Delft, The Netherlands: Association for Computing Machinery, 2016. ISBN: 9781450340809.
- [118] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. “Characterization of Backfilling Strategies for Parallel Job Scheduling.” In: Feb. 2002. DOI: [10.1109/ICPPW.2002.1039773](https://doi.org/10.1109/ICPPW.2002.1039773).
- [119] G. Staples. “TORQUE Resource Manager.” In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. DOI: [10.1145/1188455.1188464](https://doi.org/10.1145/1188455.1188464).
- [120] M. Taufer. “AI4IO: A Suite of Ai-Based Tools for IO-Aware HPC Resource Management.” In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2021. DOI: [10.1109/HiPC53243.2021.00012](https://doi.org/10.1109/HiPC53243.2021.00012).
- [121] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer. “Celerity: High-Level C++ for Accelerator Clusters.” In: *Euro-Par 2019: Parallel Processing*. Ed. by R. Yahyapour. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-29400-7.
- [122] S. Toledo. “A survey of out-of-core algorithms in numerical linear algebra.” In: *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999.

-
- [123] H. Topcuoglu, S. Hariri, and M.-Y. Wu. “Task scheduling algorithms for heterogeneous processors.” In: *Proceedings. Eighth Heterogeneous Computing Workshop (HCW’99)*. 1999. DOI: [10.1109/HCW.1999.765092](https://doi.org/10.1109/HCW.1999.765092).
- [124] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong. “PAVER: Locality Graph-Based Thread Block Scheduling for GPUs.” In: *ACM Trans. Archit. Code Optim.* 18.3 (June 2021). ISSN: 1544-3566.
- [125] J.-F. Weets, M. K. Kakhani, and A. Kumar. “Limitations and challenges of HDFS and MapReduce.” In: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. 2015. DOI: [10.1109/ICGCIoT.2015.7380524](https://doi.org/10.1109/ICGCIoT.2015.7380524).
- [126] Y. Ye, Z. Du, D. Bader, Q. Yang, and W. Huo. “GPUMemSort: A High Performance Graphics Co-processors Sorting Algorithm for Large Scale In-Memory Data.” In: *GSTF INTERNATIONAL JOURNAL ON COMPUTING* 1 (May 2011). DOI: [10.5176/2010-2283_1.2.34](https://doi.org/10.5176/2010-2283_1.2.34).
- [127] A. B. Yoo, M. A. Jette, and M. Grondona. “SLURM: Simple Linux Utility for Resource Management.” In: *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-39727-4.
- [128] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis. “Locality-Aware Task Management for Unstructured Parallelism: A Quantitative Limit Study.” In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013. DOI: [10.1145/2486159.2486175](https://doi.org/10.1145/2486159.2486175).
- [129] S. Zhou, X. Zheng, J. Wang, and P. Delisle. “Utopia: a load sharing facility for large, heterogeneous distributed computer systems.” In: *Software: practice and Experience* 23.12 (1993).

List of publications

Article in International Refereed Journal

- [J1] M. Gonthier, L. Marchal, and S. Thibault. “Taming data locality for task scheduling under memory constraint in runtime systems.” In: *Future Generation Computer Systems* (2023). DOI: [10.1016/j.future.2023.01.024](https://doi.org/10.1016/j.future.2023.01.024).

Article in International Refereed Conference

- [C1] M. Gonthier, L. Marchal, and S. Thibault. “Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems.” In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2022. DOI: [10.1109/IPDPS53621.2022.00073](https://doi.org/10.1109/IPDPS53621.2022.00073).

Article in International Refereed Workshop

- [W1] M. Gonthier, L. Marchal, and S. Thibault. “Locality-Aware Scheduling of Independent Tasks for Runtime Systems.” In: *COLOC - 5th workshop on data locality - 27th International European Conference on Parallel and Distributed Computing*. Lisbon, Portugal, Aug. 2021, pp. 1–12. ISBN: 978-3-031-06156-1.

Poster in International Refereed Conference

- [P1] M. Gonthier, L. Marchal, and S. Thibault. *Memory-Aware Scheduling Of Tasks Sharing Data On Multiple GPUs*. ISC 2023 - ISC High Performance 2023. Poster. May 2023. URL: <https://inria.hal.science/hal-04090595>.

Article in National Refereed Conference

- [N1] M. Gonthier. “Exploiting data locality to maximize the performance of data-sharing tasksets.” In: *ComPAS 2023 - Conférence francophone d’informatique en Parallélisme, Architecture et Système*. Annecy, France, July 2023. URL: <https://inria.hal.science/hal-04090634>.

Invited Posters

- [IP1] M. Gonthier, L. Marchal, and S. Thibault. *Locality-Aware Scheduling Of Independent Tasks For Runtime Systems*. HiPEAC ACACES 2021 - 17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems. Poster. Sept. 2021. URL: <https://inria.hal.science/hal-04090604>.
- [IP2] M. Gonthier, L. Marchal, and S. Thibault. *Memory-Aware Scheduling Of Tasks Sharing Data On Multiple GPUs*. HiPEAC ACACES 2022 - 18th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems. Poster. July 2022. URL: <https://inria.hal.science/hal-04090607>.

Research Reports

- [R1] M. Gonthier, L. Marchal, and S. Thibault. *Locality-Aware Scheduling of Independant Tasks for Runtime Systems*. Research Report RR-9394. Inria Grenoble -Rhône-Alpes, 2021, p. 21. URL: <https://inria.hal.science/hal-03144290>.
- [R2] M. Gonthier, L. Marchal, S. Thibault, E. Larsson, and C. Nettelblad. *Locality-aware batch scheduling of I/O intensive workloads*. Tech. rep. RR-9497. ENS Lyon ; Inria Bordeaux ; Uppsala Universitet, Feb. 2022, p. 25. URL: <https://inria.hal.science/hal-03993118>.
- [R3] M. Gonthier, S. Thibault, and L. Marchal. *A generic scheduler to foster data locality for GPU and out-of-core task-based applications*. Tech. rep. June 2023. URL: <https://inria.hal.science/hal-04146714>.

Acknowledgement

This work is part of the SOLHARIS project, supported by the Agence Nationale de la Recherche , under grant ANR-19-CE46-0009.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see <https://www.grid5000.fr>).