



On the use of hierarchical task for heterogeneous architectures

Gwenolé Lucas

► To cite this version:

Gwenolé Lucas. On the use of hierarchical task for heterogeneous architectures. Other [cs.OH]. Université de Bordeaux, 2023. English. NNT : 2023BORD0231 . tel-04316145

HAL Id: tel-04316145

<https://theses.hal.science/tel-04316145>

Submitted on 30 Nov 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
PRÉSENTÉE À
L'UNIVERSITÉ DE BORDEAUX
ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE
par **Gwenolé LUCAS**
POUR OBTENIR LE GRADE DE
DOCTEUR
SPÉCIALITÉ : INFORMATIQUE

**Programmation des architectures hétérogènes
à l'aide de tâches divisibles**

Sous la direction de : Abdou GUERMOUCHE et Raymond NAMYST

Soutenue le : 10 Octobre 2023

Devant la commission d'examen composée de :

Siegfried BENKNER	Professor, University of Vienna	Rapporteur
Alfredo BUTTARI	Directeur de recherche, CNRS - IRIT	Rapporteur
Vincenç BELTRAN	Senior researcher, Barcelona Supercomputing Center ..	Examineur
Thomas HERAULT	Research Assistant Professor, University of Tennessee .	Examineur
Isabelle TERRASSE	Scientific advisor, Airbus	Examinatrice
Abdou GUERMOUCHE ..	Maître de conférences, Université de Bordeaux	Directeur de thèse
Raymond NAMYST	Professeur des universités, Université de Bordeaux	Directeur de thèse

Programmation des architectures hétérogènes à l'aide de tâches divisibles

Résumé : Au cours des dernières décennies, les plateformes de calcul haute performance ont connu une croissance exponentielle de leur puissance de calcul au détriment d'une complexité toujours plus grande. Programmer ces plateformes pour tirer pleinement parti de leur puissance de calcul est un défi de taille. Le programmeur doit prendre en compte les différents types d'unités de calcul, la hiérarchie de la mémoire, les transferts de données au sein d'un nœud ou à travers un réseau, etc. Les supports d'exécution à base de tâches ont gagné en popularité grâce à leur capacité à exprimer des applications performantes et portables sur ces architectures hétérogènes complexes. Ce paradigme s'appuie sur divers modèles de programmation pour représenter les applications sous la forme de graphes de tâches. Plus particulièrement, le modèle de *soumission séquentielle de tâche* (*Sequential Task Flow, STF*) fournit une interface simple pour écrire des applications performantes, mais ne produit malheureusement que des graphes de tâches statiques. Ce manque de dynamisme est préjudiciable aux exécutions destinées à des systèmes hétérogènes, où la granularité des tâches peut avoir besoin d'être ajustée selon les types d'unités de calcul disponibles à un moment donné. La nature séquentielle du modèle STF crée également un surcoût à la soumission qui peut limiter le passage à l'échelle de certaines applications présentant énormément de parallélisme.

Cette thèse propose de répondre à ces problèmes en étendant le modèle de STARPU pour y inclure des *tâches hiérarchiques*, capables d'insérer des sous-graphes au cours de leur exécution. Les graphes de tâches gagnent ainsi en dynamisme, ce qui permet d'ajuster la granularité des tâches à l'exécution pour correspondre au mieux aux unités de calcul ciblées. Par ailleurs, les tâches hiérarchiques peuvent être utilisées pour paralléliser le processus de soumission et en réduire le surcoût. En effet, il est possible de traiter des tâches hiérarchiques indépendantes en parallèle. Notre modèle est complété par un gestionnaire de données capable d'adapter dynamiquement la disposition des données sans intervention du programmeur. Il contribue également à l'exactitude du graphe de tâche, car l'insertion de sous-graphes à l'exécution contourne les règles du STF. Nous parvenons à ce résultat grâce aux tâches de synchronisation qui ajustent la disposition des données autour des tâches hiérarchiques. Nous avons implémenté ce modèle et l'avons appliqué à la bibliothèque d'algèbre linéaire dense CHAMELEON. La coexistence de plusieurs granularités de tâches a permis d'améliorer les performances sur des architectures hétérogènes. Cela valide notre modèle et fournit une évaluation des avantages de l'utilisation de tâches hiérarchiques dans STARPU.

Mots-clés : Calcul haute performance, Supports d'exécution, Programmation à base de tâches, Modèles de programmation, Calcul hétérogène, Algèbre linéaire dense

Unité de recherche

Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

On the Use of Hierarchical Tasks for Heterogeneous Architectures

Abstract: In the last decades, the computing power of high-performance platforms has grown exponentially at the expense of increased complexity. Programming such platforms to take full advantage of their computing power is challenging. The programmer must take into account different types of computing units, memory hierarchy, data transfers within a node or across a network, etc. Task-based runtime systems have become popular because of their ability to express efficient and portable applications on these complex heterogeneous architectures. This paradigm relies on various programming models to represent applications as task graphs. In particular, the *Sequential Task Flow* (STF) model provides a simple interface for writing efficient applications, but unfortunately only supports static task graphs. The lack of dynamism is detrimental for execution on heterogeneous systems, where task granularity may need to be adjusted to optimally match the different types of computing units available at any given time. The sequential nature of the STF model also creates a submission overhead that limits scalability for embarrassingly parallel applications.

In this thesis, we address these issues by extending the STF model in the STARPU runtime system with *hierarchical tasks*, which can insert subgraphs at runtime. This allows for more dynamic task graphs, making it possible to adjust task granularity at runtime to best match the targeted computing units. In addition, the submission overhead can be reduced by using large-grain hierarchical tasks to parallelize the submission process. Indeed, independent hierarchical tasks can be processed in parallel on different workers. Our hierarchical task model is combined with an advanced data manager to dynamically switch between different data layouts without programmer input. The data manager also contributes to the correctness of the task graph, since the insertion of subgraphs at runtime bypasses the usual rules of the STF model. This is achieved through the synchronization tasks that adapt the data layout around hierarchical tasks. We implemented this model and applied it in the context of dense linear algebra using the CHAMELEON library. The coexistence of multiple task granularities thanks to hierarchical tasks resulted in good performance improvements on heterogeneous architectures. This allowed us to validate our model and evaluate the potential benefits of hierarchical tasks in STARPU.

Keywords: High-performance computing, Runtime systems, Task-based programming, Programming models, Heterogeneous computing, Dense linear algebra

Research Unit

Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Contents

Remerciements	1
Résumé	3
Introduction	7
1 Background & Related Work	11
1.1 Task-based programming models	13
1.2 Runtime systems for modern architectures	16
1.3 Relaxing the models to face modern problems	18
2 The StarPU Runtime System	21
2.1 Data structures	22
2.1.1 Codelets	22
2.1.2 Data handles	23
2.1.3 Tasks and jobs	24
2.2 Building the graph	24
2.2.1 Shared memory	25
2.2.2 Distributed memory	25
2.3 Executing the graph	27
2.3.1 Submission time and execution time	27
2.3.2 Scheduling	28
2.4 Data management: partitioning operations	29
2.4.1 Partitioning data manually	29
2.4.2 Automatic data manager	34
2.5 Conclusion	40
3 The Hierarchical Task Model	43
3.1 Objectives	44
3.1.1 Improving upon the STF model	44

3.1.2	Introducing more dynamic task graphs	46
3.1.3	Keeping the model simple	47
3.2	Designing hierarchical tasks for STARPU	48
3.2.1	Aiming for generality	48
3.2.2	User interface	49
3.2.3	Adapting the data layout around hierarchical tasks	50
3.2.4	Correctly inserting tasks at runtime	51
3.2.5	Fine grain dependencies	53
3.2.6	Parallel submission	55
3.2.7	Data management for the programmer	57
3.2.8	Case of read-only access mode	57
3.3	Conclusion	58
4	Hierarchical Tasks Implementation	59
4.1	Foundations of the implementation	60
4.1.1	Updating the data structures	60
4.1.2	Processing hierarchical tasks	61
4.1.3	Processing hierarchical tasks as soon as possible	62
4.2	Extending the data manager	62
4.2.1	Controlling data dependencies	64
4.2.2	Partition task insertion	65
4.2.3	Unpartition task insertion	66
4.2.4	The read-only access mode	69
4.3	Conclusion	71
5	Experimental Evaluation	73
5.1	Application of Hierarchical Tasks for Dense Linear Algebra	74
5.1.1	Recursive descriptors	74
5.1.2	Updating the kernels	76
5.2	Experimental results	76
5.2.1	Experimental settings	76
5.2.2	Hierarchical tasks overhead	78
5.2.3	Matrix-matrix multiplication	79
5.2.4	Cholesky factorization	82
5.2.5	LU factorization	85
5.2.6	Impact of the dynamic data manager	85
5.2.7	Comparison with other frameworks	87
6	First Steps of Hierarchical Tasks toward Distributed Memory	91
6.1	Introducing shared data	92
6.2	Automatic pruning	93
6.3	Communications at runtime	94
6.4	Conclusion	95

Conclusion	97
Acknowledgements	101
Bibliography	103
Publications	109

List of Figures

1.1	Sequential algorithm, corresponding STF version, and DAG.	14
1.2	Nested task parallelism management.	18
2.1	Writing a new codelet.	23
2.2	Registering a handle.	23
2.3	Submitting a task.	24
2.4	Cholesky factorization of a matrix composed of 4×4 tiles.	26
2.5	Matrix with two partition trees.	30
2.6	Example of synchronous partitioning.	31
2.7	Example of asynchronous partitioning.	33
2.8	Example of automatic partitioning.	34
2.9	Evolution of the graph and handle states associated to Figure 2.8 . . .	36
2.10	Unrolling a call of AUTOSUBMIT.	38
3.1	Submitting a hierarchical task.	49
3.2	Encapsulating some of the new parameters into the task codelet. . . .	50
3.3	Example of a single hierarchical task.	51
3.4	Processing a hierarchical task by relying strictly on the STF model. . .	51
3.5	Altering the STF model to process a hierarchical task.	52
3.6	Releasing hierarchical task dependencies to connect sub-graphs. . . .	54
3.7	Using unpartition tasks to enforce the correction of the graph.	55
3.8	Example of hierarchical tasks submitting sub-graphs in parallel. . . .	55
3.9	Example of an “identity” partition plan.	56
4.1	Processing a hierarchical task (or not).	61
4.2	Example of code using hierarchical tasks.	63
4.3	Processing the hierarchical tasks of Figure 4.2 in the submission thread. .	63
4.4	Manually defining sequential consistency for a task’s data.	64
4.5	Incorrect behavior of AUTOSUBMIT with multiple submission threads. . .	65
4.6	Checking if unpartitioning operation are needed at runtime	67
4.7	Insertion of unpartitioning tasks.	68

4.8	Insertion of a read-only unpartition task.	71
5.1	Information stored in a regular matrix descriptor.	75
5.2	Information stored in a recursive matrix descriptor.	75
5.3	Using a matrix product kernel to determine optimal tile sizes.	77
5.4	Evaluation of the overhead of hierarchical tasks.	78
5.5	Evaluation of hierarchical tasks in matrix-matrix multiplication.	80
5.6	Illustration of the matrix layouts used in the following experiments.	81
5.7	Evaluation of hierarchical tasks in Cholesky type operations on the INTEL-V100 platform.	82
5.8	Evaluation of hierarchical tasks in Cholesky type operations on the AMD-A100 platform.	83
5.9	Evaluation of hierarchical tasks in the LU decomposition.	84
5.10	Evaluation of the performance impact of the dynamic data manager.	86
5.11	Comparison of hierarchical tasks with parallel tasks and PARSEC	88
6.1	Partition tree with shared data.	92
6.2	Example of automatic pruning in a distributed task graph.	94

List of Tables

1.1	Summary of hierarchical task support in task-based runtime systems.	19
2.1	Usability of a handle depending on its state.	35
2.2	Handle states after the submission of partitioning tasks.	37
5.1	Characteristics of the experimental platforms.	77
5.2	Performances of the computational units of both experimental platforms.	79
5.3	Parameters explored to find the best performances on INTEL-V100. .	87
5.4	Parameters explored to find the best performances on AMD-A100. .	87

Remerciements



ARRIVÉ AU TERME de ~~trois~~ quatre années de thèse, il est finalement temps de rendre des comptes, en commençant par remercier toutes celles et ceux qui auront contribué à cet aboutissement.

Pour commencer par la fin, je remercie les membres du jury. J'ai été honoré de vous présenter ces travaux et de répondre à vos questions. Merci à Siegfried et Alfredo d'avoir accepté le rôle de rapporteur et relu cette thèse, j'espère que la lecture aura été sinon agréable, du moins intéressante. Merci à Vincenç, Thomas¹ et Isabelle d'avoir pris part à ce jury, j'ai beaucoup apprécié discuter des tâches hiérarchiques avec vous. J'adresse également un second remerciement à Alfredo pour avoir cumulé les rôles en présidant le jury.

Ensuite, je me dois de remercier mes nombreux encadrants, sans qui cette thèse n'aurait pas pu être ce qu'elle est, à commencer par la direction. Abdou, merci pour toute l'aide que tu m'as apporté pendant ces quatre années, pour ta patience pendant que je tombais dans tous les pièges administratifs impossibles et inimaginables, et pour ta disponibilité sans faille. Cette thèse et moi te devons beaucoup. Raymond, merci de m'avoir accueilli chez STORM durant tout ce temps. Tes conseils, ton expérience et tes histoires m'ont énormément appris. Ce n'est pas tout, car j'ai eu la chance de travailler avec un sacré groupe. Mathieu, merci pour les longs moments passés à réfléchir et déboguer, et merci de m'avoir parlé de ce stage sur les bulles qui a dégénéré en thèse. Nathalie, merci pour ton aide technique, notamment autour de STARPU. Pierre-André, merci pour tes précieuses idées sur les tâches hiérarchiques. Les réunions de thèse pouvaient être endiablées, mais nous finissions toujours par accorder nos violons et j'en suis toujours ressorti avec les idées plus claires, grâce à vous tous. Enfin, merci à Samuel de nous avoir guidé hors de quelques ornières STARPU-esque.

Un grand merci à toute l'équipe STORM de l'Inria Bordeaux. À tous les permanents, notamment ceux que je n'ai pas encore cités, Marie-Christine, Amina, Laércio, Mihail, Emmanuelle et Olivier, merci de former une équipe aussi formidable et bienveillante. C'était un plaisir de travailler à vos côtés, les innombrables discussions, de travail

¹Et merci Thomas pour la pêche aux coquilles, désormais corrigées dans cette version définitive.

ou pas, resteront d'excellent souvenirs (de même que les iconiques pots et autres goûters). Je remercie également les membres des autres équipes Inria, en particulier TOPAL, CONCACE et TADAAM, j'ai été heureux de prendre part à l'écosystème HPC bordelais.

Je ne peux bien sûr pas oublier les éphémères de l'open-space. Aux membres des neiges d'antan, ceux qui nous ont quittés ou qui font partie des meubles, merci Baptiste, Romain, Célia, Kun, Chiheb, Van Man, Philippe, d'avoir contribué à instaurer la bonne ambiance qui règne encore aujourd'hui. Aux membres des dernières averses, tout frais émoulus de leur Master ou déjà éprouvés par le poids des années (de thèse), merci Diane, Vincent, Alice, Lana, Thomas, Radja, Albert d'assurer la relève avec autant de brio. Merci à tout les autres que je n'ai pas cités mais que j'ai côtoyés avec joie au fil des ans. Enfin, merci à Maxime pour son esprit de camaraderie, bien qu'il ait eu l'audace de finir ~~dans les temps~~ le premier.

Je remercie également les relecteurs de cette thèse : Abdou, Pierre-André, Nathalie, Mathieu, Mihail, Olivier, Radja, merci pour vos corrections de mon anglaise prose, et pour vos suggestions, sur la forme ou le fond.

Pour finir, je remercie toute ma famille. Merci à mes grands-pères et grands-mères, oncles et tantes, cousins et cousines. À ma petite sœur, qui m'a toléré toute sa vie, merci pour ton sale caractère, tes histoires qui se terminent en eau de boudin, et ta maîtrise de choses qui me dépassent complètement (choisir des cadeaux, interagir avec les gens, réaliser des perfusions...). À mes parents, qui ont su me transmettre des valeurs dont je suis fier et qui m'ont permis de poursuivre ces interminables études : merci. Vous m'avez toujours laissé libre de toute pression, tout en m'accompagnant depuis le début. Sans les répétitions de dictées et les explications de factorisation avec les mains, je ne serais certainement jamais arrivé si loin.

Résumé

HISTORIQUEMENT, l'informatique a connu une évolution rapide, tant sur le plan théorique que technologique. En parallèle, elle a contribué au progrès de nombreux autres domaines scientifiques. Ceci est particulièrement vrai dans le domaine du calcul intensif, où de puissants ordinateurs permettent de résoudre des problèmes toujours plus massifs, plus rapidement et avec une plus grande précision. Impliqués dans des applications industrielles ou universitaires, ces problèmes comprennent la modélisation et la simulation de phénomènes naturels, de systèmes physiques pour l'ingénierie ou de processus biologiques.

Sur le plan matériel, l'augmentation de la puissance de calcul a pris de nombreuses formes au fil des ans. La simple accélération de la fréquence des cœurs CPU se heurte à des limites physiques et a arrêté de progresser il y a plus de vingt ans. Afin de continuer à améliorer leur puissance de calcul, les fabricants de processeurs ont fait le choix d'augmenter le nombre de cœurs. Dans ces architectures multicœurs, les logiciels doivent paralléliser leur charge de travail pour tirer pleinement parti du matériel sous-jacent. Par ailleurs, la dernière décennie a connu une montée en puissance des accélérateurs, des unités de calcul spécialisées capable d'effectuer certaines opérations plus efficacement, comme les GPU (*Graphics Processing Unit*). D'abord conçus pour accélérer le traitement des images, ils sont aujourd'hui utilisés dans les supercalculateurs en raison de leur affinité pour les calculs hautement parallèles. Les architectures modernes sont donc plus complexes que jamais : elles forment un réseau de nœuds hétérogènes combinant des CPU multicœurs et des accélérateurs. C'est par exemple le cas de FRONTIER, à la tête du dernier classement TOP500 des supercalculateurs les plus puissants du monde.

Au-delà de leur complexité individuelle, les super-ordinateurs présentent une grande variété d'architectures, avec de nombreuses configurations et types de matériel, dont la programmation nécessite différentes technologies. Pour pallier la difficulté de créer des applications capables d'exploiter ces différentes plateformes, une partie de la communauté HPC se consacre au développement de supports d'exécution qui fournissent des techniques et des outils de programmation portables. Un paradigme

notable est le modèle de programmation à base de tâches, dans lequel les applications sont représentées comme un graphe orienté acyclique (*Directed Acyclic Graph*, DAG) de tâches. Ce modèle facilite l'expression d'applications parallèles pour le programmeur en fournissant une abstraction dans laquelle des blocs de calculs séquentiels forment des *tâches* reliées par des *dépendances*.

Cependant, l'exploitation de plateformes hétérogènes par les supports d'exécution à base de tâches présente plusieurs défis. Les unités de calcul des plateformes hétérogènes ont des caractéristiques et des exigences différentes. Par exemple, les GPU sont généralement plus performants en travaillant sur de grands ensembles de données, tandis que les cœurs CPU conventionnels atteignent leurs performances maximales avec des opérations à grain fin travaillant sur une empreinte mémoire plus réduite. En outre, ces plateformes disposent généralement d'un plus grand nombre de cœurs CPU que de GPUs. Cela signifie qu'il peut être nécessaire d'augmenter le nombre de petites tâches pour améliorer les performances. Différentes stratégies existent pour résoudre ce problème, notamment en trouvant le meilleur compromis entre la granularité optimale de chaque type d'unité de calcul, ou en agrégeant des cœurs CPU pour traiter efficacement des tâches à gros grain, normalement plus adaptées aux GPUs. Une autre solution consiste à diviser les tâches à gros grain en tâches plus petites adaptées aux cœurs CPU. Le principal problème de ces approches est la nature statique du graphe de tâche, qui nous empêche de sélectionner la granularité appropriée au moment de l'exécution, en fonction des ressources disponibles.

En général, la plupart des supports d'exécution modernes basés sur les tâches souffrent d'un manque de dynamisme dans la génération des graphes de tâches. Par exemple, lors de la conception de solveurs d'algèbre linéaire basés sur des algorithmes d'approximation de rang faible, il est presque impossible de définir un graphe statique garantissant une bonne précision numérique. En outre, la représentation interne de graphes avec un grand nombre de tâches en attente peut causer un surcoût au sein du support d'exécution. C'est notamment le cas des supports d'exécution qui reposent sur le modèle de soumission séquentielle de tâche (*Sequential Task Flow*, STF), dans lequel le graphe est construit tâche après tâche. Dans ce modèle, la construction séquentielle du graphe peut également devenir un goulot d'étranglement. D'autres modèles de programmation qui reposent sur une description de haut niveau des dépendances sont moins touchés par ces problèmes, mais ils ont tendance à être plus difficiles à utiliser.

Pour résoudre ces problèmes, cette thèse propose d'étendre le processus de soumission du modèle STF dans le support d'exécution STARPU. Notre nouveau modèle de programmation introduit des tâches hiérarchiques, qui peuvent insérer un nouveau sous-graphe de tâche dans le DAG d'une application quand elles sont exécutées. Les tâches hiérarchiques permettent un contrôle plus fin du flot de soumission de l'application. En encapsulant des portions d'un DAG dans des tâches hiérarchiques, les programmeurs peuvent en retarder la soumission et réduire le surcoût lié au support d'exécution. Par ailleurs, étant donné que les tâches hiérarchiques sont également des tâches par nature et qu'elles sont donc exécutées par une unité de calcul, le processus


de soumission n'est plus géré par un unique fil d'exécution et peut être parallélisé en traitant des tâches hiérarchiques indépendantes sur différentes unités de calcul. Notre modèle est également capable de produire des graphes de tâche qui évoluent dynamiquement à l'exécution. Ceci est particulièrement utile pour adapter le niveau de granularité ou pour sélectionner différentes implémentations d'une même opération. Cependant, cette approche dynamique contourne l'ordre séquentiel de soumission sur lequel le modèle STF s'appuie pour correctement déduire les dépendances entre les tâches. Par conséquent, implémenter les tâches hiérarchiques sans créer d'incohérences au niveau des données ou du calcul est particulièrement complexe.

Puisque l'expression d'un sous-graphe équivalent à une tâche nécessite le partitionnement de ses données, nous devons être en mesure de modifier la disposition des données lorsqu'une tâche hiérarchique est traitée. STARPU gère généralement cela de manière asynchrone, avec des tâches de partitionnement servant de points de synchronisation définissant la région du DAG où une certaine disposition de données est active. Dans le modèle de tâches hiérarchiques, nous avons décidé de lier la hiérarchie des tâches à une hiérarchie des données. Cela nous permet de faire respecter la correction du graphe en utilisant uniquement les synchronisations de données nécessaires. Une première implémentation du modèle sacrifie certaines fonctionnalités pour s'appuyer sur le gestionnaire de données existant. Au cours du processus de soumission, on insère des tâches de partitionnement qui assureront la cohérence des données pendant l'exécution future. Dans cette approche, les tâches hiérarchiques sont traitées par le fil de soumission dès que possible, ce qui se traduit par une soumission récursive mais statique. Une implémentation plus avancée étend le gestionnaire de données pour retarder l'insertion des tâches de partitionnement jusqu'au moment de l'exécution. Grâce à ce nouveau gestionnaire de données, les tâches de partitionnement sont ajoutées au fur et à mesure, et si nécessaire, lorsqu'une tâche hiérarchique est traitée. Une fois le sous-graphe inséré, la tâche hiérarchique est considérée terminée et libère ses dépendances sortantes. Cela permet des dépendances à grain fin, car il n'y a pas de barrière ou de synchronisation entre des tâches hiérarchiques successives. Comme les successeurs non hiérarchiques utilisent une disposition de données différente de celle d'un sous-graphe, des tâches de départitionnement sont ajoutées entre les deux, ce qui permet d'assurer l'exactitude du graphe de tâches. Dans ces deux implémentations, la gestion des données est transparente pour les programmeurs, qui n'ont qu'à décrire les différentes dispositions possibles pour les données de leur application.

Des travaux préliminaires montrent que notre modèle de tâches hiérarchiques peut être utilisé pour élaguer les graphes de tâches distribués en encapsulant des parties de DAG dans des tâches hiérarchiques, qui ne sont alors traitées que par certains nœuds. Afin de soumettre et de traiter les tâches hiérarchiques sur plusieurs nœuds, nous avons introduit la notion de *données partagées*. Il s'agit d'une approche plus expressive que la simple restriction des tâches hiérarchiques à des nœuds individuels, mais elle nécessite de gérer soigneusement les communications de données ajoutées au moment de l'exécution.

Nous avons validé notre modèle en utilisant la bibliothèque d'algèbre linéaire dense CHAMELEON. Les descripteurs de matrice de CHAMELEON ont été adaptés pour stocker des tuiles de matrices de manière récursive. Un descripteur récursif peut donc stocker une matrice avec plusieurs niveaux de granularité. L'extension d'un noyau pour prendre en charge les tâches hiérarchiques est assez simple, puisque la fonction décrivant le graphe des tâches d'une version tuilée de l'opération est déjà disponible dans CHAMELEON. Nos expériences tentent de tirer parti des tâches hiérarchiques, soit en fournissant la bonne quantité de tâches à grain fin aux CPU, soit en affinant les tâches sur le chemin critique des opérations de factorisation. Les résultats que nous avons obtenus montrent que, bien que limitées par l'absence d'un ordonnanceur dédié, les tâches hiérarchiques ont le potentiel de grandement améliorer le comportement des support d'exécution à base de tâches tout en offrant une plus grande flexibilité au programmeur.

Introduction

OMPUTER SCIENCE historically exhibited fast-paced evolution, both theoretically and technologically. In turn, it enabled and supported progress in many other scientific fields. This is particularly true in High-Performance Computing (HPC), where powerful hardware is exploited in both academic and industrial applications to solve larger problems faster and more accurately. Such problems include the modelization and simulation of natural phenomena (e.g. climate prediction, weather forecasting), physical systems for engineering (e.g. aerodynamics, nuclear reactions), or biological processes (e.g. cellular interactions, epidemiology).

Hardware-wise, the increase in computing power has taken many forms over the years. The straightforward acceleration of the frequency of CPU cores stalled over twenty years ago². In order to keep improving their computing power, CPU manufacturers increased the number of cores. In multicore architectures, software must express parallelism to take full advantage of the hardware. Furthermore, the last decade has seen the rise in popularity of accelerators, specialized hardware that can perform certain operations more efficiently, such as GPUs. Originally designed to accelerate image processing, they are now used in supercomputers due to their affinity for embarrassingly parallel computations. Modern architectures are therefore more complex than ever, consisting of a network of heterogeneous nodes combining multicore CPUs and accelerators. For example, this is the case of FRONTIER, the front-runner of the latest TOP500 ranking³.

Beyond their individual complexity, supercomputers present a wide variety of architectures, with many configurations and types of hardware that require different technologies to program. To alleviate the difficulty of creating applications that can use these different platforms, part of the HPC community is focused on developing runtime systems that provide portable programming techniques and tools. One notable paradigm is the task-based programming model, in which applications are represented as a Directed Acyclic Graph (DAG) of tasks. This facilitates the expression of parallel

²<https://github.com/karlrupp/microprocessor-trend-data>

³<https://www.top500.org/lists/top500/2023/06/>

applications for the programmer by providing an abstraction in which sequential sets of computations are regrouped into tasks connected by dependencies.

However, exploiting heterogeneous platforms using the task-based paradigm presents several challenges. The computing resources of heterogeneous platforms have different characteristics and requirements. For example, GPU devices typically perform best with large data sets, while conventional CPU cores reach peak performance with fine-grain kernels working on a reduced memory footprint. Additionally, these platforms typically have a larger number of CPU cores than GPUs. This means that having more small tasks may be necessary to increase performance. Several efforts have been made to address this problem, such as finding the best trade-off between the optimal granularity of each device, or by using large grain tasks suited to GPU devices and aggregating CPU cores to process them efficiently. Alternatively, large grain tasks could be split into smaller tasks adapted to individual CPU cores. The main issue behind these approaches is the static nature of the task graph, which prevents us from selecting the appropriate granularity at runtime, depending on the available resources.

In general, most modern task-based runtime systems suffer from a lack of dynamism in task-graph generation. For example, when designing linear algebra solvers based on low-rank approximation algorithms, it is almost impossible to predict the right DAG to ensure good numerical accuracy. In addition, some runtime systems suffer from an overhead resulting from the internal representation of task graphs with a large amount of non-ready tasks. This is the case for runtime systems that rely on the Sequential Task Flow (STF) model, where the graph is constructed task by task, which might also bottleneck the execution. Other programming models that rely on a high-level description of dependencies are less affected by this problem, but they tend to be more difficult to use.

In this thesis, our goal is to extend the submission process of the STF model in the STARPU runtime system to produce more dynamic task graphs limiting the runtime overhead. To this end, we propose a new type of task, called *hierarchical tasks*, that can transform themselves into a new task graph at runtime. When such a task is scheduled to be executed, it can either behave like a regular task or insert a task graph that takes its place in the larger DAG. To define a hierarchical task, the programmer is only required to provide hints on top of a regular task. The runtime system can then delay the submission of parts of the task graph to support dynamic granularity and implementation selection, parallelize the task insertion process, and greatly reduce the number of tasks in the runtime system.

In order to replace a task with an equivalent sub-graph, the data used by the task must also be split into sub-data. Such modifications in the data layout are handled by STARPU's data manager, which inserts partitioning tasks at submission time to maintain data consistency at execution time. To be able to make dynamic adjustments to the data layout, we extended this data manager to insert partitioning

tasks at execution time. We can then rely on these tasks to ensure that a graph involving hierarchical tasks is consistent with a graph created with the STF model, and therefore correct.

Chapter 1 sets the context of this thesis in more details by presenting the various programming models used in task-based runtime systems. We also review the related work to present how runtime system address modern architectures and their challenges.

Chapter 2 gives a more in depth introduction to the terminology and concepts of the STARPU runtime system. Most notably, we introduce its original data manager which was extended to support hierarchical tasks.

Chapter 3 presents our hierarchical task model and the underlying motivations. This new programming model aims to address some limitations of the submission process within the STF model, resulting in more dynamic task graphs.

Chapter 4 explores the implementation of this model in the STARPU runtime system. A first approach uses the existing data manager to process hierarchical tasks entirely at submission time. Then, we extend said data manager to function at runtime, which enables dynamic modifications of the task graph during the execution.

Chapter 5 validates the hierarchical task model by evaluating its behavior in the context of dense linear algebra. We take this opportunity to explain how the matrix descriptors and kernels of the CHAMELEON library were extended for hierarchical tasks.

Finally, chapter 6 deals with some preliminary work done to apply the hierarchical task model to distributed memory.

Chapter 1

Background & Related Work

Contents

1.1	Task-based programming models	13
1.2	Runtime systems for modern architectures	16
1.3	Relaxing the models to face modern problems	18

THE ADVENT OF MULTICORE TECHNOLOGY in the early 2000s brought a sharp break with the past for scientific computing. Researchers had to rethink their methods and algorithms to take advantage of increasing levels of parallelism. Since then, the number of cores per processor has grown steadily, while accelerators and GPU devices have become increasingly popular because of their massive computing power. Supercomputing nodes now commonly include multiple multicore CPUs and multiple accelerators. Such nodes are assembled in huge numbers to achieve extreme performance in a scalable way. The large scale and heterogeneity of these architectures, equipped with processing units and memories of different speed and capabilities, and interconnects with different bandwidths and latencies, bring numerous challenges, from the choice of parallel programming models to the need for new or redesigned methods to better take advantage of such systems. This common problem is known as the performance portability issue.

In the diverse landscape of supercomputing architectures described above, and due to the increasing complexity of algorithms for scientific computing, traditional methods for implementing parallel applications based on a mix of different technologies (e.g., MPI [26], OPENMP [9] and CUDA [39]/HIP [7]), may not be sufficient to achieve high performance, scalability, performance portability, and code maintainability. A need has gradually emerged for novel parallel programming models and tools that address the diversity and heterogeneity of modern platforms in a consistent manner, relieving programmers from architectural details, while making a code efficient and portable across a wide range of architectures with minimal modification. Modern *runtime systems* are designed to meet this demand. A runtime system basically consists of a **Programming Model** (to let the programmer express the workload), a **Scheduler** (to decide when and on which device a particular operation will be executed), a **Data Manager** (to transparently handle data coherency and transfers) and a set of **Drivers** (to drive the execution of tasks on a type of computing unit).

The first purpose of runtime systems is thus to provide *abstraction*. Runtime systems offer a uniform programming interface for a specific subset of hardware (e.g., OpenGL or DirectX are well-established examples of runtime systems dedicated to hardware-accelerated graphics) or low-level software entities (e.g., POSIX-thread implementations). They are designed as thin user-level software layers that complement the basic, general purpose functions provided by the operating system calls. Applications then target these uniform programming interfaces in a portable manner. Low-level, hardware dependent details are hidden inside runtime systems. The adaptation of runtime systems is commonly handled through drivers. The abstraction provided by runtime systems thus enables portability. Abstraction alone is however not enough to provide performance portability, as it does nothing to leverage low-level-specific features to increase performance.

Consequently, the second role of runtime systems is to *optimize* abstract application requests by dynamically mapping them onto low-level requests and resources as efficiently as possible. This mapping process makes use of scheduling algorithms and

heuristics to decide the best actions to take for a given metric and application state at a given point in its execution time. This allows applications to easily take full advantage of available underlying low-level capabilities to their full extent without breaking their portability. Thus, optimization combined with abstraction allows runtime systems to offer portability of performance.

In the specific case of parallel work mapping, other approaches have occasionally been adopted instead of using runtimes. Many scientific applications and libraries, including linear system solvers, integrate their own custom dynamic scheduling algorithms [8] or even resort to static scheduling techniques [32], either for historical reasons, or to avoid the potential overhead of an extra runtime layer.

However, as multicore processors densify, as cache and memory hierarchies deepen, the resulting increase in complexity now makes the use of work-mapping runtime systems virtually unavoidable. Such runtime systems take elementary task descriptions and dependencies as input and are responsible for dynamically scheduling the tasks on the available computing units so as to minimize a given cost function (usually the execution time) under some predefined set of constraints.

Work-mapping runtime systems themselves are now facing new challenges with the recent move of the high performance community towards the use of specialized accelerating cores together with traditional general-purpose cores. Not only do they have to decide whether or not to take advantage of specific hardware features, but they also have to decide whether entire application tasks should be executed on an accelerated device or better left on a standard core.

In the case where specialized cores are located on an expansion card having its own memory (e.g., GPUs, FPGAs, ...), the input data of a task must be copied from central memory to the card's memory before the task can be run. The output results must also be copied back to the central memory once the task computation is complete. The cost of copying data between central memory and accelerator memory is not negligible. This cost, as well as data dependencies between tasks, must be taken into account by the scheduling algorithms when deciding whether to offload a given task, to avoid unnecessary data transfers. Transfers should also be done in advance and asynchronously in order to overlap communication with computation.

In the rest of this chapter, we present the major programming models used to interact with task-based runtime systems. We then give a broad overview of recent contributions to runtime system design. Finally, we focus on approaches aimed at improving the behavior of task-based runtime systems.

1.1 Task-based programming models

Modern task-based runtime systems aim at abstracting the low-level details of the hardware architecture and enhance the portability of the performance of the code designed on top of them. As it is the case in this thesis, in most cases this abstraction

relies on a *Directed Acyclic Graph (DAG) of tasks*. In this DAG, vertices represent the tasks to be executed, while edges represent the dependencies between them.

While tasks are almost systematically explicitly encoded, runtime systems offer multiple ways to encode the dependencies of the DAG. Each runtime system usually comes with its own API which includes one or multiple ways to encode the dependencies, and their exhaustive listing would be out of the scope of this thesis. However, we may consider that there are two main modes for encoding dependencies. The most natural method consists in declaring *explicit dependencies* between tasks. In spite of the simplicity of the concept, this approach may have a limited productivity in practice as some algorithms may have dependencies that are difficult to express. Alternatively, dependencies may be implicitly computed by the runtime system thanks to the *sequential consistency*. In this latter approach, tasks are provided in sequence and the data they operate on are also declared.

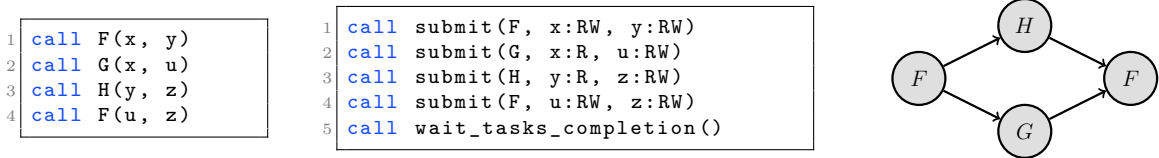


Figure 1.1: Pseudo-code for a dummy sequential algorithm (*left*), corresponding STF version (*center*) and subsequent DAG (*right*).

We illustrate this mode of expression of dependencies with a simple example relying on a minimum number of pseudo-instructions. Assume we want to encode the DAG shown in Figure 1.1 (right) relying on explicit dependencies. A task can be defined as an instance of a function working on a specific set of data, different tasks possibly being different instances of a same function. For example, in our example we can see that the first and final tasks are instances of function *F*. Tasks are instantiated with the `submit_task` pseudo-instruction (see Figure 1.1, center). Implicit dependencies aim at letting the runtime system automatically infer dependencies thanks to the so-called superscalar analysis [5] which ensures that the parallelization does not violate dependencies, following the sequential consistency. While CPUs implement such a superscalar analysis on chip at the instruction level [5], runtime systems implement it on tasks in a software layer. Superscalar analysis is performed on tasks and the associated input/output data they operate on. Assume that the first task *F* operates on data *x* and *y* in read/write mode (calling *F(x, y)*), while task *G* (resp. *H*) uses data *x* (resp. *y*) in read mode. Because of possible data hazards occurring on *x* (resp. *y*) between tasks *F* and *G* (resp. *H*), the superscalar analysis detects that a dependency is required to respect the sequential consistency.

Another important paradigm for handling dependencies consists of *recursive submission*. Indeed, it may be convenient for the programmer to let tasks trigger other tasks. Sometimes, one may need the task to be fully completed and cleaned up before triggering other tasks. Runtime systems often support this option through a

so-called *call-back* mechanism consisting of a post-processing portion of code executed once the task is completed and cleaned up.

Depending on the context, the programmer’s affinity, and the portion of the algorithm to encode, different paradigms may be considered natural and appropriate. Alternatively, one may rely on a well-defined and simpler programming model in order to design a relatively simpler code that is easier to maintain and benefit from the model’s properties. The *Sequential Task Flow* (STF) programming model consists in relying entirely on *sequential consistency* using only implicit dependencies. Thus, the STF model consists of submitting a sequence of tasks through non-blocking function calls that delegates the execution of the tasks to the runtime system. Upon submission, the runtime system adds a task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [5]. The actual execution of the task is then postponed until its dependencies are satisfied. As noted above, this paradigm is also sometimes referred to as *superscalar* because it mimics the operation of superscalar processors, where instructions are issued sequentially from a single stream, but can actually be executed in a different order, and possibly in parallel, depending on their mutual dependencies.

One challenge in scaling to large scale many-core systems is how to compactly represent extremely large DAGs of tasks. In [21, 20], the authors present a model, namely the *Parameterized Task Graph* (PTG), that addresses this problem. In this model, tasks are not enumerated but parameterized, and dependencies between tasks are explicit. For example, in the DAG represented in Figure 1.1 (right), the initial and final tasks are two instances of the same type of task that implements F . This property can be used to encode the DAG in a compact way, resulting in a smaller memory footprint for its representation, as well as ensuring limited complexity for parsing it as the problem size grows. For this reason, the memory consumption overhead in the runtime system for representing the DAG can be much lower for the PTG model than for the STF model. In addition, in the STF model, the DAG must be completely unrolled, whereas in the PTG model, the DAG is only partially unfolded during the execution following the task progression. From this point of view, the advantage of the PTG approach over the STF one can be crucial in a distributed memory context because the DAG is pruned on every nodes and only a portion of the DAG is represented on each node. This could significantly reduce the runtime overhead for the management of the DAG. On the other hand, knowing the entire DAG can be useful to compute the schedule of the DAG or to provide information to the dynamic scheduler by prepossessing the DAG. The PTG model has been successfully used and implemented within the PARSEC runtime system [22]. More recently, the *Template Task Graph* (TTG) was introduced in [43]. TTG is a generalization of PTG with respect to the dynamic aspect of graph discovery and generation. It extends the notions of parameter and dependency management (data-dependent task dependencies selection). This allows for dynamic construction of the DAG depending on the result of already computed tasks.

1.2 Runtime systems for modern architectures

Many initiatives have emerged in the past years to develop efficient runtime systems for modern heterogeneous platforms. Most of these runtime systems use a task-based paradigm to express concurrency and dependencies, using a task dependency graph to represent the application to be executed. The main differences between all the approaches are related, for the programming models, to whether or not they manage data movements between computational resources, to which extent they focus on task scheduling and finally to whether or not they handle distributed memory parallelism.

Most of the available runtime systems do not target any specific type of applications and provide a general API. Qilin [37], for example, provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of an heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB [42] technology) and GPUs (using CUDA) [39]. Another relevant framework is CHARM++ [33] which is a parallel variant of the C++ language that provides sophisticated load balancing and a large number of communication optimization mechanisms. CHARM++ has been extended to provide support for accelerators such as the Cell processors as well as GPUs [35]. Many runtime systems propose a task-based programming paradigm. Runtime systems like KAAPI/XKAAPI [28] or APC+ [29], Legion/Realm [14, 47] offer support for hybrid platforms mixing CPUs and GPUs. Their data management is based on a DSM-like mechanism: each data block is associated with a bitmap that permits to determine whether there is already a copy locally available to a specific processing unit or not. Moreover, task scheduling within XKAAPI is based on work-stealing mechanisms or on graph partitioning. The STARSS project is actually an umbrella term that describes both the STARSS language extensions and a collection of runtime systems targeting different types of platforms [13, 12]. STARSS provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. Later on, OMPSS [23, 25] was introduced by the same authors in an effort to integrate features from the STARSS programming model into a single programming model. This effort, was then pushed further in the context of the second generation programming model OMPSS-2 [45, 1]. The PARSEC [16] (formerly DAGuE [17]) runtime system dynamically schedules tasks within a node using a rather simple strategy based on a locality-aware work-stealing strategy. It was first introduced for linear algebra but was later extended to more generic applications. It takes advantage of the specific shape of the task graphs (in the sense that there are few types of tasks) to represent the task dependency graph in an algebraic fashion. The STARPU runtime system [11] provides a generic interface for developing parallel, task-based applications. It supports multicore architectures equipped with accelerator as well as distributed memory systems. This runtime is capable of transparently handling data and provides a rich panel of features. The details of this runtime systems are given in the next section. All the efforts mentioned above have contributed to proving the ease of use, the effectiveness and portability

of general purpose runtime systems to the point where the OPENMP [9] board has decided to include similar features in the OPENMP standard since version 4.0: the `task` construct was extended with the `depend` clause which enables the OPENMP runtime to automatically detect dependencies among tasks and consequently schedule them. The same OPENMP standard also provides constructs for using accelerator devices.

The common point among these runtime systems is that they all use high-level descriptions of dependencies to build the task graph at runtime and then schedule the corresponding computations on available resources. Various approaches are used to construct the task graph. For instance, most of the previously mentioned runtime systems rely on the STF model (e.g. OPENMP, STARSS, STARPU) to build the task graph. On the other hand, runtime systems such as PARSEC are based on the parameterized task graph programming model (PTG) [21]. Other runtime systems use different paradigms to express computations, such as Legion, which describes logical data regions representing the data flow and dependencies between tasks. The various programming models have different levels of ease of use and the amount of overhead they impose on the underlying runtime system. For example, the sequential task flow model is easy to use as programmers only need to provide the sequential implementation of their application and then add data access modes. However, this method comes with higher overhead in the runtime system. On the other hand, the parameterized task graph approach requires users to express their computations in a subtle high-level formalism where the dataflow is explicitly described, but has less overhead on the runtime system [30, 40].

While task-based runtime systems have been mainly research tools in the past, their recent progress makes them solid candidates for designing advanced scientific software as they provide programming paradigms that enable the programmer to express concurrency in a simple yet effective way and relieve him from the burden of dealing with low-level architectural details.

The key features which need to be constantly considered within task-based runtime systems are related to:

1. the overhead of the runtime which needs to be as low as possible,
2. the flexibility of the programming model,
3. the efficient exploitation of the underlying platform.

These aspects have been extensively studied and many contributions have been made. Most of them concern either the actual implementation of the runtime and/or the relaxation of the corresponding programming model.

Several efforts have been made to address the problem of reducing the overhead of task-based runtime systems, which mainly focus on those based on the sequential task flow model or increasing the amount of parallelism provided by these systems. In [6], the authors analyzed the limiting factors in the scalability of a task-based

runtime system and proposed individual solutions for each of the identified challenges, such as a wait-free dependency system and a scalable scheduler design based on delegation instead of work-stealing. Other approaches focus on advanced dependency management. For example, in [24], the authors proposed an eager approach for releasing data dependencies, where tasks are launched for execution as soon as their data requirements are met instead of waiting for predecessor tasks to finish execution. In [38], worksharing tasks were introduced, which internally leverage worksharing techniques to exploit fine-grained structured loop-based parallelism without requiring a barrier.

1.3 Relaxing the models to face modern problems

As shown in the introductory chapter, the granularity required by each computing resource may be very different. Finding the best granularity to optimally exploit the platform may be a difficult problem. A natural solution would be to rely on several granularities (one for each type of resource). However, handling task-graphs using several levels of granularities can be tedious from the programmer's point of view. Some preliminary work targeting heterogeneous architectures has considered splitting tasks when assigned to CPU cores in PARSEC [48, 18] and XKaapi [28]. However, in the case where a coarse-grain task is split when assigned to CPU cores, the submitted subgraph needs to be entirely completed before releasing the coarse dependency, which is equivalent to having a barrier at the end of the subgraph. This is represented in the scenario illustrated in Figure 1.2b where we can see that the red dependency between the two parent tasks is still expressed at coarse grain. An alternative approach consisted in considering the dual problem: instead of splitting tasks at runtime, resources are aggregated. Then, parallel implementation of the tasks are used to exploit the set of aggregated resources. This approach was explored in [19]. This scenario corresponds to the one depicted in Figure 1.2a where we can see that the parallelism within the task is hidden to the runtime system. From a broad point of view, the main difference between the two strategies is mainly related to the fact that resource aggregation will rely on an existing parallel implementation of the task, while the task splitting approach will require a fine grain implementation of the split task using the considered runtime system.

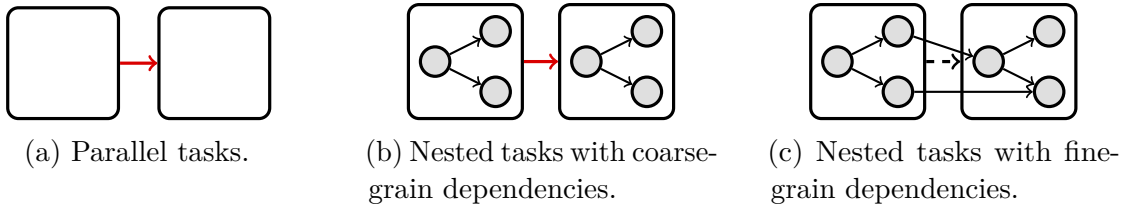


Figure 1.2: Nested task parallelism management.

Concurrently to these questions, the problem of optimizing task dependencies

management when using nested tasking was explored in [41] with the concept of *weak dependencies*. It is an extension of the OPENMP model that supports fine-grained dependencies not only between sibling tasks but also between tasks with any family relationship. Our objective in this thesis is to define a model that exhibits the same properties as task splitting while supporting fine-grain dependencies when exploiting heterogeneous systems. Figure 1.2c presents an example of such fine grain dependencies.

Runtime	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
PARSEC [48]	✗	✗	✓
OMPSS [41]	✓	✗	✗
TASKFLOW [31]	✗	✗	✓
IRIS [34]	✗	✓	✓
LIBTASK [10]	✗	✓	✓

Table 1.1: Summary of hierarchical task support in task-based runtime systems.

Several efforts have been made to allow task-based runtime systems to have more dynamic capabilities in terms of task management and generation. In TASKFLOW [31], advanced tasking schemes are introduced, including dynamic, composable, and conditional tasking. Dynamic tasking, in particular, allows for the dynamic generation of a sub-DAG from a given task. However, synchronization is added at the end of each hierarchical task to ease the dependency management, and programmers are responsible for data management and changing data layout as needed. In [34], the IRIS runtime is introduced, which can perform dynamic task partitioning either through user input or automatically via a polyhedral compiler, but no details are provided on how dependencies are handled in this context. Finally, LIBTASK, an advanced runtime system supporting hierarchical tasks in the context of low-rank linear algebra solvers is presented in [10]. This work introduces hierarchical tasks, and dependencies are expressed at the finest level. The correctness of the produced DAG is considered through extra automatic dependencies, but data management is straightforward as data partitioning is performed statically at the beginning of the execution.

Table 1.1 provides a summary of how the runtime systems mentioned in this section compare with respect to the functionality we want to achieve with hierarchical tasks. We focus on three main features. The first one is the presence of fine-grain dependencies between distinct sub-graphs. The second one is a data manager that can adjust the data layout around hierarchical tasks without user supervision. The third one is the support of heterogeneous architectures.

Chapter 2

The StarPU Runtime System

Contents

2.1	Data structures	22
2.1.1	Codelets	22
2.1.2	Data handles	23
2.1.3	Tasks and jobs	24
2.2	Building the graph	24
2.2.1	Shared memory	25
2.2.2	Distributed memory	25
2.3	Executing the graph	27
2.3.1	Submission time and execution time	27
2.3.2	Scheduling	28
2.4	Data management: partitioning operations	29
2.4.1	Partitioning data manually	29
2.4.2	Automatic data manager	34
2.5	Conclusion	40

DURING THIS THESIS we mainly worked on the STARPU runtime system, which we present in this chapter. More specifically, we give an overview of the STARPU terminology and mechanisms that are necessary for the understanding of this document. We start by introducing the main data structures of STARPU: *handles* and *codelets*. The former are an abstraction for the data used in the applications, and the latter store the different implementations of a task. We then explain how STARPU implements the STF model to construct the DAG of an application, either on a single node or on multiple nodes using a distributed paradigm. Next, we detail the execution flow of an application to complete the picture of how STARPU operates prior to the addition of hierarchical tasks. Finally, we take a look at data partitioning and its challenges from a programming model perspective. This leads us to introduce STARPU's data manager by explaining how it handles data partitioning automatically with minimal input from programmers.

2.1 Data structures

STARPU utilizes various data structures to facilitate the portability of its applications and to achieve efficient communications between different computing units or nodes. In this section, we introduce the main data structures of STARPU, *codelets*, *data handles* and *tasks* and explain how they enable users to write efficient and portable programs with ease.

2.1.1 Codelets

Programming hybrid architectures is one of the main features of the STARPU runtime system. Such architectures require different implementations of a task, each suitable for a specific type of core, for example a CPU kernel and a CUDA kernel. The role of *codelets* is to collect these kernel implementations into a single data structure, which can then be used to submit tasks that can run on any of the different computing cores of a node. Codelets also encapsulate the attributes shared by all the tasks performing the same computation, regardless of the chosen implementation. This includes debugging information, like the name of the operation, data information, and some specific options, such as restrictions on the type of cores targeted by a task, or flags to use when executing certain kernels.

Figure 2.1 presents how programmers can define a new codelet. Note that the **funcs** members (lines 3 to 5) are arrays and can store pointers to several implementations for a single type of worker. STARPU can then use its performance models to pick the best alternative on a given core.

```
1 struct starpu_codelet new_cl = {
2     /* Task implementations */
3     .cpu_funcs    = { cpu_impl1, cpu_impl2 },
4     .cuda_funcs   = { cuda_impl },
5     .opencl_funcs = { opencl_impl },
6     /* Data information */
7     .nbuffers     = 3,
8     .modes        = { STARPU_R, STARPU_W, STARPU_RW },
9     /* Debug information */
10    .name         = "new"
11};
```

Figure 2.1: Writing a new codelet.

2.1.2 Data handles

A codelet alone does not suffice to create a task, it also requires data to work on. For this reason, STARPU applications generally begin by registering the data that its tasks will access. This registration step abstracts data into *handles*, an opaque data structure that defines the properties of a piece of data within the runtime system. It allows STARPU to efficiently manipulate data, for example by maintaining coherent copies of a piece of data to limit data movements (e.g. across NUMA nodes).

```
1 float A[SIZE];
2 starpu_data_handle_t handleA;
3
4 starpu_vector_data_register(&handleA, STARPU_MAIN_RAM, A,
5                             SIZE, sizeof(float));
6
7 /* In a distributed memory context: */
8 starpu_mpi_data_register(handleA, tagA, mpi_rank);
9
10 /* Migrate the data */
11 starpu_mpi_data_migrate(MPI_COMM_WORLD, handleA, mpi_rank);
```

Figure 2.2: Registering a handle.

Figure 2.2 illustrates the registration of a vector. It makes use of one of STARPU’s predefined *data interfaces*, which facilitate the registration of commonly used data types, such as vectors (in this case) or matrices. It is however possible for programmers to define their own data interface by providing STARPU the corresponding function implementations. These functions include the registration function as well as functions used to allocate and free data buffers, to copy and transfer them between different memories, to access some properties of the handle (for example a matrix leading dimension), etc.

When using STARPU in the context of distributed memory, an additional function call is required to set the ownership of each handle to a certain node (identified by its `mpi_rank` in Figure 2.2). The owner of a handle is the node in charge of sending it to the other MPI processes using it, and of updating it in case it was modified. This operation also associates the handle with an MPI tag that will be used in data

```

1 starpu_task_insert(&new_cl,          /* Task codelet */
2                     STARPU_R, handleA, /* Data in Read-only mode */
3                     STARPU_W, handleB, /* Data in Write-only mode */
4                     STARPU_RW, handleC, /* Data in Read-Write mode */
5                     STARPU_VALUE, &cst, sizeof, /* Constant value */
6                     STARPU_CALLBACK, callback, /* Callback function */
7                     STARPU_PRIORITY, 10,      /* Task priority */
8                     0);

```

Figure 2.3: Submitting a task.

transfers. This data mapping is not definitive and the application can dynamically update it through data migration (see Figure 2.2, line 11).

Overall, the handle data structure enables STARPU to fully manage the data of an application. This mainly involves transfers between memories within a node and inter-node communications.

2.1.3 Tasks and jobs

The *task* structure combines a codelet and a set of handles to define a specific computation in the application. It instantiates the codelet and completes it with parameters that are not shared with other tasks using the same codelet. The most important of these parameters are the handles on which the task operate. As shown in Figure 2.3, they are passed to the task along with their access modes (lines 2 to 4). The other parameters (lines 5 to 7) are optional and allow users to customize the behavior of the task. These include additional arguments for the codelet, callback functions to be executed before or after the task, priority information, etc.

Each task is eventually associated to a *job* structure. Unlike tasks, which are largely defined by users during their submission, jobs are an internal data structure of STARPU. They contain all the task properties that should only be accessed by STARPU. This includes debugging information, synchronization structures (mutexes), information on the task successors (to notify them upon completion), updates on the status of a task, etc.

For the sake of clarity, we do not make a distinction between task and job in the rest of this document. The “task” designation encompasses both structures as the differences between them are purely internal to STARPU.

2.2 Building the graph

The main data structures now being defined, we can now take a closer look into the submission process and explain how STARPU implements the STF model to construct its DAGs. In this section, we first go through the submission process of the standard *shared memory* case. Then we complete it with the addition of MPI communication

for the *distributed memory* case.

2.2.1 Shared memory

Tasks are the vertices of our graphs. To complete the picture, we need to explain how the edges, the *dependencies*, are constructed to connect those tasks. While it is possible to explicitly define dependencies “by hand” between tasks, STARPU will, by default, infer them implicitly. These implicit dependencies are actually data dependencies that maintain a sequential consistency [36] order as described in the paragraph below.

In STARPU, the application code is executed by a submission thread that will be responsible of the construction of the graph. When a new task T is inserted, its handles and their access modes are inspected. If a handle H_W is accessed in *read-write* or *write* mode, a dependency is added from the last task(s) that accessed H_W to T . In turn, T becomes the last task to modify that handle and the next task accessing H_W will depend on T . If a handle H_R is accessed in *read-only* mode, a dependency is added from the last task that modified H_R . As long as the following tasks access H_R in *read-only* mode, they will only depend on that last task. This process is entirely sequential: tasks are added one by one to depend from the previously inserted ones.

Consequently, two STARPU tasks can be considered independent (and therefore run in parallel) if they either operate on different handles, or on the same handles, but in *read-only* mode.

It is also possible for the user to disable the sequential consistency for a data handle, either globally or within a task. When the sequential consistency of a handle is disabled, no dependencies will ever be inferred from it. The user is then responsible of ensuring the correctness of the task graph.

Figure 2.4 shows how a dense Cholesky factorization can be implemented in STARPU. The code 2.4a is analogous to a tiled and sequential implementation of the operation but replaces function calls to the POTRF, TRSM, SYRK and GEMM kernels with task insertions. This approach removes the need to explicitly define parallel regions, and the resulting DAG 2.4b is able to exhibit all the parallelism available.

This process can be done entirely asynchronously, for example like in Figure 2.4, by submitting the entire graph and then waiting for the completion of every task.

2.2.2 Distributed memory

STARPU provides an interface to create task graphs in distributed memory, called STARPU-MPI [44].

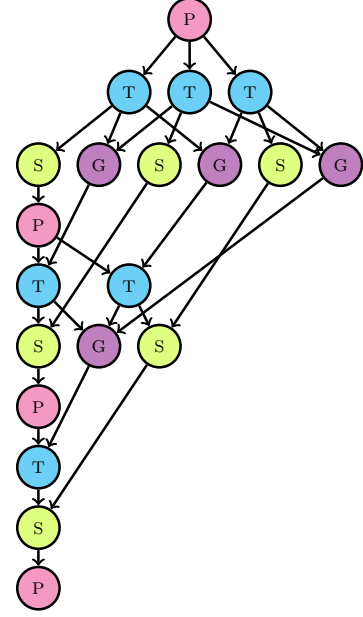
Porting a shared memory STARPU application to a distributed context is fairly straightforward and does not require much effort from programmers. The only

```

1 #define N 4
2
3 starpu_data_handle_t A[N][N];
4 [...] /* Register handles for each block of A */
5
6 for (k = 0; k < N; k++) {
7     starpu_task_insert(&potrf_cl,
8                       STARPU_RW, A[k][k], 0);
9     for (m = k+1; m < N; m++) {
10        starpu_task_insert(&trsm_cl,
11                          STARPU_R, A[k][k],
12                          STARPU_RW, A[m][k], 0);
13    }
14    for (n = k+1; n < N; n++) {
15        starpu_task_insert(&syrk_cl,
16                          STARPU_R, A[n][k],
17                          STARPU_RW, A[n][n], 0);
18        for (m = n+1; m < N; m++) {
19            starpu_task_insert(&gemm_cl,
20                              STARPU_R, A[m][k],
21                              STARPU_R, A[n][k],
22                              STARPU_RW, A[m][n], 0);
23        }
24    }
25 }
26
27 starpu_task_wait_for_all();

```

(a) STARPU code.



(b) DAG.

Figure 2.4: Cholesky factorization of a matrix composed of 4×4 tiles.

necessary modification is the mapping of the data over the different nodes as discussed in Section 2.1.2. STARPU-MPI implements a decentralized model without a master process, where all processes are equals¹. During the task submission step, each process goes through the submission of the entire task graph. The decision to execute a task on a given node is made locally at submission time and inferred from the data mapping. In the easiest case, every data handles of a task are owned by the same node, which will be responsible for its execution. In the more complicated cases, various policies can be selected to choose the node to execute the task.

If a task is assigned to a node N that does not own all of its handles, data transfers are necessary. When that situation occurs, N inserts a *recv* communication task for every handle it needs. Likewise, upon reaching the submission of that same task, the nodes owning these handles insert a *send* communication task. These send-receive pairs serve as inter-node dependencies, in addition to the usual STF dependencies that connects tasks inside the node.

This model depends on the fact that every node goes through the graph submission sequentially. This ensures that the decisions made locally are consistent between the processes. However, this approach unnecessarily inflates the runtime system overhead, since every node has to process the entire graph and to manage tasks it is not involved

¹Although a “master/slave” mode is implemented.

with. This issue can be addressed by *pruning* the DAG. The pruned DAG of a node only includes the tasks it will execute and the tasks that require communications (i.e. tasks using a handle owned by the node but executed on another one). STARPU can prune the graph automatically to a certain extent by interrupting the submission of a task once it can tell that they are irrelevant to the node. However this pruning is far from optimal as STARPU must go through a non-negligible section of the submission process before knowing if a task can be pruned or not.

A more efficient approach requires the programmer to handle pruning directly in the application. At the application level, it is possible to know which nodes are relevant for a given task (i.e. the nodes owning the different handles used by the task) *before* submitting that task. The application can therefore test the MPI rank of a process to only submit a task on a node if it is necessary.

2.3 Executing the graph

Now that we have described the construction of the task graph, the question of its execution arises. This section presents the concepts associated to the *runtime*. We first consider the asynchronous nature of STARPU and how it creates overlap between the submission process and the execution process. Next we discuss scheduling matters and give a general idea of how tasks are assigned to computing devices. These devices, in charge of executing tasks, are called *workers* in STARPU. In general, one worker is bound to each CPU core and to each accelerator. Binding multiple workers to a single CPU is possible but generally ill-advised. In the case of accelerators however, binding multiple workers to a single device can be useful (e.g. to use CUDA streams).

2.3.1 Submission time and execution time

The default model used by STARPU creates two different “timelines”, or *fronts* for the application. On the one hand, the submission front inserts new tasks into the graph. On the other hand, the execution front removes tasks from the graph as they are completed. This means that STARPU is never really aware of the entire graph, as it does not wait for the end of the submission process to start executing tasks. Instead, the graph is perceived as a set of *ready tasks* that can be executed, and a set of *non-ready tasks* waiting for their dependencies to be released.

Task insertions mainly direct the path of the submission front, but the STARPU interface contains methods to interact with the execution front. This is mostly achieved through `wait()` functions that create barriers blocking the submission process. These barriers can either wait for a specific task, a set of tasks, or all the submitted tasks. It is also possible to use `acquire()` operations to create data-centric barriers. These calls ensure that a piece of data will be available and up-to-date in the specified node and access mode. These barriers achieve this by waiting for all the tasks already submitted on the associated data handle. These operations can be blocking and

suspend the submission thread, or they can be submitted as tasks in the graph to be asynchronous.

2.3.2 Scheduling

A key actor of an application’s runtime is the scheduler that STARPU uses to assign the tasks to the various available workers. The scheduler implements a strategy aiming to optimize certain criteria, most notably minimizing the execution time. In STARPU, schedulers can be loosely defined with two operations. The *push* operation makes scheduling decisions when a task becomes ready to be executed. The *pop* operation is used by the workers when they finish executing a task and need a new one. Users can define their own scheduling strategies by providing implementations for the methods composing a scheduler. Many methods already implemented in STARPU are modular and can be re-used. This allows users to define their own schedulers or to tailor a scheduler to their application.

By default, STARPU uses the Locality Work Stealing (LWS) scheduler. With that scheduler, each worker has its own queue in which ready tasks will be *pushed*. When a worker wants a new task to execute, it *pops* the first one in its queue. If its queue is empty, the *pop* operation will attempt to steal a task from another worker’s queue, prioritizing its neighbors in the memory hierarchy of the machine to take advantage of caches. This is an example of “worker-centric” scheduler, which make decisions around the availability of the workers, handing them tasks when they request it.

Another popular scheduler in STARPU is Deque Model Data Aware (DMDA). This scheduler makes use of STARPU’s performance models to predict the completion time of a task for each device. When a task is *pushed*, it is scheduled onto the worker that minimizes that completion time (including the computation time of tasks already assigned to this worker). STARPU is then notified to prefetch the data it will use. When a worker is done with its previous task, the *pop* operation returns the first task in its queue. The DMDA scheduler has many variants, which can, for example, take into account task priorities or data availability on the devices. Such schedulers can be considered “task-centric”, meaning that their decisions are made when a task becomes ready to be executed.

Some schedulers are hybrid and implement aspects of both worker-centric and task-centric schedulers. An example of such scheduler in STARPU is HeteroPrio [4], which assigns one priority per type of computing units to each task (e.g. one for CPUs and one for GPUs). When a task becomes ready, it is *pushed* in a queue with other tasks sharing the same couple of priorities. When a worker needs work, it looks in the queues to *pop* a task. Each type of worker checks the queues in a different order, starting with those having the highest priority for its type.

2.4 Data management: partitioning operations

The last STARPU mechanism we want to introduce allows programmers to divide a piece of data into smaller chunks. This is often useful in order to express more parallelism in specific parts of the task graph, by creating data parallelism out of larger data pieces. In some applications, it is a natural way to represent data. In the context of linear algebra for example, it is tempting to register a large piece of data, such as a matrix, before dividing it into blocks. It can also be used to achieve greater memory efficiency depending on the resources of the system, and address load balancing issues. This process of dividing a piece of data into smaller chunks is called data partitioning. In the rest of this document, we call the large data *parent* and the smaller chunks *children*. In the context of STARPU, all the children are represented by handles, that we call the *sub-handles* of the parent.

This section presents the evolution of the data management in STARPU, from a manual approach requiring programmer input to a more automated process. Data management is a central notion in this thesis, and the automatic data manager in particular is at the core of the programming model we propose in the following chapters. For this reason, we present it in detail, which will make our later extensions to it clearer.

2.4.1 Partitioning data manually

While it is certainly a convenient feature, data partitioning can be a bit tricky to implement. First, it creates another view of the partitioned data that must be kept consistent with the original one. This means that some mechanism is needed to prevent users from creating a graph where a piece of data and its children are modified simultaneously. On the other hand, as long as the data accesses are *read-only*, they can happen concurrently.

For the sake of generality, it should also be possible to partition a data handle that is already the child of another data. This results in a *partition tree* associated with each higher level data. If there is a need for different partitions of the same data, that data can have multiple partition trees. For example, Figure 2.5 represents two partition trees for the matrix *A*: vertical blocks on the left side, and horizontal blocks on the right side (with one of the children partitioned vertically).

In addition to the partitioning operation, which divides a parent into children, it is also necessary to implement a symmetrical *unpartitioning* operation, which merges the children back into their parent.

Synchronous partitioning

The initial implementation of data partitioning in STARPU was synchronous: the operation happens when partitioning function calls are reached in the submission thread. To protect data consistency, partitioning calls have to be blocking. Partition

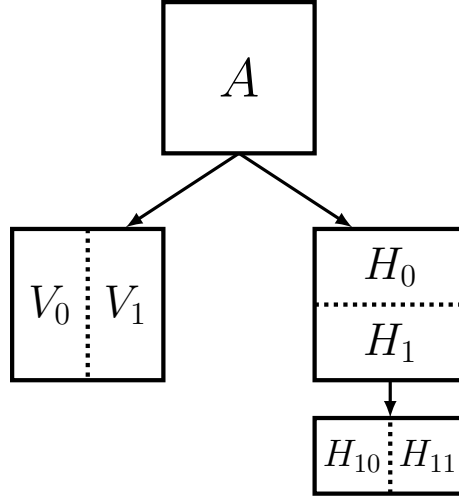


Figure 2.5: Matrix with two partition trees.

calls wait for all the tasks working on the parent handle. Unpartition calls wait for all the tasks working on the children.

The first step in introducing data partitioning in a STARPU application is to define a *data filter*. This is common to all the approaches presented in this section. A data filter is a simple structure used to describe a partitioning operation. It consists of a filter function and information on the number of children. The filter function is called once per child to fill its interface with the appropriate information, mainly by computing the pointer to the start of a sub-data and its dimensions. These filter functions are part of the methods defined for the data interfaces mentioned in Section 2.1.2, and STARPU provides a variety of them for its predefined interfaces.

Once a data filter is created, users can use partitioning and unpartitioning functions in their applications. Calling `starpu_data_partition(handle, &filter)` will wait for the completion of all tasks working on `handle` and then creates its children using `filter`. Users can then submit tasks working on the children (stored in `handle`). To use the parent `handle` in tasks again, users must call `starpu_data_unpartition(handle, STARPU_MAIN_RAM)`. Upon reaching this function call, STARPU will wait for all the tasks that are working on the children of `handle`. It will then collect these children back into one large piece of data in the designated memory node (here `STARPU_MAIN_RAM`).

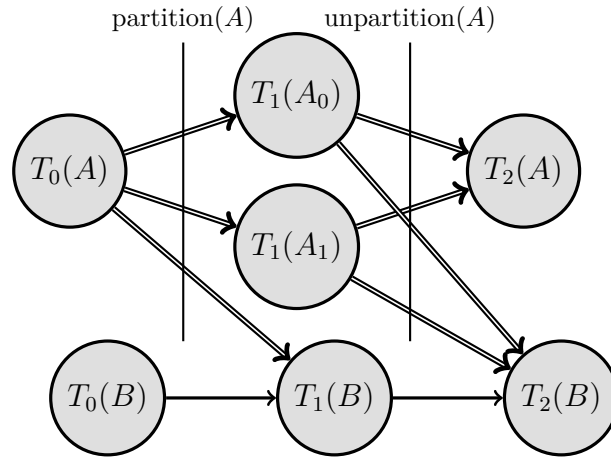
Figure 2.6 is an example of synchronous partitioning. The programmer submits two chains of three tasks, working respectively on data A and B . Instead of submitting a single task in the middle of the chain working on A , we partition A into $\{A_0, A_1\}$ and work on these sub-handles (lines 15 to 23 of Figure 2.6a), before reconstituting A with an unpartition for the final task of the chain (lines 27 to 31). Each task working on B is submitted after the corresponding task working on A . Figure 2.6b illustrates the resulting task graph. Regular dependencies are represented with simple arrows,

```

1 #define NPARTS 2
2
3 [...] /* Registration of handleA & handleB */
4
5 /* Filter dividing a handle in NPARTS vertical slices */
6 struct starpu_data_filter f = {
7     .filter_func = starpu_vector_filter_block,
8     .nchildren = NPARTS
9 };
10
11 /* Insertion of tasks working on the large data */
12 starpu_task_insert(&T0_cl, STARPU_RW, handleA, 0);
13 starpu_task_insert(&T0_cl, STARPU_RW, handleB, 0);
14
15 /* Synchronous partitioning */
16 starpu_data_partition(handleA, &f);
17
18 /* Insertion of tasks working on the sub-data of handleA */
19 starpu_data_handle_t subhandle;
20 for (i = 0; i < NPARTS; i++) {
21     subhandle = starpu_data_get_child(handleA, i);
22     starpu_task_insert(&T1_cl, STARPU_RW, subhandle, 0);
23 }
24
25 starpu_task_insert(&T1_cl, STARPU_RW, handleB, 0);
26
27 /* Synchronous unpartitioning */
28 starpu_data_unpartition(handleA, STARPU_MAIN_RAM);
29
30 /* Insertion of tasks working on the large data */
31 starpu_task_insert(&T2_cl, STARPU_RW, handleA, 0);
32 starpu_task_insert(&T2_cl, STARPU_RW, handleB, 0);

```

(a) Code using synchronous partitioning.



(b) Graph resulting from Figure 2.6a.

Figure 2.6: Example of synchronous partitioning. Simple arrows are regular dependencies. Double arrows represent the submission flow barriers caused by synchronous partitioning calls.

while double arrows represent the barriers in the submission flow blocking the insertion of a task until a synchronous call can be executed. For example, $T_1(B)$ cannot be submitted until the completion of $T_0(A)$ because the submission call is placed after a synchronous partition call. For the same reason, the insertion of $T_2(B)$ depends on the completion of the $T_1(A_i)$ tasks. This example showcases how blocking calls are used to enforce data consistency between a data piece and its children. It also reveals how this method can result in undesirable synchronizations by delaying the submission of tasks, effectively creating unnecessary dependencies. This can quickly become a bottleneck for the application. Furthermore, while it is possible to define multiple partition trees for a data (as shown in Figure 2.5) by writing different filters, manually switching from one to the other will quickly become tedious for the programmer.

Asynchronous partitioning

In response to the bottleneck issue, an asynchronous approach to data partitioning was implemented. Instead of applying the filter every time a partition is needed and accessing the children through their parents, a *plan* operation is introduced. This operation will only use the filter once, to register a new set of handles which will be the *sub-handles* of the parent handle.

Figure 2.7 is an example of asynchronous partitioning reusing the case presented in Figure 2.6. Lines 1 to 11 of Figure 2.7a are the data registration step, which defines a filter and creates a partition plan for `handleA`. Lines 13 to 27 are the task insertion steps, and Figure 2.7b shows the resulting graph. As shown in this example, once a partition plan is specified for a handle, users can use asynchronous variants of the partitioning functions presented earlier. These variants will insert partitioning and unpartitioning tasks using both the main handle and the sub-handles in *write* mode. Therefore, the rules of the STF model can be applied to correctly insert these tasks in the graph. By making the partition operation asynchronous, the barriers blocking the submission of the tasks $T_i(B)$ in Figure 2.6 are removed, and replaced by proper dependencies in the task chain using A .

At runtime, the partitioning tasks do not perform any computations (i.e. their codelet does not have any implementations). Instead, they serve as synchronization points. A *partition task* becomes ready when the tasks working on the parent handle are finished. Then, it invalidates that handle and makes the children usable. Symmetrically, an *unpartition task* becomes ready when the tasks working on the sub-handles are finished, and it invalidates the children and makes the parent usable again.

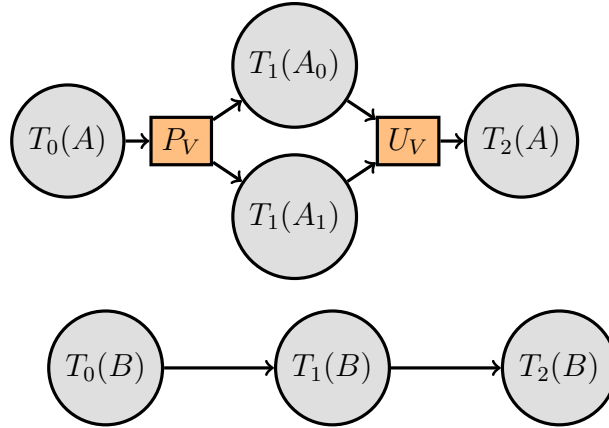
The asynchronous approach also implements *read-only* versions of the partition and unpartition functions. The *read-only* case calls for a different behavior because *read* accesses to a single piece of data can happen concurrently. This means that the partitioning and unpartitioning tasks do not have to invalidate any handles. This can help to express more parallelism in the application.

```

1 #define NPARTS 2
2
3 [...] /* Registration of handleA & handleB */
4
5 /* Partitioning in NPARTS slices */
6 struct starpu_data_filter f = {
7     .filter_func = starpu_vector_filter_block,
8     .nchildren = NPARTS
9 };
10 starpu_data_handle_t subhandlesA[NPARTS];
11 starpu_data_partition_plan(handleA, &f, subhandlesA);
12
13 /* Insertion of tasks working on the large data */
14 starpu_task_insert(&T0_cl, STARPU_RW, handleA, 0);
15 starpu_task_insert(&T0_cl, STARPU_RW, handleB, 0);
16
17 /* Insertion of a partitioning task */
18 starpu_data_partition_submit(handleA, NPARTS, subhandlesA);
19
20 /* Insertion of tasks working on the sub-data of handleA */
21 for (i = 0; i < NPARTS; i++)
22     starpu_task_insert(&T1_cl, STARPU_RW, subhandlesA[i], 0);
23
24 starpu_task_insert(&T1_cl, STARPU_RW, handleB, 0);
25
26 /* Insertion of an unpartitioning task */
27 starpu_data_unpartition_submit(handleA, NPARTS, subhandlesA, STARPU_MAIN_RAM);
28
29 /* Insertion of tasks working on the large data */
30 starpu_task_insert(&T2_cl, STARPU_RW, handleA, 0);
31 starpu_task_insert(&T2_cl, STARPU_RW, handleB, 0);

```

(a) Code using asynchronous partitioning.



(b) Graph resulting from Figure 2.7a.

Figure 2.7: Example of asynchronous partitioning.

However, just as in the synchronous case, this method of data partitioning still requires the user to add function calls at the appropriate times. It is even further complicated by the addition of read-only partitioning. This is not only tedious for the user, but it can also be a source of bugs in the application. To alleviate this problem, it may be desirable to have these operations handled automatically by STARPU.

2.4.2 Automatic data manager

```

1 #define PARTS 2
2 void submit_task_graph(int matrix[NX][NY]) {
3     starpu_data_handle_t handle;
4     starpu_data_handle_t v_handles[PARTS];
5     starpu_data_handle_t h_handles[PARTS];
6     starpu_data_handle_t h_subhandles[PARTS];
7
8     /* Data registration */
9     starpu_matrix_data_register(&handle, STARPU_MAIN_RAM,
10                                (uintptr_t)matrix, NX, NX, NY,
11                                sizeof(matrix[0][0]));
12
13     /* Partition the matrix in PARTS vertical slices */
14     struct starpu_data_filter vertF = {
15         .filter_func = starpu_matrix_filter_block,
16         .nchildren = PARTS
17     };
18     starpu_data_partition_plan(handle, &vertF, v_handles);
19
20     /* Partition the matrix in PARTS horizontal slices */
21     struct starpu_data_filter horF = {
22         .filter_func = starpu_matrix_filter_vertical_block,
23         .nchildren = PARTS
24     };
25     starpu_data_partition_plan(handle, &horF, h_handles);
26     starpu_data_partition_plan(h_handles[1], &vertF, h_subhandles);
27
28     /* Fill the matrix */
29     starpu_task_insert(&initialize, STARPU_W, handle, 0);
30
31     /* Modify the values via the vertical slices */
32     for (i = 0; i < PARTS; i++)
33         starpu_task_insert(&modify, STARPU_RW, v_handles[i], 0);
34
35     /* Check results with both horizontal and vertical slices */
36     for (i = 0; i < PARTS; i++)
37         starpu_task_insert(&check, STARPU_R, h_handles[i], 0);
38     for (i = 0; i < PARTS; i++)
39         starpu_task_insert(&check, STARPU_R, v_handles[i], 0);
40
41     /* Unregister data from StarPU. */
42     starpu_data_partition_clean(handle, PARTS, v_handles);
43     starpu_data_partition_clean(h_handles, PARTS, h_subhandles);
44     starpu_data_partition_clean(handle, PARTS, h_handles);
45     starpu_data_unregister(handle);
46 }

```

Figure 2.9a

Figure 2.9b

Figure 2.9c

Figure 2.9d

Figure 2.8: Example of automatic partitioning.

To relieve users of the burden and risk of manually managing data partition trees,

STARPU's default behavior now handles data partitioning automatically. Just as in the asynchronous case presented earlier, users call *plan* operations to define the partition trees they want to use and to register the corresponding sub-handles. During this step, they can introduce multiple partition plans for a single piece of data, and recursively partition sub-handles to create more levels of partitioning. In the code snippet 2.8, the lines 13 to 26 show how to define the partition trees of Figure 2.5. In the rest of the code, all the handles and sub-handles can be used directly to insert tasks without calling any of the previous partitioning functions.

The functions calls that inserts partitioning tasks are added internally at submission time. Five different operations are implemented. On top of the `partition_submit()` and `unpartition_submit()` and their read-only variants that were described in the previous subsection, the `readwrite_upgrade_submit()` operation turns a read-only partition into a read-write partition.

The handle structure defines the following states to describe the state of a handle during the submission of tasks:

- *Inactive*, the piece of data cannot be accessed.
- *Read-Write active*, the piece of data can be read and modified.
- *Read-only active*, the piece of data can only be read.

Additionally, an *active* handle can be:

- *Not partitioned*, if only the piece of data itself can be accessed, and not its children.
- *Read-Write partitioned*, if the piece of data can only accessed through its children.
- *Read-only partitioned*, if the piece of data and its children can be read.

The state of a handle is used to tell at submission time whether or not that handle would be usable at runtime. Table 2.1 summarizes in green the states indicating that a task can use the handle without generating an error at runtime. **RW** means that the handle can be accessed in *read-write*, *write* or *read-only* mode. **RO** means that the handle can only be accessed in *read-only* mode. A state in which a handle cannot be used at runtime is marked with **X**. Impossible states are indicated with **N/A**.

	Inactive	Read-Write active	Read-only active
Not partitioned	N/A	RW	RO
Read-Write partitioned	N/A	X	N/A
Read-only partitioned	N/A	RO	RO

Table 2.1: Usability of a handle depending on its state.

Each handle also contains information used to navigate the partition trees. This information includes a pointer to the parent handle (NULL in the case of the root

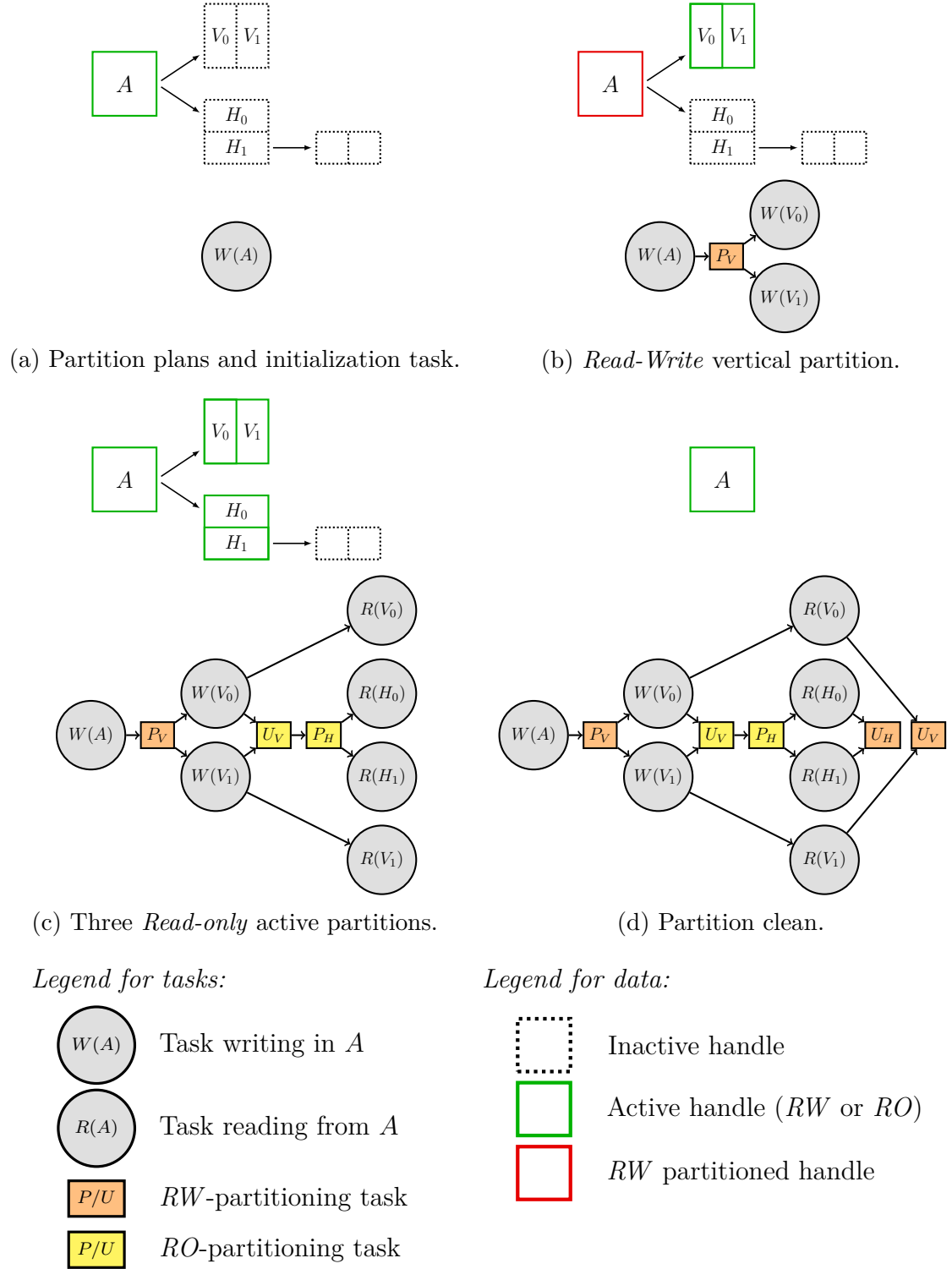


Figure 2.9: Evolution of the graph and handle states associated to the submission process of the code provided in Figure 2.8.

handle), pointers to the active children and to the siblings (NULL if there is none).

	Parent handle	Sub-handles
partition_submit()	RW active RW partitioned	RW active Not partitioned
unpartition_submit()	RW active Not partitioned	Inactive
partition_readonly_submit()	RW or RO active RO partitioned	RO active Not partitioned
unpartition_readonly_submit()	RO active RO partitioned	RO active Not partitioned
readwrite_upgrade_submit()	RW active RW partitioned	RW active Not partitioned

Table 2.2: Handle states after the submission of partitioning tasks.

Before inserting a new task into the graph, STARPU checks the states of the task’s handles to decide whether or not it should partition or unpartition them first. If the state of the handle indicates that it cannot be used, then the appropriate operation is inserted. If the handle is at the bottom of a partition tree, a chain of partitioning tasks will be recursively inserted to activate it. Similarly, chains of unpartitioning tasks can be recursively inserted to re-enable a handle higher in the tree. Every time a partitioning task is inserted, the states of the handles involved are updated according to Table 2.2.

Algorithm 1 Recursive traversal of the partition trees to activate *target* handle

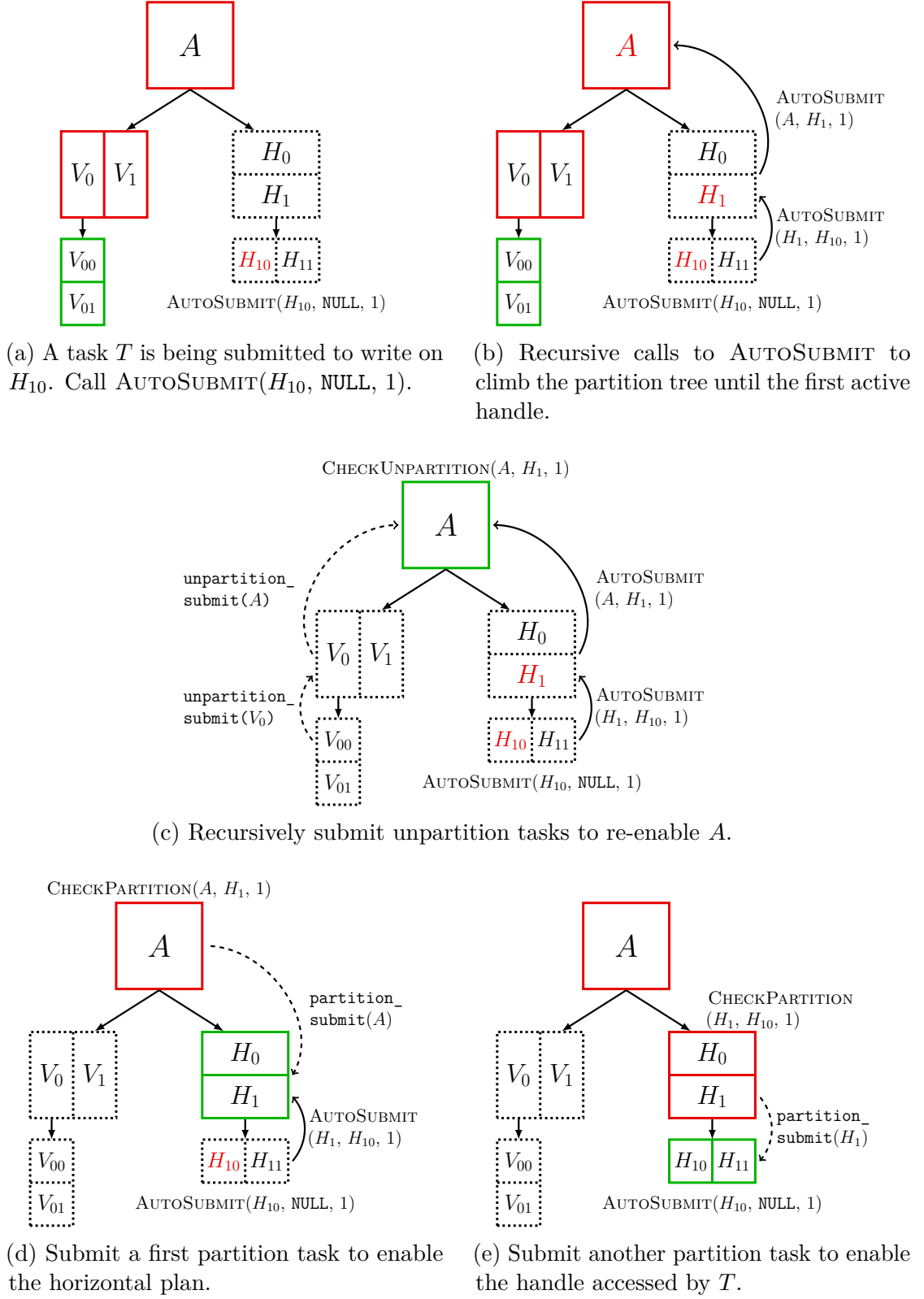
```

1: function AUTOSUBMIT(ancestor, target, write)
2:   if ancestor is Inactive or
      (write and parent is RO active) then
3:     AUTOSUBMIT(ancestor parent, ancestor, write)
4:   CHECKUNPARTITION(ancestor, target, write)
5:   if target is NULL then
6:     return
7:   CHECKPARTITION(ancestor, target, write)

```

Algorithm 1 is a pseudo-code version of the recursive function that is called on each handle of a new task to be inserted. The *ancestor* argument is the handle whose state is being checked. The *target* argument is NULL on the initial call to AUTOSUBMIT. During recursive calls, it is used to remember the *ancestor* of the previous call. The *write* argument is a boolean that is *true* if *target* uses the STARPU_RW or STARPU_W access modes and *false* if it uses the STARPU_R access mode.

Figure 2.10 illustrates how the partition tree of *A* is traversed to activate the handle H_{10} for a task *T*. Initially, the “vertical” branch of the tree is activated and


 Figure 2.10: Unrolling a call of AUTOSUBMIT .

RW partitioned (see Figure 2.10a). First, AUTOSUBMIT is recursively called to climb the partition tree until we reach an active ancestor of H_{10} matching the access mode. This corresponds to lines 2 and 3 of Algorithm 1 and to Figure 2.10b. Upon reaching that active handle, we call CHECKUNPARTITION (line 4).

Algorithm 2 Submission of unpartitioning tasks

```

1: function CHECKUNPARTITION(ancestor, target, write)
2:   if ancestor is Partitioned then
3:     if write then
4:       if target is NULL or
         ancestor active children  $\neq$  target siblings then
5:         rec_unpartition_submit(ancestor)
6:     else
7:       if ancestor state  $\neq$  RO partitioned then
8:         rec_unpartition_readonly_submit(ancestor)

```

CHECKUNPARTITION 2 is used to check whether a handle *ancestor* needs to be unpartitioned or not. If *ancestor* must be unpartitioned, then one of the function calls on lines 5 and 8 will recursively insert unpartitioning tasks by following the pointers to the active descendants of *ancestor*. The states of the handle traversed are updated accordingly. This process is illustrated by Figure 2.10c.

Algorithm 3 Submission of partitioning task

```

1: function CHECKPARTITION(ancestor, target, write)
2:   if ancestor is RO partitioned then
3:     if write then
4:       readwrite_upgrade_submit(ancestor, target siblings)
5:     else
6:       partition_readonly_submit(ancestor, target siblings)
7:   else
8:     if write then
9:       partition_submit(ancestor, target siblings)
10:    else
11:      partition_readonly_submit(ancestor, target siblings)

```

After this step, we determine if the active *ancestor* must be partitioned with CHECKPARTITION 3 (line 7 of Algorithm 1). The *target* argument is used to chose which plan should be activated by the partitioning operation, in case multiple plan were created for *ancestor*. Contrary to Algorithm 2, this algorithm inserts a single partition task, which is enough to put *target* in the correct state. As the call stack unwinds (Figures 2.10d and 2.10e), AUTOSUBMIT keeps using CHECKUNPARTITION and CHECKPARTITION to progressively update the partition tree until the initial handle is in the correct state for T to be submitted.

Figure 2.9 provides another example of this mechanism, showing the evolution of the handle states and the task graph during the submission process of the code from Figure 2.8. At the end of the data registration step, once the plan operations have been performed, only the root handle is active (see Figure 2.9a). When the first task using a sub-handle is submitted (line 33), STARPU first inserts the corresponding partitioning task. It also updates the states of the handles involved in the operation. A becomes *RW-partitioned* and the children V_i become *RW-active* (see Figure 2.9b). The next step (line 35) submits “check” tasks that access the data in read-only mode in both the vertical and horizontal layout. Because the vertical layout is already active, no partition operation is needed for the tasks using the V_i handles. On the other hand, the H_i handles are still *inactive* and A is also in an invalid state. Thus STARPU starts by inserting a read-only unpartitioning task to make A *RO-partitioned*. Then it submit a read-only partitioning task to activate the H_i handles. This results in Figure 2.9c. Finally, unpartitioning tasks are added before cleaning the partition plans (see Figure 2.9d).

2.5 Conclusion

The main goal of this thesis is to implement a model of hierarchical tasks in an existing runtime system. It is necessary to first present the runtime system in question, STARPU, in order to properly set the stage for the rest of the document. STARPU considers its tasks as the combination of a codelet and data. Codelets factorize the properties of a task, such as its different implementations targeting various computing resources. Data are registered at the beginning of an application as data handles. Tasks are submitted sequentially in a singular submission thread according to the STF model. STARPU creates incoming dependencies for each new task based on the access modes of the data pieces it uses, relying on sequential consistency.

The execution of the task graph thus constructed is overlapped with its submission. This means that STARPU never gets a full picture of the DAG of an application, but rather see it through a sliding window. On the one hand, the submission thread inserts new tasks. On the other hand, the workers remove tasks from the graph by executing them. Assigning a task to a worker can be done through various scheduling policies centered around customizable push and pop operations. The push operation occurs when a task is ready to be executed, while the pop operation occurs when a worker is looking for a task to execute.

The most important component for the introduction of hierarchical tasks in STARPU is the data manager. This mechanism enables the programmer to declare different data layouts and use them transparently in the application. STARPU is in charge of automatically creating the partitioning operations required at runtime to maintain the consistency between the data layouts. These operations are positioned in the DAG as tasks during the graph’s submission. Later in this thesis, we will have to extend this data manager to position these operations at execution time. However,

the user interface and the core principles are unchanged.

After presenting the existing elements of STARPU that are relevant for this thesis, we can build our hierarchical task model.

Chapter 3

The Hierarchical Task Model

Contents

3.1 Objectives	44
3.1.1 Improving upon the STF model	44
3.1.2 Introducing more dynamic task graphs	46
3.1.3 Keeping the model simple	47
3.2 Designing hierarchical tasks for StarPU	48
3.2.1 Aiming for generality	48
3.2.2 User interface	49
3.2.3 Adapting the data layout around hierarchical tasks	50
3.2.4 Correctly inserting tasks at runtime	51
3.2.5 Fine grain dependencies	53
3.2.6 Parallel submission	55
3.2.7 Data management for the programmer	57
3.2.8 Case of read-only access mode	57
3.3 Conclusion	58

FACING THE CHALLENGES posed by the increasing complexity of modern HPC architectures demands more and more adaptability and expressivity from runtime systems. In addition, runtime systems need to address the limitations imposed by the programming models they implement. To this end, we propose to extend the STF model implemented in the STARPU runtime system by introducing the concept of *hierarchical tasks*. A hierarchical task shares all the characteristics of a regular task with the added feature of being able to submit a sub-DAG. Such a task can either behave as a regular task and perform a given set of computations, or it can insert new tasks into the DAG. These new tasks are called *sub-tasks* of the hierarchical task that submitted them. The work done by the graph formed by these sub-tasks must produce the same results as the standard execution of the task they replaced.

In this chapter, we present the hierarchical task paradigm, its features, and how to use it. First, we review our objectives and explain the problems we are aiming to solve with hierarchical tasks. We then give a detailed presentation of our new programming model to explain how it addresses the aforementioned problems without diverging from the existing STF model. To do that, we iterate on our model to present how we address the challenges of inserting sub-graphs without creating inconsistencies in the graph. A central part of that discussion is the question of data management that emerges from the need to partition data handles to refine a task into a sub-graph. Finally, we discuss the use of hierarchical tasks in the context of distributed memory.

3.1 Objectives

Our main goal with hierarchical tasks is to introduce in STARPU a mechanism that enhances the submission process and allows a finer control over it, both automatically at the runtime system level and for users at the application level. In this section, we first go over the improvements that hierarchical tasks can bring to STARPU's programming model. We then focus on the common problems of all task-based runtime systems that we want to address with hierarchical tasks. The last part is dedicated to explaining our goals regarding the programming interface, since we want to keep our model simple to use while extending its expressivity.

3.1.1 Improving upon the STF model

In STARPU, the interactions between the submission process of the task graph and its execution can lead to different types of overhead depending on the nature of the DAG processed by the runtime system. With hierarchical tasks, we aim to reduce these overheads and target the following goals.

Parallel submission of the task graph

While the sequential nature of the STF model is the key to its simplicity (from users' perspective), it is also the root of its problems. Placing the entire burden of the task graph submission on the shoulder of a single thread can be a bottleneck for the application. If an application presents enough parallelism and the computing resources offer enough workers to exploit it, the execution front might catch up with the submission front. This can induce various levels of starvation for the workers.

For the same reason, we must minimize the interruptions of the submission thread in order to avoid bottlenecks in STARPU applications. This means that users must rely on asynchronous operations whenever possible and use barriers sparingly. However, some necessary interruptions are independent from the user. For example, the submission thread might have to wait for a mutex internal to the runtime. This can also lead to some level of starvation of the computing resources.

Hierarchical tasks address these limitations by introducing parallelism in the submission process. If we can share the insertion of tasks between different threads, then the bottleneck issue can be mitigated and interruptions of a submission thread are not as big of a deal.

More generally speaking, submitting and managing tasks with a runtime system will inevitably induce overhead. The order of magnitude of that overhead in STARPU is typically in the range of a few microseconds per task. As the number of workers increases, tasks must meet a minimal threshold in their execution time for that overhead to be negligible. If tasks are too short, the scaling of the application will be limited by the runtime overhead beyond a certain number of workers. This notion must be kept in mind as we introduce hierarchical tasks. Otherwise the benefits gained by parallel submission might be offset by the cost of the new mechanism itself.

Delay the submission of parts of the task graph

At first glance, it may seem advantageous to submit the task graph as quickly as possible. It prevents the aforementioned issues and provides the runtime system with more information about the future of the execution of the application. This improves the scheduling decisions and data transfer operations. However, the cost of task management increases along with the number of non-ready tasks accumulating in the system. It can negatively impact the runtime system overhead by increasing scheduling costs. It can also lead to excessive memory usage due to the allocation of a large number of task structures and data buffers.

In STARPU, this issue is generally handled by throttling the submission thread. There are mechanisms implemented to control the submission flow either at the runtime level, or at the application level. At the runtime level, STARPU can monitor its current memory usage (and anticipate future memory usage) to interrupt the submission flow between certain thresholds [44]. At the application level, Application Programming

Interface (API) functions or environment variables can be used to interrupt the submission based on the number of non-ready tasks already submitted [3].

With hierarchical tasks, we also aim to introduce a new mechanism to control the task submission flow. By encapsulating parts of the task graph in hierarchical tasks, it becomes possible to limit the number of non-ready tasks waiting in the system without interrupting the submission. Another benefit of using hierarchical tasks to delay parts of the submission process is that the hierarchical graph still gives STARPU an overview of the rough shape of the final graph. This overview is useful to inform certain scheduling decisions in advance or to anticipate data transfers, which is not possible when the submission thread is interrupted.

In some applications, some parts of the task graph cannot be known in advance, so it cannot be entirely submitted at startup. The submission of these unknown parts has to be delayed. For example, when factorizing sparse matrices with partial pivoting, rows are interchanged based on the numerical values of their elements. This makes it impossible to predict the exact final structure of the sparse matrices involved without completing the factorization [27]. Since the data layout is unpredictable, many regions of the task graph can only be inserted at the correct time. In this context, hierarchical tasks could be used to represent these regions and submit sub-graphs suiting the data layout as soon as it is known.

Automatic pruning in StarPU-MPI

In Section 2.2.2, we explained that when using STARPU-MPI to create distributed DAG, each node should prune irrelevant tasks and thus mitigates runtime overhead. When done automatically by the runtime system, the pruning does not fully solve the overhead issue as the submission process is still partially executed. When done by users at the application level, the pruning is more efficient but it requires implementation efforts that can be complex, especially if the data mapping is redefined.

We aim to use the submission flow control effects of hierarchical tasks to tackle this problem. Once again, the idea is to encapsulate parts of the task graph in hierarchical tasks. This would enable STARPU's automatic pruning to prevent the submission of many tasks by aborting the submission of a single, hierarchical, task. It would give users a new way to build distributed graphs without having to oversee their pruning.

3.1.2 Introducing more dynamic task graphs

Beyond leveraging hierarchical tasks to address specific shortcomings of the STARPU programming model, we also aim to tackle broader challenges faced by task-based runtime systems. In Section 1.3, we presented an overview of the current limitations of runtime systems and the state-of-the-art approaches employed to overcome them. The common trait of these limitations is the static nature of the task graphs handled by each runtime system. The use of hierarchical tasks also aims to bring more dynamism

into the expression of DAGs. By delegating parts of the DAG submission to specific tasks, the goal is to produce graphs that can evolve during execution time. This can create opportunities to shape the graph on the fly, for example by fitting the new graph to the availability of resources at a given time.

Optimal task granularity

In the context of heterogeneous platforms, there is a disparity in computing power between workers of different nature. For this reason, if one wants to fully exploit all the resources available to them, finding the appropriate task granularity (i.e. the amount of computations done in a task) is particularly important.

The use of hierarchical tasks could allow STARPU to dynamically reduce the granularity of a task by replacing it with a sub-graph that works at a smaller grain. This decision could be made by the scheduler depending on the priority of the coarse hierarchical task and the workers available when it becomes “ready”. If the sub-graph introduces enough parallelism, then the runtime system can take advantage of slower but more numerous workers (e.g. CPU cores) to run the computations faster than by waiting for a more powerful worker to be free. It is particularly interesting in the case of high-priority tasks, such as a task from the critical path or a task unlocking many other tasks.

The hierarchical task approach is adjacent to the parallel task solution, which relies on resource aggregation to efficiently execute coarse tasks on CPU cores [19] and is already implemented in STARPU. The two methods could eventually be used jointly to address the granularity issue.

More expressive task graphs

In addition to the potential performance improvements resulting from more dynamic hierarchical task graphs, hierarchical graphs can also be more expressive. For example, the capacity of hierarchical tasks to submit sub-graphs, themselves potentially composed of hierarchical sub-tasks, lends itself well to recursive algorithms.

Iterative solvers are another field that could benefit from this new expressivity. A single hierarchical task could submit a task graph that includes another hierarchical task identical to itself. Every time such a hierarchical task is ready, it can decide whether to iterate by inserting a new task graph or not. This process can be repeated until a termination criterion is met and the hierarchical task decides not to insert a task graph.

3.1.3 Keeping the model simple

Finally, an important goal during this thesis was to keep STARPU as accessible as possible. A big advantage of the STF model resides in its simplicity of use. This property is also generally a crucial feature for runtime systems in HPC. Our approach

to extend this programming model must therefore avoid diverging too much from the model that users expect to interface with. If our answers to the goals stated above are too complex to use, then their value is severely limited.

3.2 Designing hierarchical tasks for StarPU

At the beginning of this chapter, we defined hierarchical tasks as regular tasks that can insert a sub-graph to replace them. The sub-graph in question must, of course, perform computations providing the same results as the task it replaces. In the rest of this document, this substitution operation where a hierarchical task inserts a sub-graph instead of executing a kernel is called *processing* the hierarchical task. In our model, a hierarchical task is processed when all its incoming dependencies have been released and the task is considered ready.

In this section, we build up the hierarchical task paradigm by progressively expanding the model, highlighting the issues it may cause, and explaining how to solve them. We start with a presentation of the rules we have imposed on hierarchical tasks in order to provide a solution for our goals that is as general as possible. This leads us to describe the use of hierarchical tasks from the users' point of view. Next, we present how hierarchical tasks adapt the data layout to create sub-graphs with more parallelism. We can then explain how to correctly insert tasks at runtime, breaking the usual rules of the STARPU submission flow. The following part refines this approach by expressing dependencies between sub-graphs at the finest grain possible to maximize parallelism. Finally, we conclude this section with a final rule that allows us to safely use hierarchical tasks to parallelize the submission process.

3.2.1 Aiming for generality

In order to meet the objectives we set in Section 3.1, the hierarchical task model must be as general as possible. For example, in most cases, it would be possible to provide tasks of the appropriate granularity for CPU cores and GPU devices with only two levels of task granularity. However, this would severely limit the expressivity of the model. To ensure that the programming interface does not limit users, we want to be able to nest hierarchical tasks. If a hierarchical task can generate a graph that contains hierarchical tasks, this enables new expressions for the recursive and iterative algorithms mentioned in Section 3.1.2.

The user interface for hierarchical tasks also needs to be as straightforward as possible. If the creation of a sub-graph is too convoluted, then writing hierarchical graphs or adapting existing graphs to be hierarchical becomes tedious. To this end, users express hierarchical graphs at the highest level according to the standard STF model. They only have to annotate some tasks as “hierarchical” to indicate where a sub-graph (themselves expressed through the STF model) can be inserted. The annotations in question are explained in more details in the following subsection.

```

1 starpu_task_insert(&new_cl,          /* Task codelet */
2                      STARPU_R,   handleA, /* Data in Read-only mode */
3                      STARPU_W,   handleB, /* Data in Write-only mode */
4                      STARPU_RW,  handleC, /* Data in Read-Write mode */
5                      STARPU_TURN, is_hierarchical,
6                      STARPU_TURN_ARG, &is_h_args,
7                      STARPU_GEN_DAG, generate_subdag,
8                      STARPU_GEN_DAG_ARG, &gen_dag_args,
9                      0);

```

} New parameters

Figure 3.1: Submitting a hierarchical task.

3.2.2 User interface

To indicate to STARPU that a task is hierarchical and to set its properties, new parameters have been added to the function dedicated to task insertion. Figure 3.1 reuses the example of Figure 2.3 with the annotations required to turn the task hierarchical.

The most important parameter, prefixed by `STARPU_GEN_DAG`, is the function `generate_subdag()` in charge of the submission of the sub-graph. This function is user-defined and describes a DAG using the STF model. Writing that function can be more or less difficult. When trying to factorize parts of an existing graph into a hierarchical task, then the code of the function can be taken directly from the application. When writing a new application or extending the tasks of an existing one, users have to either reuse an already defined graph generating function or write it themselves. Users must make sure that the sub-graph generated will perform equivalent computations to the task’s implementations, i.e. that the task output will remain the same whether it is executed “normally” or processed as “hierarchical”.

The parameter prefixed by `STARPU_TURN` is a function (`is_hierarchical()` in Figure 3.1) that will be called when the task becomes ready. This function returns a boolean. If the return value is *false*, then the task is executed like any regular task. If the return value is *true*, then the hierarchical task is processed and the worker calls `generate_subdag()` instead of a codelet implementation.

The other two parameters are optional in the definition of a hierarchical task. They are used to provide arguments to the previously mentioned functions. These arguments are passed to `generate_subdag()` as a generic type pointer with `STARPU_GEN_DAG_ARG`. In general, they consists in a structure storing a set of data handles as well as static task parameters, which will be used to insert the sub-tasks. Similarly, `STARPU_TURN_ARG` enables programmers to give an argument to the function passed with `STARPU_TURN`.

Instead of positioning these parameters by hand in each task submission of an application, the `STARPU_GEN_DAG` and `STARPU_TURN` functions can be factorized into the task’s codelet. Figure 3.2 shows an example of codelet defining hierarchical tasks.

```

1 struct starpu_codelet hierarchical_cl = {
2     /* Task implementations */
3     .cpu_funcs    = { cpu_impl1, cpu_impl2 },
4     .cuda_funcs   = { cuda_impl },
5     .opencl_funcs = { opencl_impl },
6     /* Hierarchical task parameters */
7     .turn_func    = is_hierarchical,
8     .gen_dag_func = generate_subdag,
9     /* Data information */
10    .nbuffers     = 3,
11    .modes        = { STARPU_R, STARPU_W, STARPU_RW },
12    /* Debug information */
13    .name         = "hierarchical"
14 };

```

Figure 3.2: Encapsulating some of the new parameters into the task codelet.

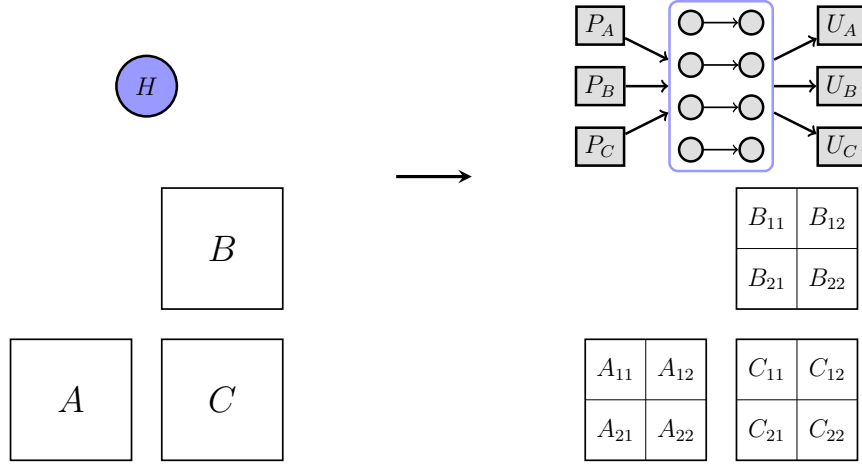
It is also worth noting that the parameters used when inserting a specific task will override the functions defined in its codelet. This means that there is no loss of generality in using codelets to define hierarchical tasks and that users still have full control over its application. They can ensure that a task with a “hierarchical” codelet is processed by submitting it with a `STARPU_TURN` function that always returns *false*. They can also use the `STARPU_GEN_DAG` parameter to replace the sub-graph generating function of a particular task with a different one without having to write a new codelet.

3.2.3 Adapting the data layout around hierarchical tasks

Due to the way in which STARPU infers dependencies from data access modes using sequential consistency, the role of data is of paramount importance in the creation of our task graphs. With the addition of hierarchical tasks, we must also consider how data coherence will be achieved between the main DAG and the sub-DAGs. In fact, to be able to express parallelism within a hierarchical task, its input data must be partitioned to a smaller granularity. The introduction of hierarchical tasks requires dynamically adapting the granularity of the data around them.

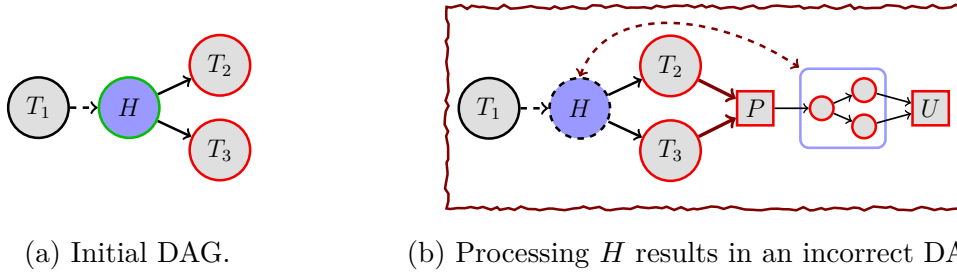
Figure 3.3 illustrates this notion with a single hierarchical task H performing a matrix multiplication. In order for H to be able to submit its sub-graph when it is processed, each original matrix has to be partitioned into tiles in a 2×2 layout. Then, if a subsequent task needs to work on the entire matrices again, an unpartitioning operation is required to protect data coherence by disabling the sub-handles and re-enabling their parents. This serves to demonstrate that a proper implementation of hierarchical tasks should be associated with a data manager. To avoid costly barriers, we want to maintain the asynchronous approach that relies on partitioning tasks¹. Therefore, the data manager must be able to dynamically partition and

¹In Figure 3.3b, the dependencies between the sub-graph and the partitioning tasks are simplified to keep the figure legible. In reality, the partitioning tasks all are connected directly to the sub-tasks that access a sub-handle.



(a) Initial hierarchical task and data layout. (b) Sub-graph and partitioned data layout.

Figure 3.3: Example of a hierarchical task corresponding to a matrix product operation $C = C + A \times B$ and its corresponding DAG.



(a) Initial DAG.

(b) Processing H results in an incorrect DAG.

Figure 3.4: Processing a hierarchical task by relying strictly on the STF model.

unpartition an application's data at runtime. This deviates from the current data manager of STARPU, which inserts partitioning tasks at submission time, as explained in Section 2.4.

However, these partitioning tasks were previously inserted following the rules of the STF model, where the creation of dependencies relies on the order of submission (see Section 2.2). Since a hierarchical task H is processed during runtime, its sub-tasks, and the partitioning tasks around them, will be inserted *after* the potential successors of H . This scenario is shown in Figure 3.4, where H is submitted between regular tasks T_i . When H is processed, T_2 and T_3 have already been submitted, and STARPU will incorrectly consider that the newly inserted sub-tasks should be placed *in front* of them, as shown in Figure 3.4b.

3.2.4 Correctly inserting tasks at runtime

To address the issue we just introduced, we remove some of the automatically inferred data dependencies on both ends of a hierarchical task. Then, we add new dependencies

in the correct place to ensure that the sub-tasks are properly positioned into the DAG. The objective is to guarantee that hierarchical DAGs remain correct and consistent with sequentially created graphs.

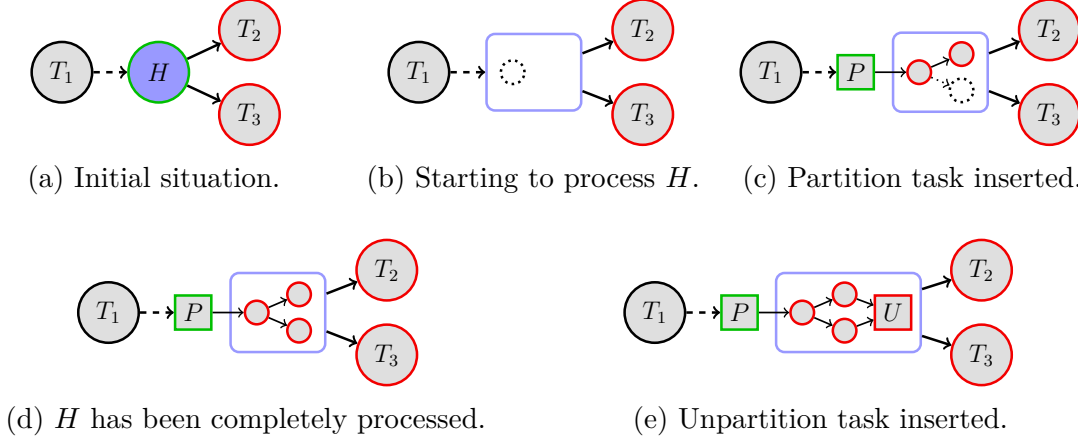


Figure 3.5: Altering the STF model to process a hierarchical task without breaking the DAG. Dotted arrows represent released dependencies, tasks with a **black** border are already executed, **green** borders indicate ready tasks, and **red** borders indicate not-ready tasks.

Partitioning before a hierarchical task

First, let us examine what should happen when a hierarchical task becomes ready and starts inserting sub-tasks. In the scenario from Figure 3.4, the roots of the issue are the partition tasks P that are placed in front of T_2 and T_3 . As mentioned before, we have to disable the data dependencies that the STF model inferred from the “coarse” handles used by the tasks T_i .

Figure 3.5 shows the evolution of the example of Figure 3.4a once the classic STF model has been altered for hierarchical tasks. When H becomes ready (Figure 3.5a) and starts being processed, the data layout matches the level of T_1 and must be fitted for the sub-tasks of H . Since our model supports the coexistence of multiple partition plan of the same data, we must wait for the beginning of the first sub-task’s submission (Figure 3.5b). Only then can we know which partition plan must be activated. With this information, the appropriate partition task is inserted. This is done *during* the submission of the sub-task, which is only inserted *after* the partition task. Figure 3.5c shows the subsequent state of the task graph, with P correctly inserted before any of the sub-tasks. The hierarchical task then continues to submit the sub-graph, which will wait for the partition task to update the data layout.

In this example, H only submitted on the sub-handles of a single data handle. For more complex cases involving more data handles (e.g. our first matrix multiplication example), the process described above is repeated to partition each parent handle into

the appropriate sub-handles. A partition task for a data D is only inserted before the submission of the first sub-task that accesses a descendant of D .

Processing the hierarchical task

Once the partition tasks are in place, we can start inserting the sub-tasks of H . As shown in Figure 3.5d, because all the sub-tasks work on sub-handles and not on the parent handles, the STF model will not connect them to the coarser tasks T_i . Moreover, accessing these sub-handles means that they are the natural successors of the partition tasks, which is the behavior we want. After all the sub-tasks have been inserted, we must restore the larger data handles for the following tasks, such as T_2 and T_3 , with unpartition tasks.

Unpartitioning after a hierarchical task

Similarly to the partition tasks, the rules of the STF model dictate that a task unpartitioning a data D must depend on the already submitted tasks accessing D or its children. We can address this issue by removing the dependencies inferred from the parent handles, just like we did for the partitioning tasks. This only leaves the dependencies inferred from the sub-handles, which will automatically connect the unpartition task to the now fully submitted sub-graph.

Since they are the last tasks to be inserted, the unpartition tasks will have no outgoing dependencies. This is an issue, as it means that T_2 and T_3 risk to be considered ready, when they should wait for the execution of the sub-graph and the unpartition tasks. A direct solution to this problem is to prevent the task H from releasing its dependencies until the completion of the entire sub-graph, including the unpartition tasks associated to it. This is represented in Figure 3.5e.

Just like in the partition case, our example only considers the unpartitioning of a single handle. If more handles were partitioned when H was processed, additional unpartition tasks would be inserted.

3.2.5 Fine grain dependencies

Preventing the release of the dependencies of a hierarchical task until its sub-graph is executed makes for a simple way to ensure the correctness of the DAG. However, when considering a hierarchical task graph at its coarsest level, having dependencies between hierarchical tasks can appear particularly strict. By creating a barrier that waits for the completion of the sub-graph before inserting the next one, these dependencies limit the expression of parallelism in the application. In contrast, removing this barrier improves the pipelining of the processing of two consecutive hierarchical tasks. For example, the sub-tasks of the second hierarchical tasks can start as soon as their dependencies in the first sub-graph are satisfied, instead of waiting for the execution of the entire sub-graph to be completed.

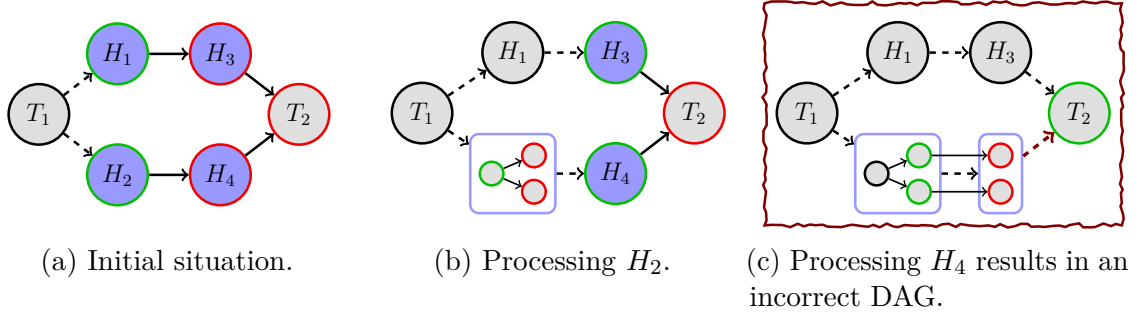


Figure 3.6: Releasing the dependencies of hierarchical tasks once they are processed to connect sub-graphs at the finest level. Partitioning tasks have been simplified out.

To achieve this, we want to release the dependencies of a hierarchical task immediately after it has been processed. Figure 3.6 illustrates this mechanism. In Figure 3.6b, the dependency of H_2 is not held until the sub-graph is fully executed but released immediately after it is submitted. Thus H_4 can start its own submission process. Because STARPU automatically infers dependencies from the data, H_4 's sub-graph will naturally plug itself on top of H_2 's sub-graph without any additional effort on the part of programmers. This ensures that task dependencies are always positioned at the finest grain (i.e. the deepest level of the hierarchy).

While the expression of fine grain dependencies emerges naturally from this approach, it creates a glaring problem. As shown in Figure 3.6c, releasing H_4 's dependency turns T_2 into a ready task too soon. This can be disastrous, as it breaks the graph of the application and can easily lead to incorrect results.

Another issue with our approach is that releasing the outgoing dependencies of a hierarchical task once it has been processed is not enough. Due to the unpartition tasks that we insert after a sub-graph, the following sub-graph must first re-partition its data. This will also result in pointless dependencies between the unpartition and the partition tasks.

The solution to both of these issues is to avoid systematically inserting unpartition tasks after the sub-graph of a hierarchical task. Instead, we can delay the submission of these unpartition tasks until we can determine that it is necessary, i.e. until a task using the parent data is about to start. This requires inserting the unpartitioning task *between* the last sub-graph submitted and the tasks accessing the parent handle. Once again, such a task cannot be correctly placed by strictly following the STF model. However, we can implement a mechanism that replaces the dependencies released prematurely by the processing of a hierarchical task.

Figure 3.7 revisits the last step of Figure 3.5 to illustrate this mechanism. With the changes we just made, when H inserts its sub-graph and terminates, its successors are now considered ready (see Figure 3.7a). However, before starting the execution of these tasks, their handles are checked and an unpartition task U is inserted. Dependencies

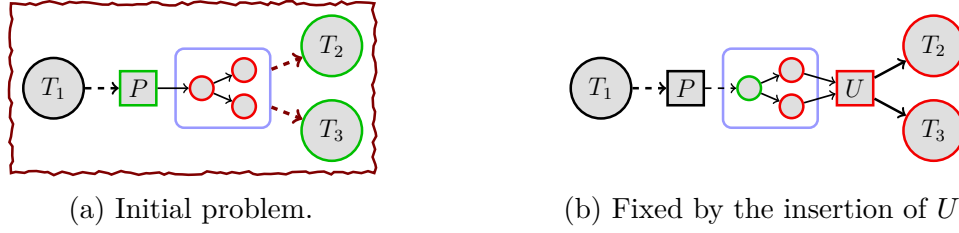


Figure 3.7: Using unpartition tasks to enforce the correction of the graph.

from the last sub-tasks to be submitted for each sub-handle are gathered by U and more dependencies are created between U and H 's successors. The execution of these successors is then postponed and they are returned to a non-ready state.

As shown in Figure 3.7b, if H carried multiple dependencies, the unpartition task U will only be submitted once and all the successors of H will depend on it.

3.2.6 Parallel submission

Now that our model is capable of inserting sub-graphs correctly while enforcing data consistency, we can examine its potential features. An important notion that stems from the nature of hierarchical tasks is the coexistence of multiple submission threads. Because hierarchical tasks are treated as regular tasks until their execution is about to start, STARPU's scheduler assigns them to a worker when they are ready. If it turns out that a hierarchical task is to be processed, the sub-graph generating function will be called by the worker in charge of the task. Therefore, processing independent hierarchical tasks at the same time in different workers parallelize the submission process.

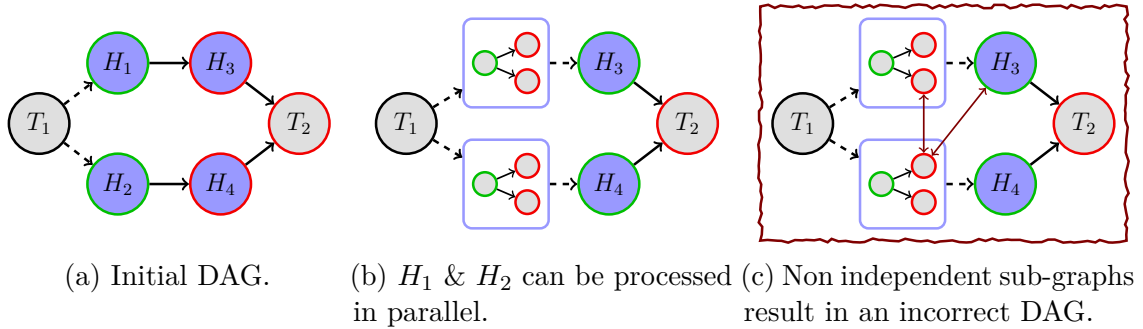


Figure 3.8: Example of hierarchical tasks submitting sub-graphs in parallel.

Figure 3.8 presents a simple case where two hierarchical tasks (H_1 and H_2) could be processed in parallel. For this approach to be correct, programmers must ensure that the coarse dependencies ($T_1 \rightarrow H_1$ and $T_1 \rightarrow H_2$) are representative of the dependencies of the sub-graphs within the larger graph. In other words, the sub-tasks must respect the *incoming* and *outgoing* dependencies of their parent hierarchical task.

Otherwise, our hypothesis that independent hierarchical tasks submit independent sub-graphs would be false, and parallel submission would produce race conditions and potentially incorrect graphs. An example of this kind of errors is shown in Figure 3.8c. In the sub-graph inserted by H_2 , a sub-task uses a data handle that is not a child of the handles on which H_2 was submitted. In Figure 3.8c, it results in new dangerous dependencies. Depending on when the hierarchical tasks are processed, the direction of the **dark red** dependencies will change. In turn, this makes the numerical result of this DAG unpredictable.

To be able to solve this problem, we placed a natural restriction around the use of hierarchical tasks, more specifically regarding their relation to data handles. The rule in question is that *hierarchical tasks should only submit tasks working on sub-handles of their own handles*.

This rule removes the risk of a sub-task using a handle tied to a different data than the handles used in the parent hierarchical tasks. It makes the dependencies of the hierarchical tasks sufficient to fully describe the location of the sub-graph in the rest of the DAG. As a result, two independent hierarchical tasks will always produce two independent sub-graphs that can be safely appended to the graph in parallel.

An added benefit of this rule is that it consolidates our model in the context of nested hierarchical tasks and partition trees with multiple levels. Under this rule, the data hierarchy is tied to the task hierarchy: each level of the task hierarchy will operate on its own level of data handles. The subsets of handles that each level of tasks will access are exclusive to this specific level. This gives us two guarantees that contribute to the correct insertion of sub-tasks (as described in Section 3.2.4):

1. Level n tasks inserted by a level $n - 1$ hierarchical task will never depend on the tasks of levels $< n$ that are already present in the graph.
2. Level n tasks are naturally placed in front of the other level n tasks from previously inserted sub-graphs, according to the STF model.

```
1  [...] /* Registration of handleA */
2  starpu_data_handle_t handleA_bis;
3
4  /* Example of filter creating a single child */
5  struct starpu_data_filter f = {
6      .filter_func = starpu_vector_filter_block,
7      .nchildren = 1
8  };
9
10 starpu_data_partition_plan(handleA, &f, &handleA_bis);
```

Figure 3.9: Example of an “identity” partition plan.

While this restriction may seem to lessen the generality of the approach by forcing the programmer to partition their data in order to use hierarchical tasks, this is not the case. In fact, it is possible to partition a handle “into itself”. Programmers

can create a new partitioning level for a data with a single sub-handle pointing to the entire data buffer. Figure 3.9 showcases how to create such a partition. In this example, `handleA_bis`, created with a `nchildren=1` filter, points to the exact same data buffer as `handleA`. However, it is only accessible after a partitioning task, and an unpartitioning task is required to access `handleA` after that point. Using this mechanism, the programmer can create sub-tasks that work on the same data as their parent while still benefiting from the model’s security.

3.2.7 Data management for the programmer

Now that we have completed our presentation of how the model expects to be interacted with, we can look at the situation from the programmer’s perspective. Creating a graph using hierarchical tasks requires dynamic adaptation of the data layout depending on which hierarchical tasks are being processed and when. For the programmer, manually managing these issues would add a whole new level of complexity to the inconveniences presented in Section 2.4.1. In particular, it would conflict with our goal of making the user’s job as simple as possible. A more reasonable approach is to extend the automatic data manager presented in Section 2.4.2 and make data management completely transparent to the user.

Specifically, we want to leave the data registration step (including partition planning) strictly unchanged, whether the programmer uses hierarchical tasks or not. They can provide new filters and partition plans to use with hierarchical tasks, but they can do that using the interface already in place in STARPU.

The only constraint placed on the user is to ensure that each level of hierarchical task only submits tasks that work on data handles of the correct level. Reciprocally, data handles of a certain level should only be accessed by tasks of the correct level. The initial graph should be submitted at the coarsest data level, with hierarchical tasks submitting sub-graphs on the next data level, and so on. The partition trees can be completely traversed by nesting hierarchical tasks.

3.2.8 Case of read-only access mode

In the previous examples, we mostly presented cases where data was accessed in *write* or *read-write* mode. The *read-only* access mode, which is especially important for parallelism in the application, requires special partitioning and unpartitioning tasks (as mentioned in Section 2.4). Otherwise, the regular partitioning tasks would create harsh synchronizations that limit the parallelism possibilities of read accesses.

These read-only partitioning and unpartitioning tasks can, for the most part, follow the same principles as those described above. Still, the fact that read-only partitions enable concurrent access to a data handle and all of its possible sub-handles can create convoluted problems. These problems and their solutions will be discussed in detail in the next chapter, which deals with the implementation of this model.

3.3 Conclusion

A hierarchical task is a task that can submit a sub-DAG at runtime instead of performing actual computations. The introduction of such tasks in STARPU aims to address certain limitations of task-based runtime systems. A first goal is to improve upon the submission process of the STF model. By processing independent hierarchical tasks, we parallelize the submission process. This addresses a potential bottleneck in embarrassingly parallel applications, where the execution front would catch up with the sequential submission front. By encapsulating part of a task graph in hierarchical task, we also delay task submission. This limits the amount of non-ready tasks in the system, reducing runtime overhead. It is also useful in applications where parts of the task graph cannot be known in advance. Another goal behind the implementation of hierarchical tasks is to increase the expressivity of our task graphs. For example, making a large grain task into a hierarchical task would enable it to turn into a sub-DAG of tasks with smaller grain at runtime. This would help us target the appropriate CPU or GPU resources based on their availability during the execution.

Our model aims to be as natural and transparent as possible for the programmer. To this end, making a task hierarchical consists of providing a pair of hints: a function describing a potential sub-graph and a function used at runtime to decide if the task should be processed or not. This can be done individually for certain tasks in the application, or generalized to an entire set of tasks through the codelet structure.

In order to express a sub-graph out of a task, it appears necessary to partition its data. In STARPU, data partitioning is completely transparent to the programmer. They can submit partition plans describing the different data layouts they will use in their application to create new handles associated to the different layouts. They can then use these handles freely in the submission of their tasks and STARPU will automatically insert partitioning tasks to protect data consistency. This user interface is maintained in the hierarchical task model with the added constraint to always use the sub-handles of a task's data in its potential sub-graph. The model can therefore support fine-grain dependencies: there are no barriers between two consecutive hierarchical tasks. When a hierarchical task finishes its sub-graph insertion, its outgoing dependencies are immediately released. The partition and unpartition tasks inserted at runtime for data consistency also ensure that the hierarchical graph remains consistent with a STF graph.

To be able to correctly position the partitioning tasks at runtime and enforce the correction of a DAG using hierarchical tasks, we must extend the data manager. We deal with this in the next chapter, covering the implementation of the hierarchical task model in STARPU.

Chapter 4

Hierarchical Tasks Implementation

Contents

4.1	Foundations of the implementation	60
4.1.1	Updating the data structures	60
4.1.2	Processing hierarchical tasks	61
4.1.3	Processing hierarchical tasks as soon as possible	62
4.2	Extending the data manager	62
4.2.1	Controlling data dependencies	64
4.2.2	Partition task insertion	65
4.2.3	Unpartition task insertion	66
4.2.4	The read-only access mode	69
4.3	Conclusion	71

IN THE PREVIOUS CHAPTER, we explained that the introduction of hierarchical tasks in STARPU requires changes in the way its current model works. Most notably, we have to update the behavior of tasks to make them capable of inserting sub-graphs as described in Section 3.2.2. Just as importantly, the support of hierarchical tasks involves extending the data manager to enable dynamic data partitioning at runtime.

This chapter is meant to present the modifications that were made in STARPU to implement hierarchical tasks. A first implementation effort focuses on the groundwork and processes hierarchical tasks at submission time. While this approach is entirely static, it completely implements the hierarchical task interface and can be seen as a powerful way to express recursive algorithms with the STF model. To improve upon the STF model and deliver on the promised dynamism, we then have to update the data manager to match the behavior described in Sections 3.2.4 and 3.2.5.

4.1 Foundations of the implementation

In this section, we explain the basics involved in the implementation of the interface we detailed in Section 3.2.2. This requires modifications that concern data structures, the submission process, and, most prominently, the execution process of STARPU's tasks. To simplify this initial presentation, we elude issues of graph correctness and data consistency by processing hierarchical tasks as soon as they are submitted.

4.1.1 Updating the data structures

The majority of the updates needed in the data structures of STARPU's codelets and tasks are fairly straightforward. The codelet simply presents two new members where users can define the functions characterizing a hierarchical task (see Figure 3.2). For the task structure, new members store the aforementioned functions (`STARPU_TURN` and `STARPU_GEN_DAG`) and their arguments. Information on the status of a task are also added, like whether or not it is hierarchical (`hierarchical` flag), and whether or not it will be processed (`to_process` flag). On top of that, some additional elements can be stored in the task structure for debugging purposes. This includes, for example, information linking a sub-task to its hierarchical parent.

When a hierarchical task is submitted, the `TURN` and `GEN_DAG` functions and their arguments are stored in the task structure. The task is then flagged as hierarchical. For now, we can consider that the rest of the submission process is identical to that of a regular task. Differences between the two types of tasks are concentrated around the point where the execution of a kernel is replaced by the insertion of a sub-graph.

Algorithm 4 A task's lifetime

- 1: *task* is submitted
 - 2: **When** *task* is ready
 - 3: Push *task* to scheduler
 - 4: Assign *task* to worker
 - 5: Execute *task* codelet
 - 6: Release *task* outgoing dependencies
-

4.1.2 Processing hierarchical tasks

This point is reached when a task becomes ready, i.e. when all of its incoming dependencies are fulfilled. When this happens, the task is inspected to check if it is a hierarchical task that should be processed. Algorithm 4 is a pseudo-code representation of the existence of a task in STARPU. The moment when we want to process a hierarchical task is after line 2.

```

1 void process_hierarchical_task(struct starpu_task *task)
2 {
3     struct starpu_codelet *codelet = task->cl;
4
5     /* Check if a hierarchical task should be processed */
6     if (task->hierarchical)
7         if (task->turn_func)
8             task->to_process = task->turn_func(task->turn_arg);
9         else
10            task->to_process = codelet->turn_func(task->turn_arg);
11
12    /* Process the hierarchical task */
13    if (task->to_process)
14        if (task->gen_dag_func)
15            task->gen_dag_func(task->gen_dag_arg);
16        else
17            codelet->gen_dag_func(task->gen_dag_arg);
18 }

```

Figure 4.1: Processing a hierarchical task (or not).

The function presented in Figure 4.1 is a streamlined version of that step. First, if the task is flagged as hierarchical, `turn_func` is called (lines 6 to 10). The return value is stored as another flag in the task and used to decide whether or not the hierarchical task should be processed (lines 13 to 17), as specified in the model.

The processing step itself is particularly straightforward, as it simply consists of calling `gen_dag_func()` with its argument. This user-defined function describes a sub-graph using the standard STF model, and can therefore be called directly by a worker to insert new tasks in the DAG.

If the task is not hierarchical, or if `turn_task` returned *false*, then its execution continues unchanged and one of its codelet implementation is launched. On the contrary, when a hierarchical task is processed, the execution of the task is interrupted

soon after, as the sub-graph is meant to replace the task that inserted it.

4.1.3 Processing hierarchical tasks as soon as possible

In Section 3.2.3, we explained how the use of hierarchical tasks often mandates adjustments in the data layout by partitioning the handles to exhibit new parallelism in the sub-graphs. In turn, this forces the use of unpartitioning operations to restore the larger handles. The next sections detailed the various modifications to implement in STARPU's data manager to achieve our goals regarding dynamic task graphs and parallel submission. However, a first implementation can focus on sub-graph insertion and rely strictly on the existing data manager. To this end, we let the submission thread process hierarchical tasks as soon as they are inserted.

This is achieved by calling the `process_hierarchical_task()` function of Figure 4.1 *as soon as possible*, before inserting another task. In Algorithm 4, this means calling the function during the very first step (line 1). Figure 4.3 is an example of the resulting submission process for the code given in Figure 4.2. Since we restore the sequential order of task submission, the existing data manager can insert the partitioning tasks correctly at submission time, without any modifications.

As mentioned above, this approach is entirely static and will not be enough to achieve some of our goals. It can instead be seen as a way of enabling recursive task submission. This gives us a first functioning implementation of the hierarchical task model with several of its features. Most notably it can be used to create DAGs mixing the granularity of their tasks, as shown in Figure 4.3. It also fully implements the interface we described in Section 3.2.2. This will later allow us to compare the fully implemented model to an equivalent graph relying on the STF model.

4.2 Extending the data manager

To overcome these limitations and finally implement the entire hierarchical task model, the data manager must be extended. Under the hierarchical task model, the whole task graph cannot be known until all hierarchical tasks have been processed. A consequence of this is that the layout of an application's data can vary at any given time of the execution, depending on how hierarchical tasks are processed. Therefore, the fundamental difference with the data manager presented in Section 2.4.2 is that the insertion of partitioning tasks must be handled at execution time rather than submission time.

In this section, we go over the changes implemented in the data manager. First, we focus on tasks modifying the data pieces they access (read-write or write mode). With these access modes, a piece of data can only be accessed by one task at a time. This ensures that we can cycle through the *partitioning states* of data handles safely. The main challenge in these first sections is to correctly position our partitioning tasks.

```

1 #define NPARTS 2
2
3 int is_hierarchical(void *arg)
4     return 1;
5
6 void generate_subdag(void *arg) {
7     starpu_data_handle_t *subA = (starpu_data_handle_t*)arg;
8     for (i = 0; i < NPARTS; i++)
9         starpu_task_insert(&read_cl, STARPU_R, subA[i], 0);
10 }
11
12 struct starpu_codelet task_cl = {
13     .cpu_funcs = { cpu_impl1 }
14 };
15
16 struct starpu_codelet hierarchical_cl = {
17     .cpu_funcs = { cpu_impl2 },
18     .turn_func = is_hierarchical,
19     .gen_dag_func = generate_subdag
20 };
21
22 int main() {
23     [...] /* Registration of handleA */
24
25     /* Partitioning in NPARTS slices */
26     struct starpu_data_filter f = {
27         .filter_func = starpu_vector_filter_block,
28         .nchildren = NPARTS
29     };
30     starpu_data_handle_t subhandlesA[NPARTS];
31     starpu_data_partition_plan(handleA, &f, subhandlesA);
32
33     starpu_task_insert(&task_cl, STARPU_RW, handleA, 0);    // T1
34
35     starpu_task_insert(&hierarchical_cl, STARPU_R, handleA, // H1
36                       STARPU_GEN_DAG_ARG, subhandlesA, 0);
37
38     starpu_task_insert(&hierarchical_cl, STARPU_R, handleA, // H2
39                       STARPU_GEN_DAG_ARG, subhandlesA, 0);
40
41     starpu_task_insert(&task_cl, STARPU_RW, handleA, 0);    // T2
42
43     [...] /* Handle unregistration */
44 }

```

Figure 4.2: Example of code using hierarchical tasks.

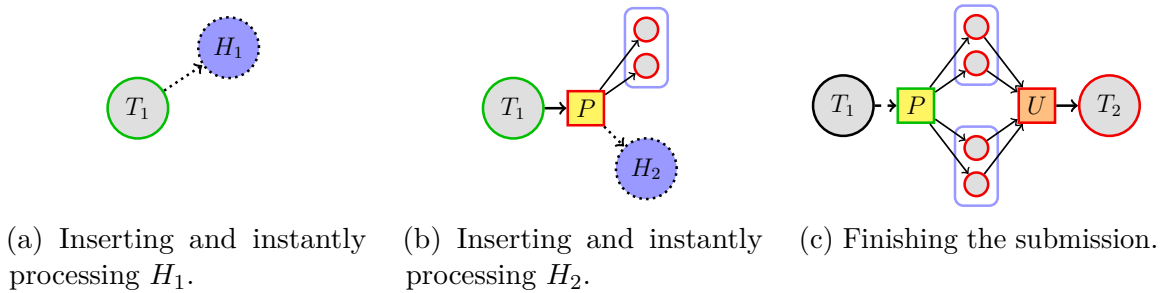


Figure 4.3: Submission of the task graph of Figure 4.2. Hierarchical tasks are processed in the submission thread.

We achieve this by removing some of STARPU's automatically inferred dependencies, and by explicitly declaring new ones. We can then tackle the case of read-only accesses. It requires dedicated partitioning tasks and safeguards to account for the additional parallelism it enables. Under the read-only access mode, multiple threads are susceptible to simultaneously submit partitioning operations on the same data. Thus, we have to ensure that the data manager is not subject to race conditions.

4.2.1 Controlling data dependencies

By default, the STF model will insert the new tasks it receives as successors of the tasks already present to respect the sequential consistency, as explained in Section 3.2.3. It is however possible to specify the behavior of a task regarding the sequential consistency of each of its handles.

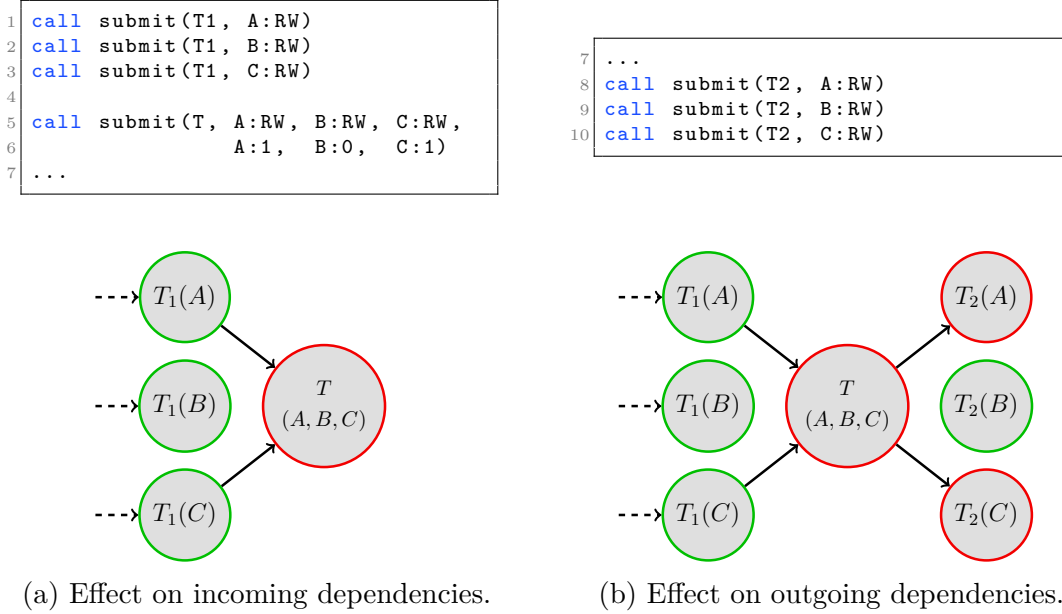


Figure 4.4: Manually defining sequential consistency for a task's data.

An example is given in Figure 4.4. In every task, each data is accessed in read-write mode. However, when $T(A, B, C)$ is submitted, we specify that the sequential consistency is ignored for the data B . Thus, $T(A, B, C)$ is inserted without being dependent from $T_1(B)$ (see Figure 4.4a). Moreover, the tasks submitted after $T(A, B, C)$ will only depend on it if they access A or C . As shown in Figure 4.4b, there is no outgoing dependency toward $T_2(B)$.

This mechanism can be used to remove undesirable dependencies from partitioning tasks. A task partitioning or unpartitioning a data handle A will be submitted with write access to the parent handle A and each of its sub-handles. By disabling the sequential consistency on the parent handle only, a partition task will not depend on any of the tasks accessing the parent handle, but it will have outgoing dependencies

toward any subsequent task accessing the sub-handles. With the same approach, an unpartition task will only depend on the tasks accessing the sub-handles.

4.2.2 Partition task insertion

When considering the insertion of partition tasks, we can heavily rely on the existing data manager. The rules of the model guarantee that a task accessing a sub-handle is submitted by a hierarchical task. When a hierarchical task is processed, the submission of its sub-tasks is identical to the submission of any other tasks. In particular, the AUTOSUBMIT algorithm presented in Section 2.4.2 is used to check the state of the handles accessed. If it finds that the data is not in the proper state, it submits a partition task. Thus, before the first sub-task is submitted, AUTOSUBMIT is called on each of the sub-handles it accesses, and inserts the appropriate partition tasks to activate them.

The only adjustment required was presented in the previous section. The incoming dependencies of a partition tasks are removed by ignoring the sequential consistency of the main handle. This ensures that it is immediately ready. This can be done safely, because a hierarchical task is only processed when all its dependencies are satisfied. Once the partition task is inserted, the states of the parent and children handles are updated to reflect the modification of the data layout.

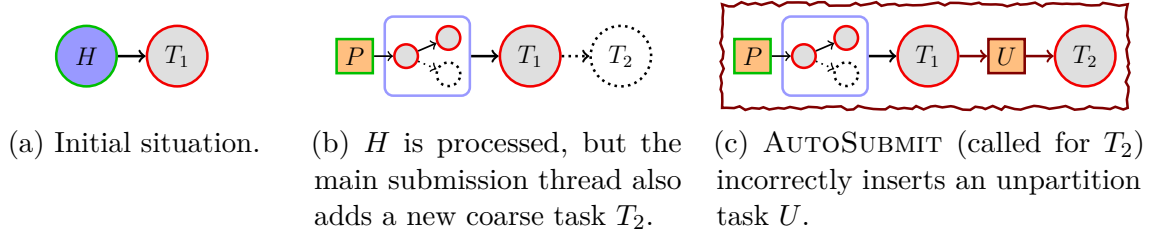


Figure 4.5: Incorrect behavior of AUTOSUBMIT when multiple submission threads coexist.

The AUTOSUBMIT algorithm was conceived to be used within a unique submission thread. In that context, the algorithm updates the partitioning states of every handle sequentially. When a handle is partitioned, all the tasks using the sub-handles are necessarily submitted before the insertion of an unpartition task. This property is not true when data is partitioned for hierarchical tasks at execution time.

If we leave the algorithm as it is, it risks to incorrectly insert unpartitioning tasks. This is illustrated in Figure 4.5. As expected, a partition task is inserted before the sub-graph of H (Figure 4.5b), but the handle state is updated after the insertion of P , indicating that it is now RW-partitioned. Figure 4.5c shows what happens if the main submission thread submits a new task T_2 at this point. Since the handle is now considered RW-partitioned, the call to AUTOSUBMIT made during the insertion of T_2 will indicate that an unpartition task must be added. This can lead into a cascade of

errors, as the handle states will be updated again. Since H is still being processed, the next sub-task will believe that the sub-handles have been unpartitioned and insert a redundant partition task.

Algorithm 5 Updated version of Algorithm 1

```

1: function AUTOSUBMIT(ancestor, target, write, isRuntime)
2:   if ancestor is Inactive or
      (write and parent is RO active) then
3:     AUTOSUBMIT(ancestor parent, ancestor, write, isRuntime)
4:   if isRuntime then
5:     CHECKUNPARTITION(ancestor, target, write)
6:   if target is NULL or isRuntime then
7:     return
8:   CHECKPARTITION(ancestor, target, write)

```

To address this issue, we must prevent AUTOSUBMIT from inserting unpartitioning tasks at submission time. A revised version is presented in Algorithm 5. We add a boolean argument *isRuntime*, which indicate whether AUTOSUBMIT was called by a thread submitting tasks or not. If *isRuntime* is *false*, we only check if *ancestor* must be partitioned. Otherwise, if *isRuntime* is *true*, we check if *ancestor* must be unpartitioned and return from AUTOSUBMIT.

4.2.3 Unpartition task insertion

After the modifications presented in Algorithm 5, the insertion of unpartition tasks has to be done at execution time. It also has to exhibit the behavior detailed in Section 3.2.5, which we summarize next. When a hierarchical task is fully processed, it releases its outgoing dependencies. This enables a potential hierarchical successor to be processed without creating a barrier between the sub-graphs. If the successor is not a hierarchical task (or if the hierarchical task is not processed), the data handle involved will have to be unpartitioned and the dependency replaced. By properly inserting our unpartitioning tasks, we aim to kill two birds with one stone.

Inserting unpartitioning tasks correctly is more complex than inserting partitioning tasks. Here are the points that we address in this section:

1. When should a task check if its handles must be unpartitioned?
2. How was the handle structure extended to enable this operation?
3. How can the execution of a ready task be postponed in case an unpartition is needed first?
4. How are the dependencies of the unpartition task positioned?

First, when a task is ready, we must determine the appropriate moment to check if

```

1 void process_hierarchical_task(struct starpu_task *task)
2 {
3     struct starpu_codelet *codelet = task->cl;
4
5     /* Check if a hierarchical task should be processed */
6     if (task->hierarchical)
7         if (task->turn_func)
8             task->to_process = task->turn_func(task->turn_arg);
9         else
10            task->to_process = codelet->turn_func(task->turn_arg);
11
12     /* Check if the data must be unpartitioned */
13     if (!task->hierarchical || !task->to_process)
14         unpartition_if_needed(task);
15
16     /* Process the hierarchical task */
17     if (task->to_process)
18         if (task->gen_dag_func)
19             task->gen_dag_func(task->gen_dag_arg);
20         else
21             codelet->gen_dag_func(task->gen_dag_arg);
22 }

```

Figure 4.6: Updating `process_hierarchical_task()` to check if unpartitioning operations are needed at execution time.

an unpartitioning task is needed. If we insert such a task between two hierarchical tasks of the same level, the data will have to be partitioned again if the second hierarchical task is processed. This would recreate the bottleneck between the first and the second sub-graphs that we want to avoid. Therefore, the decision to unpartition a data should be made once we know whether the task will be processed or not. To this end, we update the `process_hierarchical_task()` function, as shown in Figure 4.6. Every task T that does not insert a sub-graph will call `unpartition_if_needed()`. This function checks each handle of `task` and, as its name indicates, unpartition them if necessary.

In order to be able to manage the insertion of unpartition tasks at runtime, we extended the data handle structure. A first additional member, `ctrl_unpartition` stores a pointer to a *control task*. A control task is a task without a codelet: when it becomes ready, its outgoing dependencies are immediately released without having to be run by a worker. The control task of a handle H is used to manage the dependency between the task unpartitioning H and the task accessing H . A second new member is a flag indicating if an unpartition task exists for the parent of the handle, but has not been executed yet. It provides information completing the partitioning state of the handle.

Figure 4.7 shows an example of insertion of unpartitioning tasks. In Figure 4.7a, two hierarchical tasks H_1 and H_2 were processed and released their successor T . This task calls `process_hierarchical_task()` and, since it is not hierarchical, `unpartition_if_needed()`. The function checks every handle of T one by one. First, it creates a

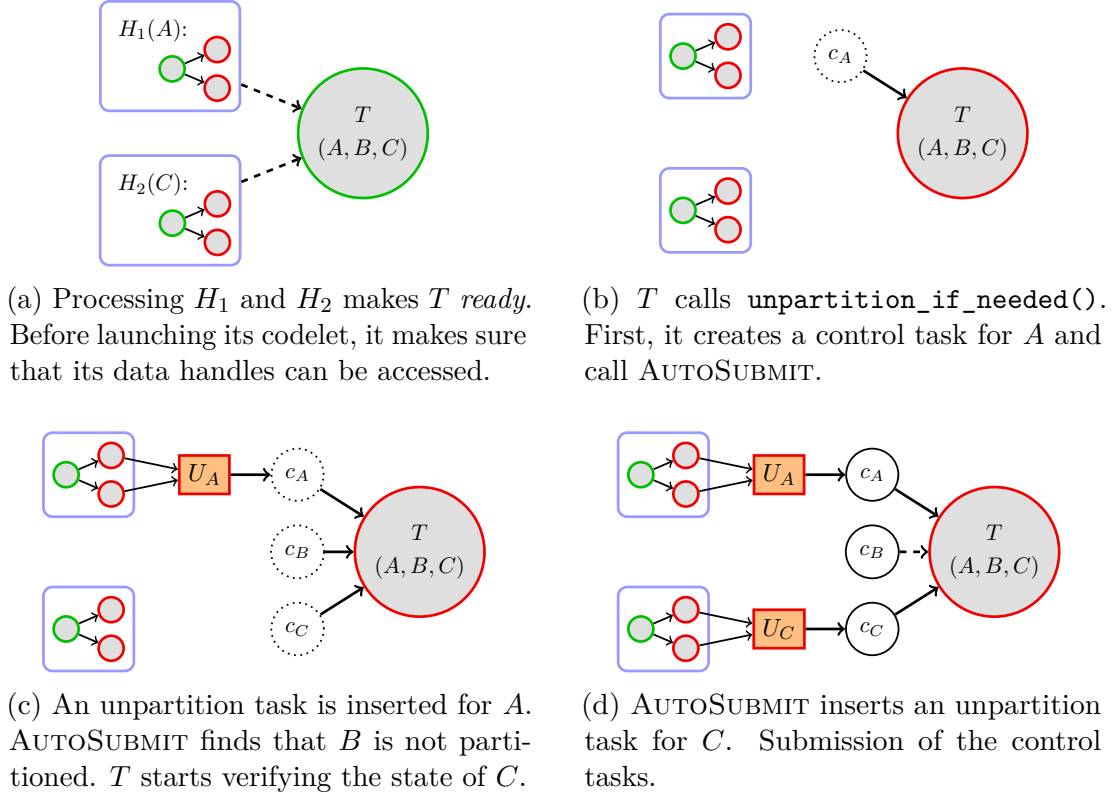


Figure 4.7: Insertion of unpartitioning tasks.

control task c_A for the handle A and declares a $c_A \rightarrow T$ dependency *without submitting* c_A . This indicates to STARPU that T is no longer ready. Furthermore, it gives us control over the “readiness” of T , since the dependency cannot be released until we submit c_A . The handle A stores c_A in `ctrl_unpartition`. Then, `AUTO SUBMIT` is called for A with the argument `isRuntime = True` to decide if an unpartition task is required or not. In Figure 4.7c, the state of A indicates that it is partitioned, and an unpartition task is inserted. The same process is repeated for the handles B and C . In the case of the handle B , `AUTO SUBMIT` finds that it is not necessary to insert an unpartition task.

If an unpartition task is inserted, we tweak its dependencies, just like we did for the insertion of partitioning tasks. The incoming dependencies come from the sub-tasks, the only ones accessing the sub-handles. No dependencies (incoming or outgoing) are inferred from the parent handle, instead, the necessary dependencies are declared explicitly. This concerns the outgoing dependency, which connects the unpartition task to the task that called `AUTO SUBMIT`. It is achieved by manually setting a dependency between the unpartition task and the control task found in the parent handle via `ctrl_unpartition`. Once every handle of the task has been checked, the control tasks are submitted (Figure 4.7d). In the case of A and C , an unpartition was inserted, so their control task is not ready. On the contrary, since B

was not unpartitioned, c_B is considered ready when it is submitted, and the associated dependency is immediately released.

Algorithm 6 Insertion of an unpartition task at execution time

```

1: function UNPARTITIONIFNEEDED(task)
2:   if task was already checked then
3:     return
4:   Mark task as checked
5:   controlTasks  $\leftarrow \emptyset$ 
6:   for H in task handles do
7:     if H has partition plans then
8:       write  $\leftarrow$  H access mode in task
9:       ctrl  $\leftarrow$  New control task
10:      Add ctrl to controlTasks
11:      Save ctrl in H as ctrl_unpartition
12:      Create dependency (ctrl  $\rightarrow$  task)
13:      AUTOSUBMIT(H, NULL, write, True)
14:   for ctrl in controlTasks do
15:     Submit ctrl
16:   return

```

The details of the algorithm implemented by `unpartition_if_needed()` are given in Algorithm 6. Lines 2 to 4 ensure that a task only goes through the algorithm once. Furthermore, since handles without a partition plan cannot be partitioned, they never have to be unpartitioned and can be ignored (line 7).

A callback function is also added to the unpartitioning tasks to reset the new members of the data handle structure. This callback is used after the completion of an unpartitioning task, but before its outgoing dependencies are released.

4.2.4 The read-only access mode

Within the STF model, the access modes declared by a task on each of its data are particularly important for the parallelism expressed in the task graph. In particular, the distinction between reading information from a data piece and writing information in it is crucial. In Section 2.4, this justified the implementation of partitioning tasks dedicated to the read-only access mode. A read-only partition task can modify the layout of a data handle without disabling it. As long as they only declare to read from a data piece, tasks can do it using the parent handle, or any of the children. Similarly, read-only unpartition tasks can re-enable the parent handle without disabling the children.

In the context of hierarchical tasks, where task submissions can occur in multiple threads simultaneously, this can lead to concurrency issues. Two sub-tasks reading

the same sub-handles can be inserted in parallel and attempt to partition the same data at the same time. Likewise, two regular tasks reading the same handle can turn ready at the same time and try to create the same unpartition task twice. This was not an issue for the previous data manager, as it could rely on the sequential nature of the submission thread.

To protect our data manager from race conditions, we rely on mutual exclusion. The critical sections to protect are the updates in the handle structure that are related to partitioning operations. This includes the partitioning state of a handle and the new members we created to help with the insertion of unpartitioning tasks. Every sub-handle of a partition tree, regardless of its level, shares the same lock as the root handle. This prevents inconsistent modifications in different branches of the partition tree.

The addition of locks removes the risk of simultaneously inserting the same read-only partition task in two different submission threads. The first thread to take the lock will insert the partition task and update the state of the handle. When the lock is released, the second thread will find the handle in a *partitioned* state and will know that it does not have to insert a partition task. For the same reason, it partially solves the issues with the insertion of unpartitioning tasks.

Algorithm 7 Updated version of Algorithm 6

```

1: function UNPARTITIONIFNEEDED(task)
2:   if task was already checked then
3:     return
4:   Mark task as checked
5:   controlTasks  $\leftarrow \emptyset$ 
6:   for H in task handles do
7:     if H has partition plans then
8:       if H has ctrl_unpartition then
9:         Create dependency (ctrl_unpartition  $\rightarrow$  task)
10:      else
11:        write  $\leftarrow$  H access mode in task
12:        ctrl  $\leftarrow$  New control task
13:        Add ctrl to ctrlTasks
14:        Save ctrl in H as ctrl_unpartition
15:        Create dependency (ctrl  $\rightarrow$  task)
16:        AUTOSUBMIT(H, NULL, write, 1)
17:   for ctrl in controlTasks do
18:     Submit ctrl
19:   return

```

Indeed, there is no longer a risk to insert the same read-only unpartition task multiple times. However, since our data manager has to position the outgoing

dependencies of an unpartition task explicitly, Algorithm 6 has to be updated to create more dependencies between a read-only unpartition task and its eventual successors. We present this update in green in Algorithm 7 (lines 9-10). We use the `ctrl_unpartition` task stored in the data handle as a meeting point to create the correct dependency.

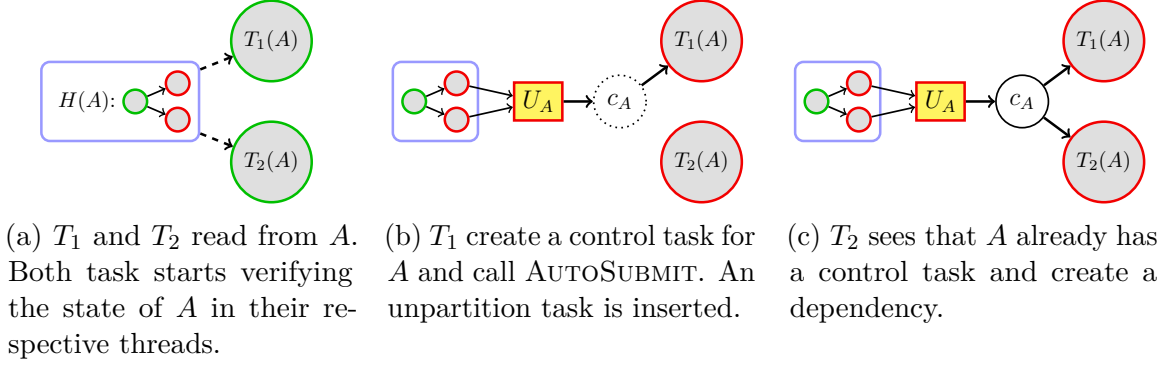


Figure 4.8: Insertion of a read-only unpartition task.

Figure 4.8 illustrates how the new algorithm covers the insertion of read-only unpartition tasks. Just like in the partitioning case, the first thread to take the lock is charged with creating the control task and inserting the unpartition task (T_1 in Figure 4.8b). The second task T_2 can only check its handle A when T_1 's thread releases the lock. Since T_1 already inserted the unpartition task, T_2 finds that a control task already exists in A and directly connects itself to it.

4.3 Conclusion

Implementing hierarchical tasks in a runtime system such as STARPU requires creating new mechanisms and adapting existing ones. The processing of a hierarchical task can be handled at different points of its life cycle. A simplified version processes hierarchical tasks as they are submitted. This limits their potential, but by keeping the submission of the whole graph sequential, we can rely on existing features of STARPU, in particular its data manager. However, to fully implement the hierarchical task model, the insertion of sub-graphs must be delayed until hierarchical tasks are considered ready.

As explained in Chapter 3, the main challenge in implementing this model on top of the STF is data management. We extended the existing data manager in order to correctly adjust the data layout around hierarchical tasks according to the model we constructed. It requires a finer control of the dependencies of partitioning tasks, in order to correctly position them around hierarchical tasks within the DAG. This involves disabling some of STARPU's automatically inferred dependencies and, if necessary, replacing them. Partitioning operations are fairly straightforward: by

construction, they are ready to be executed as soon as they are needed, when a hierarchical task starts submitting sub-tasks. Unpartitioning operations are more tricky, as they must be placed at execution time, between the last sub-tasks and their successor. Inserting such a task is fairly unnatural in the STF model. To this end, we extended the handle structure to store a control task. The purpose of this task is to enable the declaration of a dependency from an unpartition task to an already existing task.

As long as the handles are accessed in *write* or *read-write* mode, they are only used by one task (hierarchical or not) at a time, and their partitioning states are still updated sequentially. However, with the *read-only* access mode, multiple tasks might attempt to look at the same handle in parallel. We used mutual exclusion to protect our data manager from race conditions. The updates in the partitioning tree of each data piece are protected by a lock to ensure that partitioning tasks are only inserted once. When multiple tasks must unpartition the same handle before reading from it, we can rely on the control task stored in the handle to connect all of them to a single unpartition task.

Now that our model has been completely implemented, we must evaluate how it performs. The next chapter showcases the potential performance benefits of the use of hierarchical tasks for dense linear algebra.

Chapter 5

Experimental Evaluation

Contents

5.1	Application of Hierarchical Tasks for Dense Linear Algebra	74
5.1.1	Recursive descriptors	74
5.1.2	Updating the kernels	76
5.2	Experimental results	76
5.2.1	Experimental settings	76
5.2.2	Hierarchical tasks overhead	78
5.2.3	Matrix-matrix multiplication	79
5.2.4	Cholesky factorization	82
5.2.5	LU factorization	85
5.2.6	Impact of the dynamic data manager	85
5.2.7	Comparison with other frameworks	87

VALIDATING THE HIERARCHICAL TASKS MODEL requires an evaluation of the benefits it provides to an application. To illustrate those, we chose to apply hierarchical tasks to dense linear algebra. This area of application is widely used to evaluate task-based runtime systems. Linear algebra operations present very regular task graphs that could be improved on heterogeneous systems by having multiple levels of task granularity coexist.

The CHAMELEON library [2] implements task-based versions of dense linear algebra operations over multiple runtime systems (STARPU, PARSEC, OPENMP, and QUARK). In this chapter, we start by presenting how CHAMELEON was updated to use hierarchical tasks. Most notably, the data structure representing matrices was extended in a recursive version storing multiple levels of partitioning. Kernels were also modified to incorporate hierarchical tasks. We can then move on to the actual evaluation of hierarchical tasks. After presenting the experimental settings, we study the behavior of hierarchical tasks. To this end, we provide their performances for operations (of varying level of affinity with hierarchical tasks) and compare them to existing solutions.

5.1 Application of Hierarchical Tasks for Dense Linear Algebra

CHAMELEON implements dense linear algebra routines such as matrix products, matrix factorizations, or solvers of linear equations. It can be interfaced with various task-based runtime systems, including STARPU, to generate task graphs of these operations on tiled matrices. To use hierarchical tasks in CHAMELEON, we had to extend the representation of tiled matrices to account for the potential partitioning of tiles. A recursive data structure was implemented to traverse matrices with multiple levels of partitioning. Multiple kernels were also adapted to enable hierarchical tasks. This required both a method to determine what tasks of the DAG are hierarchical and a function generating the appropriate sub-graph.

5.1.1 Recursive descriptors

The first step in applying hierarchical tasks to CHAMELEON is to describe the data. In a dense linear algebra context, data are in the form of matrices. To represent these matrices, CHAMELEON relies on a data structure called *descriptor*. A descriptor contains all the information defining a tiled matrix, such as its dimensions and the dimensions of its tiles. It also includes pointers to the matrix data and to an array of tiles. Each tile stores its number of rows and columns, its leading dimension, and a pointer to its data. They are registered in STARPU and the resulting handles are also recorded in an array of the descriptor structure.

Figure 5.1 schematically illustrates a CHAMELEON descriptor. The descriptor itself

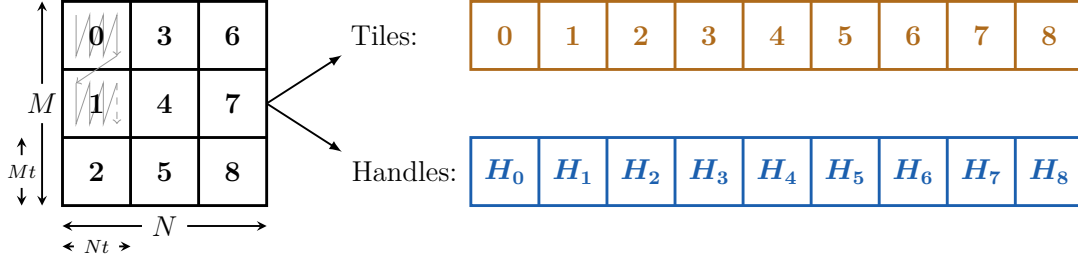


Figure 5.1: Information stored in a regular matrix descriptor.

is represented in **black**, with its array of tiles in **brown** and the corresponding array of handles in **blue**. The matrix values are stored tile by tile in column-major order.

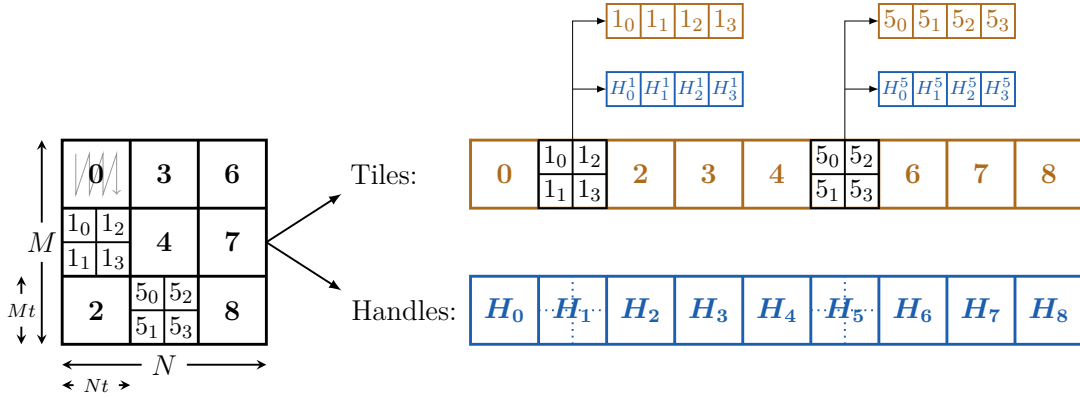


Figure 5.2: Information stored in a recursive matrix descriptor.

Since the tile handles have to be partitioned to use hierarchical tasks, the existing descriptors must be updated to reflect that. Figure 5.2 schematically illustrates a recursive descriptor, i.e. a descriptor with “partitionable” tiles. In a recursive descriptor, each tile is given an additional member defining its *format*. The format of a tile indicates whether it is a regular tile, or a *sub-descriptor*. If a tile T_i is a sub-descriptor, it includes a pointer to a descriptor representing the partitioned tile. This descriptor has its own array of tiles (the sub-tiles of T_i) and its own array of handles (the sub-handles of T_i ’s handle). Each sub-tile might also be a descriptor, making this recursive structure can be as deep as necessary. In Figure 5.2, the tiles 1 and 5 are sub-descriptors. Partition plans are created for the corresponding handles H_1 and H_5 , and the resulting sub-handles H_1^1 and H_5^5 are stored in the sub-descriptors.

Making the “partitionable” tiles into descriptors means that they can be treated by CHAMELEON as if they were regular tiled matrices. This allows us to use the existing algorithms to submit sub-graphs, once the kernels have been updated to include hierarchical tasks.

5.1.2 Updating the kernels

With our data now arranged in a hierarchical-friendly way, we update the submission of kernels in CHAMELEON to use hierarchical tasks. We implemented a simple strategy where hierarchical tasks are guided by the data layout in the recursive descriptors. In the absence of a “hierarchical” scheduler capable of making hierarchical task decisions at execution time, this static approach allows us to showcase some advantages of hierarchical tasks.

In that strategy, the task of a kernel is processed as hierarchical if all the tiles accessed by the kernel are sub-descriptors. When a hierarchical task is processed, the handles of the tiles are partitioned into the handles of the sub-tiles, and the task graph associated to that kernel is submitted. As mentioned above, most functions describing the graph of a specific kernel already exist in CHAMELEON. Since all the tiles involved are also descriptors, we can simply call the appropriate function with them, which will submit the sub-graph.

In the rest of this chapter, our objective is to validate the hierarchical task model by using it to create correct DAGs with various levels of task granularity. The layout of recursive descriptor provides us with a partitioning scheme of a matrix. We use various partitioning schemes to decide where to place hierarchical tasks and address certain limitations in the CHAMELEON applications we chose. Three partitioning schemes have been implemented for these experiments. They are described further down for the experiments they were used in.

5.2 Experimental results

This section compiles the experimental results of different configurations of hierarchical tasks and compares them with other strategies. First, we present our experimental platforms and settings and explain how we selected the tile sizes used in the rest of the section. Then, we study the overhead and performances of hierarchical tasks when using the implementation detailed in Section 4.1.3. This implementation processes hierarchical tasks as soon as possible, in the submission thread. The following part examines the impact of the extended data manager that enables dynamic modifications of the data layout at runtime. Finally, we compare the hierarchical task model to the parallel task paradigm and the PARSEC runtime system.

5.2.1 Experimental settings

First, we give an overview of our experimental settings by presenting the platforms that we used and the configuration of CHAMELEON and STARPU. We also explain how the tile sizes of our experiments were chosen, as well as the nomenclature we use in figures.

The following experiments were conducted on two architectures, presented in

5. Experimental Evaluation

Name	Processor	GPU	Memory
INTEL-V100	2 x INTEL XEON GOLD 6142, 16 cores, 2.6GHz	2 x NVIDIA V100 (16GB)	384GB
AMD-A100	2 x AMD ZEN3 EPYC 7513, 32 cores, 2.6GHz	2 x NVIDIA A100 (40GB)	512GB

Table 5.1: Characteristics of the experimental platforms.

Table 5.1.

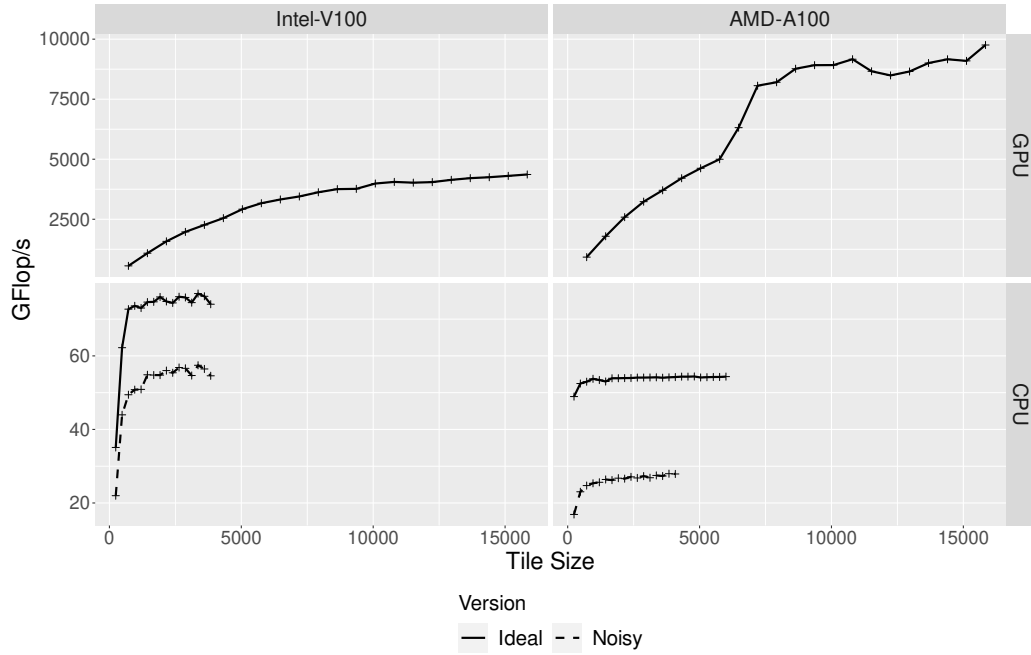
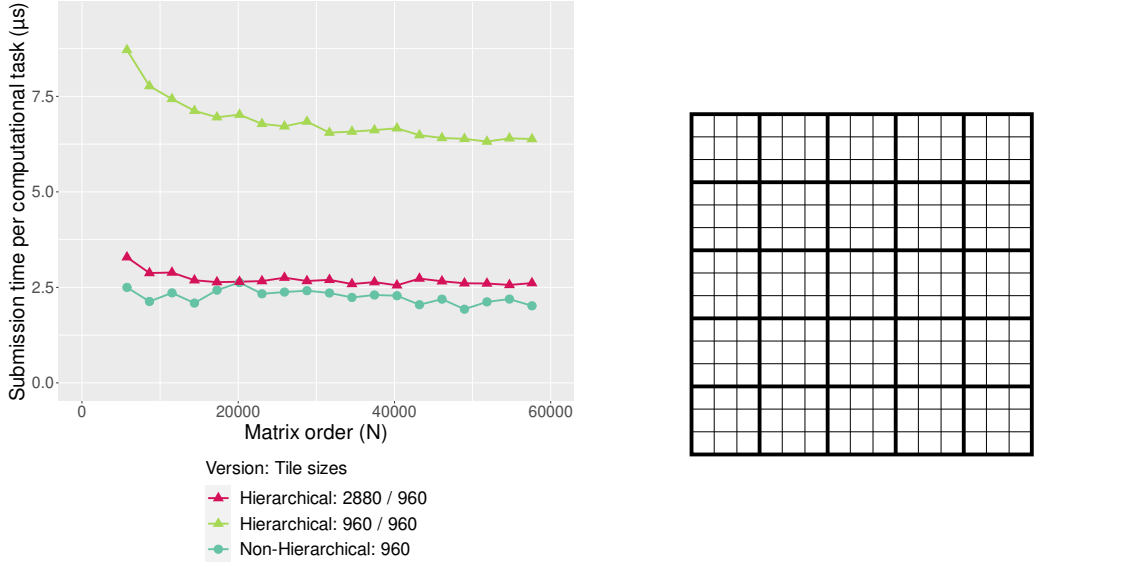


Figure 5.3: Performances of a matrix product (DGEMM) kernel. In the ideal setting a single kernel runs on a single resource (CPU core or GPU device). In the noisy setting every CPU core runs 10 independent DGEMM kernels.

Figure 5.3 shows the double precision performance of a single dense DGEMM kernel for different sizes of square matrices on both platforms. We present with a solid line the actual performance for both a single CPU and a single GPU. We can see that on the CPU side, the processor’s peak performance is achieved for matrix sizes not exceeding 500. Providing larger matrices does not improve the absolute performance on a single core. On the GPU side, we can observe that the larger the matrix size, the higher the performance. This experiment aims to find the best trade-off for the task granularity between the CPU cores and GPU devices. Having a granularity of tasks not large enough will limit GPU performance. On the other hand, using very coarse grain tasks limits parallelism and may affect CPU performance. Thus, in future experiments, we will mainly use a tile size of 2880 (resp. 5700) on the INTEL-V100



(a) Submission cost of computational tasks for DGEMM with all tiles partitioned on INTEL-V100. Lower is better. (b) Partitioning every tile of the matrix.

Figure 5.4: Evaluation of the overhead of hierarchical tasks.

(resp. AMD-A100) platform.

The dotted line is the normalized performance of a batch of independent DGEMM kernels spread evenly across all CPU cores. This accounts for the performance degradation resulting from memory contention in shared memory hierarchy levels (L3 cache, for example), and for the CPU frequency reduction due to the higher number of concurrent resources exploited.

On both architectures, STARPU and CHAMELEON are compiled with GCC 10.3.0, Nvidia CUDA 11.2, and the Intel MKL 2020. STARPU has been configured to use a single stream per GPU and to pipeline four events per stream. Unless otherwise specified, we use all CPU cores and GPU devices available on the node. All the experiments in STARPU relied on the Deque Model Data Aware Sorted (DMDAS) scheduler. A description of the basic principles of this scheduler is given in Section 2.3.2.

In the next section, the hierarchical variants use the following notation: $x/y/z/\dots$. It means that each initial tile is of size x and is partitioned according to one of the three schemes mentioned above into tiles of size y , which are in turn split into tiles of size z etc.

5.2.2 Hierarchical tasks overhead

To evaluate the overhead induced by hierarchical tasks, we consider the graph of a matrix-matrix multiplication (GEMM) using a tile size of 960. The partitioning

scheme used in this experiment recursively splits each matrix tile. An example is shown in Figure 5.4b with one recursion level: each matrix tile is divided into a finer tile size. Figure 5.4a compares the graph submission time per computational task in two configurations. The ‘960’ curve represents the non-hierarchical case. The ‘960/960’ curve shows the worst possible scenario: the DAG is composed only of hierarchical tasks and each submits exactly one task when processed. This doubles the number of tasks submitted and heavily increases the workload of the data manager, making the submission time per computational task roughly 3.5 times slower. Finally, the ‘2880/960’ curve is a more realistic scenario, where the graph is first submitted at coarse grain (with a tile size of 2880) and then refined down to the same granularity as the previous configurations (960). In this case, each hierarchical task submits $\lceil 2880/960 \rceil^3 = 27$ regular tasks when processed, thus amortizing the overhead induced by the management of hierarchical tasks.

5.2.3 Matrix-matrix multiplication

For the matrix product kernel $C = \alpha AB + \beta C$, the main objective of the hierarchical scheme is to balance the workload between CPU cores and GPU devices based on their respective computing power. A custom partitioning scheme is used. We first define a computing power for each resource, \mathcal{P}_{CPU} , and \mathcal{P}_{GPU} respectively, and then we estimate the computing power ratio of the CPU resources:

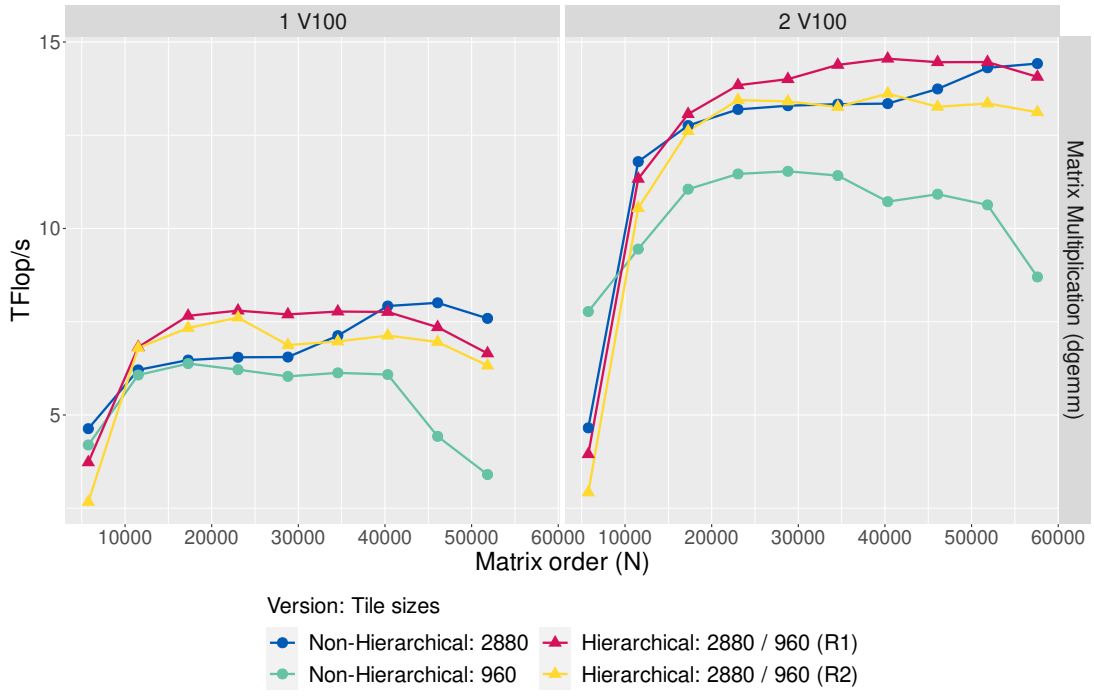
$$R = \frac{\#_{CPU} \times \mathcal{P}_{CPU}}{(\#_{CPU} \times \mathcal{P}_{CPU}) + (\#_{GPU} \times \mathcal{P}_{GPU})}$$

Platform	INTEL-V100				AMD-A100			
	\mathcal{P}_{CPU}	\mathcal{P}_{GPU}	1 GPU	2 GPU	\mathcal{P}_{CPU}	\mathcal{P}_{GPU}	1 GPU	2 GPU
<i>R1</i>	50.9	6.65e3	19%	10%	25.0	14.5e3	10%	5%
<i>R2</i>	55.0	4.50e3	26%	15%	28.0	9.5e3	16%	9%

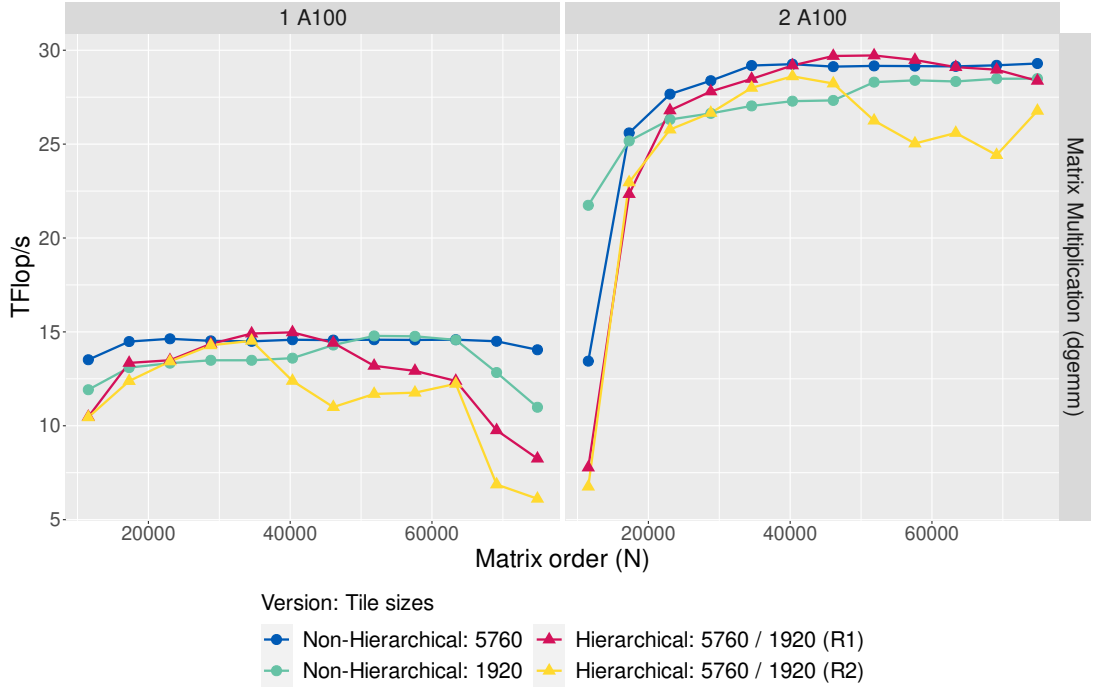
Table 5.2: Performances in GFlop/s used to compute *R1* and *R2* for each individual computational unit, and resulting percentages of tiles partitioned for the CPUs in the hierarchical version in Figure 5.5.

We finally compute the number of tiles for which the partition planning has to be done with respect to R : $\lfloor R \times Nt^2 \rfloor$, where Nt^2 is the total number of tiles of the C matrix. Table 5.2 summarizes the different values of R we selected for our experiments. *R1* is computed from the best performances reached for tiled matrix multiplication using all the CPU cores without the GPU devices and vice-versa. *R2* is computed from the plateau values from Figure 5.3, using the noisy setting in the CPU case.

We evaluate the behavior of the GEMM operation on those matrices, using one and two GPUs (Figures 5.5a and 5.5b). On the INTEL-V100 platform, this strategy allows



(a) INTEL-V100 platform.



(b) AMD-A100 platform.

Figure 5.5: Evaluation of hierarchical tasks in the matrix-matrix multiplication (DGEMM) kernel, with a fixed percentage of hierarchical tasks (see Table 5.2). Higher is better.

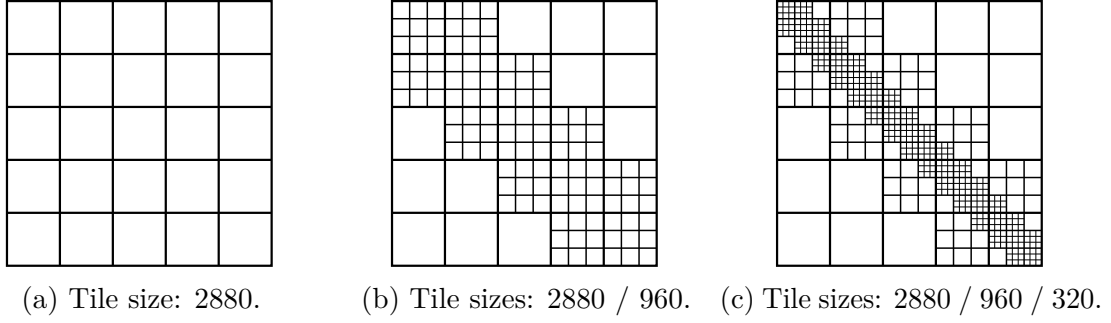


Figure 5.6: Illustration of the matrix layouts used in the following experiments.

the hierarchical versions to outperform the standard ones on medium matrix sizes by providing the CPU cores of the node with tasks working on smaller tiles. When using large tiles (2880) on bigger matrix sizes, STARPU's scheduler can start assigning more work to the CPUs without penalizing execution time and catches up with the hierarchical versions. On the AMD-A100 platform, the performance ratio between the CPU cores and the GPU devices is so high that the contribution of the CPU cores is marginal. Once the matrices become large enough, the percentage of smaller tiles becomes a disadvantage and negatively impacts the performance. This could be compensated by partitioning fewer tasks as the matrix size increase. However, even in this situation, the hierarchical version using *R1* can achieve similar performance to standard CHAMELEON. A common trend between both platforms is that the *R1* ratio provides better results than *R2*, because it was computed using values more representative of the disparity in computing power between CPUs and GPUs for this type of operation. Generally speaking, the dense matrix product is a very regular operation that is rather unfavorable to the use of hierarchical tasks. Despite that, the hierarchical variants have a good behavior and can achieve high performance with a simple partitioning strategy.

We present experimental results in the forthcoming section illustrating the hierarchical tasks' behavior on more advanced dense linear algebra operations to illustrate the flexibility of hierarchical tasks better. First, we consider operations relying on Cholesky decomposition (POTRF): POSV (linear system solving, in this case, of a single vector) and POINV (matrix inversion). These operations have complex task graphs, and in the case of POINV, validate the anti-dependency problem (*WRITE* after *READ*). Furthermore, we provide an experimental evaluation using the LU factorization without partial pivoting (GETRF_nopiv) which has a wider task graph than the Cholesky factorization and for which the criticality of some tasks (mainly those corresponding to the tiles on the diagonal) can severely degrade performance if delayed.

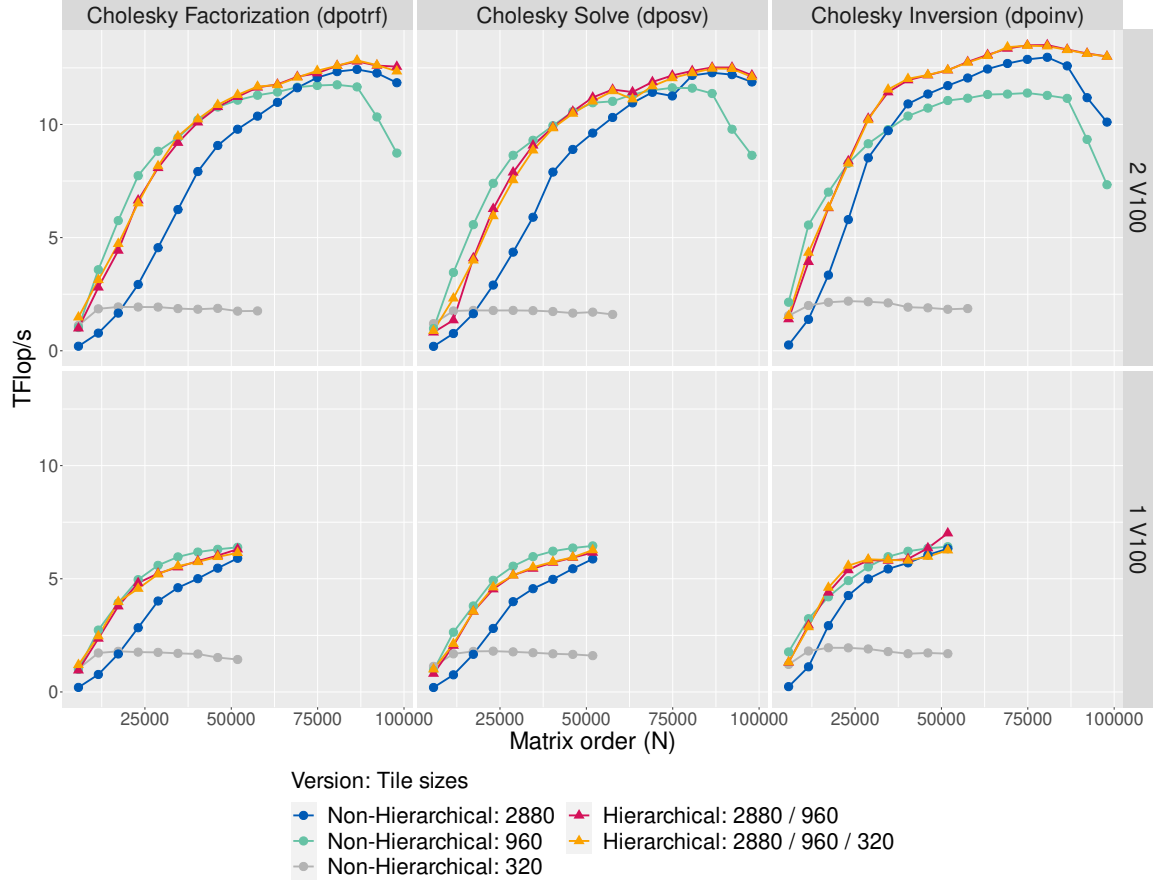


Figure 5.7: Evaluation of hierarchical tasks in Cholesky type operations (DPOTRF, DPOSV, DPOINV) with a diagonal distribution of the hierarchical tasks on the INTEL-V100 platform. Higher is better.

5.2.4 Cholesky factorization

The partitioning scheme we implemented for the following experiments splits recursively the matrix along the diagonal, sub-diagonal and superdiagonal, to mimic \mathcal{H} -Matrices algorithms and use finer granularities on the critical path. An example is shown in Figure 5.6. Figures 5.7 and 5.8 show the results of operations relying on the Cholesky decomposition: POTRF (actual Cholesky decomposition), POSV (linear system solving, with a single right-hand side) and POINV (matrix inversion). Contrary to the previous DGEMM experiment, this partitioning scheme doesn't simply attempt to supply all resources with the right amount of work. Instead, it tries to improve performance on the critical path of the operation, because CHAMELEON requires all POTRF kernels (which are on the critical path of the factorization and located on the diagonal tiles) on CPU cores. This is illustrated in both figures where we can see that the granularity having the best behavior used for the standard version use smaller tiles than the one relying on the hierarchical tasks on both platforms. Thanks to

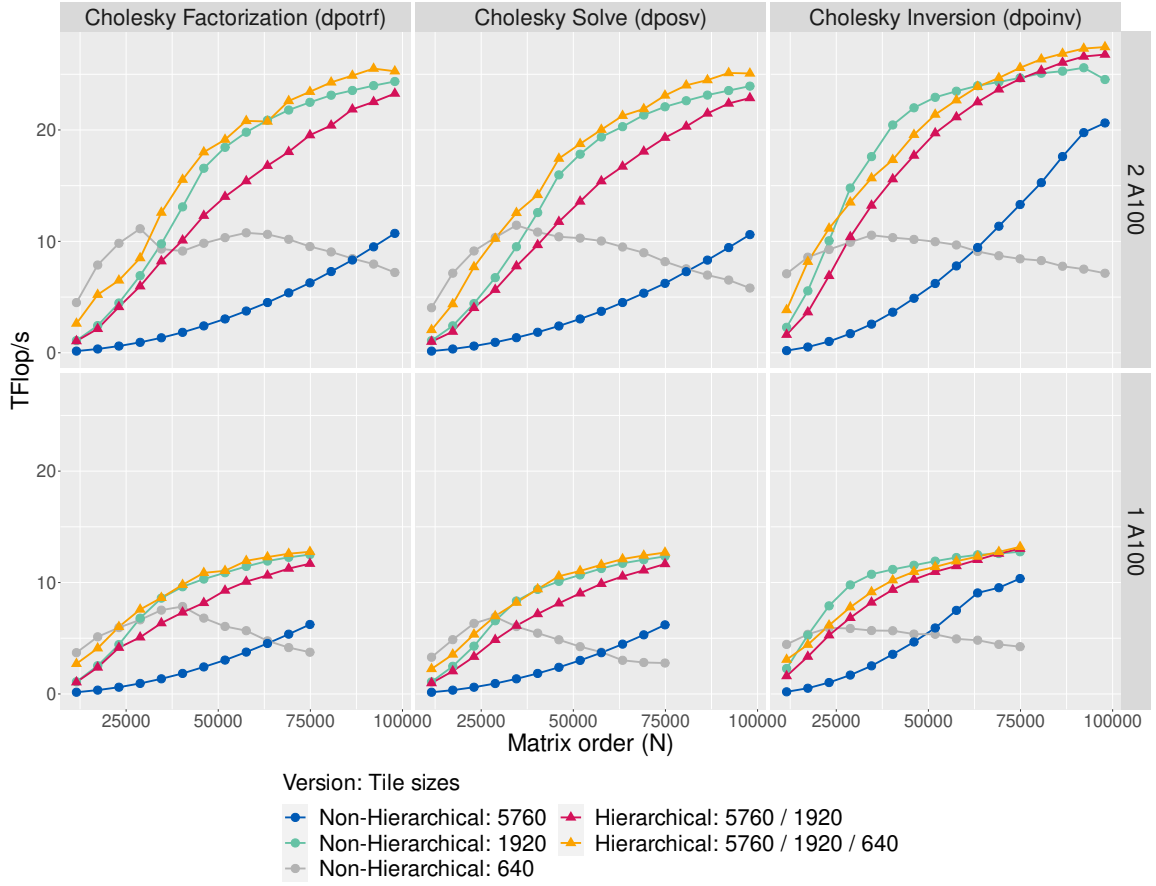
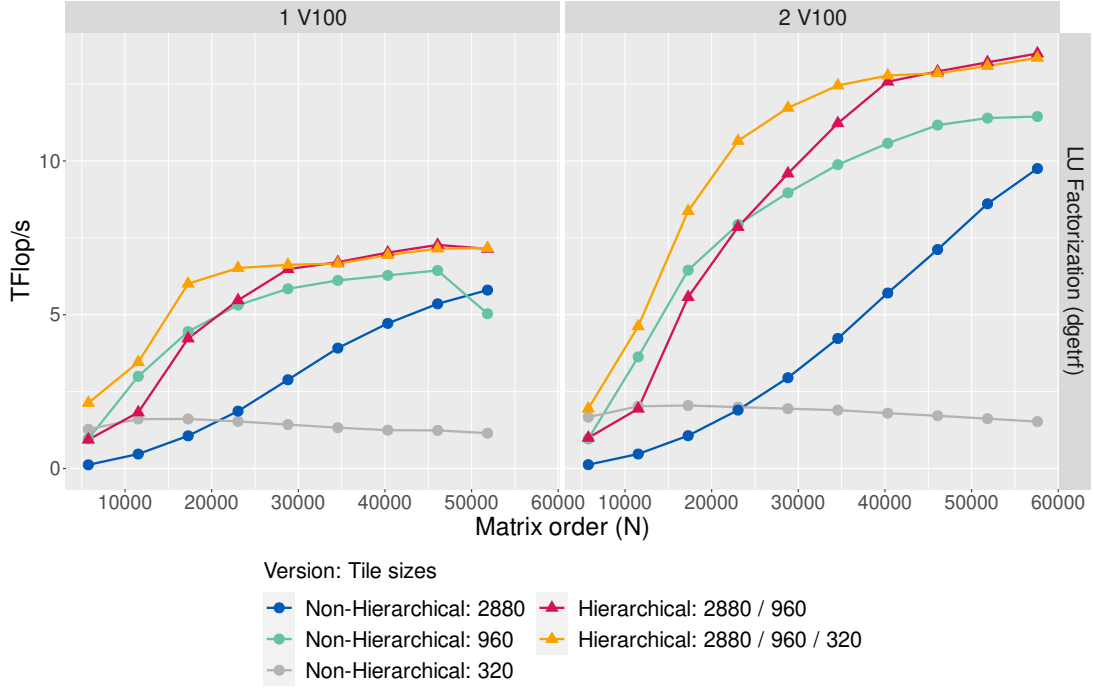
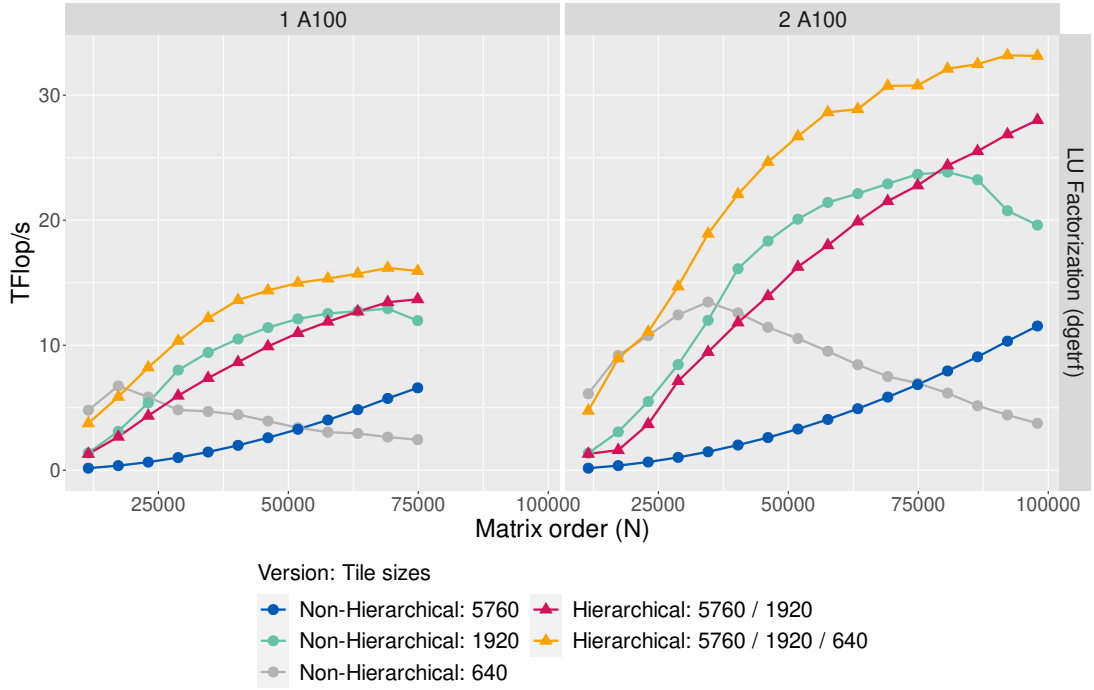


Figure 5.8: Evaluation of hierarchical tasks in Cholesky type operations (DPOTRF, DPOSV, DPOINV) with a diagonal distribution of the hierarchical tasks on the AMD-A100 platform. Higher is better.

hierarchical tasks, we can partition the tiles along the diagonal and split those large tasks into sub-graphs with a smaller granularity, allowing for better CPU utilization on the critical path. For all the kernels and platforms, enabling hierarchical tasks enhances performance for most matrix sizes. We can also observe that the regular version can catch up for huge matrix sizes. The sudden drop observed at the end of some non-hierarchical curves is explained by a conflict between the STARPU scheduler data prefetching and eviction in GPU memory, as explained in [46]. The experimental results illustrate the interest in hierarchical tasks for tackling the granularity problem of heterogeneous architectures. This highlights the fact that hierarchical tasks can find a better trade-off to fully exploit the whole platform: each resource is used with a task granularity adapted to its characteristics.



(a) INTEL-V100 platform.



(b) AMD-A100 platform.

Figure 5.9: Evaluation of hierarchical tasks in the LU decomposition (DGETRF_nopiv) with diagonal distribution of the hierarchical tasks. Higher is better.

5.2.5 LU factorization

We provide in Figure 5.9 an experimental evaluation of hierarchical tasks to enhance the dense LU factorization without pivoting (DGETRF_nopiv). This dense linear algebra kernel exhibits a high amount of parallelism. It requires the tiles on the diagonal to be processed as fast as possible to release many dependencies. Once again, we can observe that hierarchical tasks improve performance on both platforms. We can also observe that the non-hierarchical version of CHAMELEON achieves lower performance because of the critical tasks on the diagonal: since the matrix partitioning is uniform, the best tiling uses a large granularity to exploit the GPU devices fully.

Finally, we can observe that the gap between the hierarchical tasks and standard Chameleon is more significant on the more heterogeneous AMD-A100 platform. This is mainly because, on this platform, large tiles (5760) are needed to exploit the potential of the A100 devices fully. Thus, not using hierarchical tasks to adapt the granularity for each device penalizes performance.

5.2.6 Impact of the dynamic data manager

While the results we just covered illustrate the potential of hierarchical tasks, they rely on a static implementation of the model. In this section, we examine how the fully implemented model compares to the previous results. This version of the model extends the data manager of STARPU to insert partitioning tasks at runtime. It enables a dynamic DAG generation, as hierarchical tasks can now be processed at execution time, when they become ready.

Figure 5.10 presents in dashed lines the performances reached with the previously established hierarchical task configurations when the dynamic data manager is active. We can see that it results in some overhead, which was expected, since part of the submission process is delayed and the data manager has to synchronize certain tasks at runtime to maintain the consistency of their data. This overhead is particularly noticeable for small matrices (e.g. Figure 5.10a), which exhibits less parallelism. Since hierarchical tasks are processed at runtime, the delayed insertion of sub-tasks is more penalizing if it blocks a larger portion of the graph. For larger matrix sizes, the impact of the data manager is less important, as the processing of hierarchical tasks and the work of the data manager are spread more evenly across a larger task graph with more parallelism.

It is also important to note that the scheduler is a critical component of the course of the execution and even more so when we aim to process hierarchical tasks at runtime. For these experiments, we used the existing DMDAS scheduler, which is not designed for hierarchical tasks, leaving a lot of room for improvement.

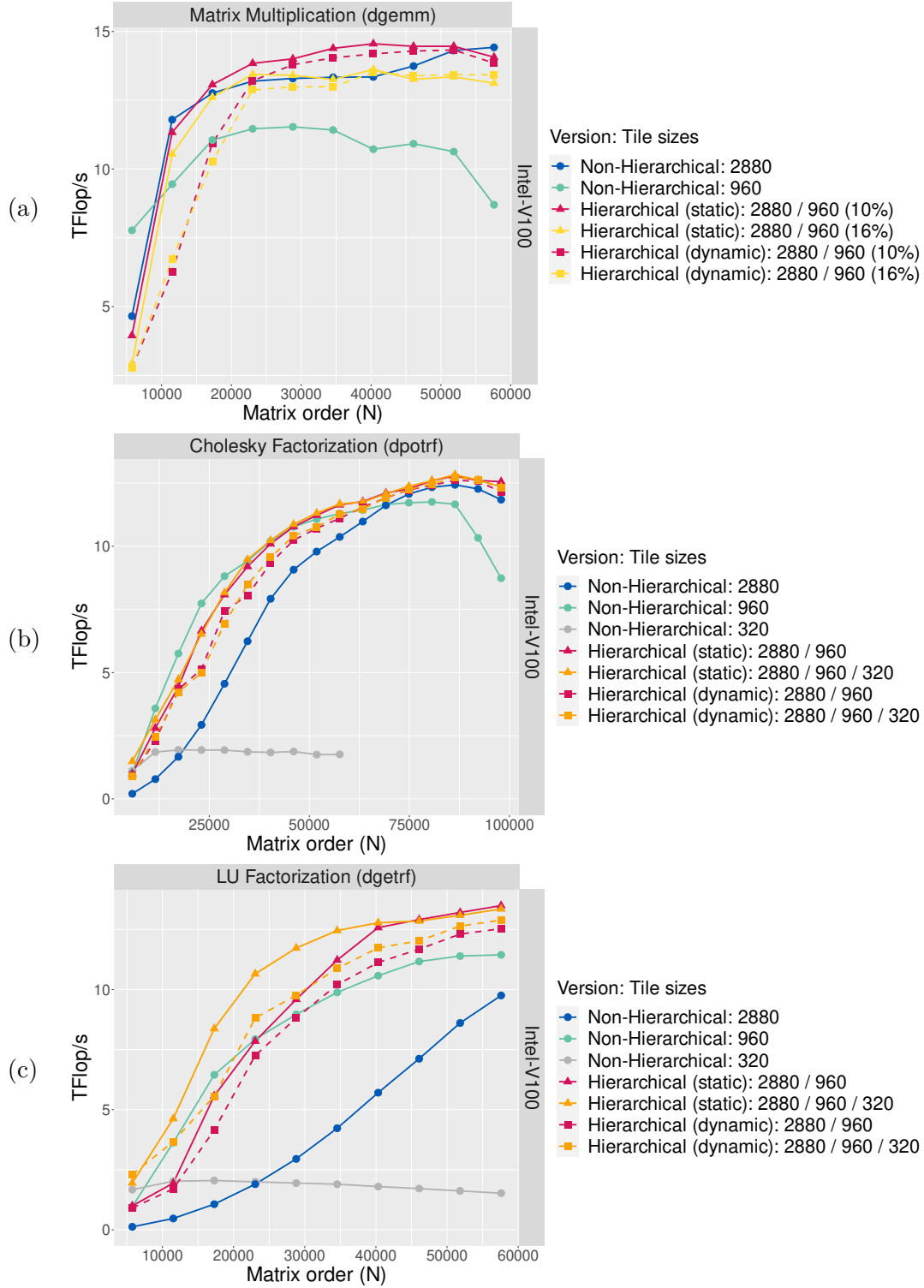


Figure 5.10: Evaluation of the performance impact of the dynamic data manager for matrix multiplication (DGEMM) and factorization (DPOTRF, DGETRF_nopiv) on the INTEL-V100 platform. Higher is better.

5.2.7 Comparison with other frameworks

In this section we compare our hierarchical task paradigm with two different approaches. We consider the so-called *parallel task* paradigm, presented in [19] and available within STARPU, where CPUs are aggregated to excute parallel tasks which helps to limit the impact of granularity issues. We also present the results of the PARSEC runtime system in the state-of-the-art library DPLASMA [15]. The comparison will consider the matrix product operation (GEMM), the Cholesky factorization (POTRF) and the LU factorization without pivoting (GETRF-nopiv). Each point for a given framework will consider the best observed performance with respect to tile size, number of streams, and task partitioning scheme (for the hierarchical variants). The parameters used are summarized in Tables 5.3 and 5.4. Finally, concerning the approach using parallel tasks, one parallel worker is assigned to each socket of the INTEL-V100 platform, to share the same cache. For the AMD-A100 architecture, two parallel workers have been assigned to each CPU socket because of the high CPU count on the platform.

INTEL-V100	Non-Hierarchical	Hierarchical	Parallel tasks	PARSEC
Tile sizes:	2880, 960, 320	2880, 960, 320	2880, 960, 320	2880, 960, 320
CUDA streams:	1, 2, 4, 8 streams	1 stream	1 stream	1 stream
Partitioning scheme:	N/A	See Table 5.2 and Figure 5.6	N/A	N/A

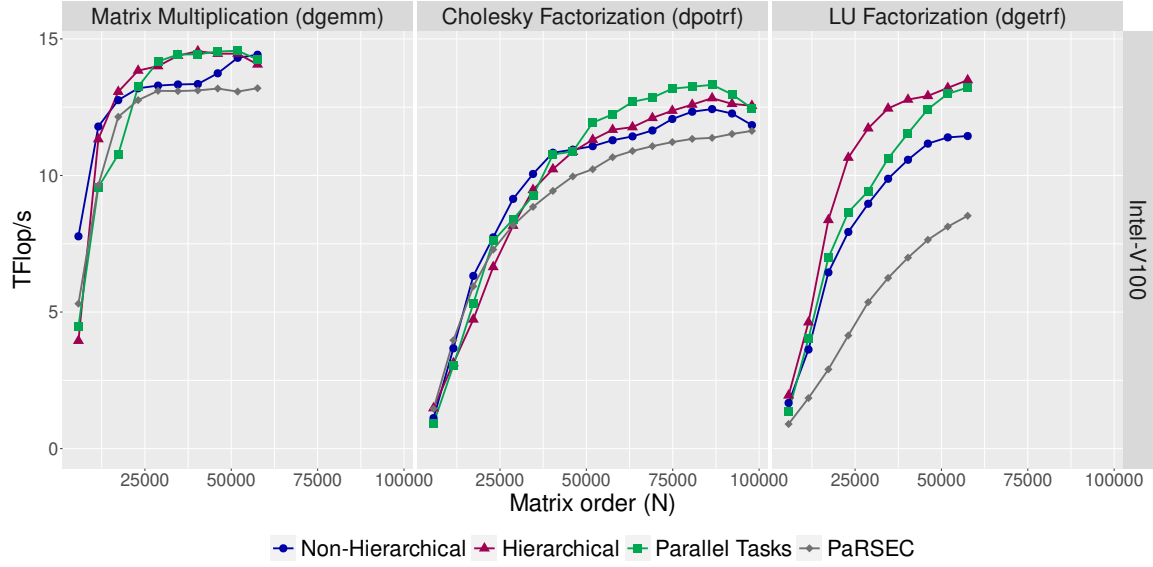
Table 5.3: Parameters explored to find the best overall performances for each operations in Figure 5.11a (INTEL-V100 platform).

AMD-A100	Non-Hierarchical	Hierarchical	Parallel tasks	PARSEC
Tile sizes:	5760, 1920, 640	5760, 1920, 640	5760, 1920, 640	5760, 1920, 640
CUDA streams:	1, 2, 4, 8 streams	1 stream	1 stream	1 stream
Partitioning scheme:	N/A	See Table 5.2 and Figure 5.6	N/A	N/A

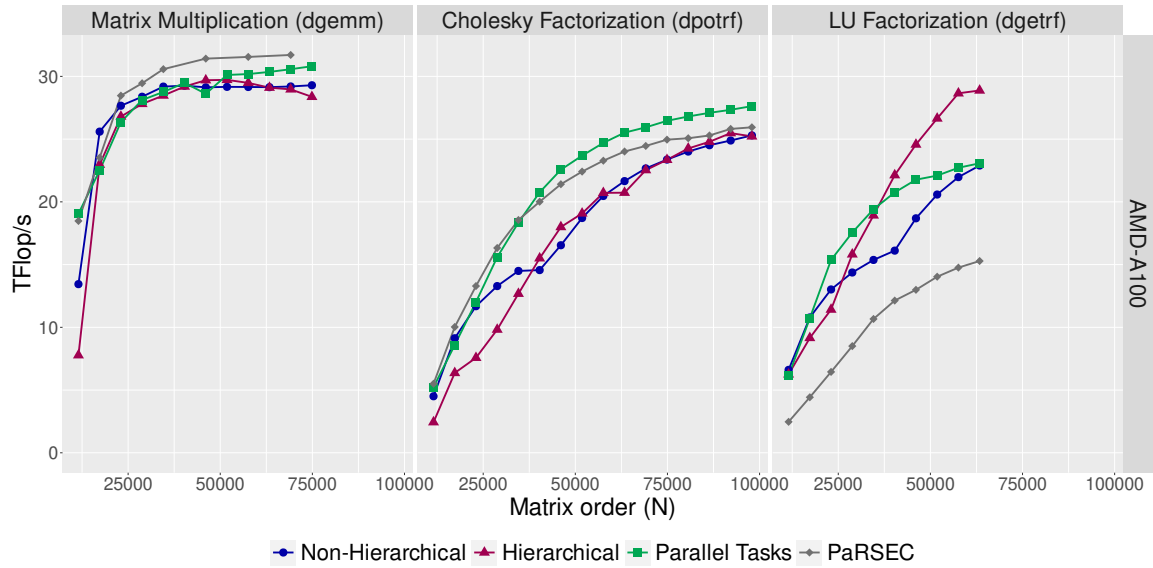
Table 5.4: Parameters explored to find the best overall performances for each operations in Figure 5.11a (AMD-A100 platform).

The matrix multiplication (GEMM) performances in Figure 5.11 (left) show us that the different approaches have a better behavior than the standalone STARPU implementation (no hierarchical tasks). We can also observe that using parallel tasks is a very efficient solution except for moderate size matrices. This can be explained by the fact that since the GEMM operation is highly parallel, the synchronizations induced by the use of parallel tasks is not impacting performance.

We also provide in Figure 5.11 (center) additional performance comparison using the Cholesky factorization (POTRF). We can observe that the parallel worker approach is the most efficient approach. Our hierarchical task variant is not able to achieve the same performance. This is mainly due to scheduling issues. It is important to remind



(a) INTEL-V100 platform.



(b) AMD-A100 platform.

Figure 5.11: Comparison of the behavior of the hierarchical task paradigm with respect to the parallel tasks and PARSEC for the matrix product operation (DGEMM), Cholesky factorization (DPOTRF), and LU factorization without pivoting (DGETRF). Higher is better.

that we use the standalone STARPU scheduler for heterogeneous systems, namely DMDAS (presented in Section 2.3.2). We did not tune or improve the scheduler to handle hierarchical tasks. Thus, there is a lot of room for improvement from this point of view. This represents a fundamental problem on its own and will be considered in future work.

Finally, we provide in Figure 5.11 (right) the observed behavior for the LU factorization kernel (without pivoting). We can see that the hierarchical variant is able to achieve the highest performance on most test cases and for both platforms. The parallel workers approach also achieves good performance but is slightly less efficient. This mainly due to the fact that for moderate to large matrices, tile sizes of 2880 are used. However, these large tiles are processed by a single parallel worker (1 CPU socket for the INTEL-V100 platform and half of a CPU socket for AMD-A100 one). This will represents a limitation when processing tasks on the critical path. One could improve the behavior by dynamically changing the amount of CPU cores associated to a parallel worker but this is out of the scope of this experimental evaluation. The uncharacteristically poor performances of PARSEC in this scenario is unfortunately due to the lack of CUDA implementation for some kernels in the LU factorization of DPLASMA.

The results presented in Figure 5.11 illustrates that even with a basic scheduling and partitioning strategy, the flexibility of the hierarchical task paradigm can find a better trade-off between parallelism and task granularity. All in all, the set of results presented in this section enlighten the potential and the interest of our approach but also points out that advanced scheduling strategies will have to be designed to achieve high performance in this new context.

Chapter 6

First Steps of Hierarchical Tasks toward Distributed Memory

Contents

6.1	Introducing shared data	92
6.2	Automatic pruning	93
6.3	Communications at runtime	94
6.4	Conclusion	95

UP UNTIL NOW, our efforts to design a hierarchical tasks model have been restrained to shared memory systems. Containing the graph to a central memory limited the need for additional data management such as network communications. Once our model became capable to function properly on a single node, a natural follow up is to extend it over multiple nodes. The most direct translation of our model for distributed systems consists in restricting hierarchical tasks to a single node. By submitting and processing hierarchical tasks within one node, we can mostly rely on the work done for the shared memory model. This is already interesting when considering heterogeneous nodes, but it is tempting to extend the model further and propose more features for distributed memory systems. In particular, being able to submit and process hierarchical tasks over multiple nodes would greatly enhance the expressivity of the model.

In this chapter, we go over the first few extensions of the hierarchical task model dedicated to distributed memory systems. Since STARPU-MPI, the extension of STARPU targeting distributed memory architectures, maps the data over the different nodes, we first present the necessity to share data ownership between multiple nodes. This enables us to insert a single hierarchical task in the local graph of different nodes. When it is processed, the sub-graph will be inserted between the nodes according to their ownership of the sub-data. We then explain how this mechanism results in a more user-friendly way to prune the graph (see Section 2.2.2 for a definition of the pruning operation). Finally, hierarchical tasks need to be able to position new communications at runtime, due to their dynamic nature.

6.1 Introducing shared data

As explained in Section 2.1.2, STARPU-MPI maps all the data of distributed applications over the nodes of the targeted platform. This data mapping is then used to determine where the tasks will be executed. In this model, a task is, very logically, not allowed to be submitted and executed across multiple nodes. In order to submit a single hierarchical task on different nodes, we must introduce a type of data that can be owned by all the nodes involved at the same time. This new type of data are called *shared data*.

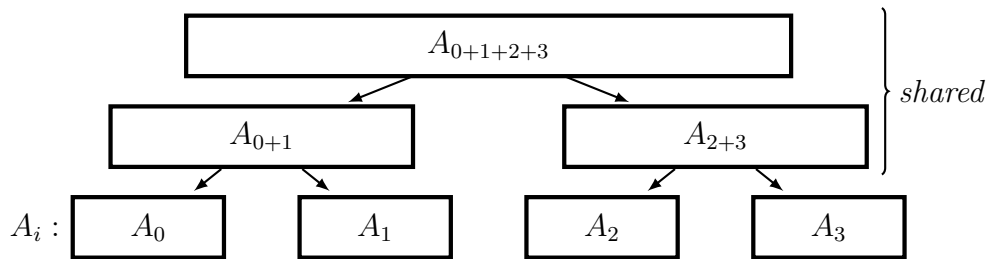


Figure 6.1: Partition tree with shared data.

A shared data is the root of a partition tree, as illustrated in Figure 6.1. The

children of a shared data can be shared or not themselves, but all the leaves of the tree must be non-shared data. In Figure 6.1, the leaves A_i are all owned by four different nodes i . A_{0+1} and A_{2+3} are shared respectively between the nodes $[0,1]$ and $[2,3]$. The root itself is shared between all four nodes. This partition tree can be extended by partitioning the leaves further, but all the children of A_i will be non-shared data owned by the node i . The shared data $A_{0+1+2+3}$ can be used to submit a hierarchical task on the four nodes. In the corresponding sub-graph, the hierarchical tasks using A_{0+1} (respectively A_{2+3}) are only executed on nodes 0 and 1 (respectively 2 and 3).

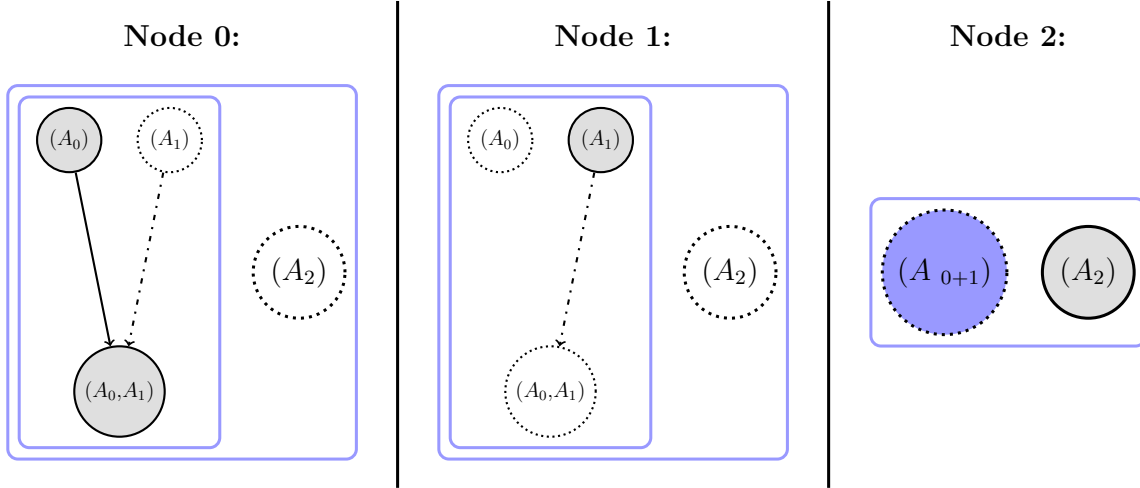
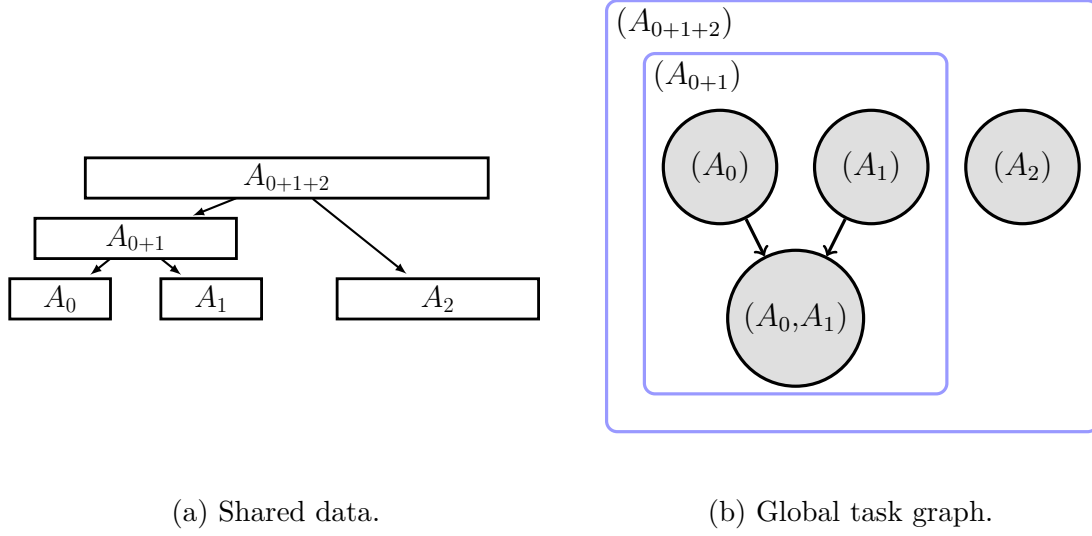
The construction of shared data is automatic. Like in any STARPU-MPI application, the user is only required to register its non-shared data into handles and to map them on the different nodes. The runtime system then automatically marks the parent handles that are considered shared.

For now, shared data are only meant to be a tool in the submission of distributed DAG with hierarchical tasks. They are not meant to be used in computations, but this possibility could be explored more in future works.

6.2 Automatic pruning

A common issue that limits the scalability of STARPU-MPI applications is the fact that every node is required to go through the entire submission of the task graph sequentially (see Section 2.2.2). To avoid this issue, programmers can *prune* the DAG of their application by avoiding to submit certain tasks on the nodes that are not involved with them (typically because they do not own any of the task's data). However, this is a tedious and complex process. To address this issue, we want to use hierarchical tasks and the shared data mechanism to achieve some pruning automatically. In fact, since a hierarchical task encloses a part of the task graph, it can be submitted on every node, and only the nodes processing it will have to see the submission process of that part of the DAG.

Figure 6.2 showcases the automatic pruning mechanism on a very simple DAG. In this example, each node i owns a data A_i in a partition tree of shared data (see Figure 6.2a). The complete task graph is shown in Figure 6.2b and distributed between three nodes like in Figure 6.2c. Initially, they all submit a hierarchical task on the shared data A_{0+1+2} , which is processed into a regular task working on A_2 (thus executed on node 2) and another hierarchical task working on A_{0+1} . Since A_{0+1} is not shared with node 2, it does not complete the submission of the hierarchical task and never processes it, effectively pruning this sub-graph. The task working on both A_0 and A_1 can be assigned to the nodes 0 or 1 indifferently depending on the STARPU policy selected (here node 0). Communications between the two nodes are then required to execute that task.



(c) Task graph distributed between the nodes. Full tasks are executed on the corresponding node. Dotted tasks indicate that the node interrupted the submission of the task. Dashed arrows are inter-node dependencies (communications).

Figure 6.2: Example of automatic pruning in a distributed task graph.

6.3 Communications at runtime

In the standard model of STARPU-MPI, communications are planned at submission time. When the submission process of a node N inserts a task T to be executed on N , it first checks if N owns all the data handles used by the task. If not, communication tasks are inserted with a *recv* operation for each missing handle. Similarly, when other nodes try to submit T and finds that it will be executed by N , the owners of the data handle missing for N submit their own communication tasks with a *send* operation. After the task is submitted in N , the nodes create communications tasks

in a symmetric manner to return the data handles to their owners. A *send/recv* pair forms an inter-node dependency that is important for the correctness of the application.

When we introduce hierarchical tasks in STARPU-MPI, this mechanism is no longer restricted to submission time. In fact, when a hierarchical task is processed across multiple nodes at runtime, new communications might be necessary. This is the case in the example of Figure 6.2c. Since the node 1 owns A_1 , this data piece must be communicated to the node 0 to execute a task using A_0 and A_1 .

The addition of communications at runtime creates data consistency issues that are analogous to the ones encountered in the previous chapters of this thesis. They were addressed by extending the task data structure to include more information on the state of communications and on the place of the task in the hierarchy. Communications at runtime could be further enhanced with *hierarchical tasks of communications*. Such a hierarchical task would be submitted using the root of the partition tree of shared data. Depending on the hierarchical level of the task that triggered its insertion, it could *send* (respectively *recv*) the root handle, or be processed into a sub-graph of (potentially hierarchical) communication tasks. This could enable the use of shared data in computations, instead of their current role as a submission tool.

6.4 Conclusion

The most evident way to use hierarchical tasks in distributed memory consists of limiting their scope to individual nodes. However, it is tempting to submit hierarchical tasks across multiple nodes and distribute the resulting sub-graph between them. This is mostly hindered by the data mapping. Each data is owned by a single node, and each task is submitted on a single node based on the ownership of the data involved, making it impossible to process a single hierarchical task over multiple nodes. In this chapter, we gave some insight into the first efforts made to address this. We introduced the concept of *shared data*, a submission tool that share the higher levels of a partition tree between multiple nodes. A hierarchical task can therefore use these shared data and be submitted and processed by different nodes. In turn, this enables some automatic pruning, as a node will never process a hierarchical task whose data are not shared with it. Similarly to the previous chapters, data consistency is a complex issue to handle, in this case because of the communications added at runtime between the nodes processing the same hierarchical task. Our solution extended the task structure to store the information needed to handle runtime communication. The idea of hierarchical tasks of communications is also currently explored to enable computations on shared data.

Conclusion

THE GROWING COMPLEXITY of high performance computing platforms has led to the pursuit of highly efficient runtime systems, which aim to abstract the underlying architectures. These runtime systems use various programming models to express computations. One of the most widely used is the so-called Sequential Task Flow (STF) model, in which tasks are sequentially inserted to form a Directed Acyclic Graph (DAG) representing the application. This model offers a natural way for programmers to express a parallel workload. A runtime system, such as STARPU, is then left in charge of scheduling computations and managing data and communications. While this approach has been shown to be effective, the sequential submission process and the resources used by non-ready tasks can result in non-negligible overhead for certain applications. In addition, it is limited to constructing static task graphs: once a set of tasks has been inserted in the system, it cannot be optimized for the resources available at runtime. This is particularly problematic in heterogeneous systems where different types of computational units have conflicting characteristics.

To address these issues, this thesis proposed to extend the submission process of the STF model in the STARPU runtime system. Our new programming model introduced hierarchical tasks, which can insert a new sub-graph of tasks into an application's DAG at runtime. Hierarchical tasks offer a finer control of the submission flow of the application. By encapsulating parts of a DAG in hierarchical tasks, programmers can delay its submission and reduce runtime overhead. Furthermore, since hierarchical tasks are also tasks by nature, and thus executed by workers, the submission process is no longer handled by a single thread and can be parallelized by processing independent hierarchical tasks on different workers. Our model is also capable of producing task graphs that evolves dynamically at runtime. This is particularly useful to adapt the level of granularity or to select different implementations of an operation. However, this dynamic approach bypasses the sequential order of submission that the STF model rely on to correctly infer task dependencies. As a result, integrating hierarchical tasks without creating data or computational inconsistencies is quite complex.

Since the expression of a sub-graph equivalent to a given task requires partitioning the task's data to find parallelism, we must be able to modify the data layout when a hierarchical task is processed. STARPU generally handles this asynchronously, with partitioning tasks serving as synchronization points defining the region of the DAG where a given data layout is active. In the hierarchical task model, we decided to tie the task hierarchy with the data hierarchy. This enables us to enforce the correctness of the graph using only the necessary data synchronizations. A first implementation of the hierarchical task model sacrifices some features to rely on the existing data manager. During the submission process, it inserts partitioning tasks that enforce data consistency during the execution. In this approach, hierarchical tasks are processed by the submission thread as soon as possible, resulting in a recursive but static submission. A more advanced implementation extends the data manager to delay the insertion of partitioning tasks until runtime. Thanks to this new data manager, partition tasks are added as needed when a hierarchical task is processed. Once the sub-graph is inserted, the hierarchical task finishes and releases its outgoing dependencies. This enables fine-grain dependencies, as there is no barrier or synchronization between successive hierarchical tasks. Because the non-hierarchical successors use a different data layout than the sub-graph, unpartition tasks are added between the two, enforcing the correctness of the DAG. In both implementations, data management is transparent to programmers, who are only required to describe the different possible data layouts of their application.

Some preliminary work showed that the hierarchical task model can be used to prune distributed task graphs by encapsulating parts of the DAG in hierarchical tasks, which are then only processed by certain nodes. In order to submit and process hierarchical tasks across multiple nodes, we introduced the notion of “shared data”. This is a more expressive approach compared to simply restricting hierarchical tasks to individual nodes, but it requires to carefully manage the data communications added at runtime.

We validated our model using the dense linear algebra library CHAMELEON. We adapted the matrix descriptors of CHAMELEON to store tiles recursively. A recursive descriptor can therefore store a matrix with multiple levels of tiles. Extending a kernel to support hierarchical tasks is fairly straightforward, since the function describing the task graph of a tiled version of the operation is already available in CHAMELEON. Our experiments attempt to leverage hierarchical tasks, either by providing the right amount of fine grain tasks to CPUs or by refining tasks in the critical path of factorization operations. The results we obtained showed that, despite being limited by the lack of a dedicated scheduler, hierarchical tasks have the potential to improve the behavior of task-based runtime systems while providing more flexibility to the programmer.

Perspectives

Designing and implementing the hierarchical task model in STARPU opens up many perspectives. The most natural direction would be to enhance this mechanism by designing a scheduler that can take full advantage of its dynamic properties. While a specialized scheduler would already makes the programmer’s job easier, it might be possible to automate the use of hierarchical tasks even more with compiler techniques. Another direction for hierarchical tasks would be to look into the applicative side of things. In fact, many applications could benefit from hierarchical tasks beyond the dense linear algebra operations showcased in this thesis.

Scheduling hierarchical tasks

Considering scheduling matters is essential to achieve the full potential of the hierarchical task model. The design of a “hierarchically oriented” scheduler will be a considerable undertaking. Such a scheduler should be able to tell **when**, **where** and **how** a hierarchical task should be processed and therefore make runtime optimization to the construction of the DAG.

When to process a hierarchical task? Using the scheduler to measure the availability of resources and anticipate future needs, it would be possible to make informed decisions on whether to process a hierarchical task or not, at runtime. If the system is already crowded with non-ready tasks, it would be wiser to avoid processing hierarchical tasks, which would risk overloading the system even more. Conversely, if the sequential submission thread is bottlenecking the application, the issue could be solved by processing many hierarchical tasks in parallel.

How to process a hierarchical task? Once a hierarchical task starts being processed, the hierarchical task mechanism could give the scheduler an opportunity to decide which sub-graph to insert. Selecting the most appropriate implementation will require the design of advanced performance models. Instead of picking a particular implementation, which require some input from the programmer, another approach would be to target the appropriate degree of parallelism for the platform.

Where to process a hierarchical task? When a hierarchical task has been processed, the question of how to schedule the freshly inserted sub-graph arises. The most straightforward answer is to simply rely on existing strategies. However, thanks to hierarchical tasks, each sub-graph constitutes a region for which the scheduler has access to additional information based on the position the hierarchical tasks used to occupy in the DAG. Such information could be leveraged to improve the behavior of the application. For example, coarse graphs could be scheduled with a more costly algorithm before using a faster scheduler for big amounts of tasks lower in the hierarchy.

Overall, addressing these issues will improve the behavior of hierarchical tasks in the runtime system.

Distributed hierarchical tasks

The hierarchical task model for distributed architectures is still in a perfectible state and some edge cases must be ironed out. There is more work to be done around data communications. For example, implementing *hierarchical communication tasks* would enable the use of shared data in computational tasks, as they are simply a submission tool for the moment. The scaling and performance capabilities of the model must also be evaluated.

Toward a more automated approach

During this thesis, we focused a lot of efforts into making the hierarchical task model as natural to use as possible. However, it still requires to manually provide hints to the tasks we want to be hierarchical. This involves writing the STF code generating the sub-graph of the hierarchical task. For some applications, this process could be automated at compile time. An advanced compiler would be able to regroup parts of the task graph into hierarchical tasks. It could rely on its information on the targeted architecture to evaluate the levels of granularity and the amount of parallelism that should be expressed.

Exploring more applications

Finally, the use of hierarchical tasks for more irregular algorithms should definitely be explored. The dense linear algebra operation we covered in Chapter 5 present a very regular workload that does not take full advantage of the features of hierarchical tasks. On the other hand, algorithms with task graphs that evolve based on the results of the computations could be expressed more naturally with hierarchical tasks. For example, sparse matrix factorizations relying on partial pivoting have unpredictable data layouts due to reordering their matrices based on the numerical value of its elements throughout the operation. Hierarchical tasks could express such algorithms by adapting the sub-graph inserted at runtime to the data layout. Iterative solvers could also be expressed elegantly, with each iteration including a hierarchical task that will be processed into the graph of the next iteration, until a termination criterion is met. Low-rank approximations are another promising lead. This method compresses matrices into a collection of dense (full-rank) and low-rank blocks in a nested structure called a hierarchical matrix (\mathcal{H} -Matrix). The solvers using this method to reduce their memory and computation cost could implement hierarchical tasks to operate more naturally on the structure of \mathcal{H} -Matrices.

Acknowledgements

The experiments presented in this document were carried out using the PLAFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see <https://www.plafrim.fr>).

This work was supported by the french ANR through the SOLHARIS project, under the grant ANR-19-CE46-0009.

Bibliography

- [1] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta Mancho. OmpSs-2@Cluster: Distributed Memory Execution Of Nested OpenMP-Style Tasks. In *Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing*, pages 319–334, Berlin, Heidelberg, 2022. Springer-Verlag. ↑ p. 16
- [2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010. ↑ p. 74
- [3] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, 2017. ↑ p. 46
- [4] Emmanuel Agullo, Béranger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for Heterogeneous Architectures. *Concurr. Comput. Pract. Exp.*, 28(9):2608–2629, 2016. ↑ p. 28
- [5] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002. ↑ pp. 14, 15
- [6] David Álvarez, Kevin Sala, Marcos Maroñas, Aleix Roca, and Vincenç Beltran. Advanced Synchronization Techniques for Task-Based Runtime Systems. In *Proc. of PPOPP '21*, pages 334–347, 2021. ↑ p. 17
- [7] AMD. HIP. <https://github.com/ROCm-Developer-Tools/HIP>. ↑ p. 12
- [8] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L’Excellent, and Jacko Koster. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. ↑ p. 13
- [9] The OpenMP architecture review board. OpenMP Application Programming

-
- Interface. <https://www.openmp.org>, 2022. ↑ pp. 12, 17
- [10] Cédric Augonnet, David Goudin, Matthieu Kuhn, Xavier Lacoste, Raymond Namyst, and Pierre Ramet. A Hierarchical Fast Direct Solver for Distributed Memory Machines with Manycore Nodes. Technical report, CEA/DAM ; Total E&P ; Université de Bordeaux, Oct 2019. ↑ p. 19
 - [11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. ↑ p. 16
 - [12] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par*, pages 851–862, 2009. ↑ p. 16
 - [13] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing Dense and Banded Linear Algebra Libraries using SMPs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009. ↑ p. 16
 - [14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 66, 2012. ↑ p. 16
 - [15] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IEEE IPDPS Workshops and PhD Forum*, pages 1432–1441, 2011. ↑ p. 87
 - [16] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Héroult, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013. ↑ p. 16
 - [17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Héroult, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012. ↑ p. 16
 - [18] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '20, New York, NY, USA, 2020*. Association for Computing Machinery. ↑ p. 18
 - [19] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-

- André Wacrenier. Resource Aggregation for Task-based Cholesky Factorization on top of Modern Architectures. *Parallel Comput.*, 83:73–92, 2019. ↑ pp. 18, 47, 87
- [20] Michel Cosnard and Emmanuel Jeannot. Compact DAG Representation and its Dynamic Scheduling. *Journal of Parallel and Distributed Computing*, 58(3):487–514, 1999. ↑ p. 15
- [21] Michel Cosnard and Michel Loi. Automatic Task Graph Generation Techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122 vol.2, Jan 1995. ↑ pp. 15, 17
- [22] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: An Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30, 2014. ↑ p. 15
- [23] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A Proposal for Programming Heterogeneous Multi-Architectures. *Parallel Process. Lett.*, 21(2):173–193, 2011. ↑ p. 16
- [24] Hatem Elshazly, Francesc Lordan, Jorge Ejarque, and Rosa M. Badia. Accelerated Execution via Eager-release of Dependencies in Task-based Workflows. *The International Journal of High Performance Computing Applications*, 35(4):325–343, 2021. ↑ p. 18
- [25] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Task-Based Programming with OmpSs and its Application. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 601–612. Springer, 2014. ↑ p. 16
- [26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.0. <https://www.mpi-forum.org/docs>, 2020. ↑ p. 12
- [27] Cong Fu, Xiangmin Jiao, and Tao Yang. Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures. *IEEE Trans. Parallel Distributed Syst.*, 9(2):109–125, 1998. ↑ p. 46
- [28] Thierry Gautier, Fabien Le Mentec, Vincent Faucher, and Bruno Raffin. X-kaapi: A Multi Paradigm Runtime for Multicore Architectures. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 728–735, 2013. ↑ pp. 16, 18
- [29] Timothy D. R. Hartley, Erik Saule, and Çatalyürek Ümit V. Improving Performance of Adaptive Component-based Dataflow Middleware. *Parallel Computing*, 38(6-7):289–309, 2012. ↑ p. 16
- [30] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic

- Task Discovery in PaRSEC: A Data-Flow Task-Based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, New York, NY, USA, 2017. Association for Computing Machinery. ↑ p. 17
- [31] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021. ↑ p. 19
- [32] Pascal Hénon, Pierre Ramet, and Roman Jean. PaStiX: A High-performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. *Parallel Computing*, 28(2):301–321, 2002. ↑ p. 13
- [33] Laxmikant V. Kalé and Sanjeev Krishnan. Charm++: A Portable Concurrent Object Oriented System Based On C++. In *OOPSLA*, pages 91–108, 1993. ↑ p. 16
- [34] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *Proc. of HPEC'21*, pages 1–8, 2021. ↑ p. 19
- [35] David M. Kunzman and Laxmikant V. Kalé. Programming Heterogeneous Clusters with Accelerators using Object-based Programming. *Scientific Programming*, 19(1):47–62, 2011. ↑ p. 16
- [36] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979. ↑ p. 25
- [37] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO*, pages 45–55, 2009. ↑ p. 16
- [38] Marcos Maronas, Kevin Sala, Sergi Mateo, Eduard Ayguadé, and Vicenç Beltran. Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism. In *Proc. of HiPC'19*, pages 383–394, 2019. ↑ p. 18
- [39] NVIDIA. Cuda. <https://developer.nvidia.com/cuda-toolkit>, 2020. ↑ pp. 12, 16
- [40] Yu Pei, George Bosilca, and Jack Dongarra. Sequential Task Flow Runtime Model Improvements and Limitations. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–8, 2022. ↑ p. 17
- [41] Josep M. Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *Proc. of IPDPS'17*, pages 809–818, 2017. ↑ p. 19
- [42] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007. ↑ p. 16

- [43] Joseph Schuchart, Poornima Nookala, Thomas Herault, Edward F. Valeev, and George Bosilca. Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 117–128, 2022. ↑ p. 15
- [44] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016*, pages 318–327. IEEE Computer Society, 2016. ↑ pp. 25, 45
- [45] OmpSs-2 specification. Barcelona supercomputing center. <https://pm.bsc.es/ftp/ompss-2/doc/spec>, 2023. ↑ p. 16
- [46] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-core Architectures. In Fernando M. A. Silva, Inês de Castro Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, volume 8632 of *Lecture Notes in Computer Science*, pages 50–62. Springer, 2014. ↑ p. 83
- [47] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an Event-based Low-level Runtime for Distributed Memory Architectures. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 263–276, 2014. ↑ p. 16
- [48] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *Proc. of IPDPS'15*, pages 156–165, 2015. ↑ pp. 18, 19

Publications

Articles in peer-reviewed conferences

- [49] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. In *HeteroPar 2022 - twentieth international workshop*, page 12, Glasgow, United Kingdom, August 2022.

National conferences

- [50] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Samuel Thibault, and Pierre-André Wacrenier. Programmation des architectures hétérogènes à l’aide de tâches hiérarchiques. In *COMPAS 2022 - Conférence francophone d’informatique en Parallélisme, Architecture et Système*, Amiens, France, July 2022.

Journals

- [51] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. *Concurrency and Computation: Practice and Experience*, 2023.

Research reports

- [52] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. Research Report RR-9466, Inria Bordeaux Sud-Ouest, March 2022.

