Composability of parallel codes on heterogeneous architectures

Andra-Ecaterina Hugo

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHEMATIQUES ET INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

Par Andra Hugo

Composability of parallel codes on heterogeneous architectures

Sous la direction de : Raymond Namyst
(co-directeurs : Pierre-Andre Wacrenier et Abdou Guermouche)

Soutenue le 12 décembre 2014

Membres du jury :

M. Frederic Desprez	Directeur de Recherche INRIA	Rapporteur
M. Pierre Manneback	Professeur des Universités	Rapporteur
M. Patrick Amestoy	Professeur des Universités	Examinateur
M. Thomas Herault	Research Scientist	Examinateur
Mme. Isabelle Terrasse	Professeur des Universités	Examinateur
M. Raymond Namyst	Professeur des Universités	Directeur de Thèse
M. Pierre-Andre Wacrenier	Maître de conférence	Directeur de Thèse
M. Abdou Guermouche	Maître de conférence	Directeur de Thèse

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse, Raymond Namyst, Pierre-André Wacrenier et Abdou Guermouche de m'avoir guidée tout au long de ces trois années de thèse et également pendant les stages que j'ai effectués dans l'équipe Runtime et les cours que j'ai suivis à l'Université de Bordeaux. Au delà de la connaissance théorique, je voudrais les remercier de m'avoir appris à mieux structurer et présenter mes idées, à être plus rigoureuse dans mon travail scientifique et à mieux argumenter les solutions envisagées pour résoudre les problèmes que j'ai pu rencontrer tout au long de ces trois années de thèse.

J'adresse aussi mes remerciements aux membres de mon jury, Frédéric Desprez, Pierre Manneback, Patrick Amestoy, Thomas Herault et Isabelle Terrasse, pour avoir accepté et pris le temps de relire mon manuscrit et pour les questions pertinentes soulevées lors de la soutenance.

Un très grand merci à Marie-Christine pour ses conseils et sa patience lors des réunions assez bruyantes dans le bureau. Je souhaite ensuite remercier toute l'équipe Runtime de m'avoir accompagnée et soutenue au cours de cette thèse. Je pense particulièrement à Samuel et Nathalie qui ont répondu à pleins de questions sur StarPU et qui m'ont aidée lors des merges des branches de branches ..., de StarPU. Merci également à ma deuxième équipe Hiepacs qui m'a montrée le coté obscur de l'algèbre linéaire.

J'adresse de remerciements également à l'équipe pédagogique de l'IUT Bordeaux qui m'a accueillie pour que j'effectue mes heures d'enseignement. Je pense particulièrement à Pierre Ramet, Olivier Gauwin, Isabelle Dutour, Christine Uny, Michel Billaud. Un très grand merci à Jean-Michel Lépin pour ses conseils et son aide. Un deuxième merci à Pierre-André, Abdou et Samuel pour leur aide pendant mes six mois d'ATER à l'Université de Bordeaux.

Je pense également à Mehdi, Mathieu et Stole qui ont rendu cette période plus joyeuse. Merci pour les soirées films, pizza, bière ou même boite de nuit. Un grand merci à Julien pour m'avoir soutenue et supportée pendant le supplice de la rédaction.

Et enfin, les dernières mais non les moindres remerciements je les adresse à mes parents pour m'avoir guidée et soutenue à vouloir toujours apprendre des nouvelles choses et à essayer de devenir meilleure.

Titre: La composition des codes parallèles sur plate-formes hétérogènes

Résumé

Pour répondre aux besoins de précision et d'efficacité des simulations scientifiques, la communauté du Calcul Haute Performance augmente progressivement les demandes en terme de parallélisme, rajoutant ainsi un besoin croissant de réutiliser les bibliothèques parallèles optimisées pour les architectures complexes.

L'utilisation simultanée de plusieurs bibliothèques de calcul parallèle au sein d'une application soulève bien souvent des problèmes d'efficacité. En compétition pour l'obtention des ressources, les routines parallèles, pourtant optimisées, se gênent et l'on voit alors apparaître des phénomènes de surcharge, de contention ou de défaut de cache.

Dans cette thèse, nous présentons une technique de cloisonnement de flux de calculs qui permet de limiter les effets de telles interférences. Le cloisonnement est réalisé à l'aide de contextes d'exécution qui partitionnement les unités de calculs voire en partagent certaines. La répartition des ressources entre les contextes peut être modifiée dynamiquement afin d'optimiser le rendement de la machine. À cette fin, nous proposons l'utilisation de certaines métriques par un superviseur pour redistribuer automatiquement les ressources aux contextes. Nous décrivons l'intégration des contextes d'ordonnancement au support d'exécution pour machines hétérogènes StarPU et présentons des résultats d'expériences démontrant la pertinence de notre approche. Dans ce but, nous avons implémenté une extension du solveur direct creux qr_mumps dans la quelle nous avons fait appel à ces mécanismes d'allocation de ressources. A travers les contextes d'ordonnancement nous décrivons une nouvelle méthode de décomposition du problème basée sur un algorithme de "proportional mapping". Le superviseur permet de réadapter dynamiquement et automatiquement l'allocation des ressources au parallélisme irrégulier de l'application. L'utilisation des *contextes d'ordonnancement* et du *superviseur* a amélioré la localité et la performance globale du solveur.

Mots-clés : Composition, Hypervisor, Support d'exécution

Title: Composability of parallel codes on heterogeneous architectures

Abstract

To face the ever demanding requirements in term of accuracy and speed of scientific simulations, the High Performance community is constantly increasing the demands in term of parallelism, adding thus tremendous value to parallel libraries strongly optimized for highly complex architectures.

Enabling HPC applications to perform efficiently when invoking multiple parallel libraries simultaneously is a great challenge. Even if a uniform runtime system is used underneath, scheduling tasks or threads coming from different libraries over the same set of hardware resources introduces many issues, such as resource oversubscription, undesirable cache flushes or memory bus contention.

In this thesis, we present an extension of StarPU, a runtime system specifically designed for heterogeneous architectures, that allows multiple parallel codes to run concurrently with minimal interference. Such parallel codes run within scheduling contexts that provide confined execution environments which can be used to partition computing resources. Scheduling contexts can be dynamically resized to optimize the allocation of computing resources among concurrently running libraries. We introduced a hypervisor that automatically expands or shrinks contexts using feedback from the runtime system (e.g. resource utilization). We demonstrated the relevance of this approach by extending an existing generic sparse direct solver (qr_mumps) to use these mechanisms and introduced a new decomposition method based on proportional mapping that is used to build the scheduling contexts. In order to cope with the very irregular behavior of the application, the hypervisor manages dynamically the allocation of resources. By means of the *scheduling contexts* and the *hypervisor* we improved the locality and thus the overall performance of the solver.

Keywords: Composability, Hypervisor, Runtime

Unité de recherche
INRIA Bordeaux Sud-Ouest-200818243Z

Résumé

L'introduction des processeurs multicœurs et des accélérateurs au sein des plates-formes de calcul haute performance a suscité de nombreux travaux de recherche autour de la portabilité des performances. Ces travaux se sont focalisés autour de supports d'exécution capables de fournir aux programmeurs des techniques et des outils permettant d'exploiter des architectures matérielles toujours plus complexes. Désormais les programmeurs disposent de supports d'exécution suffisamment matures pour exploiter de telles architectures (tels Cilk [29], OpenMP ou Intel TBB [49] pour les multicoeurs, Anthill [55], DAGuE [19], Charm++ [38], Harmony [26], KAAPI [34], StarPU [14] ou StarSs [17] pour les configurations hétérogènes) et peuvent maintenant construire des bibliothèques de calcul performantes. Ainsi la bibliothèque d'algèbre linéaire MAGMA [59], reposant sur des algorithmes particulièrement optimisés, s'appuie sur le support d'exécution StarPU pour exploiter de façon efficace et portable des architectures complexes.

L'émergence de telles bibliothèques facilite la tâche du programmeur qui a tout intérêt à recycler celles-ci pour construire son application. Cependant on observe que ces bibliothèques se comportent mal, au sens des performances, lorsqu'elles sont utilisées simultanément. En fait leur utilisation en série fournit souvent de meilleures performances. Cette dégradation des performances a été identifiée et est appelée problème de la composition parallèle [46, 42]. C'est un problème important puisqu'il représente un frein aux respects des principes de bases de la programmation (modularité, réutilisation).

Ce problème de composition survient lorsque plusieurs supports d'exécution entrent en compétition pour l'obtention des ressources et même parfois plus simplement lorsqu'on fait des appels simultanés à des routines parallèles d'une même bibliothèque. Les routines parallèles, pourtant optimisées, se gênent et l'on voit alors apparaître des phénomènes de surcharge (l'application utilise plus de threads qu'il n'y a de cœurs à sa disposition), de contention sur les bus ou défaut de cache. En effet, à des fins d'optimisation, les programmeurs de bibliothèques de calcul parallèle ont pris l'habitude d'avoir la maîtrise du nombre et du placement des threads, de l'ordonnancement des tâches de calcul et de l'utilisation des caches. Cependant ces optimisations locales peuvent s'avérer contre-productives dans un cadre plus général notamment en présence de plusieurs flux de calcul parallèles. Ce type de problème a amené la communauté à promouvoir des supports d'exécution capable d'exécuter de nombreux flots de calcul sans pour autant surcharger de threads la machine [29, 49].

À l'image des machines virtuelles, il existe aussi des supports d'exécution, tel Lithe [46], qui permettent d'attribuer dynamiquement des ressources matérielles à des flux de calcul : en cloisonnant ces flux on espère ainsi limiter leurs interférences. MPC [23] propose une solution MPI + OpenMP qui permet d'isoler les sections parallèles. Pour éviter de payer le coût de la consommation mémoire d'une telle approche, MPC se base sur une gestion TLS (Thread Local Storage) qui privatise les variables globales de chaque tâches MPI en fournissant des copies. Intel TBB [49] propose une manipulation de haute niveau qui isole les codes dans des structures appelées *arènes*, qui permettent un équilibrage de charge à travers la migration de threads mais qui sont limitées en terme de parallélisme. ForestGOMP [20] est un support d'exécution qui suit le parallélisme hiérarchique de l'application et qui propose un ordonnancement basé sur la topologie des machines. Il reste cependant un problème majeur : celui de l'attribution (automatique) des ressources à chaque partition.

Dans cet thèse, nous présentons un support d'exécution où les différents flux de calcul parallèles s'exécutent dans des contextes d'ordonnancement séparés. Un contexte d'ordonnancement encap-

sule une instance d'un support d'exécution qui s'exécute sur un ensemble d'unités de calcul. La répartition des ressources entre les contextes peut être modifiée à la demande afin d'optimiser le rendement de la machine. À cette fin, nous proposons l'utilisation de certaines métriques (utilisation des ressources, progression du calcul) par un superviseur pour redistribuer automatiquement les ressources aux contextes. Nous avons implémenté cette approche en étendant le support d'exécution StarPU [14]. Les codes et bibliothèques de calcul développées au-dessus de StarPU peuvent ainsi tirer parti de cette fonctionnalité sans modification de code. Ceci nous permet de montrer expérimentalement l'intérêt de notre approche en l'appliquant à des exemples concrets.

Des contextes d'ordonnancement au sein de StarPU

S'attaquer au problème de la composition au niveau du support d'exécution est une approche qui assure la portabilité et la réutilisation de la solution. Ainsi, on fournit un outil qui a un accès direct à l'information de bas niveau sur l'architecture de la machine et qui permet au programmeur de contrôler l'allocation des ressources. Cependant, les supports d'exécution qui reposent sur une paradigme à base de tâches sont de plus en plus utilisés. Ils permettent au programmeur de fournir beaucoup d'informations (la charge de travaille, le parallélisme des noyaux) ainsi que de contrôler le flux d'exécution. Par conséquent on a choisi d'étudier le problème de la composition au sein du support exécutif StarPU. StarPU [14] est une bibliothèque qui propose aux programmeurs une interface portable pour ordonnancer des graphes de tâches dynamiques sur un ensemble hétérogène d'unités de calcul (CPUs et GPUs). Toute tâche est associée à une voire plusieurs implémentations afin de pouvoir être indifféremment exécutée sur un CPU ou un GPU, par exemple. De plus, StarPU utilise une mémoire virtuellement partagée automatisant la disponibilité et la cohérence des données.

Nous introduisons la notion de *contexte d'ordonnancement* au niveau de StarPU, afin de fournir au programmeur un outil capable de gérer l'allocation des ressources hétérogènes à plusieurs codes parallèles. Il s'agit de cloisonner les contextes en faisant en sorte que chaque contexte détienne (ou partage) des unités de calcul. Ainsi tout thread de l'application peut s'inscrire à un contexte pour soumettre ses tâches qui seront alors prises en charge par un ordonnanceur associé au contexte. Au niveau technique, l'intégration des contextes au sein de StarPU a été mise en œuvre à l'aide d'un dispositif permettant d'ajouter ou de supprimer dynamiquement une file d'ordonnancement à une instance d'ordonnanceur et ce de façon asynchrone. Ainsi alertés, les ordonnanceurs réagissent à toute nouvelle distribution des ressources permettant ainsi le redimensionnement dynamique des contextes. Cette capacité est importante puisque les besoins des différents flux de calcul évoluent au cours du déroulement de l'application.

Nous avons proposé deux scénarios pour évaluer les contextes d'ordonnancement: composer plusieurs noyaux parallèles appartenant à la librairie d'algébre linéaire dense MKL sur une plate-forme homogènes (40 CPUs) et composer plusieurs noyaux parallèles appartenant à la librairie d'algébre linéaire dense MAGMA-MORSE sur une plate-forme hétérogènes (12 CPUs et 3 GPUs). Dans les deux cas nous avons montré qu'à travers les contextes d'ordonnancement on arrive à isoler les noyaux parallèles et ainsi améliorer leur localité et leur scalabilité. Le programmeur a donc le contrôle sur l'allocation des ressources. Néanmoins, décider combien de CPUs et de GPUs allouer à un certain code parallèle à et quand devient une tâche compliquée même pour un niveau d'expertise élevé.

Superviser pour redistribuer

Afin d'automatiser la répartition des ressources nous avons mis au point un système de supervision de l'activité des unités de calcul et de progression des contextes. Ce dispositif est mis en œuvre par un *hyperviseur* exécuté de façon distribuée par les CPUs. Le rôle de l'hyperviseur est de redimensionner les contextes d'ordonnancement lorsque des dégradations de performance sont observées. L'hyperviseur peut être invoqué depuis l'application (directement ou indirectement lors de création / destruction / modification de contexte) ou peut être déclenché automatiquement.

Lors de l'exécution, la progression du déroulement de l'application est évaluée au travers d'indications fournies soit directement par l'application, soit par les compteurs de performance ou encore par la routine d'estimation de la date de terminaison. Au besoin, une redistribution des ressources est appliquée. De plus cette redistribution peut être contrainte par le programmeur, les contraintes étant explicitées sous forme d'intervalle d'unités de calcul à respecter.

Dans ce cadre, nous avons étudié deux métriques pour piloter le redimensionnement des contextes. La première est basée sur un compteur bas-niveau mesurant l'inactivité des ressources et la seconde est basée sur la vitesse instantanée d'un contexte et sur le nombre d'opérations qu'il reste à exécuter. Le choix entre les deux est déterminé par les informations fournies par le programmeur.

Dans la stratégie *Idleness-base resizing*, les contextes sont redimensionnés lorsqu'une des ressources est inactive pendant une période plus longue que celle spécifiée par le programmeur. Expérimentalement on s'est aperçu que ce seuil peut être dépendent du parallélisme de l'application.

Dans la stratégie *Speed-based resizing* l'application procure une estimation de la quantité totale de travail (le nombre de flops) correspondante à chaque noyau parallèle et à chaque tâche. Avec ces indications, l'hyperviseur calcule la vitesse instantanée de chaque noyau parallèle ce qui permet d'estimer la date de terminaison de chaque noyau. Lorsque la différence de vitesse entre les contextes est suffisamment grande, l'hyperviseur redimensionne les contextes.

L'hyperviseur utilise plusieurs stratégies afin d'optimiser l'exécution de l'application. Ainsi, celles-ci sont basées sur diffèrents crières: minimiser le temps de terminaison des flux en cours ou maximiser leurs vitesse instantanée. Pour ce faire on utilise des programmes linéaires qui formalisent ces problèmes à travers des systèmes d'équations. Ils prennent en entrée des informations comme la quantité de travail (le nombre d'opérations flottantes ou le nombre de tâches de chaque type) que doit réaliser chaque contexte. Ils fournissent le nombre de CPUs et GPUs nécessaire à chaque contexte de façon à minimiser la date de terminaison globale ou maximiser la vitesse de chaque contexte.

Dans un premier temps nous avons évalué le comportement de l'hyperviseur dans le cadre d'un scénario simple: nous avons composé deux noyaux parallèles appartenant à la librairie d'algèbre linéaire dense MAGMA-MORSE sur une architecture hétérogènes et nous avons fourni une allocation statique inefficace. Dans ce contexte nous avons analysé la réactivité et l'efficacité de l'hyperviseur à remettre en cause l'allocation initiale et à la régler en fonction des besoins de l'application. Nous avons comparer différentes stratégies de redimentionnement en terme de précision et surcoût.

Allocation dynamique des ressources pour un solveur des systèmes linéaires creux

Afin de valider notre approche on s'est attaqué à une application compliquée, plus concrètement à qr_mumps, un solveur de systèmes linéaires creux. L'objectif principal a été d'évaluer une façon différente de gérer l'allocation des ressources et l'affinité mémoire au sein des applications HPC à

travers des mécanismes placés au niveau du support d'exécution.

Le solveur qr_mumps fourni une méthode de factorisation QR creuse basée sur une approche multifrontale. L'algorithme multifrontal considère un arbre d'élimination [51], qui représente la réduction transitive du graphe de la matrice remplie et qui décrit les dépendances entre les opérations d'élimination. Le graphe de tâches est donc construit pendant la phase d'analyse quand les algorithmes de pré-traitement sont appliqués. L'algorithme continue avec la factorisation numérique puis par les pas de résolution du système.

L'approche a été de capturer la structure parallèle de l'application de façon hiérarchique et de la projeter sur un arbre abstrait. Cette arbre est ensuite décrit à travers une hiérarchie de contextes d'ordonnancement qui va permettre une gestion dynamique de l'allocation de ressources.

L'hyperviseur intervient pour collecter des informations venant de l'application ainsi que des statistiques matérielles capturées au niveau du support d'exécution. Les stratégies de dimmentionement prennent en compte ces valeurs affin de régler le nombre d'unités de calcule assigné à chaque tâche parallèle.

On évalue l'efficacité de cette approche à travers une collection des problèmes différents venant de l'Université de Florida. On effectue des expériences sur des architectures multicœurs modernes avec des accès mémoire non uniformes. On observe a travers l'utilisation des contextes un meilleur moyen de respecter la localité des données entre les tâches. De plus la hiérarchie de contextes favorise le chemin critique. La mise en œuvre de cette solution permet d'obtenir des gains en terme de temps d'exécution atteignant 35%.

Conclusions

Dans cette thèse nous avons étudié le problème de la composition parallèle à l'aide de StarPU. Nous avons introduit la notion de contexte d'ordonnancement pour donner au programmeur la maîtrise des ressources attribuées à différents flux de calcul: les contextes peuvent être redimensionnés à volonté et les unités de calcul peuvent être partagées ou non. Nous proposons l'utilisation de métriques par un hyperviseur pour redistribuer automatiquement les ressources aux contextes. De plus, afin d'assurer la portabilité des performances, nous proposons d'utiliser des programmes linéaires dont la solution nous permet d'obtenir une bonne distribution correspondante aux indications de charge transmisses par l'application. Enfin, nous avons montré expérimentalement l'intérêt de cette approche à travers qr_mumps, un solveur des systèmes linéaires creux. Nous avons ègalement montré l'apport d'un système de supervision pour redistribuer automatiquement les ressources aux contextes.

Ces travaux ouvrent de nombreuses perspectives de recherche. En terme d'amélioration, nous envisageons d'augmenter la précision et la réactivité de la détection de sous-utilisation des ressources à travers des conteurs matérielles. Cette approche peut également contribuer à une deuxième perspective, qui est d'améliorer la négociation des ressources parmi des codes qui ne reposent pas sur StarPU. Néanmoins, nos idées se dirigent vers des nouveaux stratégies de redimensionnement qui suivent des algorithmes d'ordonnancement de la littérature ou qui prennent en compte des nouveaux métriques comme la consommation d'énergie.

Les perspectives de plus long-termes se place au niveau de la valorisation de ces travaux dans le cadre des applications complexes de couplage de codes. Toutefois, les techniques présentées doivent suivre les évolutions des machines parallèles. Par conséquent, nous envisageons d'intégrer au niveau de l'Hyperviseur des nouveaux algorithmes pour mieux partitionner les architectures many-core et ainsi mieux passer à l'échelle.

Contents

List of Figures

List of Tables

Introduction

High Performance Computing community is nowadays investing considerable effort in continuously exploiting the parallelism of their applications. Fluid-structure interaction, magneto-hydro dynamics, thermal coupling, define only a subset of recent multi-physics code coupling simulations that require extreme-scale computing in order to accelerate their execution. In order to satisfy their demands hardware manufacturers keep on designing even more complex computer architectures. Yet, very few people have the expertize to program such architectures efficiently, adding thus a tremendous value to parallel libraries strongly optimized to use such architectures. Indeed, there is a common feeling in the community that reusing existing parallel libraries is indispensable.

Consequently, building high performance computing applications on top of parallel libraries is now commonplace. However, even if a natural approach would be to rely on as many external parallel libraries as needed and allow their concurrent execution, most applications invoke only one parallel library at a time. The reason lies in current implementations of parallel libraries not being ready to run simultaneously over the same hardware resources. Usually each one of them is making the assumption that their procedures have exclusive access to the computing resources. As a consequence, in order to fully tap into the potential of many-core architectures, parallel libraries typically invoke different optimizations in order to bypass the underlying operating system and better use the underlying resources. For instance, parallel libraries typically fix one thread per core in order to have a better utilization of the cache memory. However, when co-executing several parallel procedures, each one has its own set of threads and this optimization does not longer apply. As a result, applications resulting from the composition of parallel libraries usually exhibit poor performance.

The only safe solution is then to serially or sequentially use external parallel libraries. However, this situation is alarming because important software development concepts like modular and abstract structured programming are no longer used in HPC. Thus, parallel libraries do not attain their initial purpose: hide the complexity of designing an efficient and reusable collection of parallel algorithms.

Systematically used from the earliest days of programming in sequential applications, *software reuse* should also be possible in HPC. We think one of the main challenges for upcoming years is to make this possible by arbitrating the use of the computing resources between the co-existent parallel libraries. This can only be achieved by the negotiation of resources. A first step is to explore each party's requirements and workload. Then the goal is to seek "win-win" or mutually beneficial resource distributions.

Goals and Contribution of this thesis

The batch-job schedulers of cluster computing platforms have already dealt with the problem of nodes allocation for simultaneously executing applications. However, this issue is hardly addressed when executing parallel codes within the same application on the same machine. Every day more heterogeneous, with more cores and accelerators, these machines are hardly used at their full capacity. Code-coupling application would surely benefit of efficiently executing tightly coupled parts of their computation at the intra-node level as well. However, inside the node the negotiation of resources cannot only consider the load of the parallel codes but also their affinity towards the different types of computing resource. Indeed, programmers cannot deal with this problem alone. They need a low level tool that can provide an abstraction of the heterogeneous architecture of the machine and at the same time high level mechanisms to integrate user level information. Runtime systems represent thus the most appropriate level of the software stack for implementing such a tool as it can allow a strong interaction with both the hardware and the application. The contributions of this thesis cover the different aspects of the composability problem managed at the runtime level. More particularly, we identify the following main topics:

- **Isolation of parallel libraries**. We propose using the notion of *Scheduling Contexts* in order to restrict the execution of the parallel libraries on a section of the machine.
- **Supervision of the execution of the application**. We propose a tool called the *Hypervisor* that monitors the execution of the parallel kernels and collects both low-level (for instance the efficient execution of the computing resources) and high-level information (application hints).
- **Dynamic allocation of computing resources**. The hypervisor integrates different algorithms based on linear systems that provide solutions to dynamically reallocate computing resources between the parallel libraries.
- **Validation of the approach**. We validate our approach on simple case scenarios as well as on complex High Performance libraries like `qr_mumps` sparse direct solver.

All the contributions described in this thesis have been implemented and experimented in the StarPU runtime system. Although the concept remains independent of the StarPU implementation and can easily be adapted to other runtime systems. Most of the contributions have been the subject of several refereed publications which are listed at the end of this document.

Organization of this document

Chapter 1 presents a short overview of the High Performance Computing evolution both in term of hardware and programming environments leading thus to the need to *reuse and compose parallel codes*. Chapter 2 introduces the notion of *Scheduling Contexts* as a solution to the isolation of parallel libraries. Chapter 3 describes the *Hypervisor* as a StarPU plugin able to provide the necessary tools to dynamically redistribute computing resources to parallel libraries. Chapter 4 shows that the concept of dynamic co-existence of parallel codes allows improving the performance of `qr_mumps`, a sparse direct solver. We finally conclude and describe our perspectives in Chapter 5.

Chapter 1

Keeping up with the evolution of computer architectures

Computer architectures are constantly evolving in order to satisfy the every day more demanding large scale applications. Both industrial and research areas require a tremendous amount of computing power, memory and storage. Therefore, the computer manufacturers constantly try to add more computation resources, more memory, providing every day more challenges to the programmers using them.

1.1 From multi-cores to many-cores

The first attempts to introduce parallelism inside an application have been made at the instruction level of the processing unit, where the computer architects increased the clock frequencies by allowing the simultaneous execution of multiple instructions (Instruction Level Parallelism - ILP). When these techniques have reached the limits of power consumption and heat dissipation the manufacturers started focusing on adding more processing units (Thread Level Parallelism - TLP) to the computation platform.

With platforms having several processing units connected between them, applications started relying on an important increase in parallelism. However, the symmetric access to a common memory (Symmetric multiprocesseurs - SMP) generated an important contention on the bus memory. Cache hierarchies provided a good solution to limit the bus access but despite this, computing platform with several dozens of processing units could not scale on such an architecture.

NUMA (Non Uniform Memory Access) architectures became then more popular as they allow having different memory banks distributed all over the machine. Processing units grouped around a memory bank forming a NUMA node have a fast access to their data, however accessing a distant NUMA memory nodes is more costly. Thus, despite all the effort, the contention on the bus memory is still a problem unless the application has a solution for the locality management.

Therefore, the evolution of computers is now driven by a run towards higher and higher numbers of cores per chip. This trend was largely anticipated by Graphical Processing Units (GPUs). Originally designed for processing images, GPUs are now highly specialized computing devices meant to handle a particular class of applications with well defined characteristics: large computational requirements and substantial, fine-grained parallelism. Application Programming Interfaces (API) for GPUs such as CUDA or OpenCL have been rapidly evolving in the last few years: GPU pro-

grams can now be written in familiar programming languages (such as C or Fortran) according to a Single Program Multiple Data (SPMD) parallel programming model which opened the way for the collective effort that is commonly known under the name of *General Purpose GPU* (GPGPU) computing [45]. However, applications programmers find it difficult to achieve considerable performance when using it. Not only the programming paradigm is different compared to the CPUs one but also the memory coherence management becomes less straightforward when using it together with other types of processing units. Despite this, hybrid computing becomes a solid trend as there is an increasing need to exploit any computing power.

1.2 Programming environments in HPC

Complex applications running on such platforms require important programming efforts in order to efficiently make profit of the heterogeneity of the processors, the frequency of their execution, the hierarchical memory access and the memory capacity.

Most of the times we can consider parallel applications as graphs with nodes representing the operations to be performed and edges representing ordering constraints imposed by dependencies. Programming these applications on large parallel platform consists in mapping the graphs on the machine. This task becomes less straightforward for a high connectivity of the graph and for a highly non uniform architecture. Runtime systems are considered to be a solution that simplifies the interaction of the application with the hardware. By using a runtime system, programmers target obtaining for their applications what we call *performance portability*, that is finding a trade-off between performance improvement and compatibility with different architectures.

Runtime systems represent a user level software that complements the basic, general purpose functions provided by the operating system calls. Applications delegate them all the parallelism management (e.g. scheduling, task dependencies, etc.) and the runtime systems optimize the use of the underlying hardware resources. For instance, the hybrid use of general purpose cores together with specialized accelerating ones (e.g. GPU) requires solving a difficult mapping problem, in order to decide which computation could efficiently execute on which type of core. Moreover, as GPUs have their own local memory, performing operations on such resources requires transferring data from main memory to the GPU's memory, potentially overlapping communication with computation and obviously considering the transfer time when mapping an operation on the GPU. The runtime systems usually take in charge all this and make this operations as transparent as possible for the application.

According to [61] the runtime systems can be split in several approaches to building and accessing these graphs. We show thus in Table 1.1 a set of runtime systems grouped according to this criteria (inspired by the table presented in [61]).

In implicit graph representations systems, runtime systems do not have access to the graph of operations of the application and therefore they rely on the programmer to indicate the order of the operations, to launch parallel work and manage the synchronizations. For instance MPI and X10 are systems in which the programmer manages when, how and in which order the operations of the application should execute. On the other hand, in the explicit graph representations systems, runtime systems have direct access to the graph and thus manage the synchronizations and scheduling.

Some explicit systems are static, that is the graph of operations is built at compile time. Some others are dynamic, that is they discover and manage the graph at runtime as it is generated on-line

Implicit Graph Representation Systems
MPI [52] OpenMP [13] TBB [49] X10 [54] Charm++ [38] UPC [22] CUDA [1] OpenCL [53] MPC [23] ForestGOMP [20]
Explicit Graph Representation Systems
Sequoia [27] PaRSEC [19] StarPU [14] XKaapi [31] Legion [18]/Realm [61] OmpSs [17]

Table 1.1: Categorization of parallel libraries

by the application. Static systems have the advantage of not inducing any runtime overhead to the scheduling of the graph. However dynamic runtime systems provide a better flexibility to adapt to more complex architectures as well as to applications based on irregular parallelism.

Therefore, on a general basis, implicit graph representation systems are based on a control flow approach, while the explicit graph representation systems are based on a data flow approach. However, even if there are significant differences between the two approaches, some applications may require using both of them.

1.2.1 Implicit representations systems

Further on we discuss about a few of the runtime systems in table 1.1 mentioned in the implicit representation systems category. We choose one of each of the main data communication paradigms: shared memory, message passing, partition global address space (PGAS).

OpenMP [13] standard relies on a *shared memory paradigm* and it was first defined in 1997 by a consortium formed of universities, hardware and software vendors. It provides a set of annotations used to parallelize sequential programs written in C/C++ or Fortran. Thus, programmers can automatically generate the creation of threads, their synchronization as well as their termination. Generally used for loops, OpenMP automatically creates the number of threads equal to the number of CPUs and assigns an equivalent number of indexes of the loop to each thread. GOMP or Intel OpenMP are some of the most common implementations of the standard.

MPI [52] standard relies on a *message passing paradigm* and it was first proposed in 1993 by a consortium formed of universities, hardware and software vendors. It provides an interface that allows portable network communication. The vendors provide the network adapted implementation such that the applications are not aware of the underlying hardware. Some of the most used high Performance implementations are OpenMPI [30] and MPICH [33].

MPI is based on the idea of having a sender and a receiver that constantly change of roles between them. It relies on a SPMD programming model (Single Program Multiple Data), having thus a single program executed by several processes that exchange messages during the execution

whenever synchronizations or distant data is necessary. This feature revealed to be useful for intra-node parallelism too, as it induces a good isolation of the programs without the synchronization issues of shared memory systems. Despite this, one important disadvantage leaded to having the programmers think of different solutions for intra-node implementations. Each process has its own virtual address space, requiring multi-processes systems to allocate the necessary memory for each process, and having potentially several copies of the same data whenever several processes required it. As the number of cores increased the memory consumption became a serious problem and programmers had to start using a different paradigm inside a node.

X10 [54] is a parallel programming language relying on a *PGAS (Partition Global Address Space) paradigm*. Implemented by IBM and released for the first time in 2004, X10 is a language focusing on concurrency and distribution, while making profit of an object oriented programming tradition.

X10 uses the notion of Place to encapsulate binding of activities and globally addressable data. The management of synchronizations is done by means of a system of hierarchies between the activities. Nevertheless, the programmer is fully in charge with calling functions for asynchronous execution or synchronization. By means of atomic sections, X10 enforces mutual exclusion of shared data.

1.2.2 Explicit representation systems

Static or dynamic, explicit representation systems rely on what we call *task-based paradigm*. Increasingly adopted recently, it provides an easy way to express concurrency and dependencies. Thus, a task represents a piece of computation that executes a certain kernel on a predefined set of data. Usually organized into a DAG (Direct Acyclic Graph) the tasks are used by the applications to describe the operations to be executed. We say applications use a data flow approach because the dependencies between different pieces of computation dictate the synchronization and the order of their execution. This is a portable approach that responds to the requirements of hybrid architectures because, provided an adapted implementation of the kernel, the task may execute on a CPU or on an accelerator. An efficient execution of such a DAG corresponds to finding a suitable mapping of the tasks on different processing units [15].

PaRSEC [19] previously called DAGuE, is a runtime system implemented at ICL (Innovative Computing Laboratory) at the University of Tennessee and it was first released in 2010. It is a distributed DAG scheduling engine, where tasks are sequential computations and edges are communications. Each process has its own instance of the scheduler, and all the communications are implicitly made by the runtime. Therefore, a PaRSEC user has to provide the DAG of tasks together with the data distribution.

PaRSEC uses a static description of the graph, by expressing all the task dependencies before starting the execution. This insures a limited overhead of the scheduling. However, the graph can be queried dynamically in a distributed fashion. PaRSEC provides a dynamic and fully distributed scheduler based on cache awareness, data locality and task priority.

StarPU [14] is a C library developed in the Runtime team at Inria Bordeaux and it was first released in 2009. It provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units (*i.e.* CPUs and GPUs). The two basic principles of StarPU are firstly that tasks can have several implementations, for some or each of the

various heterogeneous processing units available in the machine, and secondly that necessary data transfers to these processing units are handled transparently by the runtime system. StarPU tasks are defined as multi-version kernels, gathering the different implementations available for CPUs and GPUs, associated to a set of input/output data. To avoid unnecessary data transfers, StarPU allows multiple copies of the same registered data to reside at the same time on several processing units as long as it is not modified. Asynchronous data prefetching is also used to hide memory latencies.

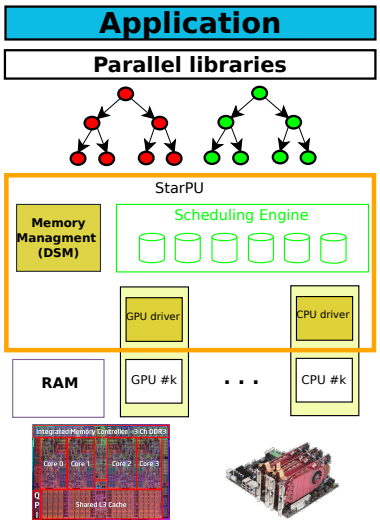


Figure 1.1: The architecture of the StarPU runtime system.

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way (see Figure 1.2). Implementing a scheduler consists in creating a set of queues, associating them with the different processing units, and defining the code that will be triggered each time a new task gets ready to be executed, or each time a processing unit is about to go idle. Various designs can be used to implement queues (e.g. FIFOs or stacks), and they can be organized according to different topologies. Several built-in schedulers are available, ranging from greedy and work-stealing based policies to more elaborate schedulers implementing variants of the Minimum Completion Time (MCT) policy [60]. This latter family of schedulers builds on auto-tuned history-based performance models that provide estimations of the expected durations of tasks and data transfers.

These models are actually performance history tables dynamically built during the application run. By observing the execution of the tasks, the runtime is able to capture the speedup and the affinities between the tasks and processors. Therefore, without the programmer's involvement, the runtime can provide a relatively accurate performance estimation of the expected requirements of the tasks allowing the scheduler to take appropriate decisions when assigning the tasks to a computing resource.

Similarly to other dynamic explicit graph representation systems, StarPU discovers the task

dependencies dynamically showing thus more flexibility but more overhead of the scheduling too. It provides a natural way to estimate the workload of an application, by keeping a history-based reference of the execution time of the tasks, the memory bandwidth, the data transfer costs, etc.

1.3 Dealing with the composability of parallel libraries

Many years of research conducted to important breakthroughs for many parallel libraries (linear algebra, physics, etc.). By relying on available mature implementations of runtime systems (e.g. Cilk [2], OpenMP or Intel TBB [49] for multicore machines, Parsec [19], Charm++ [38], KAAPI [34], StarPU [14] or OmpSs [17] for heterogeneous configurations) programmers are able to rely on thread/task facilities to develop efficient implementations of parallel libraries, like for example: Intel MKL [25], FFTW [28], FMM [11], etc. They are now able to provide parallel procedures optimized to execute extremely efficiently by considering memory and scalability bottlenecks of the parallelism.

With both hardware and software users continually striving to reach performance, application programmers cannot afford spending any more effort in reimplementing already optimized procedures. By relying on existing parallel libraries, they not only factorize code and reuse existing implementations but they also delegate performance portability concerns. The widely used BLAS (Basic Linear Algebra Subroutines) procedures are one example of basic computations, that high-level programmers usually do not consider reimplementing.

Building high performance computing applications on top of specific parallel libraries is now commonplace [32]. However, even if a natural approach would be to rely on as many external parallel libraries as needed and allow their concurrent execution, most applications invoke only one parallel library at a time. The reason lies in current implementations of parallel libraries not being ready to run simultaneously over the same hardware resources. This problem, referred to as the parallel *composability* problem [46, 42] has already been raised by the programmers of different applications.

Intel MKL for instance is known to be one of the libraries that promotes calling the parallel kernels sequentially instead of simultaneously. Despite this they propose a solution for experienced low level programmers, they advise making profit of the fact that the parallel library is well integrated with the Intel OpenMP runtime system, and reuse thus its abstract view of the machine [3].

This situation is actually alarming, because it reveals that well-known programming principles such as code *composability* and code *reusability* are currently not applicable to High Performance Computing.

There is a wide panel of applications that face this problem, ranging from code-coupling applications (e.g. molecular dynamics coupled with finite elements methods), where opportunities for executing concurrent parallel kernels are still under-exploited, to linear algebra libraries, and more precisely sparse linear algebra methods and fast multipole methods. Typically, numerical factorizations of sparse matrices involve the execution of various dense linear algebra kernels. Some of these kernels operate on small and medium blocks, and thus exhibit poor scalability on high numbers of cores. In such situations, running several kernels concurrently to preserve good scalability of each instance may greatly help to improve overall performance.

1.3.1 Oversubscription/undersubscription problem

Calling simultaneously several parallel procedures is a difficult matter because usually parallel libraries make the assumption that each procedure has exclusive access to the architecture of the machine. The main reason is that in the past years, computer architectures were not large enough (in term of processing units or memory) to host several parallel codes. Therefore, in order to fully tap into the potential of *many-core* architectures, parallel libraries typically allocate and bind one thread per core to bypass the underlying operating system's scheduler. Specialized parallel libraries, such as BLAS for instance, strictly follow such a rigid approach, to better control cache utilization. As a result, applications resulting from the composition of parallel libraries usually exhibit poor performance, because each library is unaware of other libraries' resource utilization and they run into the *oversubscription* problem. Indeed, even composing parallel codes belonging

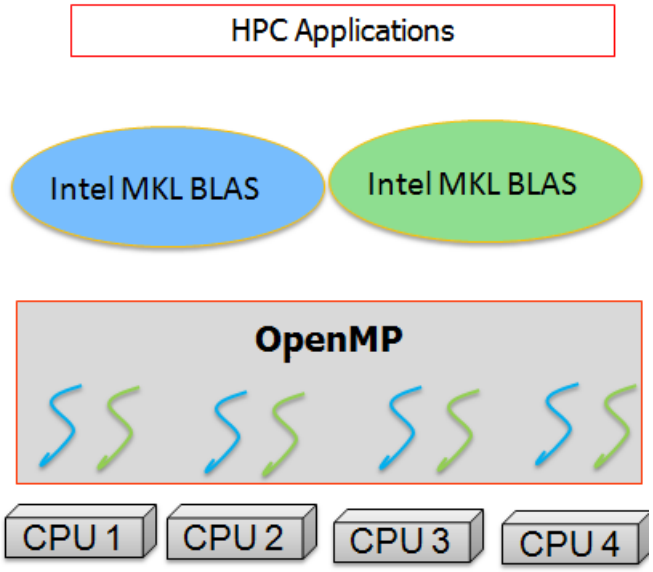


Figure 1.2: HPC application calling 2 BLAS procedures simultaneously

to the same library or relying on the same runtime exhibit severe performance issues when trying to simultaneously run independent parallel blocks within the same application. Determining the side-by-side or the nested parallelism and impose a certain allocation of resources is not an easy matter. Intel MKL, for instance, allows the application to specify the resources to allocate to each parallel code by using the OpenMP runtime to control them. However, the specificity of each parallel code (in terms of parallelism, granularity or memory access) requires additional knowledge in order to allocate the good number of processing units at the right moment of the execution.

However, for certain systems the oversubscription is necessary in order to perform well. The authors of [36] evaluate the task oversubscription on the performance of MPI, OpenMP and UPC and show that it can improve the execution time of certain applications running in competitive environments. Indeed, applications with less memory access and implicitly less synchronization requirements are less vulnerable in systems competing for resources. On the other side, systems

with significant parallelism and data dependencies are likely to benefit from the oversubscription to improve the load balancing and CPU utilization.

Nevertheless, oversubscription has been historically used to cover the latency, and complex parallel machines disposing of a hierarchical memory level structure could use it to overcome the cost of distant memory access. Despite this, we have to bear in mind that such a solution on many-cores machine is hardly scalable. Assigning tasks to each processing unit of the many-core architectures is a difficult matter. Applications partitioning the workload at a fine-grained level increase the parallelism by generating an important number of operations, but also impose an overhead on the execution time of the application due to the management of all these operations. Instead, the coarse-grained partitioning of the workload represents a burden to the parallelism but avoids the overhead. However, finding the best trade-off between fine and coarse grained partitioning seems to vary with the type of parallel code and the type of architecture.

This issue has led a few runtime system designers to provide implementations able to avoid *resource oversubscription* when multiple libraries simultaneously request the scheduling of tasks/threads. Static isolation of parallel codes may however lead to the undersubscription problem, where some libraries may have more resources assigned than they can use.

1.3.2 Parallel environments handling the oversubscription/undersubscription

In the following section we analyze different environments that provide a solution to the oversubscription and sometimes to the undersubscription problem, we then discuss their benefits and their limitations.

1.3.2.1 Intel TBB and Cilk

Intel TBB [49] is a runtime system that throughout the notion of *arena* allows for a pool of workers to execute different work stealing algorithms. However, when executing simultaneously different parallel codes in complex applications, the work stealing algorithms are not efficient any more. Low workload parallel codes expected to finish executing right away are burdened by high workload parallel codes. For this reason in the 3.0 version, Intel TBB allowed having several *arenas* used to isolate parallel codes and workers to migrate between arenas in order to respect the load balancing.

However, the degree of parallelism of *arenas* is limited, because the programmer has to indicate the maximum number of threads that can join the arena, therefore even if thread migration is allowed, no more than a certain number of threads can do this. This finally interferes with the distribution of resources since it can allow certain kernels to progress faster than others. Moreover, NUMA aspects of fair location are hardly respected as threads can migrate from one NUMA node to another.

Well integrated with the TBB runtime, Cilk [2] also tackled this problem by providing several parallel patterns (map, reduce, fork-join, etc.) that can be composed in an application. The solution for the composability of these codes was based on the philosophy of separating the notion of mandatory parallelism (tell the system what must run in parallel) and optional parallelism (tell the system what can run in parallel). Thus, the programmer decides on the type of parallelism, and the runtime takes advantage of this flexibility to balance the load across threads. However, Intel OpenMP was not integrated with TBB and Cilk, providing no tool to manage the oversubscription problem.

1.3.2.2 ForestGOMP

ForestGOMP [20] is an OpenMP runtime compatible, offering a structured way to efficiently execute OpenMP applications on top of hierarchical architectures. Developed in the Runtime Team at Inria Bordeaux and first released in 2007, ForestGOMP proposes a topology aware OpenMP thread scheduling. It uses BubbleSched in order to automatically generate bubbles out of OpenMP parallel regions.

BubbleSched [56] provides a solution to detect the hierarchical structure of the application and map it on the hierarchical architecture of the targeted machine. By relying on the notion of "bubble", the runtime isolates groups of threads that present a certain affinity (for instance data sharing, collective operations). User-level threads belonging to the MARCEL library [44], can then be scheduled on the machine by using different strategies.

1.3.2.3 PaRSEC

Driven by the need to enforce locality and to better scale on large NUMA machines, PaRSEC adopted a solution tackling the composability problem. By statically isolating certain parts of the graphs of tasks the programmers expected to better match the architecture. However, in order to enforce this structure of the application, branches are separated to be executed by what they call VPs (Virtual Processes) allowing an inner parallelism completely local. However, outer communication is made through shared memory in order to diminish the memory allocations.

1.3.2.4 MPC

Mixing MPI and OpenMP is a good solution to benefit from both shared memory and message passing. Thus, isolating OpenMP parallel code inside an MPI process provides a solution to the oversubscription problem. However, it usually leads to the undersubscription problem as this approach is purely static and does not allow any load balancing. Moreover, mixing the inner and outer synchronizations usually forces one thread to manage MPI communications only, thus interfering with the performance of the inner threaded execution.

MPC [23] is a runtime system that also focuses on mixing OpenMP and MPI multi-threaded programming models, and harness the benefits of the two paradigms. Developed by the CEA (The French Alternative Energies and Atomic Energy Commission) and first released in 2008, MPC is a hybrid parallelism framework exposing a thread based MPI runtime (each MPI rank is a thread) and an OpenMP compatible runtime. Focused to diminish memory allocations, all threads (MPI ranks) share the same address space. It targets thus optimizing both network communication and shared memory one, by providing a dynamic workload balancing.

In order to avoid shared memory synchronization issues as well as locality constraints, the MPC runtime propose a TLS (Thread Local Storage) management that privatizes every global variable to obtain thus one copy per MPI task. Therefore, it is a solution to flexibly deal with hybrid MPI and OpenMP programming, as well as a good way to stack multiple parallel programming runtime systems.

However, mixing two different implementations of runtime systems leads to a certain overhead, as the interaction between the two interferes with the resource utilization, leading sometimes to an undersubscription situation. Therefore, MPC proposes using the oversubscription in order to create more parallelism and thus manage the load balancing between the MPI tasks.

In order to limit the consequent overhead on the scheduler, MPC proposes using the notion of *micro VP(micro Virtual Processors)* in order to create an abstraction of the MPI tasks, and then distribute the *micro threads* to each micro VP such that the latter schedules them locally.

This represents a good solution to the composability problem by finding an equilibrium between the undersubscription and the oversubscription situation. Using light implementations for the micro-threads and then isolating them in micro-VPs enforce computations to execute locally with a limited overhead of the context switch. However, it does not represent a solution to the undersubscription problem as it does not provide a dynamic load balancing system.

1.3.2.5 Invasive Computing

Martin Schreiber in his dissertation [50] defended on January 2014 at the University from Munich (Germany), proposes an algorithm to deal with the oversubscription implemented on top of the X10 language. Initially designed for cluster based parallelizations, he describes a dynamic resource management for multi-processes applications. His algorithm relies on three notions: *invade*: requiring new computing resources, *infect*: replicating the program (no actual copy on shared memory systems) onto the successfully invaded resources and starting the program and *retreat*: freeing the previously invaded computing resources.

He proposes a Resource Manager that communicates with the applications through message passaging by using a client-server paradigm. The Client applications, parallelized with TBB or OpenMP, send messages with one of the three constraints (min/max number of resources, application's scalability or application's workload) to the Resource Manager(the server). The latter replies, at the end of a time step, in order to provide information with different scheduling decisions.

Dealing with distributed systems as well as shared memory system definitely requires having a uniform algorithm to manage both level of heterogeneity. Therefore, invasive computing seems to be a good solution to manage oversubscription at the cluster level and at the machine level as well.

1.3.2.6 OpenCL

OpenCL [53] standard offers a common API for programs that execute across heterogeneous architectures. It was released for the first time in 2008 by the Khronos Compute Working Group, a consortium formed with representatives from CPU, GPU, embedded processors and software companies. OpenCL defines core functionalities that all devices support, insuring thus portability and correctness of the code. However, performance results vary according to the specificity of each hardware architecture as well as to the corresponding implementation.

OpenCL is a tool that allows the programmer to manage the devices, the memory allocations, the data transfers, including launching the kernels on the target devices. Meanwhile, OpenCL provides a more high-level feature that is called *device contexts*. Before doing any of the previous tasks, the programmer has to create a context associated with one or more devices. Further on the memory allocation is associated to a context, instead of a device as other paradigms require. This feature creates a constraint to have devices with similar requirements grouped together, as memory is limited to the least-capable one. One drawback is that they also have to belong to the same vendor.

Nevertheless, OpenCL provides a good feature to deal with the composability on a heterogeneous platform. Directly connected to the memory allocation it favors the memory consumption and the locality.

1.3.2.7 Lithe

Lithe [46] is a framework developed as part of the DEGAS project, with the joint effort between UC Berkeley, LBNL, UT Austin, and Rice University. First released in 2009, Lithe provides an environment where threads no longer represent the virtual abstractions of the CPUs, and where each processing resource is represented by a hart (hardware thread). A parallel library can schedule an arbitrary number of tasks, which are very similar to lightweight threads, on top of a fixed set of harts managed by Lithe. Thus, instead of having the false illusion of unlimited processing resources, we have a fixed number of harts, cooperatively exchanged between libraries.

Lithe is based on a hierarchical system where schedulers are always attached to their parent scheduler upon creation. Therefore, children register themselves to their parents, and the parents provide harts to their children. A child may dynamically request additional harts. If accepted by its parent, it will receive the control of the harts through a transition context. Similarly, a parent receives the control of the harts back when one of its children finishes their computations.

By isolating the parallel sections, Lithe provides a solution to the interference generated by the simultaneous co-existence of these codes. However, problems like how and when a parent library should yield harts to a child library are still open to discussions.

1.4 Discussion

Many years of research and development made us having today complex architectures targeting petaflop performance. Nevertheless, if we want to harness their full computing potential one may find them difficult to program. Therefore, specialized parallel libraries usually leave this tasks on behalf of runtimes, delegating the low level optimizations and the portability requirements.

Modularity and reusability concepts imply that applications should rely on these libraries for executing specialized basic operations. However, composing different parallel libraries in HPC applications is not a straightforward task. Parallel libraries are not aware of one another and the application may run into the oversubscription problem. Static solutions fixing the number of threads at compile time not only require significant code modifications but also sometimes lead to undersubscription problems.

Several state of art solutions try to deal with this problem in different ways. A generic solution for shared memory systems is *Lithe*. It proposes removing the false illusion of unlimited resources and allows exchanging them between the parallel codes. Thus, Lithe eliminates the interference generated by the oversubscription and is able to allow different parallel libraries to co-execute on the same machine. MPC, on the other hand, advocates light well controlled oversubscription. It relies on the hybrid MPI and OpenMP approach in order to isolate parallel sections, but without the burden on the memory consumption that such a method implies. Intel TBB provides a much higher level programing environment where parallel codes are isolated into arenas. At the opposite side ForestGOMP focuses on the hierarchical parallelism of the applications in order to separate parallel sections and provides a topology aware OpenMP thread scheduling. All in all, these solutions provide different ways to deal with the composability problem, but they do not have quality information coming from the application in order to tell how many resources each kernel should use and at what point a resource should be reallocated.

Nevertheless, runtime systems should provide the flexibility to allow the interaction with the application. By means of a tool capable of integrating high level knowledge of the application together with runtime information, the end user would be able to implement a portable solution

to manage the composability problem. For instance, the programmers of a simulation library have recently proposed a solution called "Invasive computing" that provides a good model of exchange of information between the application and the Resource Manager. Obtained by means of off-line analysis or provided by the programmer, the workload or the scalability of the client applications allow the Resource Manager to estimate the resource requirements. However, this approach is implemented at the application level where the user is entirely in charge of the load balancing. Using a heterogeneous architecture with different types of processors (e.g accelerators) would require strong adjustments.

Therefore, tackling the composability problem at the runtime system level would ensures both the portability and the reusability of the solution. Indeed, this approach would benefit from low level information concerning the architecture of the machine and could thus allow the programmer to have full control on the resource allocation of the application. Meanwhile, explicit graph representations tend to be a good abstraction to allow the applications provide information (the load, the parallelism of the kernels) to the runtime. The programmer can benefit from the fact that the DAG of tasks represents a system of events and thus supply information correlated to the progress of the execution of the application. In this thesis we have chosen StarPU as it relies on a task-based paradigm and additionally it provides a history based estimation of the execution time of the tasks. We believe that using performance modeling can strongly benefit to the study of the composability problem.

Chapter 2

Scheduling Contexts: a solution to compose parallel codes

Following the trend of the hardware and software evolution, HPC applications have to apply the modularity and reusability concepts. Thus composing different parallel libraries inside the same application becomes inevitable. However, dealing with the composability problem requires having a runtime tool able to manage the oversubscription or the undersubscription problem in a portable way.

We have seen in the previous chapter that the composability problem has already been raised among the development community and several solutions have been proposed. In this chapter we focus on a task-based runtime system approach. We explain how parallel libraries fail to perform well even if they share the same runtime system. We propose then a runtime level solution that gives the programmer the control on the resource allocation of its application and we evaluate the relevance of this approach.

2.1 Composability-related issues on top of task-based runtimes

Dealing with the composability in a task-based runtime system reveals several issues raised by the interleaving of execution flows. Even if the task flows run on top of the same system, scheduling algorithms, locality aspects and specific optimization can be strongly affected.

Scheduling issue Assigning a task to a computation resources is usually done by means of a complex algorithm that satisfies multiple constraints. Parallel kernels often have specific requirements in term of scheduling strategy, for instance it has to consider the granularity of the parallelism, dependencies between the tasks, priorities etc. When composing different parallel kernels we may run into the situation in which kernels use different strategies, sometimes incompatible between them. Being able to deal with multiple schedulers simultaneously is a first step towards the composability.

Locality issue Secondly, the lack of isolation of the parallel kernels may reveal locality problems. Most task-based runtime systems use an online scheduling policy to assign the tasks submitted by the application to the various processing units. When confronted to simultaneous task flows, these online scheduling techniques may fail to deal with memory sharing, interleaving tasks working on different data input. This may result in a deterioration of data locality and scheduling quality.

Local optimization issue Programmers often tune their codes to force runtime scheduling decisions. For instance they take task submission order into account, they pre-allocate memory attached to a specific device or they introduce priorities (e.g. tasks along the critical path are often given high priorities). Such hints cannot be inferred automatically by a runtime system.

Running several unrelated parallel codes on top of a task-based runtime, may ruin such optimizations, allowing the tasks coming from different execution flows to mix and thus to compete for resources in a way that cannot be controlled any more by the programmer.

Possible solution: Ideal meta-scheduler Composing multiple parallel codes efficiently while limiting their mutual interference could theoretically be seen as a global scheduling problem. Indeed, multiple parallel kernels relying on different schedulers could simply be merged, provided that a *meta-scheduler* could meet the requirements of each individual kernel. This problem is related to the co-scheduling of multiple parallel jobs which share the underlying processing units. This has been extensively studied for cache-sharing in multi-processor platforms. Theoretical studies show that the problem is NP-Complete and can be exactly solved only for very simple architectures [58]. Moreover, scheduling policies may be so diverse that the optimization criteria would be different (e.g. time to complete, power consumption). In such situations, there would simply be no rationale that would help the meta-scheduler to prioritize tasks coming from different parallel kernels.

Thus, such a meta-scheduler would have no other choice but to allocate separate resources to each parallel code. It would also have to dynamically adapt to new incoming kernels and their associated scheduling policies, and hence would probably have to perform a dynamic resource allocation between kernels. Such a meta-scheduler would suffer from a scalability problem though, since it would have to maintain a global view of the whole set of computing resources despite the fact that each parallel kernel would only use a subset of them.

2.2 Proposed solution: scheduling contexts to isolate parallel codes

We propose a runtime solution able to isolates parallel codes inside the application and to provide an abstract view of the machine to each one of them. In this sense, we introduce the notion of Scheduling Context, defined as a structure able to encapsulate a parallel code and restrict its execution on a section of the machine. By means of the scheduling contexts, a parallel code runs on top of an abstract machine, whose set of resources can be dynamically changed during its execution. This allows the programmers to control the distribution of computing resources (i.e. CPUs and GPUs) over co-executing parallel kernels. Directly or by means of automatic tools, the programmer can dynamically partition the underlying pool of computing resources and assign each part to a parallel kernel.

Scheduling contexts give the programmer the control over the execution of the parallel kernels. Thus, different optimizations can be made depending on the structure of the application and the underlying hardware architecture. For instance, the programmer can use the scheduling contexts to enforce the locality by allocating only resources sharing a NUMA node to a certain kernel. On the contrary the programmer can consider avoiding memory bus contention and favor resource allocations accordingly.

We consider the scheduling contexts as black boxes, without interfering with the scheduling policy used by the parallel kernel. Thus, the inner scheduler is more scalable as it executes on fewer computing resources and it is adapted to the algorithmic requirements of kernel.

2.3 Scheduling contexts on top of StarPU

By implementing the Scheduling Contexts at the runtime layer, we have access to an abstraction of the machine necessary to manipulate computing resources and to consider both hardware and software information.

We decided thus to implement the Scheduling Context layer within StarPU runtime system (previously presented in section 1.2.2) in order to study their behavior on heterogeneous machines. StarPU is a runtime system that tightly integrates data management and scheduling support. It proposes a unified abstraction of different processing units, which allows us to easily manipulate resources between and inside the contexts.

More concretely, StarPU uses the notion of *worker*, that is the CPU thread in charge with executing the tasks assigned to the corresponding processing unit. We use this representation in order to specify which processing units execute the tasks of a certain scheduling context (see Figure 2.1).

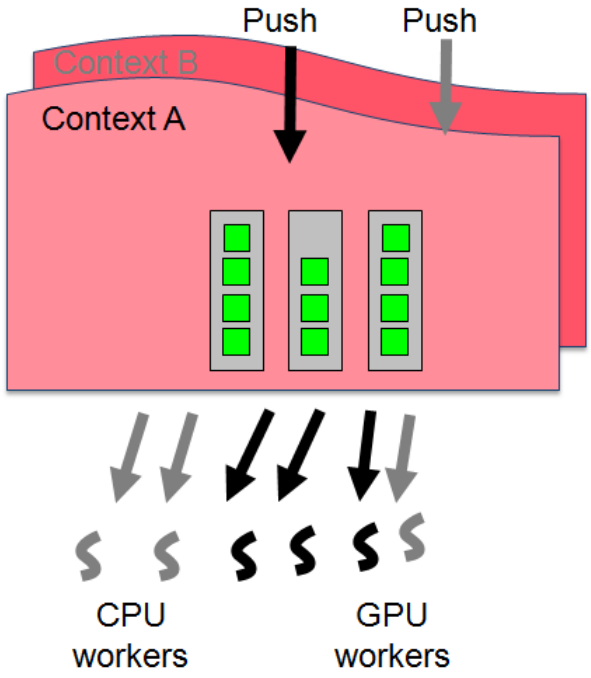


Figure 2.1: Scheduling contexts

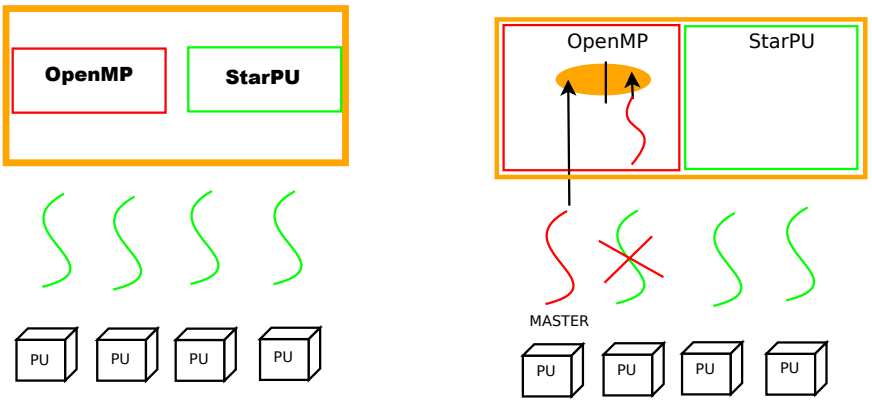
Similarly to lightweight virtual machines, *Scheduling Contexts* allow a flexible partition of the machine and unmodified parallel kernels to coexist.

2.4 Allocate resources to scheduling contexts

StarPU provides an abstraction of the heterogeneous machine. Each computing resource has an ID corresponding to the worker executing on it. When creating a scheduling context we provide a set of IDs that the inner scheduling uses to assign tasks on them. If the inner scheduler is a StarPU

one, the set of IDs we provide corresponds to the set of IDs of the workers used to execute the tasks. Thus, the scheduler assigns tasks only to them.

However, if the inner scheduler is not implemented by means of StarPU, it may have its own set of workers. For instance, we compose an OpenMP and a StarPU parallel task (see Figure 2.2). Each parallel section is isolated inside a scheduling context. The inner scheduler of the OpenMP runtime system creates its own team of threads. Allowing both StarPU and OpenMP set of threads would be unnecessary.



(a) Composing OpenMP with StarPU

(b) Scheduling context isolating the OpenMP runtime system

Figure 2.2: Resource allocation for non-StarPU parallel kernels

Though, we transmit the set of IDs of the StarPU workers to the inner scheduler and we pause the StarPU ones, except one, the *master worker*. This latter one is the only StarPU worker allowed to pick up the parallel task. When the rest of the team of StarPU workers are paused the master stays awake taking the role of application thread in the inner runtime. We see in Figure 2.2) the green threads are the StarPU workers and the red ones the OpenMP threads. The green ones are paused, and the master worker joins the team of OpenMP threads.

Further on we analyze how we can build the set of resources and how important is the programmer's input.

2.4.1 Lightweight virtual machines

First of all the *scheduling contexts* represent a tool for the programmer, that allows assigning dynamically resources to different parallel kernels. High performance programmers usually need a tool that gives control to directly manipulate resources and implement optimizations adapted to

their application. They have the tools to analyze the characteristics of their kernels and the ability to understand the performance of their application.

Programmers can create the number of scheduling contexts they need and they can decide which parallel code should execute in which context. Thus our approach provides the flexibility to structure the parallelism of the application. Moreover, they can specify how resources should – roughly or precisely – be distributed among the contexts. To this end, we give the programmer a way to define a specific distribution. They can either specify a table of processing units identifiers, or an interval of number of processing units (minimum, maximum) required for a particular scheduling context.

In Appendix C we have an example of how the programmer can use the scheduling contexts in order to statically decide the best resource distribution for two simultaneously executing parallel kernels: a factorization and a computational fluid dynamics kernel. Nevertheless, we provide a detailed performance analysis of the scheduling contexts in the Evaluation section.

2.4.2 Information provided by the application

Most of the time even if the programmer has all the information necessary to compute the resource allocation for the scheduling contexts, he needs a tool that automatizes the computation process and that can receive the input and provide the resulting allocation.

Therefore, when creating the scheduling contexts, the programmer can provide an estimated workload for the scheduling contexts and leave the runtime compute the resource allocation. We will see in the next chapter that the runtime tool in charge with doing this estimation and reallocating computing resources dynamically is the *Hypervisor*. However, if all necessary information is available before starting executing the parallel kernels, it can also provide an initial distribution. We refer then to two strategies that will be presented in the next chapter `FEFT` and `TEFT` but that can be also used to provide a static allocation of resources.

`FEFT` takes as input the workload of the scheduling contexts presented as the number of floating points operations as well as an estimated speed (number of floating points operations per second) of the types of computing resources (for instance CPUs and GPUs). By means of a linear program solving the Equation 3.1 presented in Chapter 3 we obtain the number of computing resources of each type necessary to execute the provided amount of work.

`TEFT` also takes as input the workload of the scheduling contexts but presented as type of tasks. By using one of the features StarPU provides, we can obtain the execution time of each type of tasks on each type of resource. More precisely, for each task StarPU measures its execution time on the assigned computing resource and saves this information in a performance model file. Thus, whenever re-executing the same task we already have information of its possible duration.

By using the performance models of the tasks to be executed in scheduling contexts, we can estimate, for regular applications, the resources required by each parallel kernel such that we can execute the application in a minimum amount of time. Therefore, we use the Equation 3.2 presented in Chapter 3 and compute from the beginning the set of worker IDs necessary for each context. However, in the context of more irregular applications this statical approximation is not enough and the *Hypervisor* takes action in order to dynamically reconsider the initial distribution.

2.5 Scheduling policy inside the context

The scheduling contexts isolate the parallel kernel without interfering with the scheduling algorithm applied inside each context. Computing resources are assigned to the scheduling contexts and the inner scheduler is only informed of the new resource allocation. Even if the parallel kernels are implemented on top of StarPU, schedulers run unmodified as guest schedulers in an isolated manner.

Nevertheless, if the kernel is implemented on top of a non-StarPU runtime, the inner scheduler is informed of the resource allocation required by StarPU. By executing an inner runtime specific code just before executing the parallel kernel, the inner scheduling policy is aware of the resources on which it is allowed to run. For instance, we isolate an OpenMP parallel code inside a scheduling context. StarPU indicates to OpenMP the number of computing resources as well as the logical IDs on which the parallel code can execute. Before running the parallel code we execute an OpenMP specific code that binds the corresponding team of threads to the set of computing resources indicated by StarPU.

StarPU allows executing different callbacks whose implementation is provided by the programmer. For instance, the programmer can provide the implementation of the callback executed immediately after a certain task is finished or just before being submitted. We refer now to the prologue callback, which is executed by the worker assigned with the task, just before starting executing it.

We require the programmer to provide the implementation of the prologue callback to be executed just before the non-StarPU parallel kernel. Familiar with the runtime executed inside the scheduling context, the programmer can use the necessary functions in order to indicate the new resource allocation imposed by StarPU.

For instance, if we want to isolate an OpenMP parallel kernel inside the context, we have to indicate in the prologue `omp_set_num_threads(num_threads)`, such that before executing the kernel, the inner runtime is aware it executes only on a restricted number of cores.

2.6 Share computing resources

Some specific types of processing units, such as GPUs, can not always be exploited at their full potential by some kernels. This is mainly due to the fact that a given parallel kernel may not have enough tasks capable of running on such accelerators. To tackle this problem, StarPU allows any resource to be time-shared between several contexts. When a processing unit is shared by several contexts, StarPU uses a round-robin algorithm between the different contexts in order to fetch the next task to run.

This mechanism can be expensive when there are significant differences of workload between contexts executing on shared resource. We can waste time searching for work in a context with no tasks available instead of favoring other contexts having tasks ready to execute. Therefore, the programmer can provide priorities for contexts, such that some contexts can fetch more tasks before other do.

Since computing resources may be shared by multiple contexts, the associated task schedulers need to cope with processing units interleaving tasks coming from multiple contexts. We have thus modified the schedulers provided in StarPU in order to be able to correctly predict the expected termination time for the resources shared between contexts. This is done by making the contexts inform each other when they schedule tasks on these resources. Thus, each scheduler associated

to a context is aware of all the tasks assigned to the shared resources, even the ones coming from other contexts.

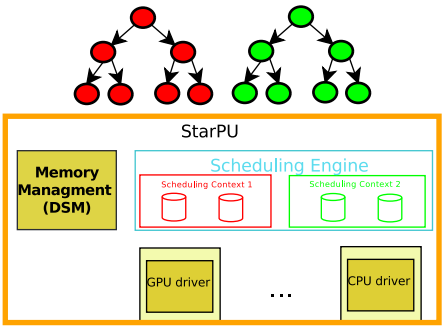


Figure 2.3: Scheduling Contexts in StarPU.

2.7 Implementation

We place the Scheduling Context layer above the Scheduling Engine of StarPU, without actually interfering with the implementation of the schedulers (Figure 2.3) such that the scheduler of any other runtime could be used. The "black box" philosophy allows the scheduler to receive information regarding the computing resources it should execute on and to return a valuable distribution of tasks over the restrained group of resources. Thus, the main challenge is to distribute computing resources to schedulers so that they better meet the constraints of the application.

The scheduling policies may be diverse: they can aim at minimizing the termination time of the kernel, minimizing the memory occupancy or maximizing the efficiency of the processing units. Therefore, kernels are executed in a confined way so as to improve data locality, lower memory contention and increase performance of the whole application.

Each scheduling context is associated with a scheduling policy, which allows several schedulers to coexist with limited interference within a single parallel application. Most importantly, a scheduling context can have a restricted view of the hardware: a list of "visible" processing units (regular cores, accelerators, etc.) is maintained for each context by the runtime system.

2.8 Execution model

Scheduling contexts can be created or destroyed dynamically, as libraries or kernels are not necessarily initialized at the same time and they may not be used during the entire application. When creating a context, the programmer indicates the set of computing resources and the scheduling strategy to be used for executing the parallel code. If a non-StarPU strategy is used this information is not required.

Further on, using the scheduling contexts is non-intrusive for the implementation of the application. When submitting the tasks, the programmer has to indicate to which of the previously created context the task should go.

```
int resources1[3] = {CPU_1, CPU_2, GPU_1};
int resources2[4] = {CPU_3, CPU_4, CPU_5, CPU_6};

/* define the scheduling policy and the table
  of resource ids */

sched_ctx1 = starpu_create_sched_ctx("mct",resources1,3);

sched_ctx2 = starpu_create_sched_ctx("greedy",resources2,4);
```

```
// thread 1:

/* define the context associated to kernel 1 */
starpu_set_sched_ctx(sched_ctx1);

/* submit the set of tasks of the parallel kernel 1*/
for( i = 0; i < ntasks1; i++)
 starpu_task_submit(tasks1[i]);
```

```
// thread 2:

/* define the context associated to kernel 2 */
starpu_set_sched_ctx(sched_ctx2);

/* submit the set of tasks of parallel kernel 2*/
for( i = 0; i < ntasks2; i++)
 starpu_task_submit(tasks2[i]);
```

Figure 2.4: Programming with Scheduling Contexts

2.8.1 Scheduling contexts using StarPU scheduling strategies

In Figure 2.4 we can see a simple use case of an application composing two parallel kernels. The first step requires creating the scheduling contexts and in the current example we use two different StarPU scheduling policies for the contexts: *mct* and *greedy*.

2.8.2 Scheduling contexts using Non-StarPU scheduling strategies

We consider now a use case where one of the scheduling contexts executes an Intel MKL parallel kernel using an OpenMP scheduler.

In order to allow StarPU (which we refer to as the *outer runtime system*) communicate the resource allocation to the guest scheduler (that we call *inner runtime system*) managing the resources for the parallel kernel we require the programmer to provide an implementation for the prologue callback. We can see in Figure 2.5 that this implementation consists in creating the OpenMP threads and binding them to the logical ids provided by StarPU. When executing the Intel MKL task, the inner scheduler reuses previously created threads, that are correctly fixed on the computing resources that StarPU decided.

2.9 Evaluation

In this section, we present a series of experiments which evaluate the impact of using scheduling contexts within applications requiring multiple parallel kernels to be executed concurrently.

```
#include "mkl.h"

void cl_prologue()
{
 /* bind openmp threads to CPUs */
#pragma omp parallel num_threads(3)
  bind_current_thread_to_cpuid(resources[omp_get_thread_num()]);
}

void codelet_cpu_func()
{
  /* call the mkl parallel kernel */
  DGEMM("N", "N", &m, &n, &k, &alpha, A, &m, B, &k, &beta, C, &m);
}
```

```
int resources[3] = {CPU_1, CPU_2, GPU_1};

/* define the table of resource ids */

sched_ctx = starpu_create_sched_ctx(resources,3);

/* submit the set of tasks to the context*/
for( i = 0; i < ntasks; i++)
{
 /*indicate the prologue function to execute */
 tasks[i].prologue = cl_prologue;

 /* indicate the codelet of the task*/
 tasks[i].codelet = codelet;

 /* submit the task to the context */
 starpu_task_submit_to_ctx(tasks[i], sched_ctx);
}
```

Figure 2.5: Executing Intel MKL parallel codes within Scheduling Contexts

We focus here on straightforward scenarios in order to study how two (or more) concurrent kernels compete for resources and exhibit how our scheduling contexts solve this problem in a generic way.

2.9.1 Experimental scenarios

We focus on two parallel linear algebra libraries **MAGMA-MORSE** and **Intel MKL** (see Appendix A for description). In both cases we select the Cholesky factorization kernel (`potrf`) for its simplicity and regularity. Further on we build parallel applications running simultaneously several Cholesky factorizations and we evaluate the performance improvement brought by the scheduling contexts.

We implement simple programs calling multiple instances of MAGMA-MORSE factorizations simultaneously and we consider the total execution time of the application, because scheduling contexts are expected to improve the overall behavior of the application and not just the performance of each parallel kernel. Moreover, to ensure best performance for MAGMA-MORSE Cholesky factorizations kernels, we use two blocking factors for all our experiments, one favorable to GPUs of 960 x 960 elements and one favorable to CPUs of 192 x 192 elements.

The main libraries we used for our experiments are some of the most efficient (together with

FLAME [37]) and widely used libraries for dense linear operations on top of heterogeneous (MAGMA) or homogeneous systems (MKL). Thus, we use them to illustrate the behavior of the scheduling contexts. We build artificial but well-controlled scenarios that mimic the configurations that can be met within complex applications, like sparse linear solvers or domain decomposition methods. These applications feature a graph of several dense BLAS operations, many of which can run concurrently.

2.9.2 Experimental architectures

We evaluate the relevance of our approach using two types of platforms (see Appendix B):

- Homogeneous shared memory memory one featuring 4 deca-core Intel processors: `riri`
- Heterogeneous one featuring 2 hexa-core Intel processors and 3 NVIDIA GPUs : the `mirage`

2.9.3 Homogeneous architecture

Matrix order	Serial	Interleaved flows	4 Contexts
14976	14.32	114.98	13.30
19968	32.30	176.55	31.21
29760	105.27	552.78	99.86
39360	239.03	387.70	228.91

Table 2.1: Composing mkl kernels (Time in s)

We make our first experiments on `riri`, a machine having 4 NUMA nodes with 10 computing resources on each one of them. Therefore, we build an application composed by four parallel kernels executing Cholesky factorizations belonging both to Intel MKL and MAGMA-MORSE library. We evaluate the effect generated by the interference of multiple parallel kernels executed simultaneously and we study possible solutions, that is serially executing the kernels or simultaneously executing them by using the Scheduling Contexts to insure isolation.

We start by studying the composition of four MKL kernels of the same size. We evaluate the execution time of the application when applying the two solutions, serial execution and scheduling contexts, with respect to the version executing the four parallel kernels without any programming optimization. We notice in Table 2.1 that allowing the kernels execute simultaneously without isolating them in any way, deteriorates significantly the performance of the application. The main reason is that each parallel kernel creates its own set of 40 OpenMP threads, thus generating an important overhead on the scheduler of the operating system and running into the oversubscription problem.

We observe that executing the four kernels serially can be a relevant solution as the kernel scales very well on the `riri` machine. In the third scenario we isolate the four parallel kernels in scheduling contexts of 10 CPUs each. We see in Table 2.1 that using the scheduling contexts slightly improves the execution time compared to the scenario executing them serially. The reason is that isolating the kernel on processing units sharing the same memory bank avoids any transfers towards other ones.

We now evaluate the composition of four MAGMA-MORSE kernels. We notice in Table 2.2 that the performance behavior of this application is different from the previous one. We notice that

Matrix order	Serial	Interleaved flows	4 Contexts
14976	53.02	20.88	20.01
19968	124.28	50.49	47.92
29760	407.13	160.62	163.58
39360	528.26	399.29	386.97

Table 2.2: Composing MAGMA kernels (Time in s)

serially executing the four kernels is not a good solution, as the kernel does not scale good enough on the 40 cores. The main reason of this problem is the scheduling strategy used by the kernels. We use here the MCT strategy that was designed especially for heterogeneous architectures. The algorithm scales more difficultly on homogeneous architecture with many workers. The main idea of this algorithm is that before assigning a task to its computing resource we first iterate on each worker and estimate if executing the task on the current worker makes the application execute in a minimum amount of time. Obviously, when having several workers the overhead of the algorithm becomes significant. In this case executing the kernels simultaneously is mandatory.

Moreover, we notice that the interference of the four parallel kernels is not so important in this case. The main reason is that the MAGMA-MORSE Cholesky Factorization kernel has a task based implementation and interleaving tasks coming from the four kernels does not induce an overhead on the operating system. Despite this, using the scheduling contexts still improve the overall execution time of the application because they enforce the locality.

Thus, we observe that even if the two libraries, Intel MKL and MAGMA-MORSE, have different scheduling strategies, scaling or managing the parallelism differently, the scheduling contexts represent a solution to fix the oversubscription issues without interfering with the guest scheduling strategy.

2.9.4 Heterogeneous architecture

Further on we evaluate the scheduling contexts on top of `mirage`, a heterogeneous platform, and we show that scheduling contexts help to better enforce data reuse and locality. In this section we focus only on the MAGMA-MORSE Cholesky factorization kernel as its implementation is adapted to the use of GPUs.

We present an experiment where we execute three independent parallel kernels, performing Cholesky Factorizations on matrices of 19200 x 19200 elements with a tile size of 960 x 960. We compare the situation where tasks coming from the three kernels interleave during the execution, with the one where we use contexts to isolate them. For the latter, we build contexts having three CPUs and one GPU each and we associate a kernel to it.

We are interested here to see the importance of scheduling contexts when the architecture of the machine provides a non uniform memory distribution. Thus, we use them in order to enforce locality and avoid unnecessary data transfers.

In Table 2.3 we evaluate statistics concerning the chances of finding the needed data on a certain device memory. We observe that by using the contexts we reach a hit rate of 90 % which is almost 10 % higher than when not isolating the kernels. Tasks belonging to a certain context execute only on the assigned GPU, therefore the chances to find their data on place are greater than when using three GPUs. We notice that the total amount of data transferred is drastically reduced (more than

50 % reduction) when using the contexts.

	Interleaved flows	3 contexts
Hits on Host memory	92.67%	91.35%
Hits on GPU 1 memory	76.97%	87.96%
Hits on GPU 2 memory	74.26%	88.08%
Hits on GPU 3 memory	74.64%	87.86%
Total hits	80.26%	88.93%
Total transferred data (in GB)	51.98	22.82
Execution time	8.55 s	8.43 s

Table 2.3: Data transfer statistics of concurrent execution of three factorizations on `mirage` platform

The scheduling strategy plays an important role in managing the data transfers. The main objective of MCT strategy in our case is to overlap as much as possible data transfers with computation in order to finish executing the tasks in the earliest time. Even if the scheduling decision at a certain point of the execution is correct and this constraint is respected with the risk of increasing the data transfers, the scheduling policy does not have enough view of what will happen latter. We can face a situation (see Figure 2.6) where a task belonging to kernel 1 is executed on the GPU 2 even if the data of the kernel 1 has already been transfered to GPU 1. It is a good scheduling decision because the remaining workload finishes executing in the earliest time. Then a task belonging to kernel 2 is scheduled to GPU 1 even if data of the kernel 2 is present on GPU 2. Thus, data transfers were mainly overlapped with computation, but the overall execution time was affected.

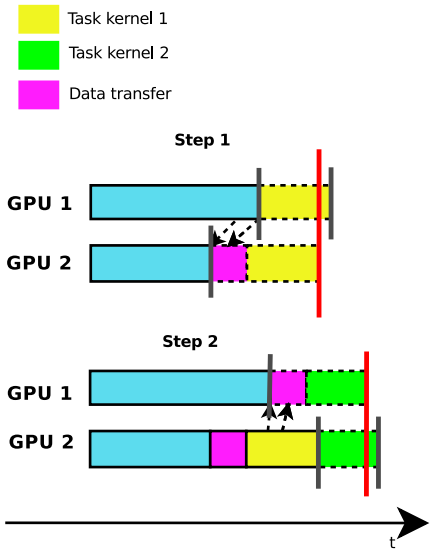


Figure 2.6: Using MCT to schedule two flows of tasks.

This scheduling strategy may easily be fixed using different techniques prioritizing certain tasks. However, scheduling contexts represent a possible solution without actually trying to fix the inner strategy.

In the second scenario we show that enforcing locality for applications executing simultaneously more than 3 parallel kernels has an important impact on the execution time. Therefore, we compose 9 independent Cholesky factorizations of matrices of the same size (19200 x 19200 elements), 9 being the number of available CPU workers on our test platform. We also measure the serial execution of the nine kernels (i.e. the nine parallel kernels are executed one after the other in a single context).

	Serial execution	Interleaved flows	3 contexts	9 contexts
Hits on Host memory	89.19%	91.96%	90.25%	90.16%
Hits on GPU 1 memory	78.29%	75.50%	91.44%	91.87%
Hits on GPU 2 memory	74.73%	70.90%	90.98%	91.48%
Hits on GPU 3 memory	74.73%	70.11%	91.37%	92.08%
Total hits	79.33%	78.12%	90.97%	91.35%
Total transferred data (in GB)	160.70	172.05	66.65	74.62
Execution time	29.20 s	27.79 s	25.32 s	25.27 s

Table 2.4: Data transfer statistics of concurrent execution of 9 factorizations on `mirage` platform

We observe in Table 2.4 that concurrent parallel kernels isolated in contexts show important performance improvement compared to the mono context version. In our experiment the overall application has reduced its execution time by 15% when isolating task flows.

The performance degradation of the single context version is mainly coming from GPUs exploitation. That is, an increase of the amount of data transferred between GPUs and the main memory is observed whenever several independent parallel kernels share the GPUs. In this case, scheduling a task on a shared GPU implies transferring the necessary data and possibly evicting data used by the previous task. Therefore, tasks belonging to different kernels but executing on the same device may constantly evict one another's data. Nevertheless, this can significantly vary according to the scheduling policy (StarPU or not) but using the scheduling contexts allows improving locality without interfering with the scheduling policy.

Therefore, by separating the nine kernels in three contexts, or even in nine, the number of kernels which use a given GPU is smaller and therefore contention is reduced. To further illustrate this phenomenon we measure the amount of memory transfers between CPUs and GPUs in the three cases. In Table 2.4 we consider the misses in the GPUs memory and we observe that when using an appropriate number of scheduling contexts we have around 10% of memory misses while when using a single context version we have around 20% of memory misses. Furthermore, the amount of data transferred between CPUs and GPUs is around 67 GB when using several scheduling contexts whereas it reaches 170 GB when using a single context. We reproduced these measurements on larger kernels (i.e. with matrix of order 30 000) and observed roughly the same behavior (multiple context-based configurations are around 30% faster than single context ones). In this case scheduling contexts represent a tool necessary to reduce contention on the GPUs and eventually improve the performance.

Moreover, if we evaluate the case where we execute the nine kernels in a serial way we observe that we run into the same problem, the amount of transferred data is still greater than the one obtained when using the scheduling contexts. The main reason is that when executing a kernel on multiple GPU devices, depending on their availability StarPU scheduler may decide to execute a task on a GPU that does not have the required data fetched. This implies prefetching it [16] without however having an important effect on the performance as data transfers may be covered

with computations. On the contrary, when using the scheduling contexts to isolate the kernels, StarPU prefetches data only on the GPU on which the kernels are allowed to execute.

It is interesting to notice that separating the kernels in 3 contexts or in 9 contexts does not change significantly the behavior of the application The reason is that in both cases one GPU is shared by three kernels. When using 9 contexts, on GPU is shared between 3 contexts, but when using 3 contexts, each context executes 3 kernels sharing the assigned GPU. Thus, we noticed that having a wise management of the GPUs is an important matter and contexts represent a useful tool to do this.

2.10 Discussion

To enable high performance computing applications to exploit multiple parallel libraries simultaneously, we introduce the *Scheduling Contexts*, a programming tool able to isolate the parallel libraries. We provide thus a generic solution, where inner scheduling strategies are not altered, instead they only execute on a restrained set of computing resource.

In a simple scenario, all the libraries run on top of a common runtime system, the programmer has then a full control of the resource allocation. In the more general case the libraries do not necessarily share the same runtime system. Thus, we propose a mechanism to inform the inner scheduler, as long as it supports such guidelines, of the resource allocation the outer runtime system requires. In this chapter, we have illustrated this approach using StarPU as outer runtime and Intel OpenMP as inner runtime.

Scheduling contexts represent also a tool for tackling the locality problems. By isolating parallel codes on a restrained set of computing resources, the contexts may insure faster access to the allocated memory in case of distributed memory (e.g. NUMA, heterogeneous architectures) and also a better scalability. Indeed, this approach gives the programmers the full control on how resources are used by parallel libraries. Nevertheless, deciding how many computing resources of each type one can allocate to concurrent parallel kernels requires a certain degree of expertise. The programmer can either provide a set of computing resources to each context or he can use external tools to compute the necessary resource distribution. However, without being attached to the actual execution of the application, the programmer can difficultly indicate the moment when such a distribution is needed. We believe that this is a runtime system matter.

In the following chapter we propose a tool called the *Hypervisor* integrated as a plugin to StarPU, that monitors the progress of the applications and provides different algorithms to dynamically compute the resource distribution over the *scheduling contexts*.

Chapter 3

The Hypervisor

We have seen in the previous chapter that scheduling contexts allow to reuse existing optimized codes and help improving the application facing the composability problem by isolating parallel kernels executing simultaneously. However, sometimes programmers cannot foresee entirely the resource requirements of their application. Unexpected behaviors can arise and the optimizations made before starting executing the applications are not efficient any more during its execution. We can take for instance a fluid mechanics problem, where at some point we have intensive operations on some areas and later on the activity diminishes on those areas. State of art solutions usually focus on statically determining this behavior and allocating the resources accordingly. Yet, this is not only very difficult to do but may also be impossible when the necessary information is determined at runtime. Therefore, we need a runtime tool capable of identifying these situations, react and adapt the resource distribution to the new requirements such that the overall execution time can be improved.

Therefore, we consider a dynamic approach that aims at foreseeing performance problems experiencing the application running within a certain configuration and adapting the resource allocation on the fly. Our contribution is based on helping the programmers to better optimize their applications by diminishing the static analysis in favor of dynamic observations improving reactivity and dynamic decision taking.

In the following section we present *the hypervisor*, a tool capable of resizing the scheduling contexts whenever their initial configuration deteriorates the performance of the application. We focus on the dynamic approach to determine unexpected behaviors of the application and to trigger the redistribution of resources.

3.1 Monitoring scheduling contexts' activity

In order to determine when the application is misperforming we use the hypervisor as a tool to monitor both the resources and the application (see Figure 3.1). The hypervisor, either polls the runtime at some intervals of time in order to check the evolution of the application or uses callbacks triggered by the runtime that update statistical information.

Monitoring the execution of an application is a discrete process. By collecting information at the instructions or the tasks level, the performance counters are available only when these operations are ready. The granularity of the information represents the compromise between the precision and the overhead of the hypervisor. Limiting the overhead of the hypervisor requires dividing the

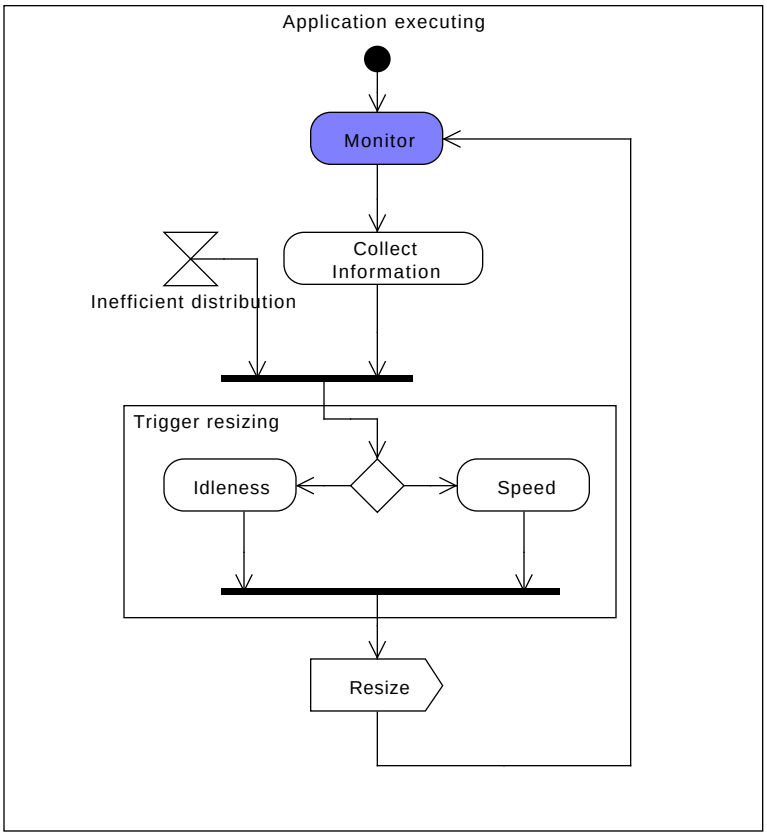


Figure 3.1: A tool to monitor

application into intervals of time. We allow the hypervisor to use the performance counters only when enough data was collected to give relevant insights.

3.2 Collecting information

While monitoring the execution of an application, the hypervisor gathers performance information concerning the current run. In Figure 3.2 we see this information is later used to resize scheduling contexts.

Collected information may consist of the duration of idle periods or the speed at which processing units make progress, the number of tasks or instructions executed, the number of tasks ready to be scheduled or executed.

The hypervisor uses these statistics to dynamically estimate the resource requirements. Nevertheless, these estimations may depend on the applications and on the problems. Very irregular applications may not permit relying on history to predict future. Despite this, usually these complex application can be regular on phases. However, when the future execution is not predictable, the hypervisor may collect hints from the programmer hoping more useful information may come from there.

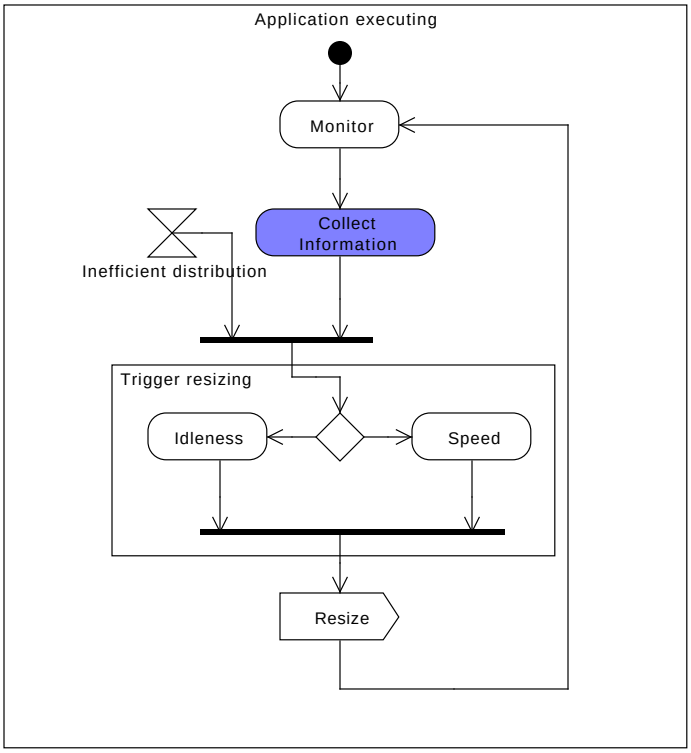


Figure 3.2: A tool to collect information

These hints may typically fix the lower- and upper- bounds that the hypervisor should not cross when allocating computing resources to a given context. For example, if the programmer wants to run two different parallel kernels simultaneously within the same application, he provides the hypervisor with a range of processing units that he considers necessary to the execution of each kernel (e.g. at least 1 GPU and between 2 and 4 CPUs for the kernel). Based on this information, the hypervisor adapts the size of the contexts according to its metrics while respecting the constraints given by the programmer.

The collected data is used to compute the speed of the contexts and estimate the termination time of the application. Thus, the hypervisor blends in a light *Statistics Manager* that stores information about contexts and resources performance. Additionally, a *Resizing Engine* is responsible for redistributing computing resources based on performance prediction of the application.

3.3 Triggering the resizing process

By using the hypervisor the programmer can require the creation, the modification and the destruction of the scheduling contexts. The algorithmic structure of the application can provide information concerning dynamic changes in the parallelism of concurrent kernels. However, the hypervisor aims at detecting dynamically when the application is not executing efficiently (see Figure 3.3). The inefficient behavior of the application is determined using the performance counters, that

provide information of when processing units are not exploited at maximum efficiency and triggers the modification of the scheduling contexts.

3.3.1 When to trigger the resizing ?

In order to automatically detect performance deterioration in the execution of the application we use two metrics: processor idleness and low speed of parallel kernels. The first one represents the basic proof of the undersubscription problem. However, for certain applications an interval of inactivity of a resource may represent a normal behavior (for e.g. the numerical algorithms require a barrier at a certain point) while for others the same interval may mean that the previous distribution of resources was inefficient. Determining when a resource is idle consists in knowing after how much time of inactivity the hypervisor may consider moving this resource from one context to another. The length of this interval must be provided by the programmer as the irregularity of the execution of an application is difficult to determine without off-line analysis. However, by the means of an empirical evaluation this interval may be easily determined.

The second metric consists in observing when parallel kernels are under-using their resources. In order to determine this behavior we compute the speed at which kernels execute (number of executed flops inside an interval of time). We compare this value to the ideal speed of execution of the kernel (computed by considering the speed of the processing units when allocated to the context executing the kernel). By observing consistent differences between the two we can tell that the kernel is not executing at the capacity it was expected to.

3.3.2 How to trigger the resizing ?

Either the application (the programmer) or the runtime (the hypervisor analyzing different metrics) may indicate that the current distribution of resources over the scheduling contexts is no longer efficient. Whenever this happens, the worker or thread involved in updating statics and determining inefficient behavior of the contexts takes the role of hypervisor and executes different algorithms (see section 3.4.3) that permit adapting the allocation of resources. Thus, the hypervisor is just a piece of code executed at some point by the worker threads.

3.4 Resizing the scheduling contexts

The hypervisor constantly collects performance information and considers the new statistics whenever it decides to redistribute resources over the scheduling contexts. Thus, resources inefficiently used in a context can be reallocated to another one where computation requirements are more significant.

3.4.1 (Re)allocate resource to scheduling contexts

Processing units can be assigned statically to scheduling contexts before starting the execution of the application. Different algorithms allow foreseeing the computations requirements of the application by the means of the programmer's indications. However, processing units can also be reassigned during the execution of the application.

Depending on the type of the scheduler of the contexts, resources can be moved more roughly or more smoothly from one context to another. For instance schedulers assigning a task queue to each

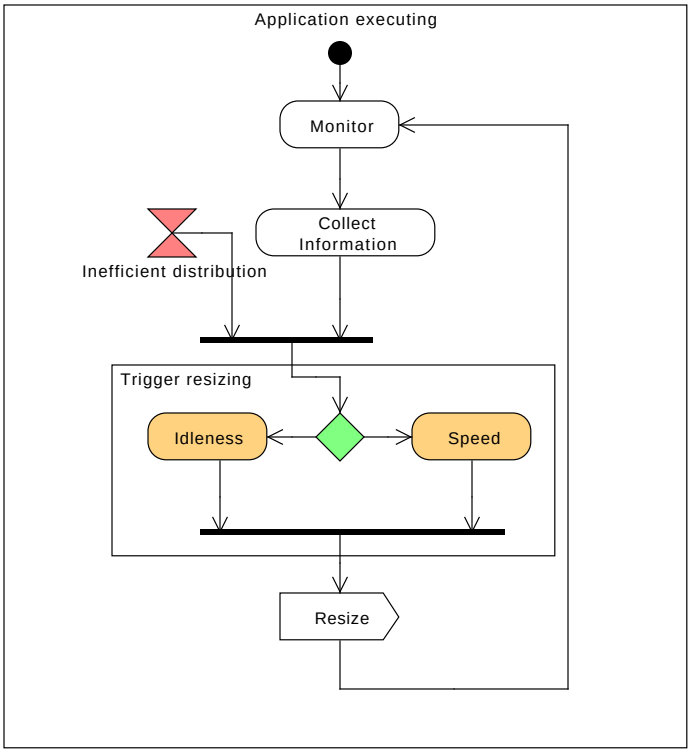


Figure 3.3: A tool to trigger

processing unit move them once they finished executing their own tasks in that context, meanwhile resources have to be moved without any delay for single shared task queue schedulers.

The reactivity of the hypervisor depends on the interval of time the scheduling strategies allow the workers to execute tasks of the old context after a resizing decision. By increasing this interval, different estimations (termination, speed, etc.) concerning the scheduling contexts may not be valid anymore. However, too rough reallocation of resources may generate data migration (for instance the data the processing unit was working on was allocated on its corresponding NUMA node, and the other processing units work on a different NUMA node) which may be penalizing in terms of the execution time.

Once the resizing process is executed the hypervisor restarts monitoring the application and the computation resources in order to continue resizing scheduling contexts if needed (see Figure 3.4)

3.4.2 Adapt to resource reallocation constraints

Dynamically reallocating resources to parallel kernels implies managing the communication between the decision taker, the hypervisor, and the decision obeyer, the parallel kernel. We refer to *maleable tasks* when their corresponding scheduling contexts are able to resize dynamically without any time constraint, that is the decision obeyer can apply the decision taker's request immediately. We refer to *moldable tasks* when the decision obeyer can apply the resizing only before or after a task executes. For instance, MKL parallel kernels need to keep the number of resources constant during

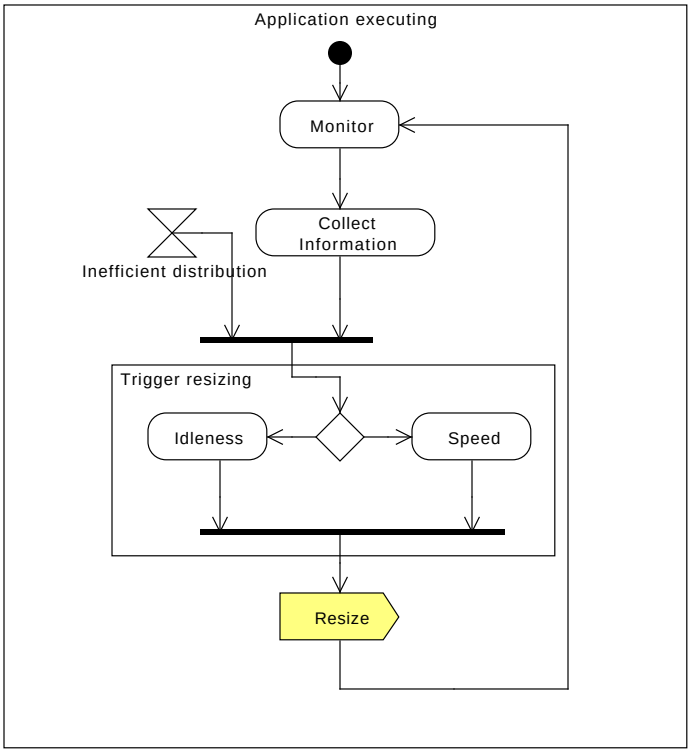


Figure 3.4: A tool to resize

their execution, holding any resizing in the meantime.

Nevertheless, the rigid execution of the moldable tasks may prevent the hypervisor from correcting the resource distribution, thus affecting the performance of the application. However, the role of the hypervisor is to adapt to these situations and to try to accomplish in the best possible way the requirements of the application.

3.4.3 Strategies to resize scheduling contexts

Different algorithms for reallocating resources may be implemented. The structure, the type of parallelism and the information available from the application may lead to different strategies.

3.4.3.1 On-demand

Firstly, we consider situations where the application is able to drive the resizing process entirely or when simply the programmer requires full manipulation of the resizing tool. For these situations, the hypervisor provides a resizing policy where the application can indicate the moment in the execution when a resizing process should be triggered as well as the set of resources that should be moved. Meanwhile, in a data flow approach the programmer does not have direct access to the execution flow at a certain moment. For this reason we use the task graph to model time and by tagging a task whose execution is observed by the hypervisor the programmer requires a

redistribution of resources when that task finishes executing. Therefore, the request as well as the potential configuration are saved and applied to the parallel kernels when the tagged task is executed. We can see in Figure 3.5 that once that certain task is executed the resizing process is triggered and 4 CPUs moved from the yellow context to the green one.

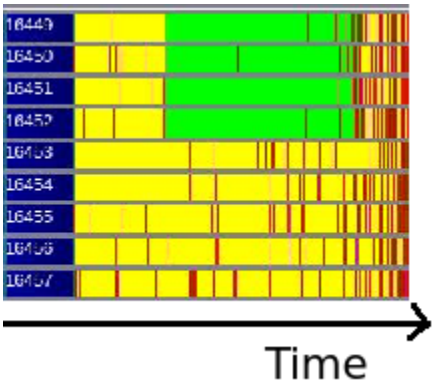


Figure 3.5: On-demand strategy in StarPU

3.4.3.2 Minimize the makespan

Further on we consider strategies that considers some input coming from the application and computes dynamically a correct distribution of resources. One strategy is to redistribute the resources such that the scheduling contexts finish at the same time by minimizing the makespan. In order to do this, the application needs to provide some information concerning the average workload left to execute in each scheduling context.

`FEFT` One solution is to provide this workload as an average amount of flops remaining to be executed and solve the Equation (3.1). We refer to this strategy as `FEFT` (Flops executed in the Earliest Finish Time)

In the linear programming problem described by Equation (3.1) we compute the number of CPUs and GPUs needed by each context such that the program will end its execution in a minimal amount of time. Note that this is a rough approximation since we do not consider either the data dependencies between the task or the granularity of the tasks.

$$\max\left(\frac{1}{t_{max}}\right) \text{subject to} \left(\begin{array}{l} \left(\forall c \in C, n_{\alpha,c} v_\alpha + n_{\beta,c} v_\beta \geq \frac{W_c}{t_{max}}\right) \\ \wedge \left(\sum_{c \in C} n_{\alpha,c} = n_\alpha\right) \\ \wedge \left(\sum_{c \in C} n_{\beta,c} = n_\beta\right) \\ \wedge \left(\forall c \in C, n_{\alpha,c} < max_{\alpha,c}\right) \\ \wedge \left(\forall c \in C, n_{\beta,c} < max_{\beta,c}\right) \end{array}\right) \tag{3.1}$$

In this linear program C denotes the set of contexts, $n_{\alpha,c}$ and $n_{\beta,c}$ represent the unknowns of the system, that is the number of CPUs and GPUs that are assigned to a context c, W_c is the total amount of work associated to the context c, t_{max} represents the maximum amount of time spent by a context to process its amount of work, v_α and v_β represent the speed (i.e. floating point operations per second) of a CPU respectively GPU on the platform, n_α and n_β are the total number of CPUs, respectively GPUs available on the machine. Equation (3.1) expresses that each context should have the appropriate number of CPUs and GPUs such that it should have finished its assigned amount of work before the deadline t_{max}. The initial value of the speed of the CPUs and GPUs has an important impact on the decision. Of course, this linear program can be easily generalized to platforms with more than two types of resources.

`TEFT` A second solution is to provide the workload as the number of tasks of each type remaining to be submitted. By the means of the calibration files StarPU updates at each run, we obtain the execution time of each type of task (executing a certain kernel using a data entry of a certain size) on each type of processing unit. The Equation (3.2) is then able to provide a potential scheduling of these tasks indicating thus potential resource requirements of each scheduling context. We will refer to this strategy as `TEFT` (Tasks executed in the Earliest Finish Time).

By recording the number of submitted tasks according to their type (the kernel they run, the size of data they operate on and the context they belong to) we can compute an optimal distribution of tasks among processing units. The main improvement of this algorithm is that it takes into account the type of tasks and does not consider the application as a simple amount of work to be done. The problem corresponding to the resources allocation in this case is solved through a non-linear program where we want to minimize the global completion time.

To solve this non-linear program, we use a dichotomy on the value of the variable causing the non-linearity (t_{min} in our case) to find its optimal value. At each step of the dichotomy process, we try to find a feasible solution to the linear program described in Equation (3.2). Note that this linear program does not need a specific objective function since we just need to check if a feasible solution can be computed with the given value of t_{min} (t_{min} being the execution time). Of course, the correctness of this technique relies on the accuracy of the performance models.

$$\min\left(t_{min}\right) \text{subject to} \left(\begin{array}{l} \left(\forall w \in W, \forall c \in C, \sum_{t \in T_c} n_{t,w} \cdot d_{t,w} \leq t_{min} \cdot x_{w,c}\right) \\ \wedge \left(\forall c \in C, \forall t \in T_c, \sum_{w \in W} n_{t,w} = n_t\right) \\ \wedge \left(\forall w \in W, \forall c \in C, x_{w,c} \in \{0,1\}\right) \\ \wedge \left(\forall w \in W, \sum_{c \in C} x_{w,c} = 1\right) \end{array}\right) \tag{3.2}$$

In this linear program C denotes the set of contexts, W is the set of workers, T_c represents the set of types of tasks of a context c, $x_{w,c}$ is a boolean variable denoting whether or not a worker w belongs to the context c, $n_{t,w}$ represents the number of tasks of type t performed by the worker w, $d_{t,w}$ is the estimate duration of the execution of the task t on the worker w and finally n_t is the number of tasks of type t encountered during execution. Equation (3.2) expresses the fact that each worker has to execute its tasks assigned by the context it belongs to before the total execution

time t_{min}. Furthermore, each worker has to be part of exactly one context. Finally, it is important to notice that the complexity of this second approach is greater than the one of the first approach (either in term of number of constraints, number of variables, or because of the dichotomy process).

Even if it is sometimes difficult for the application to provide the entire workload left in each contexts, as it is usually the case for most irregular applications, it is possible to get a good workload estimation for the current phase of computations (e.g. for a single iteration). Therefore, the hypervisor can complete the information gradually and correct the resizing decisions at each step. We will refer to this kind of strategies as *completion time based* resizing policies.

3.4.3.3 Maximize the instant speed

Clearly, collaborative strategies mixing knowledge both from the application and the runtime system have a better potential to behave optimally. However, when no global workload information is available, either because the application is very irregular or because the programmer is not an expert, resizing policies may rely only on runtime feedback. The most relevant strategies tend to optimize a local metric: make the instant speed of the different contexts converge to the same value, maximize the throughput of the platform (i.e. assign a resource to the context which uses it in the most efficient way), etc.

A possible strategy that can resize scheduling contexts without information coming from the application is one that aims at balancing the instant speed of the parallel kernels. Considering that we have no initial guess about the total workload which will be processed by each context, the instant throughput of each context is determined based on computations that have already been completed by the context. The main idea is to divide the execution of the application into several intervals. The hypervisor monitors the performance of the resources in the last interval and tries to adjust the contexts so that they execute at the same speed in the next interval. For this purpose we use the non-linear program given by Equation (**??**), where we use a dichotomy on the value of the variable t_{min}.

In this non-linear program C denotes the set of contexts, W is the set of workers, $x_{w,c}$ is a boolean variable denoting whether or not a worker w belongs to the context c. $s_{w,c}$ is the instant speed of the worker w in the context c, $\theta_{w,c}$ is the number of flops needed to be executed by the worker w in context c out of the total number of flops θ_c. Equation (**??**) expresses the fact that each worker has to execute the number of flops assigned by the context it belongs to before the total execution time t_{min}. Furthermore, each worker has to be part of exactly one context and to have some flops assigned to it if it is chosen to belong to that context. Finally, it is important to notice that the complexity of this approach is important. As for the Equation (3.2), the dichotomy process requires the execution of the linear program several times until the good solution is found. Finally, it is important to emphasize that the approach described in this section can be improved if the application provides some high level information about its execution. For instance, if the application is able to provide the proportions of workload assigned to each context, it is straightforward to update Equation (**??**) to take into account such informations. We will refer at this kind of strategies as *instant speed based* resizing policies and to this one in particular as `ISpeed` (Instant Speed)

3.5 Implementation

The hypervisor is implemented as a plugin to the runtime system. Currently on top of StarPU (see Figure 3.6), this plugin can be easily adapted to any other runtime system whenever the latter manages a similar interface of the scheduling contexts.

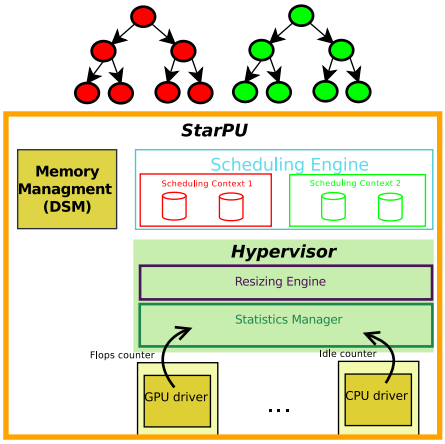


Figure 3.6: Placing the Hypervisor in StarPU

The hypervisor is a tool having a direct interaction with the application, with the runtime and managing an open platform for resizing policies.

3.5.1 Collaborating with the application

In order to allow the hypervisor to collect information, the application must first initialize it (`sc_hypervisor_init`). At this point a connection between the runtime and the hypervisor is established. Clearly, the application is also responsible for terminating the hypervisor whenever the application does not need it any more (`sc_hypervisor_shutdown`).

The application collaborates with the hypervisor by indicating which scheduling contexts can be monitored and resized. Therefore contexts are registered (`sc_hypervisor_register_ctx`) to the hypervisor and unregistered (`sc_hypervisor_unregister_ctx`) when the application requires no more monitoring or resizing.

Complicated high performance applications usually have already been strongly optimized and thus they can sometimes provide useful low level information. For instance, allowing the application to require the redistribution of the resources over the scheduling contexts at a precise point in the execution is a flexible way of taking into account the algorithmic properties of the application. Moreover, scheduling contexts can be directly modified (by adding, removing or moving processing units from contexts).

A strong interaction between the application and the hypervisor is required (see Figure 3.7) whenever the application is irregular and the entire workload of the contexts cannot be estimate at the beginning of the execution. Therefore, the hypervisor handles the updates of number of flops coming from the application (`sc_hypervisor_update_elapsed_flops` or `sc_hypervisor_update_total_flops`).

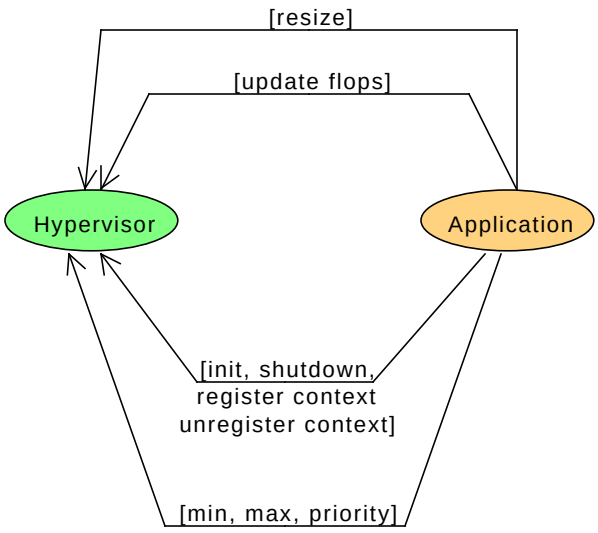


Figure 3.7: Collaborating with the application

The hypervisor is a tool adapted to listen to different hints coming from the application like the minimum or the maximum number of processing units a context could need, the priority a context should have in front of another, the limit of time after which a processing unit can be considered idle in a context, or the relevant interval of time to compute the speed of a context.

3.5.2 Collaborating with the runtime

On the other hand, the hypervisor collaborates with the runtime along two axis (as we can see in Figure 3.8): by letting the runtime transmit information throughout callbacks and by observing directly different performance counters the runtime system, in our case StarPU, provides.

The runtime system notifies the hypervisor whenever certain events arise (see Figure 3.9). By means of these callbacks the hypervisor collects performance information and also triggers the resizing process if certain conditions are satisfied.

Some situations require precise values of the performance counters that the callback events may delay to provide. In these cases the hypervisor interrogates directly StarPU about the hierarchy of the scheduling contexts, the processing units a context has as well as the number of ready tasks of a context at a precise moment.

In order to structure the monitored information of each context, the hypervisor manages a wrapper for each scheduling context, updating any information concerning the execution of the context and of their processing units. For instance the wrapper maintains statistics about the time the context started executing its tasks, the idle time and the execution time of each processing unit in the contexts, the number of pushed, poped and submitted tasks to/from the context, the number of elapsed and total flops of a context, etc.

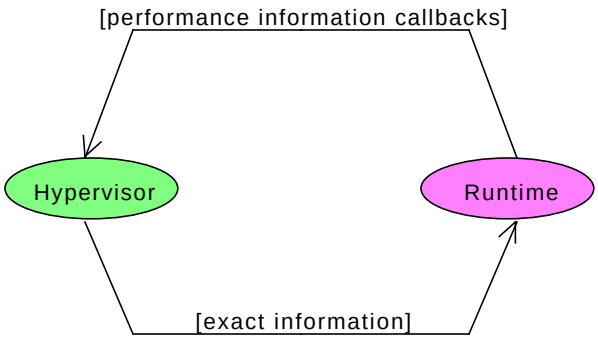


Figure 3.8: Collaborating with the runtime

```
struct starpu_sched_ctx_performance_counters
{
        void (*notify_idle_cycle)(unsigned sched_ctx_id, int worker, double idle_time);
        void (*notify_poped_task)(unsigned sched_ctx_id, int worker);
        void (*notify_pushed_task)(unsigned sched_ctx_id, int worker);
        void (*notify_post_exec_task)(struct starpu_task *task, size_t data_size,
                                      uint32_t footprint, int hypervisor_tag, double flops);
        void (*notify_submitted_job)(struct starpu_task *task, uint32_t footprint,
                                     size_t data_size);
        void (*notify_empty_ctx)(unsigned sched_ctx_id, struct starpu_task *task);
        void (*notify_delete_context)(unsigned sched_ctx);
}
```

Figure 3.9: The hypervisor updates performance information throughout runtime callbacks

3.5.3 Resizing policy platform

The policy needed to resize the scheduling contexts may vary depending on the application, on the information it can provide and that can be processed by the hypervisor without important overhead. For instance, some applications can estimate the workload of the scheduling contexts and policies like the one applying the Equation (3.1) can be used. Others can provide the number of tasks of each type, and if the number of types of tasks is too large the Equation (3.2) can become very expensive.

We provide several policies for different classes of applications:

1. idle policy: idle processing units are moved from their current context to a different one.
2. `On-demand` policy: the programmer tags certain tasks, and after executing those tasks the hypervisor is triggered to move idle processing units to other contexts.
3. `FEFT` policy: a minimum completion time based strategy applying the Equation (3.1) to redistribute resources over all or a part of scheduling contexts.

```
struct sc_hypervisor_policy
{
        /* name of the strategy */
        const char* name;

        /* indicate if it is a policiy create by the user or not */
        unsigned custom;

        /* Distribute workers to contexts even at the begining of the program */
        void (*size_ctxs)(unsigned *sched_ctxs, int nsched_ctxs , int *workers,
                          int nworkers);

        /* Require explicit resizing */
        void (*resize_ctxs)(unsigned *sched_ctxs, int nsched_ctxs , int *workers,
                            int nworkers);

        /* Take a decision when the worker was idle for another cyle in this ctx */
        void (*handle_idle_cycle)(unsigned sched_ctx, int worker);

        /* Take a decision when another task was pushed on this worker in this ctx */
        void (*handle_pushed_task)(unsigned sched_ctx, int worker);

        /* Take a decision when another task was poped from this worker in this ctx */
        void (*handle_poped_task)(unsigned sched_ctx, int worker,struct starpu_task *task,
                                  uint32_t footprint);

        /* Take a decision when the worker stoped being idle in this ctx */
        void (*handle_idle_end)(unsigned sched_ctx, int worker);

        /* Take a decision when a certain task finished executing in this ctx */
        void (*handle_post_exec_hook)(unsigned sched_ctx, int task_tag);

        /* Take a decision when a job was submitted in this ctx */
        void (*handle_submitted_job)(struct starpu_codelet *cl, unsigned sched_ctx,
                                     uint32_t footprint, size_t data_size);

        /* Take a decision when a certain ctx was deleted */
        void (*end_ctx)(unsigned sched_ctx);
};
```

Figure 3.10: Structure of a new resizing policy

4. `TEFT` policy: a minimum completion time based strategy applying the Equation (3.2) to redistribute resources over all or a part of scheduling contexts.

5. `ISpeed`: an instant speed based policy applying the Equation (**??**) to redistribute resources over all or a part of scheduling contexts.

Whenever these policy are not adapted to the requirements of the application, we provide a platform allowing the programmer to implement new policies and adapt the resizing decisions to the application's properties. Therefore, in order to create a new policy the programmer must define the set of functions (see Figure 3.10) to be executed when specific events occur during the execution.

3.6 Execution model

We provide in Figure 3.11 an example illustrating how the programmer can use the hypervisor, by indicating the scheduling contexts to be resized as well as different constraints for the resizing

```
/* select an existing resizing policy */
struct hypervisor_policy policy;
policy.custom = 0;
policy.name = "idle policy";

/* initialize the hypervisor and set its resizing policy */
sched_ctx_hypervisor_init(policy);

/* register context 1 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx1);

/* register context 2 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx2);

/* define the constraints for the resizing */
sched_ctx_hypervisor_ctl(sched_ctx1,
   HYPERVISOR_MIN_CPU_WORKERS, 3,
   HYPERVISOR_MAX_CPU_WORKERS, 7,
   NULL);
```

Figure 3.11: Configuration of the hypervisor.

process. We can for example, indicate the minimum and the maximum number of workers allowed in a certain context. For this we use the function `sched_ctx_hypervisor_ctl` in order to have the resizing process constrained by this interval.

3.7 Evaluation

In the following section we present a set of experiments which enlighten the importance of a dynamic mechanism, necessary to polish the initial distribution of the resources. We show how the hypervisor improves the performance of the application by taking decisions of when and what resources to move from one context to another.

3.7.1 Experimental architectures

The parallel library we use in this section provides CPU and GPU implementation of their codelets, therefore the following experiments are only presented on `mirage` heterogeneous architecture (see Appendix B for description).

3.7.2 Experimental scenarios

We evaluate the behavior of the hypervisor by creating synthetically worst-case scenarios determined empirically. In these scenarios we simulate applications arriving at a point in their execution where the distribution of the resources is no longer efficient.

These scenarios consist in composing parallel kernels of the MAGMA-MORSE library implemented on top of StarPU (see Appendix A for description). The studied applications, composing several such kernels, are implemented as a graph of tasks and represent malleable parallel tasks, allowing flexible dynamic redistribution of resources.

3.7.3 On-demand resizing

First of all we show that the hypervisor is a tool built for the programmer, able to receive input from the application in order to better react to its requirements and thus improve its overall performance. In the following scenario, we show that the programmer can thus use the hypervisor to optimize his application, by asynchronously triggering the resizing process.

In this section we refer to the following parallel codes:

- MAGMA-MORSE kernel of the Cholesky Factorization executed on a matrix of 29760 x 29760 (30 000 x 30 000) elements using a block size of 960 x 960 elements (referred to as `[Large_Cholesky2]`)
- MAGMA-MORSE kernel of the Cholesky Factorization executed on a matrix of 14976 x 14976 (15 000 x 15 000) elements using a block size of 192 x 192 elements (referred to as `[Small_Cholesky2]`)

We now consider the scenario where we compose different kernels executing Cholesky factorizations. In this particular case we start two different streams of parallel kernels. The first one is composed of three serially executed `[Large_Cholesky2]` and the second one is composed of three serially executed `[Small_Cholesky2]`. In this scenario, the first stream is executing efficiently over the entire set of resources and from time to time the second stream steps in and interferes with the parallelism of the first one.

We evaluate two situations. In the first one both streams are interleaved and compete thus for the computing resources. In Figure 3.12 (first 9 lines correspond to the progress of CPUs and the 3 last lines represent the progress of the GPUs, each color depicts a context) we present the second one, where the two stream are separated in scheduling contexts. Initially all the platform is used by the large stream context and no computing resources is assigned to the small stream context. Further on, when the latter steps in to execute a parallel task (respectively when it finishes executing the parallel task), the application triggers the resizing process and indicates to the hypervisor that the corresponding context needs (resp. releases) some resources (4 CPUs).

	Execution time
Overlapping contexts	19.7 s
On-demand strategy	17.2 s

Table 3.1: Application driven resizing policies

In Table 3.1 we notice an improvement of 2 seconds by resizing dynamically the contexts when the small stream steps in. We see that letting the two streams blend over the same resources has an important impact on the performance of the overall application. Assigning periodically some resources to the small stream improves the locality of `[Small_Cholesky2]` and additionally limits the negative effects on the cache management of `[Large_Cholesky2]`.

3.7.4 Automatic resource distribution

We advocate the hypervisor as a tool helping the programmer to better optimize the applications. One choice is then to do it directly by triggering the resizing process at certain points of the execution in order to allow an explicit allocation of resources, as we have seen in the previous case.

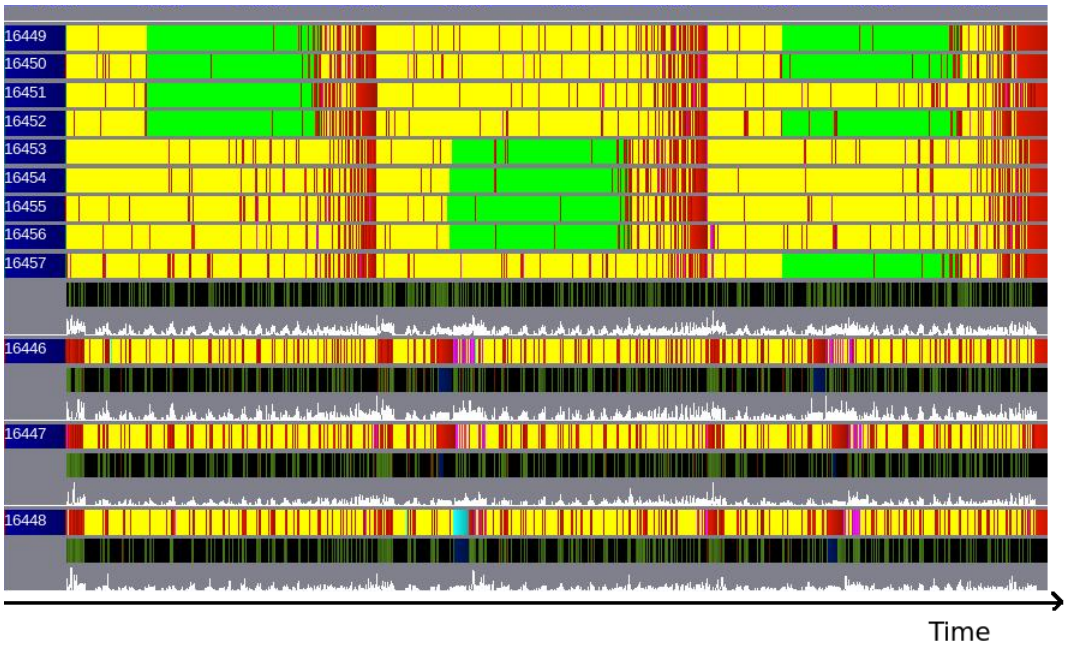


Figure 3.12: Reallocate resources to `[Small_Cholesky2]` and `[Large_Cholesky2]` streams using the On-demand strategy

However, the hypervisor can also automatize this process and compute the moment to trigger the resource allocation as well as it can decide on the actual distribution of computing resources.

We show thus in the following section how the hypervisor can do this and the benefits of allowing the application to delegate some tasks to the runtime tools. Further on in this section we will refer to the following parallel codes:

- MAGMA-MORSE kernel of the Cholesky Factorization executed on a matrix of 19968 x 19968(20 000 x 20 000) elements using a block size of 192 x 192 elements (referred to as `[Small_Cholesky]`)

- MAGMA-MORSE kernel of the Cholesky Factorization executed on a matrix of 39360 x 39360 (40 000 x 40 000) elements using a block size of 960 x 960 elements (referred to as `[Large_Cholesky]`)

We create an application composing the `[Small_Cholesky]` with `[Large_Cholesky]` kernels. In this scenario we observe that the block size of the tasks is different for the two kernels, having thus tasks executing more efficiently on the CPUs than on the GPUs. We conduct a first experiment where we statically detect an ideal initial distribution of resources for the two kernels of the application. Further on, we conduct several experiments where we use this information in order to assign an unadapted set of computing resources to the two kernels. We show that the hypervisor is able to detect and correct this initial distribution such that the overall execution time of the application is close to the one where the resources were correctly allocated from the beginning.

3.7.4.1 Computing ideal initial distribution with TEFT

In this section we use the TEFT resizing strategy to allocate the computing resources. Based on off-line statistics concerning the workload of the kernels and of their execution time on the mirage configuration, TEFT is able to compute statically the resource distribution for the two scheduling contexts. Thus, we obtain, that the best resource allocation corresponds to giving most of the CPUs to the [Small_Cholesky] at all the GPUs to the [Large_Cholesky].

We conduct the following experiments to detect empirically that the ideal distribution provided by TEFT actually corresponds to the most efficient one. In this scenario the resources for each context are allocated statically at compile time, and no resizing is allowed during the execution of the two. However, when one of the two contexts finishes, its computing resources are redistributed to the remaining one.

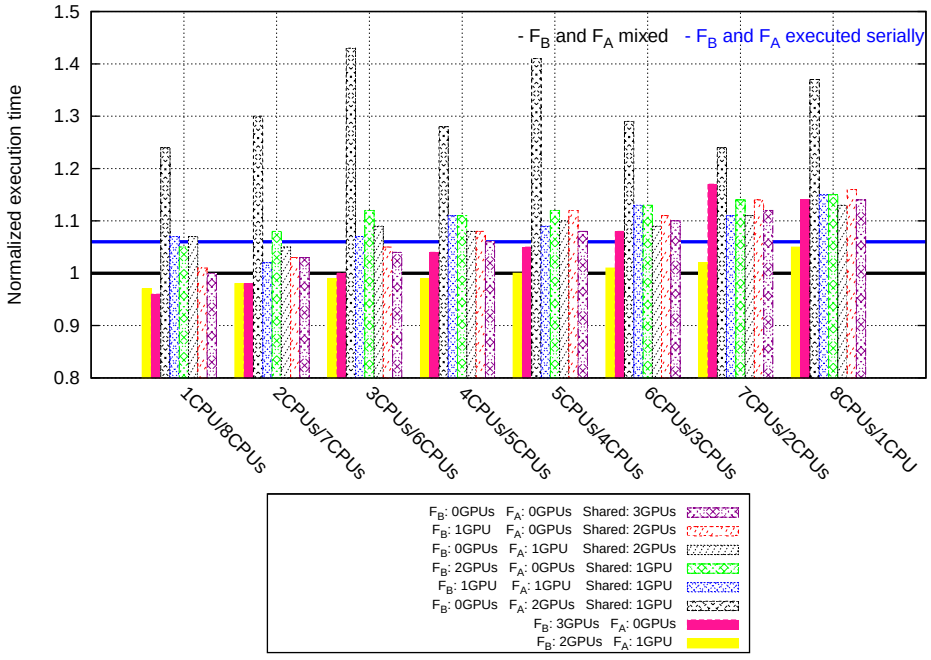


Figure 3.13: Composing [Large_Cholesky] (F_B) and [Small_Cholesky] (F_A)

We evaluate in Figure 3.13 different configurations, and we normalize their execution time with respect to the one using no contexts with an execution time of 32.19 s. Therefore, we observe that the distribution favoring [Small_Cholesky] to have all the CPUs and [Large_Cholesky] to have all the GPUs is the most efficient one. This is mainly due to the size of the matrices (more computation going to the GPU) as well as to the block size of their tasks, the 192 block size being more efficient on the CPU and 960 one on the GPUs.

It is interesting to notice that allowing GPUs to be shared between the two kernels has a great

impact on the execution time. This can be explained by the fact that interfering with the execution on the GPU of the large workload kernel implies evicting data on the accelerator in favor of a small computation.

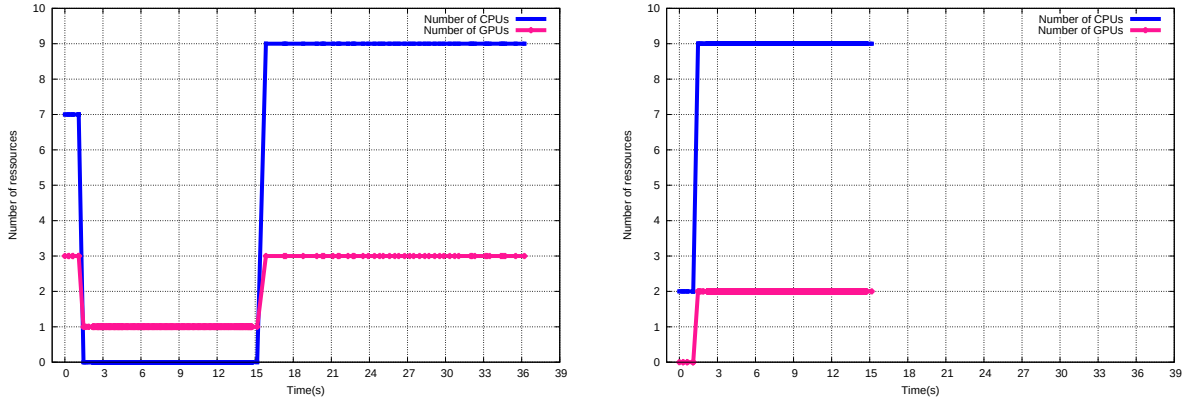
Therefore, we observe that the best resource allocation for the application composing `[Small_Cholesky]` and `[Large_Cholesky]` is 3 GPUs and 1 CPU to `[Large_Cholesky]` and 8 CPUs for `[Small_Cholesky]` with an overall execution time of 30.61 s. We observe that the resulting distribution corresponds to the one provided by `TEFT`.

In the following sections we evaluate the reactivity and the efficiency of the Hypervisor when dealing with worst-case scenarios. In order to do this we allocate an unadapted set of processing units to the scheduling contexts executing the kernels, that is 3GPUs and 7 CPUs to `[Large_Cholesky]` and only 2 CPUs to `[Small_Cholesky]`. We verify in the following section if the Hypervisor is able to detect on time or not this error and dynamically reallocate the computing resources.

3.7.4.2 No input from the application: Instant speed based policy

We start analyzing the first situation where the programmer does not give any information concerning the workload of the application and of the corresponding kernels. The hypervisor uses the instant speed policy (`ISpeed`) to resize the scheduling contexts. This type of policy is especially designed for applications that are not able to provide this kind of information. It aims at balancing the instant speed of the parallel kernels without considering any of their properties.

In Figure 3.14 we can see the use of the computing resources in each context during the execution of the application.



(a) Resources distribution in Context 1

(b) Resources distribution in Context 2

Figure 3.14: Resource distribution over the contexts when using the instant speed based policy.

We see in Figure 3.14 that the decisions regarding the distribution of resources do not lead to the optimal distribution of resources, that is all the GPUs to `[Large_Cholesky]` and all the CPUs to `[Small_Cholesky]`. The policy has no information concerning the workload of the scheduling contexts and decides to allocate resources such that the contexts run at the same speed. Therefore, the factorization of the large matrix is slowed down by forcing it to execute on a single GPU while

the one of the small matrix is accelerated by executing on 2 GPUs and all the CPUs. However, the fact that `[Small_Cholesky]` has a smaller workload makes it finish earlier than `[Large_Cholesky]`. This finally leads to a serial execution of the two kernels, that was shown to be an inefficient solution (see blue line in Figure 3.13).

The main reason of this inefficient execution, as in the case of the serial one, is that the kernel executing `[Small_Cholesky]` has a tile size particularly well suite for the CPU only, forcing it to execute on the GPUs does not accelerate its execution. Meanwhile, the kernel executing a factorization on the large matrix suffers an important penalty that is not able to catch up in the end of the execution when it has all the processing units.

This strategy lacks some information concerning the future behavior of the kernels. Knowing even a fraction of the total workload of the contexts would give a more long term view to the strategy, making it more efficient.

3.7.4.3 Interacting with the application through workload information

Without any information coming from the application the hypervisor can hardly guess how the application would behave on long term. Providing an estimated workload of the parallel kernels would allow detecting what the corresponding contexts would require as resources. Therefore, we consider two strategies `FEFT` and `TEFT` that take into account the workload information in order to reallocate computing resources to the scheduling contexts.

We have seen in the previous section that `TEFT` is able to provide a good distribution of the resources statically. It can also be used to compute the resource redistribution dynamically. However, we observe in Table 3.2 that the cost required to do this computation is significantly higher than the one needed by `FEFT`.

	FEFT	TEFT
worst case scenario	1.26	2321.57
best case scenario	0.07	16.48

Table 3.2: Cost of a redistribution of resources (ms)

The `TEFT` policy may sometimes be very time consuming, and this is mainly due to the high precision of the dichotomy algorithm. Depending on the situations `TEFT` can provide a good solution with an overhead which can reach a prohibitive cost (up to 2 s). Indeed, for an execution time of 30s (which is the case in our experimental scenario), an overhead of 2 s generated by a single redistribution will deteriorate the overall performance of the application. Therefore, in the following section we only consider `FEFT`.

Allowing a good redistribution of resources is strongly related with the moment when this is required. Depending on the reactivity but also on the correctness of the monitored information the resizing strategy can be more or less efficient.

Therefore, we measure the impact of dynamically detecting when an application is not executing efficiently by using the idleness metric. We focus on the following scenario: when an inefficient execution is detected we trigger the `FEFT` resizing policy.

Detect idle resources The idleness parameter describes the inactivity time since that last task finished executing on a certain processing unit. In Figure 3.15 we vary this parameter in order to

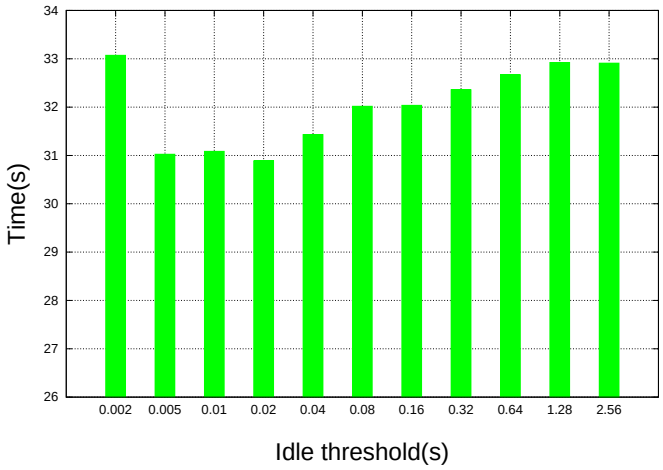


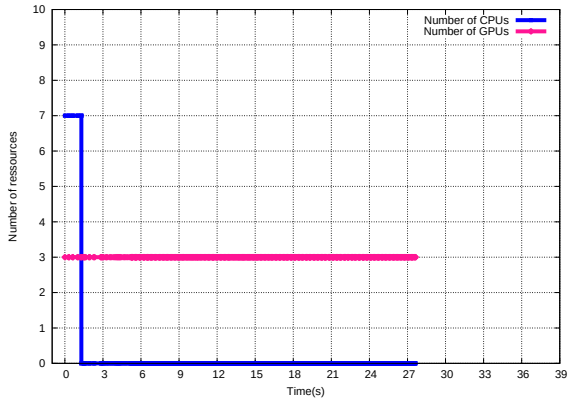
Figure 3.15: Choosing the idle threshold to trigger resizing.

determine the impact of the reactivity of the hypervisor on the execution time of the application. As expected, when the value of this parameter exceeds a threshold, the execution time is negatively impacted. This is mainly due to the fact that resizing decisions are taken too late letting the resources being idle for too long. On the other hand when the threshold has a small value the resizing decision is taken early enough such that the rest of the execution is not altered. In this case the reactivity of the hypervisor is high and this may imply a certain overhead that cancels a part of the performance improvement generated by the resizing.

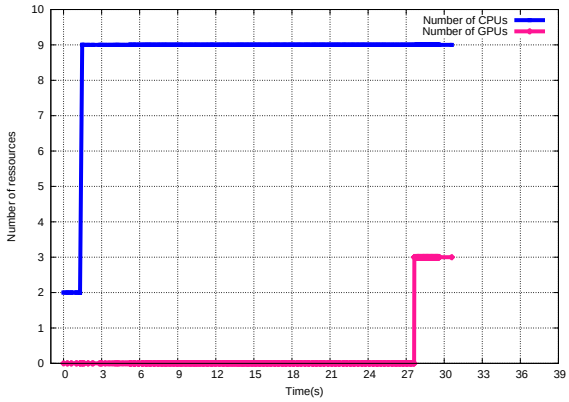
`FEFT` resizing strategy. We now analyze the behavior of the resizing strategy once it was triggered. We consider here one scenario, whose execution time is used to compute the average values presented in Figure 3.15. We present its execution trace, when the resizing process is triggered for an idle threshold of 0.02 s, using the `FEFT` policy.

The two traces in figure 3.16 complete each other, as they represent the progression of the application from the point of view of each context. They present the evolution of the number of computing resources of each type in the two contexts during the execution. We can see that at the beginning they have an incorrect distribution of resources, but once the Hypervisor determines an incorrect behavior and triggers the resizing process the scheduling contexts resize dynamically.

Here we have a case where the `FEFT` strategy takes the right decision at the right moment. By using the idleness criteria, the hypervisor is able to detect soon enough the need to resize the contexts and thus the contexts are able to benefit from the new distribution of resources without an important impact on the performance of the application. After 1 s from the beginning of the execution the value of the speed of the processing units can have a good value, which leads to a good resizing decision.



(a) Resources distribution in Context 1



(b) Resources distribution in Context 2

Figure 3.16: Resource distribution over the contexts when using the completion time based policy.

3.8 Discussion

Scheduling contexts represent a solution for the oversubscribtion problem. Isolating the simultaneously executing parallel kernels allows avoiding sharing the computing resource. However, a static resource allocation of computing resources may lead to the undersubscription problem if kernels can not scale or use all the resources allocated to them. In this chapter we focused on a set of methods to allocate resources according to the specificities of the kernels.

We focused on two completion time based resizing strategies, `FEFT` and `TEFT`, both based on linear programs. The `FEFT` considers a rough workload information and an on-line monitored speed of the computing resources. Therefore, solving the corresponding linear program is rather straightforward. On the other hand, the `TEFT` strategy relies on a precise computation of the necessary resource allocation. It is a good theoretical approach as it gives a correct distribution of resources. For the heterogeneous architectures it is very reliable as it takes into consideration the affinity of the tasks to certain types of computing resource, since the input of the `TEFT` linear program is based on the StarPU performance model. Therefore, solving the `TEFT` linear program is costly, even more because it relies on a dichotomy algorithm. Indeed, such an approach can be more adapted to larger executions, needing a good precision (for instance with tasks having a large granularity) and being able to pay the cost of the redistribution.

Despite the lack of precision, `FEFT` features less overhead and allows thus the hypervisor to be more reactive and to dynamically adapt the resource distribution. This improves the performance of the application as it can benefit from the resizing decision sooner. Therefore, in the next chapter we will apply and evaluate these techniques on more complex applications.

Chapter 4

Dynamic resource allocation for a sparse direct solver

In the previous chapters we proposed a solution to the composability problem, that is to isolate parallel codes into *scheduling contexts*. We used the *hypervisor* to detect under-used computing resources and to dynamically reallocate them in order to match the capacity of the machine and the parallelism of the application. We validated our approach through simple case scenarios that mimicked the behavior of complex applications.

We now treat a complex application, a sparse direct solver more precisely, facing an irregular parallelism. Dynamically adapting the irregular structure of the solver to the complex architectures of the machines becomes a very difficult issue. By using a data-flow approach generated by the DAG of sequential tasks the application makes profit of a fine granularity and an increased parallelism. However, when the DAG becomes significantly big, the overhead of the runtime starts being noteworthy and the complexity of the scheduling decisions becomes a real problem. Our approach to this problem is to pack sub-DAGs into malleable tasks such that the outer scheduler has fewer tasks to manage as well as the inner one that is thus able to apply local optimizations. Both become more scalable, in charge with less tasks to schedule and less resources to assign them to.

Nevertheless, it is important to notice that programmers usually can provide high level information (like numerical properties, structure of the DAG, etc.) in order to improve the performance of the application. On the other side, the runtimes have important low level knowledge of the hardware properties (like types of processing units, memory distribution, etc.) that they usually rely on when assigning resources to the tasks. Our purpose is to build a bridge between the two sides, applications and runtimes, and allow applications to provide information to the runtime and to have this latter one consider this information when performing the execution of the tasks. Therefore in this chapter we propose to use the *scheduling contexts* as a runtime tool able to pack the sub-DAGs. Further on, we use the *hypervisor* to consider the input coming from the application and thus allocate resources to scheduling contexts.

We focus on the `qr_mumps` sparse direct solver [8], facing irregular parallelism as well as a complex algorithmic structure and we investigate how to push further the interaction between the application and the runtime system. We show that by allowing `qr_mumps` provide informations about the structure of its task graph to the hypervisor, this latter is able to perform a better mapping on the underlying topology and the whole stack behaves better. With an improved interaction

scheme this approach enhances the locality and provides a solution to leave the application guide the behavior of the underlying runtime system.

4.1 qr_mumps

The sparse QR factorization method which we consider in this chapter is the multifrontal method. Like other direct methods, the multifrontal algorithm is based on an elimination tree [51], which is the transitive reduction of the filled matrix graph and represents the dependencies between the elimination operations. The task graph is built during the *analysis* phase where all the preprocessing algorithms are applied. The algorithm continues with the actual numerical *factorization* and the solve steps.

At the factorization step, we consider the DAG as a way to express the dependencies between the computational tasks: each node i of the tree is associated with k_i unknowns of the original matrix and represents an elimination step of the factorization. The coefficients of the corresponding k_i columns and all the other coefficients affected by their elimination are assembled together into a relatively small dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node.

The multifrontal QR factorization consists in a tree traversal in a topological order (i.e., bottom-up) such that, at each node, two operations are performed. First, the frontal matrix is **assembled** by stacking the matrix rows associated with the k_i unknowns with uneliminated rows resulting from the processing of child nodes. Second, the k_i unknowns are eliminated through a **complete** QR **factorization** of the front; this produces k_i rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block* formed by the remaining rows and that will be assembled into the parent front together with the contribution blocks from all the front siblings. A detailed presentation of the multifrontal QR method, including the optimization techniques described above, can be found in Amestoy *et al.* [12].

The baseline of the `qr_mumps` solver, is the parallelization model proposed by Buttari [21] where frontal matrices are partitioned into block-columns, which allows us to decompose the workload into fine-grained tasks. Each task corresponds to the execution of an elementary operation on a block-column or a front; five elementary operations are defined: 1) the **activation** of a front consists in computing its structure and allocating the associated memory, 2) **panel** factorization of a block-column, 3) **update** of a block-column with respect to a previous panel operation, 4) **assembly** of the piece of contribution block in a block-column in the parent front and 5) **cleanup** of a front which amounts to storing the factors aside and deallocating the memory allocated in the corresponding activation. These tasks are then arranged into a DAG where vertices represent tasks and edges the dependencies among them. Figure 4.1 shows an example of how a simple elimination tree (on the left) can be transformed into a DAG (on the right); further details on this transition can be found in the paper by Buttari [21] from which this example was taken. The execution of the tasks is guided by a dynamic scheduler which allows the tasks to work asynchronously.

4.2 qr_mumps on top of StarPU

Recently, Buttari *and al.* have proposed a modified version of the `qr_mumps` software [8], designed on top of the StarPU runtime system. The main idea is to have StarPU perform the actual execution of the DAG of tasks by ensuring that the dependencies are satisfied at execution time and that the memory management is coherent.

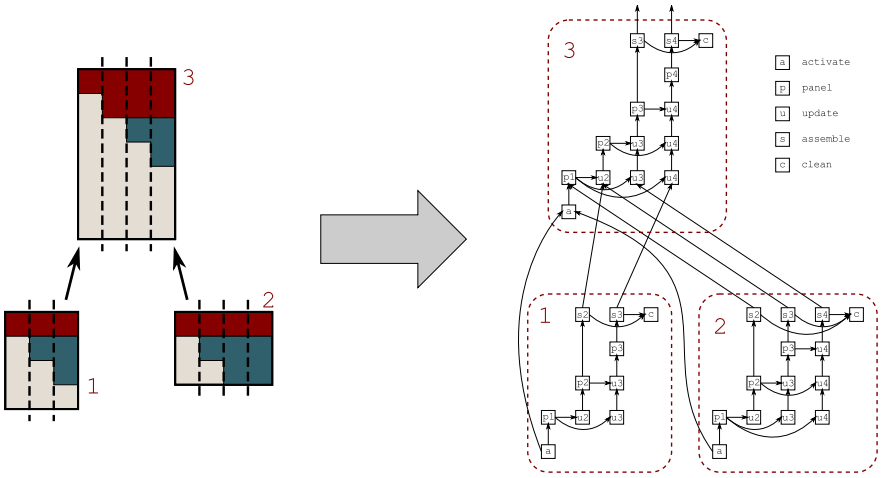


Figure 4.1: An example of how a simple elimination tree with three nodes is transformed into a DAG in the `qr_mumps` code. Vertical, dashed lines show the partitioning of fronts into block-columns. Dashed-boxes group together all the tasks related to a front.

From the algorithmic point of view `qr_mumps` provides to StarPU a tree of DAGs working on dense matrices (the fronts). StarPU receives a large DAG as input and blindly tries to schedule all the tasks as efficiently as possible given the scheduling policy indicated. Not being aware that the tasks of each DAG share common data, tasks can execute on any available worker.

Moreover, the DAG associated to the factorization of medium to large size matrices can have hundreds of thousands of tasks. Not only the resulting overhead of submitting all these tasks to StarPU can be important but it can also require too much memory to allocate at the beginning of the factorization. The solution was to progressively submit tasks by means of the activation tasks. Thus, they are responsible not only for allocating memory but also for submitting the tasks for their assembly and factorization (panel, update, assembly and cleanup).

We propose to make `qr_mumps` use the *scheduling contexts* previously described as a tool available in the StarPU runtime release. The solver can easily describe the notion of tree of DAGs to the runtime as a tree of malleable tasks. By considering groups of tasks together, the runtime can favor locality and isolate them on sets of computing resources sharing the same memory bank. In order to be able to dynamically adapt to the fact that unexpected tasks can be submitted at some point (progressively submitted by means of the activation tasks) and that the precise workload is not available before starting the factorization, we propose to use the *hypervisor* in order to dynamically adapt the scheduling contexts to the new requirements in term of parallelism.

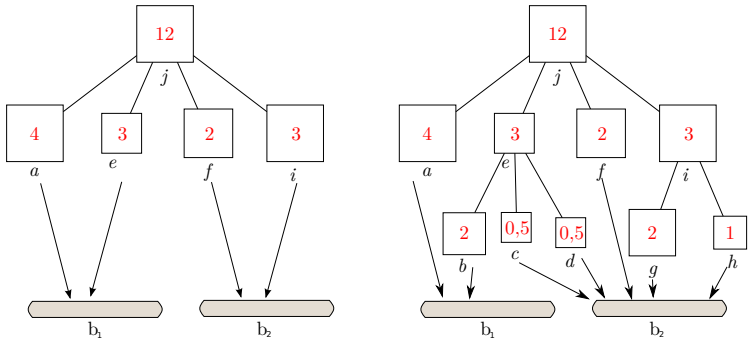
4.3 A partial task graph mapping strategy

At the first level of the tree the number of fronts can be significant and very heterogeneous in term of size. Therefore, assigning the DAG of tasks working on a front to a scheduling context may lead to having a large number of contexts of very different workloads.

We propose in this section to use a partial mapping strategy during the analysis phase that aims at ensuring load balancing between the scheduling contexts.

The idea is to use a classical static scheduling algorithm, namely *proportional mapping* [47], for

tree-shaped task graphs and to adapt it to our context. This algorithm uses a local mapping of the processors to the nodes of the task graph. To be more precise, starting from the root node to which all the processors are assigned, the algorithm splits the set of processors recursively among the branches of the tree according to their relative workload until all the processors are assigned to at least one subtree (these number of processors correspond to the red numbers associated to the nodes of the tree given in Figure 4.2). This algorithm is characterized by both a good workload balance and a good locality. It is important to note that the approach described below can be adapted to other scheduling algorithms for trees of malleable tasks like the ones presented in [48, 41].



(a) First step .

(b) Second step .

Figure 4.2: Mapping algorithm.

We propose to upgrade the stopping criterion for the top-down process of the proportional mapping so that the locality will be enhanced on multicore systems. Our improved version needs an additional input which represents a number of *bundles* (each bundle represents a group of computational resources). The bundle concept is linked to the architecture of the underlying platform: a bundle corresponds to the set of resources sharing a given level of the memory hierarchy. Generally the number of bundles may be the number of actual processor sockets or even the number of memory banks.

During the top-down process, the current layer of nodes and their corresponding number of computational resources are assigned to the bundles in a sorted-item bin-packing way: we traverse the layer in the topological order and associate the nodes to the bundles until they are full.

For example, in Figure 4.2(a), we use as input to the algorithm two bundles (b_1 and b_2) containing 6 processors each. At the first step of the algorithm, the bundle b_1 is associated with nodes a and e while the bundle b_2 is associated with f and i. This leads the bundle b_1 to exceed its capacity of 6 and thus the algorithm needs to go further using the top-down scheme (note that node a is a leaf in this example). In the next step (see Figure 4.2(b)), b_1 is associated to nodes a and b and b_2 to nodes c, d, f, g and h leading all the bundles to be perfectly filled and thus stopping the top-down process.

More generally, during the top-down process, a layer of nodes in the tree is accepted if the projection of the nodes of the tree over the bundles satisfies the constraint of each bundle. Naturally, for trees which can be met in sparse direct solvers, the constraint (i.e. the number of resources) associated with each bundle needs to be slightly relaxed to ensure that a feasible configuration can

be found: a tolerance parameter may be used to check the acceptance criterion.

Once the top-down process has been completed, we associate each subtree rooted at a node above the accepted layer (malleable task) with an abstract context associated to the set of resources resulting from the proportional mapping. This produces a hierarchy of contexts which are given to the runtime system together with the amount of work (resp. the number of resources) associated with each context before the factorization begins. Before starting the factorization, `qr_mumps` can only provide an estimation of the workload of each context, as the actual value depends on the task size which is determined during the factorization. However, dynamic updates with the real workload is provided during the factorization.

Nevertheless, the previously computed number of resources could be rational. In this case, the resource located at a border of a scheduling context is shared with the neighboring one (the runtime system uses time-sharing of the resource among the contexts it belongs to). From the implementation point of view, this represents a slight modification of the factorization in the sense that the scheduling context to which a given task belongs is an attribute of the task. We introduce in the next section a dynamic management strategy for this hierarchy of scheduling contexts. Note that from a pure software point of view, the fact that the tasks are now assigned to a context represents a very marginal modification.

4.4 Build a hierarchy of scheduling contexts

We now step on the other side and consider how the runtime receives as input the tree of DAGs and how the hypervisor hierarchically resizes the scheduling contexts in order to enforce the locality of the parallel tasks of the application. Some ideas are borrowed from the Bubble Scheduling approach [57].

By describing the hierarchical parallelism at the application level, the runtime uses the scheduling contexts in order to isolate branches of the tree on sets of processing units. Thus, we can consider these branches malleable tasks.

The execution is a bottom-up traversal of the task graph where a parent node cannot be treated before its children have been processed. Thus, from the scheduling contexts hierarchy point of view, the active scheduling contexts correspond to a layer at the bottom of the tree. The layer of active scheduling contexts moves towards the root during the execution.

In the ideal situation, mapping the tree of contexts on the hierarchical architecture of the underlying platform leads to a perfect exploitation of the resources. However, this is not the case for real-life applications. Indeed, applications like `qr_mumps` usually deal with very irregular task-graphs where predicting the actual execution time is challenging. For this reason, mechanisms to dynamically step in whenever imbalance appears are mandatory.

To this effect we used the hypervisor in order to dynamically adapt the resource allocation over the scheduling contexts such that unexploited computing units can be used by other scheduling contexts. Our main constraint in this situation is to keep the computation local in the sense of the memory hierarchy and exchange resources only between neighboring scheduling contexts, sharing cache or NUMA node memory.

4.5 Hierarchical resizing of the Scheduling Contexts

Our approach to the locality problem is to reallocate the resources hierarchically and to take the resizing decisions locally at each level of the tree of the scheduling contexts. Contexts with the same parent access nearby data and as soon as they finish executing they provide contribution blocks (see Section 4.1) to their parent. The hypervisor enforces locality and allocates the resources such that this group of contexts finishes in the minimum amount of time. In other words, each level of contexts has its own deadline and the execution of the branches of the application progresses locally on the corresponding group of processing units. In Figure 4.3 each group of sibling contexts can exchange processing units as long as they finish their execution time before a certain deadline (e.g. D5, D6, etc.)

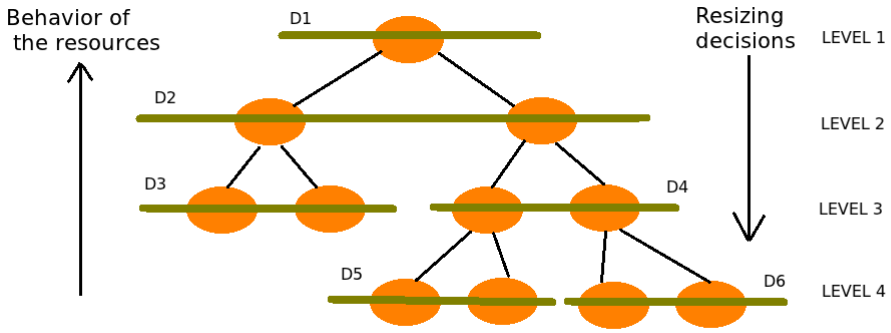


Figure 4.3: Resizing hierarchical contexts by having local deadlines

In order to keep the decisions consistent, resizing information (e.g. the speed of the processing units) is transmitted bottom-up and resizing decisions are taken at the higher level and then propagated downwards.

The hypervisor uses the `FEFT` strategy to decide the redistribution of resources. We recall the linear program solve by the strategy, described by the Equation 4.1 computing only the number of CPUs as `qr_mumps` currently provides only CPU implementation of the kernels. By using the workload value of the kernels provided by the application, in our case `qr_mumps` solver, this algorithm distributes proportionally the resources over the scheduling contexts. We obtain a rough computation of the number of processing units needed by each context such that the program ends its execution in a minimal amount of time.

$$\max\left(\frac{1}{t_{max}}\right) \text{subject to} \left(\begin{array}{l}\left(\forall c \in C, n_{\alpha,c} v_{\alpha,c} \geq \frac{W_c}{t_{max}}\right) \\ \wedge \left(\sum_{c \in C} n_{\alpha,c} = n_\alpha\right)\end{array}\right) \tag{4.1}$$

We recall that in linear program solving the Equation 4.1 C denotes the set of contexts, $n_{\alpha,c}$ represents the unknown of the system, that is the number of processing units that are assigned to a context c, W_c is the total amount of work associated to the context c, t_{max} represents the maximum amount of time spent by a context to process its amount of work, $v_{\alpha,c}$ represents the speed (i.e. floating point operations per second) of the processing units belonging to the context

c, n_α is the total number of processing units. Equation (4.1) expresses that each context should have the appropriate number of CPUs such that it finishes its assigned amount of work before the deadline t_{max}.

The hypervisor solves this equation several times during the execution such that it can consider and insert new collected information like: the speed of the processing units when executing a certain kernel, more precise values of the workload of the kernels or bounds for the number of allocated processing units. A good reactivity of the hypervisor is required as the `qr_mumps` solver may dynamically update the workload information associated with the scheduling contexts.

4.6 Upper bounds to the allocation of resources

When using the `FEFT` strategy the hypervisor computes the correct number of processing units required for a certain context given its workload. However, allocating these resources to the contexts may still lead to having idle workers. The problem comes from the fact that the malleable tasks are actually composed of elementary tasks that are sequential. When we have fewer elementary tasks than computing resources the malleable tasks tend to be rigid. Meanwhile, the linear program is not aware of this because it does not consider any granularity of the tasks forming the workload. Therefore, no matter the resources allocated to a context by means of the `FEFT` strategy, if there are not enough tasks to execute, the computation cannot be accelerated.

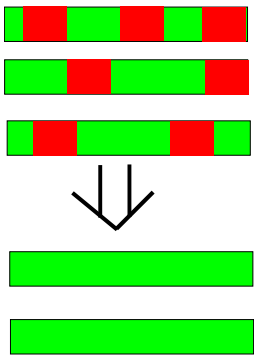


Figure 4.4: Use idle time to compute *max*

Algorithmic constraints at application level do not allow yet diminishing the granularity of the tasks (Florent Lopez in his PhD thesis is currently studying granularity related issues inside `qr_mumps`). Tasks are executing operations on block-columns, therefore diminishing the granularity would impact the intrinsic performance of the kernel . Moreover, more tasks would lead to additional overhead on the runtime.

A second solution would be to use parallel elementary tasks instead of sequential ones. By using the techniques presented in Chapter 2, the application could just use parallel BLAS operation instead of sequential ones and thus provide another level of parallelism to the existent hierarchical one. However, this solution can difficultly improve the behavior of the application. We cannot obtain a high parallelism by using parallel BLAS operations on block-column input data. The developers of `qr_mumps` are currently working on implementing the kernels to support a 2D structure of the block. They could potentially benefit from this approach.

We propose a runtime solution that does not interfere with the implementation of the application

and that adjusts to the provided information. We compute the idle time of the resources in a context in a previous period of time (we have seen in Chapter 3 how the hypervisor monitors the idle time) in order to predict how many resources that contexts would need in the next period. This value represents an additional constraint to the Equation 4.2, that is max_α, a rough information concerning the parallelism of the kernels.

$$\max\left(\frac{1}{t_{max}}\right) \text{subject to} \left(\begin{array}{l} \left(\forall c \in C, n_{\alpha,c} v_{\alpha,c} \geq \frac{W_c}{t_{max}}\right) \\ \wedge \left(\sum_{c \in C} n_{\alpha,c} = n_\alpha\right) \\ \wedge \left(\forall c \in C, n_{\alpha,c} < max_{\alpha,c}\right) \end{array}\right) \tag{4.2}$$

However, this solution (see Figure 4.4) can be limited by two problems. Firstly, if resources have been idle in the previous period we can easily compute an upper constraint for the number of resources in the next period, but if there has not been any idle resource it is difficult to say what would be the value of *max* in the next period. Therefore, we considered two possible solutions: we can either suppose that the parallelism of the kernel is determined by the number of ready tasks(whose dependencies are satisfied) available or we can add to the linear program this upper bound constraint only at certain moments. We chose the second solution for `qr_mumps`. We implemented an algorithm that considers this max only when the observed speed of the processing units is very far from an average speed value computed from the beginning of the execution of the application. We consider that only in such a case we require a more precise value of the distribution of the resources. Moreover, `qr_mumps` usually has a good speed at the beginning of the execution (tree parallelism is sufficient to ensure performance at that moment), thus this constraint would not even be necessary for this portion of execution.

The second problem is that there could be resources left unallocated as a consequence of the additional constraint to the Equation 4.1. Different solution may be considered, for instance not using them at all or splitting them between the contexts. In the `qr_mumps` situation we decided to share them between all the scheduling contexts, such that at any moment a context could benefit from them.

In Figure 4.5 we can see a simple example using 4 contexts (each line corresponds to the execution progress of a CPU, each color depicts a context and red traces the idle time) where the last few CPUs are shared between 4 contexts. We can see that scheduling contexts are being dynamically resized, each white line event representing the sign that a redistribution was triggered.

The maximum number of processing units needed by the leaf contexts is computed locally and this information is propagated hierarchically until the root where it is used as an upper bound in the linear equation (4.1).

4.7 Triggering the reallocation of resources

An important aspect in improving the execution time of an application is determining when resources are no longer efficiently used in their scheduling context, they are slow or even idle. We use the hypervisor as a tool that collects information concerning on one hand the behavior of the application with the provided distribution of resources and on the other hand the efficiency

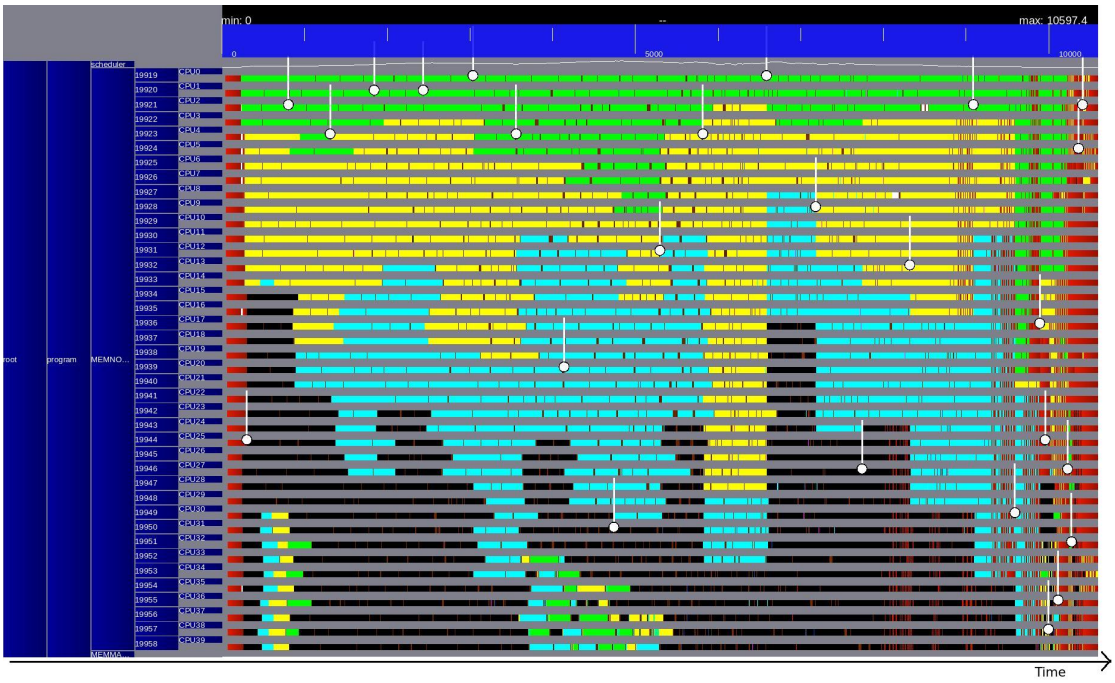


Figure 4.5: Example of `qr_mumps` solving a simple problem using 4 scheduling contexts over 40 CPUs

of the processing units when executing the parallel tasks. Therefore, the runtime is in charge with monitoring the application and providing the hypervisor information like: the moment a task started/finished executing, its workload, the time a worker spent without executing any task in a certain context. Meanwhile, the application is in charge of providing and adjusting dynamically the information concerning the workload of the application and of each task.

The hypervisor synthesizes this information and computes at some period of time the interval in which the processing units were idle and also the instant speed of a scheduling context or of a type of processing unit. This latter represents the number of flops executed by a context respectively a type of worker (CPU in our case) in a certain sample of time. The hypervisor uses the speed based criteria we presented in the previous chapter 3.3 and compares the observed value of the speed of the contexts with ideal speed resulting from the previous solution of the linear program (4.1). Whenever either the difference between the actual speed and the ideal one exceeds a threshold or the computing resources are in an idle mode for longer than a given limit of time we consider that the current distribution of processing units is not valid and the resizing process is triggered. Thus, the linear program (4.1) is solved relying on dynamically monitored values of the speed of the computing resources. The reactivity of the hypervisor is adapted to the irregularity of the problem. We have evaluated the best trade off between the need to resize the contexts and the overhead it implies.

The hypervisor monitors the scheduling contexts hierarchically and triggers the resizing of the scheduling contexts at a certain level. Verifications are then going bottom-up and stop at the level where there is no need to resize (i.e. at this given level the speed of each context is consistent with the ideal speed). We can see in figure 4.6 the expected speed of each context in black, and the actual speed in green if it is the same as the expected one and in red if it is different. We can see that

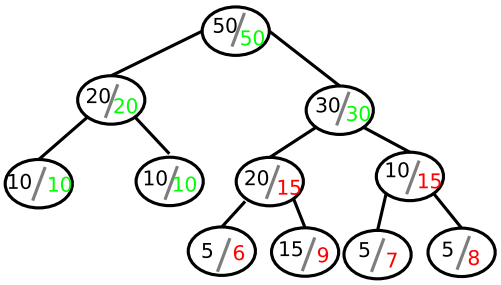


Figure 4.6: Hierarchically trigger resizing of scheduling contexts

the left sub-tree of contexts requires triggering the resizing process. Hence, the resizing decisions at higher levels of the tree of the scheduling contexts are then propagated top-down towards the layer of active contexts.

4.8 Evaluation

In this section we evaluate the behavior of our hierarchical approach and illustrate how it improves locality and performance. This is done on a set of sparse linear systems solved on two types of architectures: SMP and NUMA. The experimental evaluation illustrates the gains in terms of execution time and enhancement of the locality. Moreover, the cost of the hierarchical strategy is evaluated.

4.8.1 Experimental environment

As stated above, we evaluate the behavior of our approach on two platforms (see Appendix B):

- `riri` which has uniform access to the memory.
- `ares` is a cache coherent Non Uniform Memory Access (ccNUMA)

Both platform are shared memory memory ones in order to match the requirements of `qr_mumps`.

The experiments were conducted on a set of matrices mainly from the University of Florida Sparse Matrix Collection[1] presented in Table 4.1. The exceptions being the ultrasound80 matrix (Propagation of 3D ultrasound waves, provided by M. Sosonkina) and the conv3d64 matrix (provided by CEA-CESTA and generated using AQUILON[2]). All the matrices have been reordered using a fill-reducing matrix permutation produced by METIS[3] (version 5.0.2). We divided this set of

[1] http://www.cise.ufl.edu/research/sparse/matrices
[2] http://www.enscpb.fr/master/aquilon
[3] http://glaros.dtc.umn.edu/gkhome/views/metis

#	Mat. name	m	n	nz	op. count (Gflops)
1	TF15	7742	6334	80057	93.90
2	tp-6	142752	1014301	11537419	381.82
3	esoc	37830	327062	6019939	891.58
4	Rucci1	1977885	109900	7791168	5316.94
5	pre2	659033	659033	5834044	777.67
6	ultrasound80	531441	531441	33076161	64777.40
7	conv3d64	836550	836550	12548250	108491.50

Table 4.1: Matrices test set. The operation count is related to the matrix factorization with METIS column permutation.

matrices in two groups: the so called small problems: TF15, tp-6, esoc, Rucci1, pre2 and the large problems: ultrasound80 and conv3d64. The behavior of the small problems has not been evaluated on more than 40 cores, as they are not able to scale on so many processing units. All codes were compiled with the GNU v. 4.7.2 suite and linked to the Intel MKL sequential BLAS and LAPACK libraries. All the tests were run with real data in double precision. Finally, it is important to mention that for a small number of processing units, the cores used for the experiments are chosen according to a compact strategy according to the memory hierarchy.

4.8.2 Experimental evaluation

We begin the evaluation section by measuring the cost of the dynamic algorithm used to distribute the hierarchical structure of the scheduling contexts over the processing units. We measure the time spent calling the hypervisor and trying to redistribute the resources in order to match the structure of the application and the machine. In the table 4.2 we can see that the cost of the hypervisor is more important on smaller matrices, because they do not have enough computation in order to compensate the time spent to improve the execution time. However, larger problems like conv3d64 have enough workload such that they better benefit from the hypervisor. Moreover, conv3d64 has a relatively regular assembly tree, therefore the hypervisor is less needed and implicitly less called in practice.

On the `ares` platform the overall overhead is more important than on the `riri` platform. This is mainly due to the NUMA aware architecture of the platform `ares`. Therefore, when monitoring the applications the hypervisor detects an important number of cases of slow contexts or idle resources which require its help in order to adjust to this architecture. We can see that for small problems like tp-6 we spend 0.89% of the time in the hypervisor even when running on 8 cores. This shows that taking advantage of the hypervisor for this problem is more difficult due to its costs compared to its execution time. The overhead of the hypervisor varies also with the structure of the graph of tasks of the problem. According to its structure we may need more or less contexts and implicitly the resizing may be more or less expensive.

Further on, we evaluate the behavior of the hierarchical contexts approach to structure the parallelism of different problems. Buttari *and al.* [8] have studied the performance behavior of `qr_mumps` on top of StarPU and they compared it with different state-of-art solutions. Therefore, in the following section we compare the execution time of our approach to the one using StarPU without any contexts.

In Figure 4.7 we show the ratio between the execution time of the version using the contexts

(a) Cost of the resizing process on `riri`.

	8 cores	16 cores	24 cores	32 cores	40 cores
TF15	0.03	0.02	0.02	0.04	0.10
tp-6	0.48	0.11	0.06	0.10	0.24
ESOC	0.02	0.01	0.04	0.09	0.12
rucci1	0.03	0.02	0.03	0.03	0.04
pre2	0.01	0.01	0.03	0.06	0.06
ultrasound80	0.04	0.02	0.03	0.02	0.007
conv3d64	0.04	0.03	0.04	0.03	0.03

(b) Cost of the resizing process on `ares`.

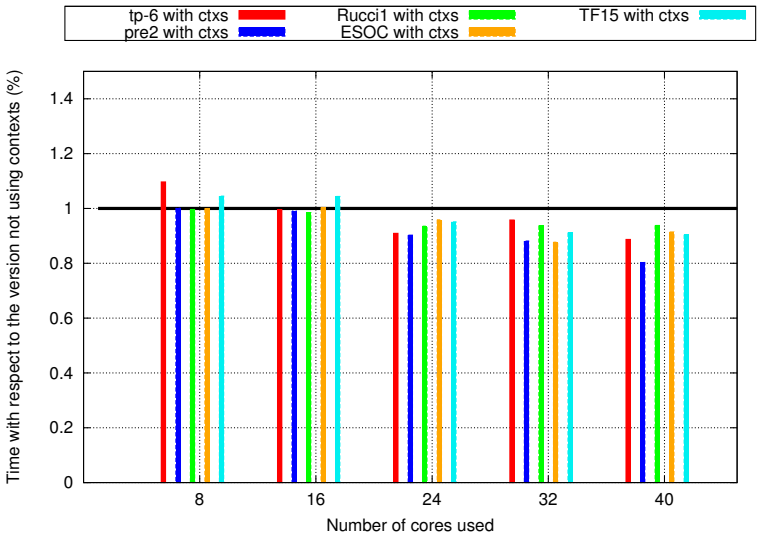
	8 cores	16 cores	24 cores	32 cores	40 cores	64 cores
TF15	0.80	0.07	0.17	0.22	0.27	-
tp-6	0.89	0.66	0.29	0.78	0.53	-
ESOC	0.05	0.04	0.13	0.21	0.23	-
rucci1	0.09	0.05	0.08	0.10	0.13	-
pre2	0.04	0.03	0.10	0.16	0.17	-
ultrasound80	0.09	0.03	0.08	0.05	0.07	0.14
conv3d64	0.15	0.03	0.09	0.05	0.08	0.16

Table 4.2: Cost of the resizing process with respect to the total execution time (%)

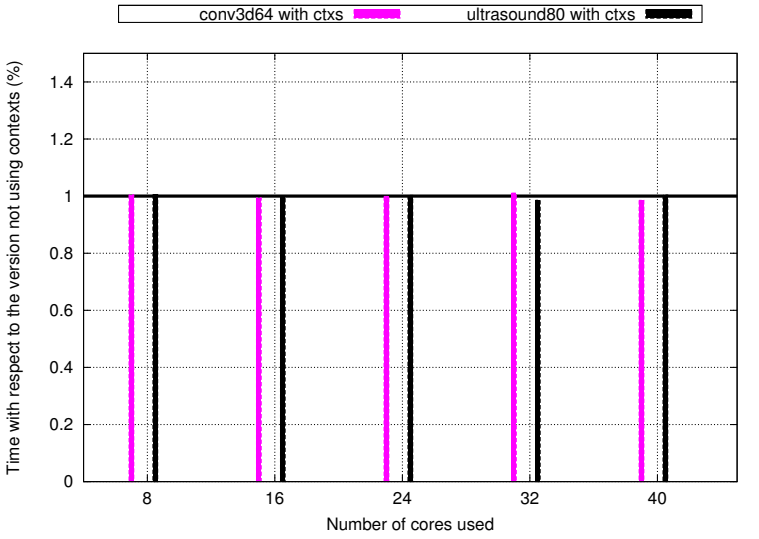
and the basic StarPU one not using the contexts. In table 4.3 we can see the referenced execution time of the version without contexts. This complements the data provided by Figure 4.7 and 4.8. We can see that the scalability of the solver is satisfactory when the problem is large enough to be treated on the considered number of cores.

We can observe in Figure 4.7 that the execution time of the hierarchical version of the solver has a comparable performance with the regular StarPU implementation on small number of processors. However, when we increase the number of processes, we can observe that the hierarchical version starts to take advantage of the locality and thus improves the performance, such that we obtain a decrease of the execution time of up to 15%. The hierarchical version does not always outperform the regular version, especially when the problems have a regular form of the assembly tree. There is more room to improve performance if the assembly tree is irregular and unbalanced. One side effect of our hierarchical algorithm is that it will assign a lot of resources to the branch of the tree corresponding to the critical path. Thus, on such irregular trees, this algorithm may reduce the length of the critical path (by increasing the number of resources) leading to a decrease of the execution time.

In Figure 4.8 we can notice a similar behavior for the small problems. However, on larger test problems (see Figure 4.8(b)), the execution time gain obtained from the use of our hierarchical approach grows with the number of resources. This is mainly due to the fact that the hierarchical strategy enhances data locality and isolates branches of the assembly on a specific set of cores taking advantage of the strongly non-uniform memory hierarchy. We can observe gains going up to 30% on some cases like the conv3d64 on 64 cores. On highly NUMA architectures like `ares` the locality is an important matter. We can see that using our hierarchical approach on top of 32 cores or 64 cores for the large matrices improves the behavior of the applications. This is mainly due to the fact that the `ares` has 4 NUMA groups of 16 cores and by isolating sections of the assemble



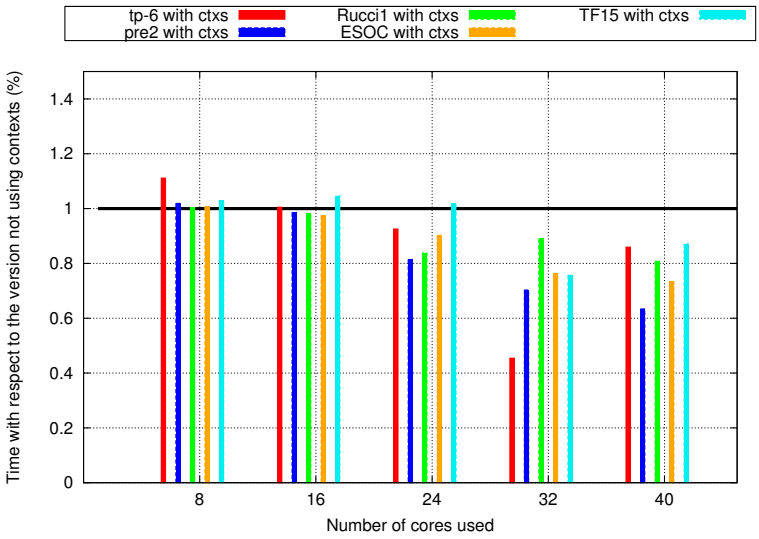
(a) Small problems



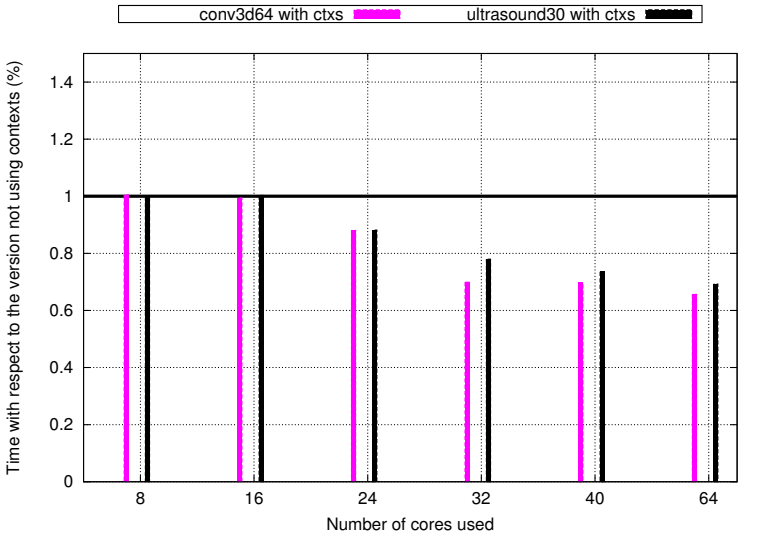
(b) Large problems

Figure 4.7: Execution time of the hierarchical version of `qr_mumps` with respect to the non contexts StarPU version on the `riri` platform

tree on different NUMA nodes we avoid data transfers between the memory nodes. Using 40 cores,



(a) Small problems



(b) Large problems

Figure 4.8: Execution time of the hierarchical version version of `qr_mumps` with respect to the non contexts StarPU version on the `ares` platform

(a) Execution time on `riri`.

	8 cores	16 cores	24 cores	32 cores	40 cores
TF15	2.27	1.82	2.05	2.37	2.28
tp-6	11.06	9.08	9.69	10.14	11.43
ESOC	21.65	13.19	14.83	16.27	16.92
rucci1	99.10	55.26	43.74	40.51	41.07
pre2	19.64	10.95	9.64	13.38	14.48
ultrasound80	1066.59	584.92	421.30	345.62	304.79
conv3d64	1779.88	966.80	680.49	546.78	463.84

(b) Execution time on `ares`.

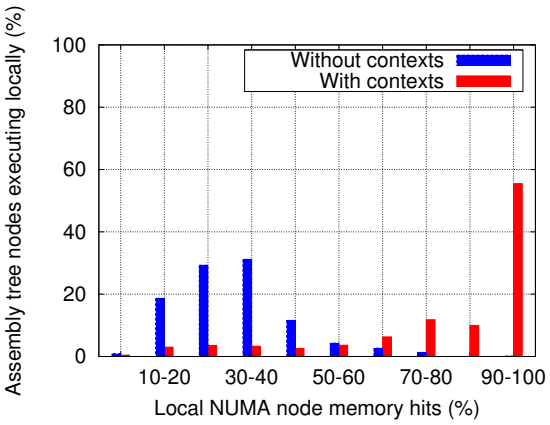
	8 cores	16 cores	24 cores	32 cores	40 cores	64 cores
TF15	2.65	2.30	2.85	3.93	3.88	-
tp-6	11.23	9.39	11.18	16.02	14.05	-
ESOC	21.27	13.43	20.85	24.73	27.50	-
rucci1	93.29	52.55	53.25	53.48	59.45	-
pre2	18.98	10.99	13.43	22.84	25.44	-
ultrasound80	1172.22	751.47	550.13	502.73	510.29	527.16
conv3d64	2166.15	1405.10	1032.14	890.85	813.71	784.87

Table 4.3: Execution time in seconds of different test problems of the regular `qr_mumps` implementation on top of StarPU

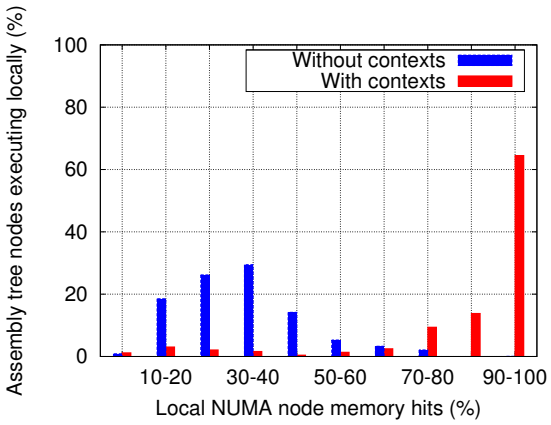
for example, implies executing on two complete groups and another additional 8 cores in another group. The increase in processing units does not compensate the costs of data transfers to those isolated cores. We can see this behavior especially for small matrices for which the computations do not counterbalance the data transfers.

To push further the analysis of the results we present in Figures 4.9 and 4.10 a plot illustrating the improvement of the locality of memory access when using the hierarchical approach for two of our test problems on the two platforms. First of all, to be consistent with the underlying architecture, we considered executions on 40 (resp. 32) cores for `riri` (resp. `ares`). Moreover, we remind that memory allocations are done during the execution of the activation task corresponding to each node of the assembly tree (see Section 4.1). This plot considers the amount (percentage) of assembly tree nodes for which the amount of corresponding StarPU tasks were executed on the same socket as the activation task. For example, if we consider the plot presented in Figure 4.10(a), we can observe that for the hierarchical approach (red bars) almost 90% of the nodes of the assembly had between 90% and 100% of their corresponding tasks executed on the same socket as the activation task. On the other side, the regular StarPU implementation has around 70% of the nodes of the tree having between 20% and 30% of their tasks executed on the same socket as the memory allocation. The locality has an important impact on large problems like conv3d64. The improvement on its execution time on `ares` is mainly due to the fact that at least 40% of the assembly nodes face between 40% and 60% of memory hits, compared to the non context version that mainly faces between 20% and 30% memory hits.

If we reconsider the results showed in Figure 4.8, we can see that indeed conv3d64 and Rucci1 both improve their execution time when using the hierarchical strategy on `ares`. Therefore, we can confirm that by enforcing the locality on non-uniform memory access platforms we can improve the

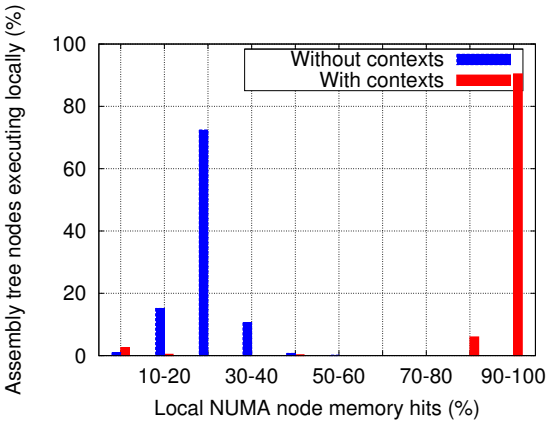


(a) Rucci1 on the `riri` platform using 40 cores.

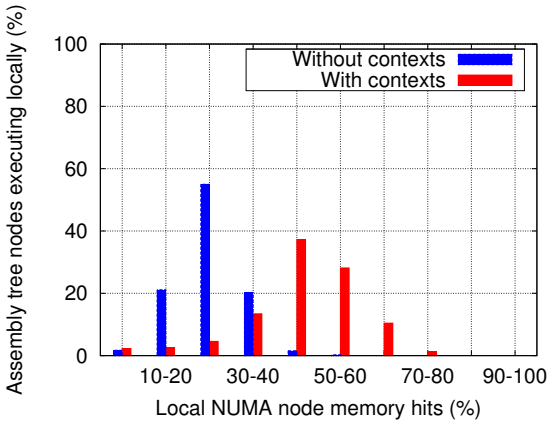


(b) Rucci1 on the `ares` platform using 32 cores.

Figure 4.9: Locality of data references for the Rucci1 problem.



(a) conv3d64 on the `riri` platform using 40 cores.



(b) conv3d64 on the `ares` platform using 32 cores.

Figure 4.10: Locality of data references for the conv3d64 problem.

overall execution time of the application.

4.9 Discussion

In this chapter we focused on studying the composability problem inside `qr_mumps` sparse direct solver, a complex library facing an irregular parallelism. Based on an elimination tree of DAGs of tasks, `qr_mumps` requires simultaneous execution of parallel codes on different dense matrices.

Our approach consists in capturing the parallel structure of the library in a hierarchical manner and projecting it on an abstract tree. This hierarchy is eventually mapped on scheduling contexts at the runtime system level which is responsible for dynamic resource management. Using information coming from the application together with hardware metrics captured at runtime, the hypervisor is able to better adjust the number of processing units allocated to each parallel task. Our experiments show that by continuously enforcing locality between related tasks, our approach exhibits a gain of up to 35% on test cases coming from real-life applications.

The results of this work showed that a tight interaction between the applications and the runtime system can improve performance with only slight modifications on the application side. We believe that this approach can easily be extended to a larger application domain.

We observe however that in this case scheduling contexts represent a solution to a scheduling problem. Meanwhile, a local scheduling constraint could probably assign certain tasks to execute on a processing unit belonging to a certain memory bank. Despite this, our approach provides the flexibility to use any local scheduling policy inside the contexts as well as any outer scheduling policy when managing the malleable tasks. Independent of the actual strategy, the scheduling contexts together with the hypervisor are able to enforce the locality.

The next step to this work is to extend our algorithm to heterogeneous platforms equipped with accelerators. This study would imply solving difficult problems when scheduling malleable tasks. Basic proportional mapping solutions would not be suitable in this case because accelerator have different requirements in term of computation capacity as well as memory access. Therefore, load balancing the scheduling contexts would become a difficult matter. On the other side, the hypervisor should consider different memory constraints when reallocating an accelerator from one context to another.

Heterogeneous architectures also raise new questions concerning the granularity of the tasks in `qr_mumps`. Scheduling contexts can be a good tool to manage higher granularity tasks. Adapted to the GPU memory management, these tasks can be expressed as moldable tasks and executed on several CPUs isolated inside scheduling contexts. Provided an on-line outer scheduler for parallel tasks, we can use the techniques presented in Chapter 2 for Intel MKL BLAS procedures, in order to introduce an intra task parallelism.

Chapter 5

Conclusion

High Performance Computing is continuously pushing towards solving faster harder and bigger problems. In order to respond to these endless requirements computer manufacturers design more and more complex machines, featuring different types of processors with several cores and several levels of memory. Yet, very few people have the expertize to program such architectures efficiently. Therefore, relying on existing parallel libraries strongly optimized to benefit from all their properties is a commonplace solution. However, applications invoking simultaneously several parallel libraries may usually exhibit poor performance. The reason lies on the fact that their implementations are based on an exclusive access to the computing resources, unaware of the co-existence of other libraries.

While the resource allocation for simultaneously executing applications has already been addressed for the cluster computing platforms, the problem has hardly been considered at the intra-node level. Indeed, arbitrating the resource allocation on a machine within an application implies different types of considerations, for instance the speed of the CPUs compared to the one of the accelerators, the memory management of multiple types of computing resources, etc.

Hence, the programmer needs a tool that can provide an abstract view of the machine, yet high level mechanisms to steer the execution of his application. Placing this tool at runtime system level allows then a good interaction between the hardware and the software management.

Contributions

Throughout this thesis we proposed a solution to the co-execution of parallel libraries on the same machine. We have implemented the concept of **Scheduling Contexts** inside the StarPU runtime system in order to isolate the simultaneously executing parallel codes. We showed that throughout this approach we can avoid the interference caused by several execution flows competing for resources.

We have implemented a plugin for StarPU called the **Hypervisor**, that monitors the execution of the parallel application and dynamically allocates computing resources to the scheduling contexts. We have proposed different resizing strategies that use linear programs to compute the necessary distribution of resources. The Hypervisor targets improving the overall execution of the application by considering the load of the kernels and the speed of the computing resources. The plugin is fully operational and available within the last release of StarPU.

The experimental results proved the relevance of our approach, demonstrating that the Hyper-

visor is a necessary tool for monitoring and dynamically assigning computing resources to simultaneously executing parallel kernels. We showed that in the context of heterogeneous architectures the *Hypervisor* is able to consider the properties of different types of computing resources (i.e. CPUs and GPUs), also acting, in some cases, as a balancing tool.

The concept of dynamic co-existence of parallel codes allowed improving `qr_mumps`, a sparse direct solver. Facing a very irregular parallelism, `qr_mumps` requires executing several independent parallel dense computations. By means of the *scheduling contexts* and the *hypervisor* we improved the locality and thus the overall performance of the solver.

Perspectives

The concepts proposed in this thesis provide some solutions to isolate parallel codes and to manage resource negotiation on heterogeneous architectures. Meanwhile, they stimulate the HPC community towards raising new questions and research opportunities.

Improving the under-subscription detection By means of the runtime abstraction of the computer architecture we detected when the computing resources are idle or when they are underused. However, more precises metrics could be considered in order to allow a better reactivity and a less frequent reexamination of the resource distribution. Such metrics could rely on hardware performance counters for instance. This approach could benefit from precise access information related to the frequency of the processing units, the cache utilization, etc. However, they require the management and filter of a significant set of events.

Improving non-StarPU parallel libraries resource negotiation We think that the concept of *scheduling contexts* is necessary when managing the composability problem. Isolating parallel libraries potentially relying on different runtime systems represents a solution to efficiently use the underlying computing resources. However, dynamically adapting their resource distribution implies that the co-existing runtime systems should react to the addition or removing of their resources at runtime. Moreover, computing the most adapted resource distribution requires that they should be able to provide some kind of information concerning their execution progress. If we take the example of OpenMP, it allows being isolated from other parallel sections, by providing mechanisms to fix the number of threads (`omp_set_num_threads(num_threads)`) and to bind them on any CPUs, but it does not allow redistributing the resources during the execution of the parallel section as it builds the team of threads just before entering it. Therefore, arbitrating the resource allocation over scheduling contexts that isolate such runtime systems requires finding different metrics and algorithms to interact with them.

Extending the resizing strategies We can find in the literature different solutions to compute static resource distributions for malleable tasks. However, on-line scheduling algorithms for heterogeneous architectures are still ongoing work on the corresponding fields. We think, the *Scheduling contexts* and the *Hypervisor* represent a framework that allows implementing and evaluating such techniques. Nevertheless, integrating such an algorithm as a resizing strategy inside the Hypervisor could provide an efficient distribution of resources, and could be applied to complex parallel applications.

Code-coupling applications An ongoing research perspective of this work is a more efficient support for code-coupling applications. Indeed, they could benefit from the *scheduling contexts* in order to allow executing simultaneously different modules on the same machine. Isolating parallel modules allows a better scalability as well as enhanced locality of the application. By means of the *hypervisor* we can provide a good load balancing and improve the use of the heterogeneous architectures.

Scaling to large architectures One of the main requirements for paving the way to exascale computing is *scalability*. By means of the *scheduling contexts* we enforced locality and scalability of scheduling decisions inside the parallel codes. However, as we face a constant evolution of the many-core architectures, the applications may require executing more parallel codes simultaneously, and thus using a significant number of scheduling contexts. In such a situation the *Hypervisor* could spend a lot of time taking the resizing decisions. Indeed, by using the hierarchical scheduling contexts we focused on taking resizing decisions only until a certain level in order to diminish the number of scheduling contexts and the number of cores considered for the resource distribution. Nevertheless, we could extend our resizing strategies in order to better take into consideration this aspect.

Partitioning accelerators (Intel Xeon Phi) Applications executing on heterogeneous machines usually face granularity problems. Architectures with both regular CPUs and accelerators raise questions concerning the granularity of the tasks to be executed on each one of them. Programmers usually use fine granularity for the tasks executed on CPUs and coarse granularity for the ones executing on the accelerators. However, this solution has an important impact on the parallelism due to the coarse granularity tasks. Moreover, it makes the scheduling decisions become even more complicated. Scheduling contexts provide a potential solution for such a problem as they may allow partitioning accelerators. Therefore, several fine grain parallel tasks can execute simultaneously on an accelerator like for example Intel Xeon Phi (having 60 cores x 4 hyperthreads).

Appendix A

Experimental libraries

As a result of high performance portability achieved thanks to the hardware abstraction layer introduced by the runtime system, several parallel libraries were implemented on top of them. These libraries usually prove state of art performance and are thus directly used in different HPC applications or even other parallel libraries. For instance, the BLAS (Basic Linear Algebra Subprograms) are a set of low-level routines that perform linear algebra operation. With several highly optimized implementations (BLIS, ATLAS, GotoBLAS), these low-level routines are reused by different other libraries like LAPACK, MAGMA in order to match different software and hardware requirements.

Intel MKL [25] (Intel Math Kernel Library) is a mathematical library that includes routines and functions optimized for Intel processor-based computers, that perform a wide variety of operations on vectors and matrices. It provides different BLAS and LAPACK routines intensively optimized to make profit from cache-management techniques. MKL is well integrated with the Intel OpenMP runtime system in order to manage the shared memory parallelism.

MAGMA [10] is a dense linear algebra library similar to LAPACK but for heterogeneous CPUs and GPUs. Developed at the University of Tennessee it was first released in 2008. It relies on the idea of representing algorithms as a collection of BLAS-based tasks that are executed over the underlying heterogeneous architecture. It is an explicit representation system, that statically manages a DAG-shaped dependency graph and schedules a large number of spawned tasks.

MAGMA-MORSE library [59, 9] is a dense linear algebra library developed at the University of Tennessee, first released in 2010. It is an implementation of MAGMA library based on StarPU [7] or quark [40] which can efficiently exploit hybrid platforms. By relying on runtime systems, these parallel libraries delegate the scheduling, data transfers and memory coherence to the underlying system. The algorithmic part of the library is separated from the architecture requirements, allowing a higher productivity for the programmer of the library not concerned with the low-level technical issues.

The Computational Fluid Dynamic (CFD) benchmark from the Rodinia benchmark suite [24] is implemented as an iterative solver for the three-dimensional Euler equations for compressible fluids. Such a scheme is very representative for unstructured grid problems, which represent an important class of applications in scientific computing. This benchmark has been rewritten to sit

on top of StarPU. The parallelization of this solver is done through domain decomposition. The number of tasks is proportional to the number of domains and the number of iterations. The tasks are independent at each iteration while there are dependencies between an iteration and the next one.

Appendix B

Experimental machines

- `riri` which has uniform access to the memory. It is composed of 4 Intel E7-4870 processors having 10 cores clocked at 2.40 GHz and having 30 MB of L3 cache for a total of 40 cores. The platform is equipped with 1 TB of memory.
- `ares` is a cache coherent Non Uniform Memory Access (ccNUMA) platform containing 8 Intel E7-8837 processors having 8 cores clocked at 2.67 GHz and having 24 MB of L3 cache for a total of 64 cores. The platform is equipped with 256 GB of memory organized in groups of 64 GB enforcing a high NUMA factor.
- `mirage` platform, a heterogeneous system composed of two Intel hexa-core processors X5650 at 2.67 GHz having 12 MB of L3 cache for a total of 12 cores and 36 GB of main memory, equipped with three NVIDIA Tesla M2070 GPUs having 6 GB of memory each. Note that 3 of 12 cores are devoted to execute NVIDIA GPU drivers.

Appendix C

Using the Scheduling Contexts to compose a CFD and a Cholesky Factorization kernel

The application consist on composing a CFD kernel, part of RODINIA benchmarks, and a Cholesky Factorization kernel belonging to the MAGMA-MORSE library (see Apendix A for description). Both the Cholesky Factorization (with the block size of the task 960) and the CFD kernel execute efficiently on the GPU, requiring thus the scheduling contexts in order to avoid the interferences.

Further on in this section we will refer to the following parallel codes:

- MAGMA kernel of the Cholesky Factorization executed on a matrix of 14976 x 14976 elements using a block size of 960 x 960 elements (referred to as `[Small_Cholesky_960]`)
- the CFD solver on 2957K cells throughout 200 iterations partitioned in two sub-domains (referred to as `[CFD]`)

This scenario composing a factorization kernel with a CFD one in Figure C.1 shows the importance of isolating the two. For this scenario, we normalize the execution time of each configuration with respect to the one using no contexts with an execution time of 16.42 s. Therefore, we observe that allocating an unadapted set of resources to the two kernels has an important impact on the execution time. We partitioned the CFD decomposition domain in two the obvious requirement in term of processing units is 2 GPUs. As long as the factorization is concerned, the important size of the matrix as well as the size of the task block (that is 960) implies that it needs at least 1 GPU in order to be efficient. As we can see in the graphic the best compromise seems to be to give 2 GPUs to CFD kernel and the left GPU together with the CPUs to the factorization.

We can see that giving any CPU to the CFD kernel is unnecessary as the scheduling policy does not use any CPUs as long as all the necessary data is on the GPU memory and any transfer would be very expensive. Moreover, the execution time of a task on the CPU is 10 times larger than on the GPU, thus minimizing the execution time of the kernels requires scheduling the tasks on two GPUs only.

It is important to notice at what point removing a GPU from the CFD kernel as well as from the factorization has a negative impact. When removing the only GPU from `[Small_Cholesky_960]` the kernel lacks computation resource. However for `[CFD]` the the execution time is affected also

by additional data transfers, as the entire required data for the two sub-domains cannot fit on the memory of a single GPU.

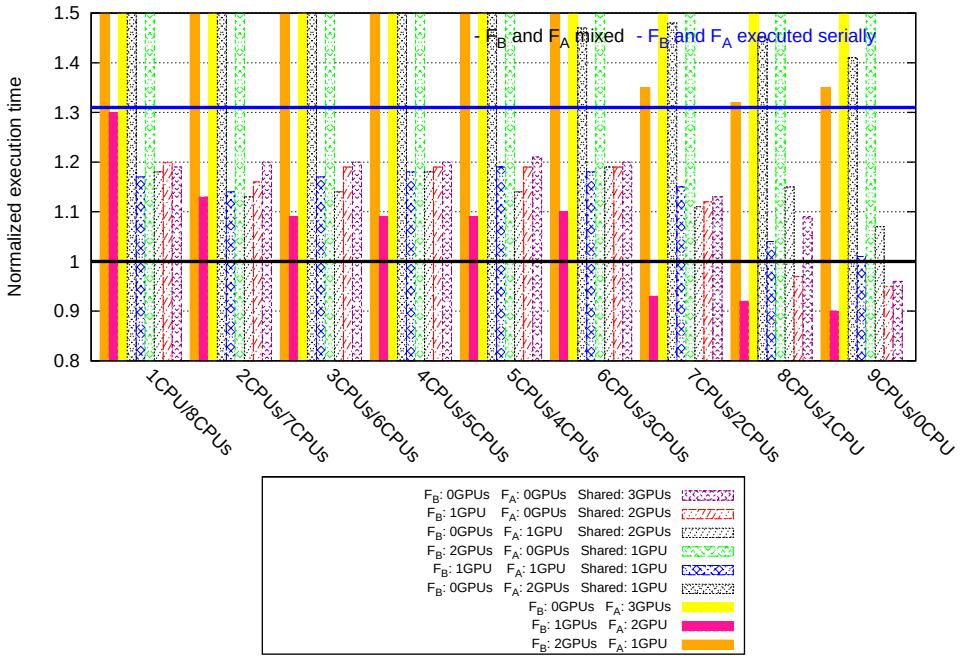


Figure C.1: Composing `[Small_Cholesky_960]` and `[CFD]` normalize with respect to the

Appendix D

Bibliography

[1] Cuda zone. http://www.nvidia.com/cuda.

[2] Release notes. https://software.intel.com/sites/default/files/managed/ae/5b/release-notes-c-2013-l-en.pdf.

[3] Using intel mkl with threaded applications. https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-using-intel-mkl-with-threaded-applications.

[4] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.

[5] *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 2010.

[6] *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 2013.

[7] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A hybridization methodology for high-performance linear algebra software for GPUs. *in GPU Computing Gems, Jade Edition*, 2:473–484, 2010.

[8] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing - 19th International Conference*, pages 521–532, 2013.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. volume Vol. 180, 2009.

[10] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[11] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for multicore architectures. *SIAM J. Scientific Computing*, 36(1), 2014.

[12] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.

[13] The OpenMP ARB. The openmp api specification for parallel programming, 2012. http://openmp.org/.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[15] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective.* Phd thesis, Université Bordeaux 1, 2011.

[16] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th Euro-Par Conference*, pages 863–874, Delft, The Netherlands, August 2009.

[17] E. Ayguadé, R.M. Badia, F.D. Igual, J. Labarta, R. Mayo, and E.S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par*, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In Hollingsworth [35], page 66.

[19] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(Issues 1):37 – 51, 2012.

[20] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Muller and Eduard Ayguadé*, 38(5):418–439, 2010.

[21] A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices, 2013.

[22] W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, George Mason University, May 1999.

[23] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with MPC. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings*, pages 1–14, 2010.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[25] Intel Corporation. MKL reference manual. http://software.intel.com/en-us/articles/intel-mkl.

[26] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[27] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[28] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[29] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[30] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[31] Thierry Gautier, João V. F. Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013* [6], pages 1299–1308.

[32] Luigi Genovese, Brice Videau, Matthieu Ospici, Thierry Deutsch, Stefan Goedecker, and Jean-François Méhaut. Daubechies wavelets for high performance electronic structure calculations: The bigdft project. *Compte Rendus Mecanique*, 339(Issues 2-3):149 – 164, 2011.

[33] William Gropp. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Linz, Austria, September 29 - October 2, 2002, Proceedings*, 2002.

[34] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In Pasqua DAmbra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin / Heidelberg, 2010.

[35] Jeffrey K. Hollingsworth, editor. *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*. IEEE/ACM, 2012.

[36] Costin Iancu, Steven A. Hofmeyr, Filip Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings* [4], pages 1–11.

[37] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. Van Zee. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput.*, 72(9):1134–1143, 2012.

[38] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *SC* [5], pages 1–11.

[39] Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors. *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, volume 4641 of *Lecture Notes in Computer Science*. Springer, 2007.

[40] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. *LAPACK working note*, lawn220, 2009.

[41] R. Lepere, G. Mounie, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European Journal of Operational Research*, 142:242–249, 2002.

[42] Andrey Marochko. Tbb 3.0 task scheduler improves composability of tbb based solutions., 2012. http://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based-solutions-part-1/.

[43] José Moreira and James R. Larus, editors. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. ACM, 2014.

[44] Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers*.

[45] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 03 2007.

[46] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. *SIGPLAN Not.*, 45:376–387, June 2010.

[47] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[48] G. N. Srinivasa Prasanna and Bruce R. Musicus. The optimal control approach to generalized multiprocessor scheduling. *Algorithmica*, 15(1):17–49, 1996.

[49] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[50] Martin Schreiber. *Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management*. Dissertation, Institut für Informatik, Technische Universität München, January 2014.

[51] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[52] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press Cambridge, MA, USA, 1998.

[53] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010.

[54] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay A. Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and APGAS at petascale. In Moreira and Larus [43], pages 53–66.

[55] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[56] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar*, Rennes,France, 2007. ACM.

[57] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In Kermarrec et al. [39], pages 42–51.

[58] Kai Tian, Yunlian Jiang, Xipeng Shen, and Weizhen Mao. Optimal co-scheduling to minimize makespan on chip multiprocessors. *Lecture Notes Computer Science*, 7698, 2013.

[59] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, 2010.

[60] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

[61] 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.

Appendix E

Personal Publications

[62] A.-E. Hugo. Le problème de la composition parallèle : une approche supervisée. In *21èmes Rencontres Francophones du Parallélisme (RenPar'21)*, Grenoble, France, January 2013.

[63] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple starpu applications over heterogeneous machines: A supervised approach. In *IPDPS'13 Workshops, workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, pages 1050–1059, 2013.

[64] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple starpu applications over heterogeneous machines: A supervised approach. *International Journal of High Performance Computing Applications (IJHPCA)*, 28(3):285–300, 2014.

[65] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. A runtime approach to dynamic resource allocation for sparse direct solvers. In *2014 International Conference on Parallel Processing (ICPP) - The 43rd Annual Conference*, 2014.