Scheduling Tasks over Multicore machines enhanced with acelerators: a Runtime System's Perspective

Cédric Augonnet

N° d'ordre: 4460

UNIVERSITÉ DE BORDEAUX 1

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

THÈSE

présentée pour obtenir le grade de

DOCTEUR

Spécialité : Informatique

Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective

---

par **Cédric AUGONNET**

soutenue le 9 Décembre 2011

Après avis de :

M. Albert	COHEN	Directeur de recherche INRIA	Rapporteur
M. Jean-François	MEHAUT	Professeur des Universités	Rapporteur

Devant la commission d'examen composée de :

M. Henri	BAL	Professeur des Universités	Examinateur
M. Albert	COHEN	Directeur de recherche INRIA	Rapporteur
M. David	GOUDIN	Ingénieur Chercheur au CEA	Examinateur
M. Jean-François	MEHAUT	Professeur des Universités	Rapporteur
M. Raymond	NAMYST	Professeur des Universités	Directeur de Thèse
M. Jean	ROMAN	Professeur à l'IPB	Président
M. Samuel	THIBAULT	Maître de Conférences	Directeur de Thèse

À Camille,

Remerciements

Au terme de ce périple qu'est la thèse, je souhaite remercier toutes celles et ceux qui m'ont permis de parcourir ce chemin tortueux. Je remercie en premier lieu mes encadrants pour m'avoir guidé pendant ces quatre dernières années. Merci Raymond de m'avoir donné l'opportunité de travailler sur un tel sujet et de m'avoir encadré tout au long de cette aventure, depuis mon premier stage en licence jusqu'á la fin de ma thèse. Merci Samuel d'avoir toujours été là pour me faire prendre le recul nécessaire afin de canaliser un enthousiasme quelque peu débordant. Je suis d'ailleurs particulièrement fier d'avoir été ton premier thésard. Je tiens aussi à remercier les autres membres de mon jury, que j'ai eu la chance de recontrer à différentes reprises au cours de ce périple : Henri Bal – dont le cours aura suscité en moi une véritable passion pour le calcul parallèle –, David Goudin et Jean Roman. Je remercie particulièrement Jean-François Mehaut et Albert Cohen pour avoir accepté et pris le temps relire mon manuscrit.

Une thèse, c'est avant tout un travail d'équipe. Merci donc aux membres de Runtime d'avoir rendu ce travail possible et de m'avoir supporté malgré un handicap administratif flagrant (pardon Sylvie !). Merci à celles et ceux qui ont participé à l'aventure StarPU et qui m'ont permis d'avancer bien plus que je ne l'aurais imaginé : Samuel, Raymond, Nathalie, Nicolas, François, Ludovic, Sylvain, Olivier ... Et bonne chance aux personnes qui reprendront le flambeau ! Je remercie également Alexandre, Denis, Emmanuel, Marie-Christine, Olivier et Pierre-André pour leurs conseils avisés. Merci aux autres personnes avec lesquelles j'ai pu partager les joies de la thèse : Babeth la spécialiste des makis, Bertrand le marathonien, Broq l'homme le plus drôle du monde, Diak mon co-bureau invisible, Jéjé le sosie officiel de Michael Lonsdale, Paulette et ses compils de Thunderdome, et enfin Stéphanie à qui je dois certainement 3000 euros chez Nespresso. Par ailleurs, je tiens à m'excuser auprès de mes différents co-bureaux pour leur avoir infligé mes frasques musicales, un fouillis sans nom, et ma décoration à base de poneys. Qui aurait cru qu'on puisse chauffer un bureau à l'aide de BLAS3 ?

Merci aussi aux personnes avec qui j'ai eu la chance de collaborer à Bordeaux et dans les projets ProHMPT, MediaGPU, PEPPHER, ou encore celles qui m'ont accueilli à l'Université du Tennesse Knoxville et d'Urbana-Champaign. Merci notamment à Emmanuel, Hatem, Mathieu et Stan de m'avoir ouvert les portes de l'ICL.

Je pense également à mes amis qui ont rendu cette période un peu plus légère. Merci à Cécile, Cyril, Amélie et Christophe pour les soirées Wii/Master System 2 sur fond de Rhum et Tequila. Spécial big up à Juliette et François qui ont toujours été présents dans les moments importants, et ce malgré la distance. Merci aussi à eux (et à Camille) pour m'avoir supporté en tant que colocataire : il en fallait du courage pour tolérer ma boite à meuh, ma carte musicale Coca-Cola,

et mes moultes décorations des plus seyantes ! Au passage, une caresse à Babouch mon chat d'adoption, pour ces longues heures passées sur mes genoux pendant que je m'affairais à rédiger ce document ! Merci enfin aux geeks de l'ENS Lyon qui m'ont fait découvrir l'informatique et la Guinness : Dain, Grincheux, Julio, Kaisse, Louloutte, Nanuq, Nimmy, Stilgar, TaXules, Tchii, Theblatte, Tonfa, Vinz, Youpi et Zmdkrbou.

Parceque sans eux je n'en serais probablement pas arrivé là: merci à mes parents d'avoir toujours pris sur eux pour me soutenir, malgré la distance. J'espère que vous ne m'en voulez pas trop pour toutes ces fêtes de Noël passées à travailler entre le saumon et la bûche ; ou pour cette époque durant laquelle je découvrais Internet tout en vous empêchant de recevoir le moindre coup de fil (une petite pensée au passage pour tous les autres parents victimes du forfait AOL 99 francs illimité). Merci à mon autre famille pour sa grande générosité et pour m'avoir fait apprécier les joies du Sud-Ouest. Merci Michel pour toutes nos discussions, pour les leçons de conduite, et pour avoir sacrifié à maintes reprises votre réserve de champagne, afin d'accompagner les moments importants. Merci Geneviève pour tous les croissants, pains au chocolat (ou chocolatines !) et autres bouteilles de Jurançon. Merci à Julie et Jerry, et à Charlotte et Jean-Clément pour m'avoir accueilli si souvent : que c'est bon d'être une pièce rapportée dans ces conditions !

Enfin plus que quiconque, je tiens vraiment à remercier Camille, qui a toujours été là pour moi. Merci pour tout cet amour, cette douceur, et ce réconfort ! Merci à toi pour les sacrifices consentis, et pour tout ce temps passé à me soutenir malgré mes innombrables absences. Je te suis infiniment reconnaissant de m'avoir ouvert les yeux sur ce que sont les vrais bonheurs de la vie, que tu rends chaque jour un peu plus belle.

Abstract

Multicore machines equipped with accelerators are becoming increasingly popular in the High Performance Computing ecosystem. Hybrid architectures provide significantly improved energy efficiency, so that they are likely to generalize in the Manycore era. However, the complexity introduced by these architectures has a direct impact on programmability, so that it is crucial to provide portable abstractions in order to fully tap into the potential of these machines. Pure offloading approaches, that consist in running an application on regular processors while offloading predetermined parts of the code on accelerators, are not sufficient. The real challenge is to build systems where the application would be spread across the entire machine, that is, where computation would be dynamically scheduled over the full set of available processing units.

In this thesis, we thus propose a new task-based model of runtime system specifically designed to address the numerous challenges introduced by hybrid architectures, especially in terms of task scheduling and of data management. In order to demonstrate the relevance of this model, we designed the StarPU platform. It provides an expressive interface along with flexible task scheduling capabilities tightly coupled to an efficient data management. Using these facilities, together with a database of auto-tuned per-task performance models, it for instance becomes straightforward to develop efficient scheduling policies that take into account both computation and communication costs. We show that our task-based model is not only powerful enough to provide support for clusters, but also to scale on hybrid manycore architectures.

We analyze the performance of our approach on both synthetic and real-life workloads, and show that we obtain significant speedups and a very high efficiency on various types of multicore platforms enhanced with accelerators.

Résumé

Les machines multicœurs équipées d'accélérateurs deviennent de plus en plus populaires dans le domaine du Calcul Haute Performance. Les architectures hybrides réduisent la consommation énergétique de manière significative et sont donc amenées à se généraliser dans l'ère du *manycœur*. Cependant, la complexité induite par ces architectures a un impact direct sur leur programmabilité. Il est donc indispensable de fournir des abstractions portables afin de tirer pleinement parti de ces machines. Les approches qui consistent à exécuter une application sur des processeurs généralistes et à ne déporter que certaines parties prédéterminées du calcul sur des accélérateurs ne sont pas suffisantes. Le véritable défi consiste donc à concevoir des environnements où les applications sont réparties sur l'intégralité de la machine, c'est-à-dire où les différents calculs sont ordonnancés dynamiquement sur la totalité des unités de calcul disponibles.

Dans cette thèse, nous proposons donc un nouveau modèle de support exécutif fondé sur une abstraction de tâche et spécifiquement conçu pour répondre aux nombreux défis en termes d'ordonnancement de tâches et de gestion de données. La plate-forme StarPU a été conçue lors de cette thèse afin de démontrer la pertinence de ce modèle. StarPU propose une interface expressive permettant d'accéder à un ordonnancement flexible, fortement couplé à une gestion de données efficace. À l'aide de cet environnement et en associant les différentes tâches avec des modèles de performance auto-calibrés, il devient par exemple très simple de concevoir des stratégies d'ordonnancement prenant en compte les temps de calcul et les surcoûts liés aux mouvements de données. Nous montrons que notre modèle fondé sur un paradigme de tâche est suffisamment puissant pour exploiter les grappes de calcul d'une part, et les architectures *manycœurs* hybrides d'autre part.

Nous analysons les performances obtenues non seulement grâce à des tests synthétiques, mais aussi à l'aide d'applications réelles. Nous obtenons ainsi des accélérations substantielles, ainsi qu'une très bonne efficacité parallèle sur différents types de plates-formes multicœurs, dotées d'accélérateurs.

Contents

List of Figures

List of Tables

Introduction

Hybrid accelerator-based computing

ADDRESSING the never-ending race for more performance has led hardware designers to continuously make their best to design ever more evolved processors. Instead of relying on the sole evolution of transistor integration, parallel architectures leverage sequential processors by replicating their processing capabilities. Nowadays, we have thus reached an unprecedented level of parallelism in clusters of hierarchical multicore machines. After the *frequency wall* which led to this multicore computing era, architects must now address the *energy wall* to enter the manycore era. Power consumption has indeed become a serious issue which prevents from designing multicore chips with hundreds or thousands of full-fledged cores. Such power considerations are indeed not limited to embedded platforms anymore. They have not only become a concern for large HPC platforms, but they are also becoming a problem for standard mainstream machines. Complex generic purpose hardware is indeed very expensive to design. Replicating simpler CPU cores permits to save gates, and therefore to reduce the amount of energy required for simpler operations. It is also possible to hard-code specific functions in hardware, which is much more efficient than when achieved by a general purpose processor, both from a performance point of view, and from an energy consumption point of view.

The manycore revolution may therefore be characterized by heterogeneous designs, either using accelerating boards or directly by the means of hybrid heterogeneous manycore processors. Even though accelerators have been existing for a long time, hybrid computing is a solid trend. They are not only in the HPC community, as illustrated by the numerous machines based on accelerators in the Top500 [1], but they are also getting adopted in mainstream computers thanks to the use of commodity hardware such as GPUs.

Goals and Contributions of this thesis

As a result of this hardware revolution, the manycore era will put a significant pressure on the software side. By proposing simpler processing units, architects assume that programmers will manually take care of mechanisms which used to be performed by the hardware, such as cache consistency. Therefore, there is a growing gap in terms of programmability between existing programming models and the hardware that keeps changing at an unprecedented pace. While multicore already introduced numerous software challenges, accelerators raise this difficulty to an even greater level, especially when combining accelerators with manycore processors. Programmers cannot deal with such a complexity alone anymore, so that we need to provide them with a better support throughout the entire software stack. Runtime systems play a central role by exposing

convenient and portable abstractions to high-level compiling environments and highly optimized libraries which are needed for end-users. The contributions of this thesis therefore cover the different aspects which must be addressed at the level of the runtime system. More particularly, we identified the following requirements:

- **Provide a unified view of all processing units.** Most efforts to provide support for accelerators initially consisted in making it easier to offload all computation on accelerators. Beyond this mere offloading model, tapping into the full potential of a hybrid accelerator-based platform requires that computation should actually be spread across the entire machine. A portable model should thus provide a unified abstraction for all processing units, including CPU cores.

- **Structure the application using task parallelism.** Scientific programmers can hardly rewrite all their codes every time there is a new hardware innovation. As a result, programmers need a portable interface to describe parallel algorithms so that they can be executed on any type of parallel machine enhanced with accelerators. Doing so by the means of interdependent tasks provides an architecture-agnostic description that can be mapped efficiently on a variety of parallel platforms. Tasks not only provide a generic abstraction of computation, they also allow programmers to explicitly specify which pieces of data are accessed by the different tasks. Such an expressive representation enables a lot of optimization opportunities for a runtime system (*e.g.* data prefetching).

- **Schedule tasks dynamically.** Statically mapping tasks between the different processing units often requires a significant understanding of both parallel programming and of the underlying hardware, which is not compatible with our portability concerns. Dynamically scheduling tasks within the runtime system makes it possible to relieve programmers from this delicate problem and to obtain portable performances. Since there does not exist an ultimate scheduling strategy that fits all parallel algorithms, runtime systems should provide a convenient way to plug-in third party scheduling policies.

- **Delegate data management to the runtime system.** Designing scalable manycore architectures often requires to relax memory coherency to some extent. As a result, programming accelerators usually implies to explicitly request data transfers between main memory and the accelerators. These transfers are achieved by the means of low-level architecture-specific mechanism (*e.g.* asynchronous DMA) which are not compatible with our portability concerns. Portable applications should therefore defer data management to lower-level software layers such as the runtime system which can dynamically ensure data availability and data coherency throughout the machine. Due to the huge impact of data contention on overall performance, data management should be tightly integrated with the task scheduler. This for instance avoids offloading computation when the data transfer overhead is higher than the performance gain actually achieved by offloading computation.

- **Expose an expressive interface.** The interface exposed by the runtime system should be expressive enough to allow programmers to supply scheduling hints whenever possible, so that the runtime systems need not *guess* approximately what programmers *know* perfectly. On the other hand, runtime systems should provide higher-level software layers with performance feedback. This for instance affords them with input for performance analysis tools and auto-tuning mechanisms.

- **Bridge the gap between the different bricks of the software stack.** Library designers are supposed to be domain-specific experts, but they are not necessarily parallel programming experts. Likewise, abstractions provided by runtime systems should be helpful when designing parallel compilers targeting accelerator-based platforms. Compilers are indeed meant to generate or optimize kernels, but they are not necessarily supposed to provide efficient runtime libraries to coordinate the execution of the code that was generated. In both cases, relying on the high-level abstractions offered by a runtime system allows designers of parallel libraries and parallel compilers to concentrate on the design of efficient parallel algorithms and to provide or generate fully optimized compute kernels instead of handling low-level non-portable issues, so that they can painlessly take advantage of the numerous hardware evolutions.

All the contributions described in this thesis have been implemented and experimented in the StarPU runtime system [Web]. StarPU is freely distributed as an open-source C library composed of more than 60 000 lines of codes. As described in Chapter 8, StarPU is used internally by various real-life applications. It has also been the subject of several refereed publications which are cited at the end of this document.

Organization of this document

Chapter 1 presents an overview of accelerator-based computing, both in terms of hardware and of software. Chapter 2 analyzes the suitability of our task-based paradigm and introduces the StarPU runtime system which implements this model. Chapter 3 describes StarPU's flexible scheduling engine which permits to design portable scheduling policies. Chapter 4 depicts the granularity concerns introduced by manycore platforms and considers different algorithmic approaches to provide StarPU with a suitable granularity. Chapter 5 presents how StarPU integrates in a cluster environment. Chapter 6 gives an overview of the performance debugging facilities available in StarPU. Chapter 7 contains an experimental validation of the model implemented by StarPU, and Chapter 8 provides examples of applications actually using StarPU and describes its integration within the computing ecosystem. We finally conclude and describe our perspectives in Chapter 8.6.

Chapter 1

Context and Motivation

Chapter Abstract

This chapter gives an overview of the related work. We first describe the evolution of accelerator-based platforms, how GPUs have evolved to processors capable of performing general purpose computation, and we describe the design of some manycore architectures. Then we study the suitability of some programming models in the context of hybrid computing. We give an overview of the programming environments implementing these models, either at a low-level or at a higher level using compiling environments or libraries for instance. After describing the related work in terms of scheduling and data management over hybrid accelerator-based machines, we consider other runtime systems also targeting such platforms. StarPU, the runtime system that we propose in this thesis, will progressively be introduced, and eventually compared with existing runtime systems.

1.1 Manycore and Accelerator-based Architectures

Application-specific accelerating boards with specific hardware capabilities have been used for decades in the context of scientific computing and within embedded systems. Due to the excessive cost required to create completely new pieces of hardware, the advent of accelerators is however relatively recent for mainstream computing. It was only made possible by reusing existing mainstream technologies such as graphic cards, but in a different way than what they were originally designed for. Accelerating technologies come under many different forms: besides physical accelerating boards typically connected to the machine through the PCI bus, some multicore processors also feature cores with specific capabilities that are used to accelerate computation.

1.1.1 Accelerating compute boards: from ASICs to GPU computing

ASICs The use of specifically designed accelerating boards to perform compute-demanding operations dates back around 1980. Application Specific Integrated Circuits (ASIC) are integrated circuits which are customized for a special purpose. An early successful example of ASIC was for example the ULA (Uncommitted Logic Array) which was a chip that handled graphics on 8-bit ZX81 and ZX Spectrum computers. Large ASICs with a processing unit coupled with memory are sometimes called Systems-on-Chip (SoC). Assembling special purpose processing elements is also typical when designing embedded platforms. The building blocks of these platforms are usually called *"Intellectual Property"* or IP. An IP core typically consists of a Verilog or a VHDL design that can be integrated within a larger circuit. Each IP is supposed to be a power-efficient core that was designed for a specific purpose, which limits the overall power consumption of the resulting embedded system. For instance, there exists DSPs which goal is to offload TCP-related computation with a minimum power consumption within an embedded processor [197]. It is however extremely expensive to design a new ASIC, so that very high volumes are required to realistically consider designing an ASIC for a specific purpose. This approach is therefore often not really suited to accelerate mainstream general purpose applications.

FPGAs The Field-Programmable Gate Arrays (FPGA) invented by Xilinx in 1985 are a flexible alternative to ASICs. They contain programmable logic blocks which can be wired together using

a hardware circuit description language (*e.g.* Verilog or VHDL) to form a re-configurable circuit. FPGAs are therefore much more easily customized for a specific application than ASICs which cannot be reprogrammed. FPGAs are more generic pieces of hardware than ASICs, so that they usually consume more power and perform slower than their hard-coded counterparts even if they can be a first step before actually designing an ASIC. Modern FPGAs are large enough to host Digital Signal Processors (DSP) or memory blocks, but they remain very expensive and are often not robust enough to be used in production.

ClearSpeed boards Instead of repeatedly redesigning common mathematical operators on an FPGA, scientists can take advantage of an ASIC that would be produced at a relatively high volume. ClearSpeed accelerating boards implement operators that can be applied on a wide range of signal processing applications [43]. ClearSpeed cards are also shipped with a Software Development Kit that provides standard libraries such as BLAS, Random Number Generators, FFTs, etc. These cards were included in very large clusters such as the Tsubame Grid Cluster that wad ranked 9-th in the top500 list established in November 2006 [1]. Adding these accelerating boards in the Tsubame Grid Cluster increased performance by 24 % but only increased power consumption by 1 %. In spite of these significant advantages, there is still too low a volume to make these cards competitive against general purpose processors featuring accelerating cores or against the millions of graphic cards produced every year which can be an order of magnitude less expensive.

1.1.2 Computing with Graphic Processing Units

Graphics Processing Units (or GPUs) are a typical example of accelerating boards that were initially designed to provide an hardware-based support for a specific compute-intensive task, and was later on used for other purposes. As they are – by nature – designed for highly parallel problems, GPUs which became more and more programmable in order to deal with always more complex graphic problems indeed started to be used to solve completely unrelated problems later on. In this section, we briefly describe the evolution of GPUs, and how they evolved from fixed hardware implementations of standard graphic APIs to fully programmable processors applicable to general-purpose applications.

From fixed-function pipelines to fully programmable shaders

The first graphic cards which appeared in the 1980s were used to display 2D primitives. These are actually typical illustrations of the trend that consists in creating dedicated hardware accelerators to enhance software-based approaches. In the mid-1990s, CPU-assisted 3D graphics became so popular that hardware-accelerated 3D graphic cards were introduced in many mass-market consoles (*e.g.* PlayStation and Nintendo 64). Software implementations of the OpenGL standard graphic API which appeared in the early 1990s became so popular that a number of hardware implementations emerged. Likewise, Microsoft's DirectX programming API which is conceptually similar to OpenGL became popular in the late 1990s. Both APIs were originally implemented into the hardware by the means of fixed-function pipelines.

Due to the increasing needs encountered in very demanding markets such as gaming or even movie production, advanced 3D graphics cannot always be performed efficiently enough using a fixed-function pipeline. In order to perform specific graphic treatment and to obtain realistic

physical effects for games, the fixed-function graphic pipeline has thus been extended with customizable stages that are controlled by the means of user-provided pieces of code called *shaders*. Shaders have initially been introduced in 1988 by the PIXAR animation studio in its RIS standard (RenderMan Interface Specification). They have been introduced into the DirectX 8.0 and OpenGL 1.4 3D programming interfaces standards later on in 2001.

Initially, there were only pixel shaders (called fragment shaders in DirectX) to customize the output of the rasterizer. Pixel shaders are used to specifically compute per-pixel colors. Vertex shaders were added afterwards to be able to modify the colors and the coordinates of the vertices and of the textures composing the 3D objects being rendered. Finally, geometry shaders made it possible to actually add or remove vertices from a mesh which for instance allows procedural geometry generation (*e.g.* to create a fur on top of an object).

In 2006, OpenGL's *Unified Shader Model* (or *Shader Model 4.0* in DirectX 10) provided a single ISA for all three types of shaders. While the previous generations of shaders would typically extend the fixed pipeline with separate programmable units (*e.g.* to process pixels or vertices), a single type of processing unit called *Shader Core* could be used to implement all shaders. While a fixed number of cores would have previously been devoted to each stage of the pipeline, having a *Unified Shader Architecture* allows to dynamically assign shaders to the different unified cores, with respect to the actual workload. Having a unified shader architecture therefore enables more flexibility: in the case of a scene with a heavy geometry the GPU can for instance assign more cores to the vertex and geometry shaders, and less cores to pixel shaders, thus leading to a better load balancing. It is however worth noting that unified shader models and unified shader architectures are not strictly related. The Xenos graphic chip of the Xbox360 designed by ATI for instance already had a unified shader architecture to implement a superset of the Shader Model 3.0, which is not unified. On the other hand, we can also implement unified shader model with different types of shader units. From that date, a unified shader architecture was adopted by all major GPU makers: it is for example available in NVIDIA's GeForce 8 Series, in ATI's Radeon HD 2000, and in Intel's GMA X3000 series.

General Purpose GPUs

Originally, programmers had to program GPUs by the means of graphics API. Microsoft's HLSL produces DirectX shader programs, GLSL produces OpenGL shader programs [168], NVIDIA's Cg [135] outputs both OpenGL or DirectX shaders and has a similar syntax than HLSL. Even though shaders were already successful among the graphics community, they were neither really accessible to mainstream programmers nor to scientific programmers who are not used to casting their problems into graphics problems. With unified shader architectures, GPUs provide a more uniform ISA that makes it easier to design higher-level languages that automatically generate shaders. Brooks [33], Scout [138], and Glift [127] are examples of stream-based languages that provide programmers with high-level constructs that do not require to manipulate graphic primitives anymore. Such high-level environments permitted to implement general purpose algorithms on top of GPUs, which really marked the advent of General Purpose computing on GPUs, usually denoted as GPGPU. OWENS *et al.* gives an extensive study of early GPGPUs efforts [151]. Noteworthy, this study already denotes that almost all types of algorithms had already been implemented on GPUs in 2006 even though it would be inaccessible to most programmers.

Compared to other accelerating technologies, technological opportunism is a major factor of the success of GPGPUs. Instead of having to redesign new chips, GPU designers such as NVIDIA

or ATI/AMD can take advantage of the huge volumes observed in the gaming market. This for instance makes it possible to release several chips per year, which is not a realistic pace for standard CPU designers. Another advantage of GPGPUs is that most computers are already equipped with potentially powerful graphic cards, which makes it possible for programmers to try getting the benefits of accelerator-based computing without having to necessarily buy new hardware.

From GPGPU to GPU Computing

While most early GPGPUs efforts consisted in offloading the entire computation on one or several GPUs, it quickly appeared that GPUs were not suitable for all types of computation. When coupling multiple accelerators, the impact of data transfers also forced programmers to figure out that parts of the computation should rather stay on CPUs. As programmers realized that GPUs are only part of the computing ecosystem, instead of a complete replacement for it, the GPGPU approach often became referred to as GPU computing instead. An important challenge of GPU computing also consists in properly combining the power of both CPUs and GPUs altogether.

As GPU computing becomes more mature, a number of standard libraries (*e.g.* BLAS, FFT or Random Number Generators) have been implemented on top of CUDA and OpenCL. Full-fledged debuggers also make it possible to design actual industrial applications accelerated with GPUs. Even though kernel programming remains a delicate problem, programmers are therefore provided with almost standard programming environments. From a hardware point of view, GPUs have also evolved since they were pure graphic cards used in a non-standard way. Not only they now keep including features initially included in standard processors, such as double precision or cached memory, but they also feature advanced data management capabilities such as fully asynchronous DMA-based memory transfers. While the earliest CUDA-enabled chips would only allow a single kernel to be executed over the entire GPU at the same time, NVIDIA Fermi GPUs [145, 196] allow to execute multiple kernels concurrently. Finally, peer GPU-to-GPU transfers as well as direct transfers between GPUs and network cards allow to properly integrate GPUs in a real HPC cluster environment.

Besides all this vendor-specific software and hardware evolution, standardization is an important step required to obtain mature tools that permit to develop portable GPU-accelerated applications. This is exactly the goal of the OpenCL standard which provides a standard device interface to manipulate accelerators on the one hand, and a portable language to write vectorized kernels on the other hand.

1.1.3 From specialized cores to hybrid manycore processors

Besides external accelerating boards such as FPGAs or GPUs, various manycore processors also feature heterogeneous processing cores that permit to offload compute intensive or critical operations on specific pieces of hardware instead of replicating numerous full-fledged cores which might consume an excessive amount of energy. In this section, we present a few examples of processors which illustrate the different approaches adopted to develop scalable manycore architectures.

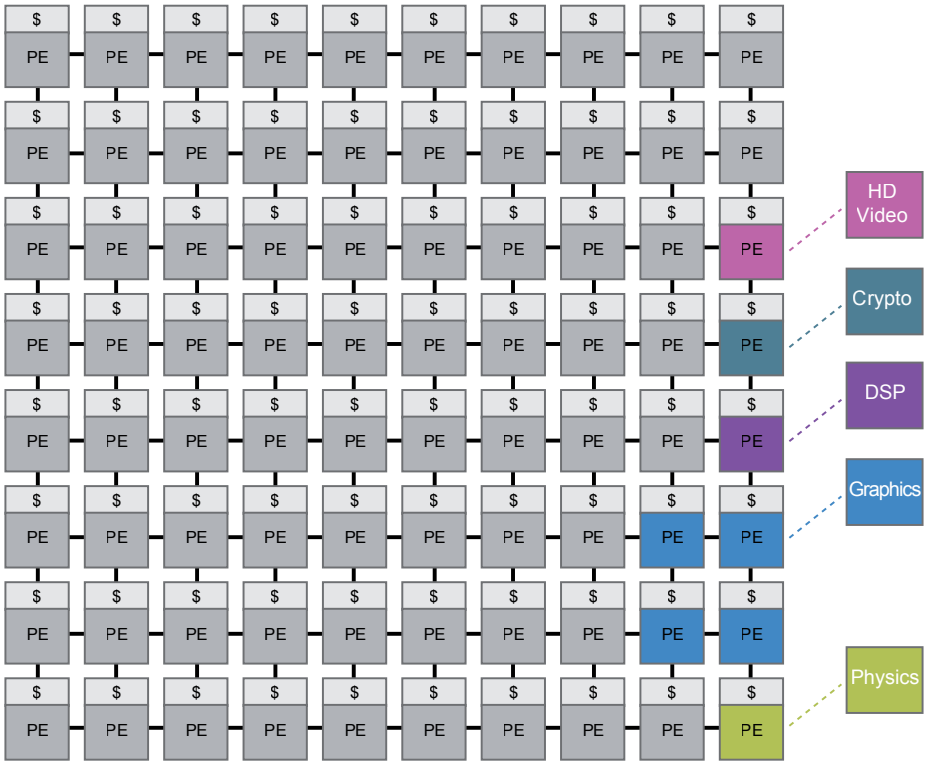


Figure 1.1: Intel Terascale Tile Arrangement: some cores provide fixed-functions and are dedicated for a special purpose.

Intel's TeraScale 80-core chip, Intel's Single-chip Cloud Computer and Intel Manycore Integrated Core Architecture

Intel's Terascale architecture is depicted on Figure 1.1. The TeraScale project was actually designed to implement a prototype of *manycore* processor, and to study the numerous challenges which appear at both hardware and software level [189]. It is composed of 80 cores arranged as a 2D interconnected mesh. The Processing Element (PE) of each core has a simple design to consume as little energy as possible. Maintaining cache coherency and the huge bandwidth requirements required by hundreds of cores is a real concern as data transfers could occur between any pair of cores. There is therefore no cache consistency enforced within the processor, and all memory transfers are fully explicit. By making the design of the memory sub-system simpler, this also greatly reduces the significant energy consumption of the memory interconnect. From a programming point of view, this however makes it much harder to program because all data management must be performed directly at the software level. The TeraScale is therefore interesting to study because most accelerators (*e.g.* FPGAs or GPUs) and most manycore processors (*e.g.* IBM's Cell) adopted such an explicitly-managed memory hierarchy.

As shown on Figure 1.1, another interesting aspect of the TeraScale is that a few cores also feature specific fixed functions which makes it possible to accelerate some compute intensive operations using these cores. This avoids designing 80 full-fledged cores which would consume too much energy without having to give up powerful capabilities such as HD video processing. It is however up to the application to ensure dealing with this heterogeneity. While the TeraScale pro-

cessor introduces a significant amount of other interesting hardware features that should make it possible to design scalable manycore processors (*e.g.* failure-resilient cores, dynamic per-core power management etc.), MATTSON *et al.* denote that "hardware designers seem to ignore software designers, [because] processors have reached an unprecedented complexity" [136]. On the other hand, they acknowledge that "it is difficult to write software with predictable performance when the state of the cache is so difficult to control" so that cache coherence should not be required when designing a manycore chip.

The 48-core Intel Single-chip Cloud Computer (Intel SCC) manycore processor [104] is a follow-up of the TeraScale project which underlines the need for suitable programming models. In order to optimize energy consumption, the frequency of the different cores can be changed dynamically. A core running a data intensive function that tends to stall on memory accesses needs not be executed at full speed: the resulting energy savings can be reinvested to execute compute intensive codes on other cores at a higher frequency.

An instance of the operating system is supposed to run on each core of the SCC. The different cores communicate with a light-weight MPI-like implementation communicating by the means of direct transfers between the local memory attached to the different cores [188]. The SCC is therefore conceptually similar to a 48-node cluster integrated on a chip. Having such an MPI-centric model provides programmer with a familiar environment which helps to adapt existing codes, but there is still a huge amount of work required at the software level to deal with the heterogeneous nature of the architecture which appears through varying frequencies or when enhancing some cores with fixed-function units.

Intel's Manycore Integrated Core (MIC) architecture is another project resulting from the previous architectures. The first prototype of MIC is called Knight Ferry. It is an accelerating board connected to the host through a PCI slot. The Knight Ferry contains 32 cores manufactured with a 32nm technology and running 4 thread each. The Knight Ferry chip also has a 8MB coherent shared cache which indicates the desire to keep this architecture as programmable as possible. Besides, the MIC is compatible with the x86 ISA, so that programmers are supposed to be able to run legacy C, C++ and Fortran codes relying on existing parallel programming environments such as OpenMP or TBB, and it is supposed to support the OpenCL standard too. The first actual product should be called Knight Corners and will have 50 cores manufactured with a 22nm technology.

The Cell processor

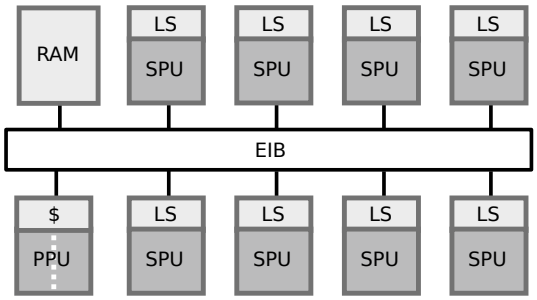


Figure 1.2: Architecture of the Cell Processor.

Figure 1.2 gives an overview of the architecture of the Cell processor. It is based on a hetero-

geneous design that combines a slightly modified PowerPC core with 8 very simple SIMD RISC cores [156]. The main core is called Power Processing Unit (PPU) and is capable of running an operating system. It is responsible for controlling the 8 coprocessors which are called Synergistic Processing Units (SPU). Each SPU only embeds 256 KB of Local Store (LS) which contains all data accessible by the SPU (*i.e.* data and code). Data exchanges between the different parts of the Cell processor are implemented by the means of explicitly managed asynchronous DMA transfers. Such transfers transit through the Element Interconnect Bus (EIB) which is a four-way ring.

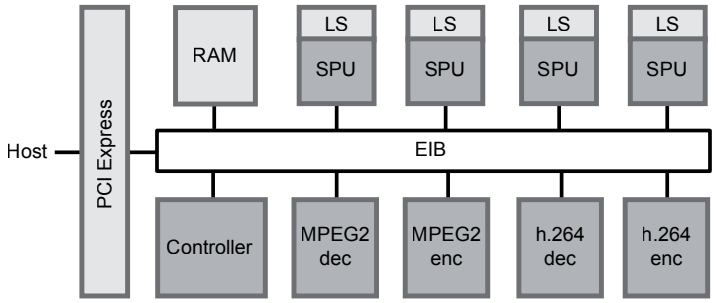


Figure 1.3: Logic schematic of the Spurs Engine chip.

The Cell was mainly used as the processor of Sony's PlayStation 3. It was also used in HPC platforms, either as Cell-based servers (*e.g.* IBM's BladeCenter QS22 servers) or integrated within accelerating compute boards such as Mercury Cell Accelerator Boards [27]. It was also used for compute intensive multimedia tasks. Toshiba's Spurs Engine external PCI cards for instance relies on a processor based on the Cell processor to perform efficient real-time movie transcoding (*e.g.* H264 or MPEG4) [101]. Contrary to standard Cell chips, the PPU was replaced by on-chip codecs which directly feed the four SPUs with computation, as depicted on Figure 1.3.

The Cell processor benefits from a very high internal bandwidth, but all transfers must be programmed manually by the means of low-level asynchronous DMA transfers. Writing an efficient code for this architecture therefore requires a significant expertise, especially to deal with the extremely limited size of the local stores on each SPU (256 KB). Load balancing is another challenging issue on the Cell. The application is indeed responsible for evenly mapping the different pieces of computation on the SPUs, without consuming too much memory bandwidth either. In spite of its novelty, the interest for the Cell processor has therefore greatly diminished with the advent of GPGPUs and more particularly of CUDA which provides programmers with much easier programming models. This lack of programmability has also had a significant impact on the suitability of the Cell processor on the gaming market. Programming a Cell processor is indeed much harder than programming a game on Xbox360's processor which is composed of three standard PowerPC homogeneous cores.

While IBM's initial plan was to develop a improved version of the Cell with 32 SPUs, this project has finally been discontinued (even though it is claimed that the ideas of the Cell processor should be used again for future IBM processors). Even though most efforts directly related to the Cell processors have now been stopped, the design of the Cell has clearly influenced that of other manycore processors such as Intel's Larrabee which was supposed to be arranged as a ring of heterogeneous cores. Besides the novelty in terms of hardware design, an important lesson learned from the Cell experience is that programmability should not be overlooked anymore.

Standard libraries should be provided to make it significantly easier to adapt existing codes on such a complex platform: the lack of a real BLAS/FFT implementation on the Cell is indeed almost sufficient to explain its failure on the scientific programming market.

Intel Larrabee

The Larrabee architecture is Intel's response to GPGPUs [169]. While some people claimed that CPUs should become useless when compared to the raw performance offered by GPUs, Intel suggested that GPUs' vector processing capabilities should be integrated directly into the CPU. The Larrabee processor was therefore supposed to provide a CPU-based implementation of the various 3D graphic stacks (*e.g.* DirectX and OpenGL), which would make GPUs useless.

Even though the overall design of the Larrabee processor is rather similar to that of the Cell processor because the various cores are also organized around a ring bus, the two approaches differ significantly. While the Cell processor explicitly exhibits the heterogeneity between the PPU and the SPUs, Larrabee's core implement an extension of the widely spread x86 ISA. Instead of having programmers to manually deal with complex low-level DMA transfers, the Larrabee architecture is cache coherent, which can however lead to scalability pitfalls. Similarly to GPUs, multiple threads are executed simultaneously on each core, to hide the significant memory latency, and each Larrabee core contains a 512-bit vector processing unit, so that it can process 16 single precision floating point numbers at the same time.

By the time the Larrabee was supposed to be available, both its pure performance and graphics capabilities were out of date compared to GPUs released at the same time, so that Intel decided to stop the project. Even though the Larrabee was never released, the Advance Vector Extensions (Intel AVX) [67] implemented in Intel Sandy Bridge processors provides 256-bit vector units which offer significant streaming capabilities comparable to those available in GPUs for instance. Contrary to IBM's Cell, the choice of the x86 ISA and the cache coherent memory hierarchy indicates that Intel wanted to design a chip with programmability and productivity in mind. This resulted in performance which is not in-par with that of the other accelerator-based architectures that accepted to give up cache coherency and to actually expose heterogeneity. On the one hand, having programmability and productivity in mind is important to avoid creating a chip that only a few parallel programming experts are able to master. On the other hand, it also appears that we need to consider simpler hardware: we may have to expose programmers to heterogeneity and/or to relax memory consistency to some extent.

AMD Fusion APUs

Buses are classical bottlenecks for accelerating boards. Instead of putting the accelerator on an external board connected to the host via a PCI-e bus, AMD's approach consists in putting the accelerator directly inside the processor package as shown on Figure 1.4 which depicts an AMD Fusion chip. This processor contains two standard AMD Phenom II CPU cores, and an Accelerated Processing Unit (APU) based on the AMD Radeon chip which implements the DirectX 11 and the OpenGL 4.1 graphic APIs.

Tightly integrating the accelerator within the processor provides a very low latency, comparable to the latency observed between two cores of the same chip. Another motivation for packaging the accelerator within the processor chip is that external GPU boards are relatively expensive. When produced in high volumes, integrated architectures such as the AMD Fusion is supposed

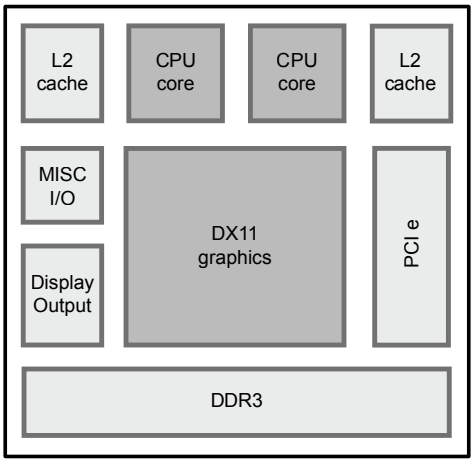


Figure 1.4: Organization of an AMD Fusion processor.

to be much cheaper. However, each square millimeter of silicon is a precious resource when designing a CPU. As a result, there is only limited space available for APUs, which is taken off the space previously available to standard CPU cores.

Similarly to Intel's Larrabee, a possible drawback of this approach is the difficulty to follow up the pace of the gaming market, so that it is hard to provide a chip that can remain in par with the performance of discrete GPUs which can be upgraded more easily.

1.1.4 Discussion

While SMP and multicore architectures have made it possible to overcome some of the physical limits encountered with purely sequential processors, a significant number of challenges reappear when designing large multicore chips.

Having numerous full-fledged modern CPU cores would be too costly if we do not reduce the complexity of the different cores. On the other hand, only having very simple cores would not be sufficient either. For instance, there must be some core(s) capable of running an OS in order to preprocess and post-process data handled by the accelerators. On the Cell processor, the significant processing power imbalance between the PPU and the 8 SPUs for example often makes it hard to supply enough work to the coprocessors. Combining many simple energy efficient cores with a few powerful cores therefore seems to be a promising approach considered by the major manycore processor designers, such as Intel, AMD and IBM, as illustrated by the various examples we have in this section.

Such heterogeneity can be achieved using either tightly coupled heterogeneous cores, or by introducing loosely coupled external accelerating boards. Accelerating boards are easy to upgrade, but the IO bus usually constitutes a major bottleneck which can be a limiting factor. Loosely coupled accelerators might also suffer from a significant kernel launch overhead which makes them unsuitable for latency-sensitive applications. Tightly integrating accelerators within the processor (*e.g.* Cell's SPUs or AMD Fusion APU core) provides much lower latency, and avoids numerous communications across the IO bus. Even though it seems advantageous to consider tightly integrated accelerators, each square millimeter of silicon is precious when designing a CPU core. While AMD has only been able to dedicate about 75 square millimeters for its APU by remov-

ing space normally available for a standard core in the Fusion chip, it is worth noting that the GF100 Fermi processor packages more than 3 billion transistors on a huge 500 square millimeters die. It also takes much longer to extend a modern CPU architecture than to upgrade a dedicated accelerating technology. By the time Intel would have been ready to commercialize its Larrabee processor, much more powerful GPUs were already available on the market.

Tightly integrating accelerators within manycore processor is therefore a difficult problem from an industrial point of view. A sensible trade-off might be to actually package accelerators as a self-standing socket so that one could easily upgrade a multi-socket motherboard to integrate up-to-date accelerators which performance are similar to that of external compute boards. Efforts to provide a unified interconnect such as AMD's HyperTransport or Intel's QuickPath Interconnect technology (QPI) should make such a design extremely efficient.

In order to hide heterogeneity, Intel's approach consists in providing a single well-known ISA (*i.e.* x86) for compatibility purpose. While this supposedly makes it possible to run existing applications on upcoming processors, this does not ensure that the application will run efficiently. Promising that we can reimplement a full complex ISA to run legacy general purpose applications on top of a very simple RISC architecture does not make much sense from energy and performance point of views if the software stacks ends up providing a software implementation of the various instructions available in modern CPU cores. Instead of expecting the hardware to automatically deal with heterogeneity, programmers need to adopt more suitable programming paradigms which take advantage of accelerators.

1.2 Programming models

While architects have proposed numerous hardware solutions to design scalable manycore architectures, a significant pressure has shifted onto the software side. Besides heterogeneity which introduces serious challenges in terms of load balancing, programmers need to deal with the lack of globally coherent shared memory by implementing explicit memory transfers between the different processing units. In this section, we present some programming models used to address these issues on accelerator-based platforms.

1.2.1 Multithreading

With the advent of multicore architectures, programmers have been forced to consider multithreading and parallel programming for shared-memory. Such models indeed permit to take into account architecture specificity such as memory hierarchy. Multithreading is however a notoriously complicated paradigm, especially when it comes to synchronization problems, even though there exist higher-level programming environments which are directly based on multi-threading (*e.g.* OpenMP [24]). Designing a really scalable multithreaded application is often challenging on current multicore architectures with as many as a hundred cores per machine. As studied by BROQUEDIS *et al.*, we already need hybrid paradigms with nested parallelism to obtain scalable performance on hierarchical machines [29]. Provided scalability is already a concern nowadays, ensuring that an algorithm written today will scale over an arbitrary machine in the future is even harder.

Load balancing is also up to the multithreaded application programmer. Provided this is hard on multicore platforms when the number of cores gets high or when the application is not regular

enough, this is really a concern when it comes to dispatching work within a heterogeneous system. It is worth noting that when load balancing is really a concern, a common approach consists in creating pools of tasks in which the different processing units pick up work at runtime. Besides manycore architectures, the use of threads within an accelerator-based platform often raises the problem of data management. The various extensions proposed [13] to support accelerators in the OpenMP standard have indeed shown a real lack of support for distributed memory in such a model that is fundamentally based on a shared memory.

1.2.2 Message passing

The need to support distributed memory suggests that we should either rely on a Distributed Memory System (DSM) or more likely that we should adopt a paradigm based on message passing. While there is no widely adopted standard based on a DSM, the MPI standard is clearly a natural candidate to accelerate parallel applications in such a distributed context. A pure MPI approach would however be very hard on such complex platforms. Indeed the recent trend for hybrid models such as MPI+OpenMP show that we have already encountered the limit of pure MPI approaches on clusters of multicore machines: adding accelerators to these complex platforms will certainly not improve this situation. Besides these programmability considerations, message passing is a convenient way to express the data transfers that need to be done between the host and the accelerators. Implementing such data transfers by hand however introduces significant portability issues because programmers need to directly combine MPI with vendor-specific APIs (*e.g.* CUDA) or at least with accelerator-specific APIs (*e.g.* OpenCL). The cudaMPI and glMPI libraries therefore implement GPU-to-GPU transfers over a cluster of machines enhanced with accelerators with an MPI-like semantic [124]. While it hides most low-level technical problems, cudaMPI only provides a communication layer which allows the different processing units composing a heterogeneous environment to communicate. cudaMPI does not offer any load-balancing facilities, and argues that the dynamic process remapping mechanism used in the Adaptive MPI (AMPI) [94] implementation of MPI could be used to address this difficult issue.

Similarly to multithreading, MPI's SPMD model does not really make it easy to deal with heterogeneous processing resources. Ensuring a good load balancing is not trivial when programmers have to manually map tasks (or data) on a machine that is composed of a number of CPU cores and a number of possibly heterogeneous accelerators which are not known in advance. In most cases, MPI applications that are enhanced to support accelerators adopt a *pure offloading model*: each accelerator is managed by an MPI process which role consists in offloading as much work as possible on the accelerator. Even though accelerators are sometimes paired with a fixed number of CPU cores, it is up to the programmer to ensure that the load is properly balanced between the different processing units within an MPI process.

In order to properly integrate accelerators in the HPC ecosystem, various environments make it possible to mix MPI with other programming paradigms for accelerators, such as CUDA. The S_GPU library for instance makes it possible to efficiently share CUDA devices between multiple processes running on the same node. When there are more MPI processes than CUDA devices, this permits to ensure that the CUDA devices are kept in use even when one of the processes is not using accelerators. Integrating these new programming environments offering support for accelerators within MPI applications is crucial because of the tremendous amount of existing MPI codes, and the wide adoption of the MPI standard among the parallel programming community. More generally, hybrid programming paradigms mixing MPI with another environment dealing

with intra-node parallelism (*e.g.* OpenMP or TBB) is becoming increasingly successful.

1.2.3 Data parallelism

A convenient way to remove portability concerns and to boost programmers' productivity consists in using higher-level approaches which are able to transparently parallelize a specific piece of computation over the different processing units. Models based on data parallelism (*e.g.* HPF [68]) for instance only require that the programmer should partition data so that the different data subsets can be processed in parallel. It is then up to the implementation of the data parallel environment to ensure that the different pieces of data are properly dispatched over the different processing units that can possibly be heterogeneous. This is either achieved through static code analysis, or by the means of a runtime system that takes care of hiding non-portable issues to the programmer.

Skeleton-based or template-based libraries are other examples of environments [78] which hide the inner complexity of the machines by performing internally the mechanisms that would be too complicated to implement by hand. Intel Ct (formerly known as Rapidmind) is an example of a data parallel library that allows C++ programmers to express parallelism by applying common parallel patterns (*e.g.* map, reduce, pack, scatter, etc.) on arrays of data [137, 102]. Intel Ct is automatically able to dispatch computation between the different processing units such as CPU cores, Cell's SPEs or GPU devices. In addition to the C++ Standard Template Library (STL) available on CPU cores, the Thrust library provides templates to execute common C++ operations and to manipulate common data structures on CUDA devices.

In spite of the significant gain in terms of programmability, these approaches are however not suited to all types of computation because rewriting an application as a combination of predefined skeletons requires a certain understanding of the algorithm. Similarly to fork-join parallelism, the scalability of pure data parallelism may also be a concern. Extracting enough parallelism within a single data parallel operation is indeed not always possible: not only the number of processing units may become very large, but the amount of computation is also sometimes not sufficient to efficiently use all the processing resources of a machine enhanced with multiple accelerators that all require to process very large amounts of data at the same time.

1.2.4 Task parallelism

Task parallelism consists in isolating the different pieces of computation into *tasks* that apply a computation kernel on a predefined data set. Tasks can be independent or organized into directed graphs that express the dependencies between the different pieces of computation. With the advent of highly parallel architectures along with explicitly managed memory hierarchies, the simplicity and the flexibility of the task paradigm have become especially successful.

A portable representation of computation In order to allow concurrency, tasks are typically submitted in an asynchronous fashion. Most of the actual work is performed in the tasks. Applicative threads are mostly control threads which goal is to describe computation, and not to perform it. Describing computation as a DAG of tasks provides a portable representation of the algorithms because executing the application efficiently is equivalent to finding a suitable mapping of the tasks on the different processing units. Such mapping is either achieved by the means of static code analysis, or often by relying on a runtime environment. The role of the application

is only to ensure that the kernels implementing the tasks are as efficient as possible, and to provide a sufficient number of tasks to keep all processing units busy. Such a separation of concerns between the design of application-specific kernels and the actual mapping of these tasks on an arbitrary platform allows to obtain portable performances. Tasks are especially convenient for accelerators because data accesses are explicit. We can transparently implement a message passing paradigm that permits to transfer a piece of data to a specific processing unit before it is accessed by a task. Synchronization between the different tasks is also implicitly derived from the task graph, which is significantly easier than to deal with low-level synchronization mechanisms (*e.g.* condition variables or mutexes when using threads).

A late adoption In spite of these advantages, the adoption of such model is relatively recent, even if parallel environments based on task parallelism have existed for a long time. In the early 1990s, the Jade language for instance proposed an extension of the C language that implements a coarse grain task-based model for heterogeneous clusters by offering a single address space and a serial semantic. In 1993, RINARD *et al.* already remarked that *"some tasks may require special-purpose hardware, either because the hardware can execute that tasks computation efficiently, or because the hardware has some unique functionality that the task requires."* [165] Deeply modifying industrial codes to introduce a task paradigm is also still a real concern that needs to be considered with care. Programmers are indeed often reluctant to the idea of having an opaque system which takes care of something as critical as task scheduling. This is especially a concern for HPC programmers who typically want to keep a full control over their application in order to obtain predictable performance. However, due to their growing complexity, it is becoming harder and harder to tap into the full potential of the variety of modern architectures with handed-coded algorithms.

Considering the fast pace of architectural evolution, the programmability and the portability gained with a task paradigm have often overcome the desire to keep a full control over the machine. The relatively wide adoption of the TBB [103] task-based programming environment on multicore architectures, and the introduction of tasks in the third version of the OpenMP standard [150] clearly illustrate this new trend. Numerous other languages have also been extended to support task parallelism such as HPF 2.0 [28] or .NET by the means of the task parallel library (TPL) [128]. Rewriting a code with a task paradigm is more intrusive than approaches based on code annotations, but it allows much more powerful optimization. It is also worth noting that annotation-based environments (*e.g.* StarSs) are often based on tasks internally. Task management overhead is another issue often raised against tasks, especially in the case of fine-grain parallelism for which this overhead cannot be neglected. It has however been shown that it is often possible to implement task parallelism very efficiently: the work-stealing mechanisms used in Cilk [70], Cilk Plus [45] and KAAPI [90] for instance allows scheduling tasks in a multicore system with a very low overhead in spite of a fine granularity.

A generalized adoption A large number of scientific libraries have already been modified to adopt a task paradigm. Dense linear algebra is a classical example of domain where programmers are ready to spend a lot of time to manually tune their compute kernels and their parallel algorithms. The High Performance Linpack (HPL) reference benchmark which solves large dense linear systems has for instance been adapted continuously since the 1980s [155, 120, 65]. Because of the high degree of parallelism and the complexity encountered in current hardware, all major implementations of the LAPACK libraries are however being rewritten using a task paradigm. The

PLASMA library [97] relies on the Quark task scheduler to fully exploit multicore platforms [198]. The FLAME project [159] takes advantage of the SuperMatrix runtime system which schedules the sub-operations of its blocked algorithms [40]. Having such classically fully hand-written libraries rewritten using dynamically scheduled tasks is a clear sign that productivity and portability have become critical concerns.

The adoption of tasks in multicore environments is not limited to HPC applications. The MAC OSX 10.6 and iOS4 operating systems for instance heavily rely on Apple Grand Central Dispatch library (GCD) which is another lightweight library that permits to create tasks which are submitted in a task pool [10]. Apple for instance claims that it is much lighter (about 15 instructions) than creating threads for instance [10]. Tasks are either described with function pointers or as code *blocks* which are obtained by the means of extensions to the C, C++ and Objective-C languages [11]. Various streaming frameworks internally based on tasks (*e.g.* Scalp [143] or AETHER's S-net [81]) also provide a portable abstraction for multimedia, image processing and other mainstream applications.

1.3 Programming Environments

A significant number of programming environments make it possible to develop codes targeting accelerator-based platforms. The most common – and usually hardest – way to program accelerators is to use the low-level toolkits provided by the vendors. In case they are available on the targeted hardware, libraries are also a very productive mean to develop optimized applications with a low entry cost. Not all applications can however be expressed exclusively as a combination of library invocations, so that efficient compiling environments are also required to generate efficient code on a variety of possibly heterogeneous platforms. These compilers can either provide support to generate efficient kernels for accelerators, coordinate computation throughout a complex hybrid accelerator-based system, or both at the same time. In order to fulfill performance portability requirements, auto-tuning facilities are also required to ensure that portable codes are executed as fast as possible on all platforms.

1.3.1 Low-level Vendor Toolkits

Most accelerating technologies are shipped with vendor-specific programming toolkits which permit to directly execute code on the accelerator. Such toolkits are usually chosen by default when programmers try to implement an algorithm on a specific accelerating technology. Even though they typically enable the best performance by providing a direct access to the hardware, most vendor-provided toolkits are however non portable and require very architecture-specific knowledge from programmer. While such vendor-specific toolkits are thus not necessarily suited for end-users, numerous third-party software are built on top of them

The `libspe` is the library provided by IBM to offload computation onto the SPUs of the Cell processor. It is a very low-level C library that requires programmers to manually launch lightweight threads on the SPUs. SPU kernels are written in C and are manually vectorized using compilers' intrinsics. Data transfers between SPUs and main memory (or SPUs) are implemented by the means of DMA transfers, and inter-core synchronization is achieved using mailbox mechanisms [99]. IBM Accelerated Library Framework (ALF) [49] is another programming environment available on the Cell processor. ALF is implemented on top of the libspe and provides higher level

constructs which automate common tasks such as loading a piece of code on a SPU, or performing a Remote Procedure Call (RPC) on a SPU. Both interfaces only provide low-level support, so that programming a Cell processor by the means of libspe or ALF requires a significant expertise. As a result, only few people were actually able to develop applications targeting the Cell processors. The lack of good standard libraries also incited many scientific programmers to abandon the Cell to adopt architectures with more mature tools and a more accessible programming environments.

During the same year, both NVIDIA and ATI/AMD shipped respectively their own proprietary general purpose language along with their first hardware implementation of Unified Shader Architectures (*i.e.* NVIDIA's GeForce 8 series and ATI's Radeon R600). NVIDIA's CUDA (Compute Unified Device Architecture) is based on the C language, so that it was quickly adopted by a wide community of mainstream programmers. CUDA exposes vector parallelism through the use of numerous GPU threads which are much lighter than OS threads. Instead of having a single thread controlling the entire device as on Cell's SPUs, each CUDA thread only handles a very limited number of instructions. Having hundreds of thousand of threads allows the GPU driver to actually hide most of the data access latency by overlapping multiple threads on the same processing unit. A huge number of algorithms have been ported on NVIDIA GPUs with CUDA [148]. Even though getting good performance only came at the price of significant programming efforts, the tremendous performance improvements observed on suitable algorithms makes it worth for a large number of general-purpose mainstream programmers. There are actually numerous attempts to make CUDA a *de facto* standard to exploit the vector parallelism available in modern architectures. MCUDA [174], PGI CUDA x86 [180] and the Ocelot project [54] for instance implement an x86 backend to execute CUDA code directly standard x86 processors. Likewise, FCUDA makes it possible to execute CUDA kernels on an FPGA [153]. NVIDIA also provides an OpenCL implementation for their GPUs, but most advanced hardware capabilities are exclusively exposed in CUDA.

On the other hand, ATI/AMD's CTM hardware interface (Close To Metal) only exposed some assembly-level interface which was far too low-level for most programmers, so that CTM was never really adopted. The first production release of ATI/AMD's GPGPU technology was finally called Stream SDK, which is based on the Brooks [33] higher-level streaming model. AMD finally decided to switch to the OpenCL standardized technology in order to have a wider audience (*i.e.* to capture applications that were already designed on NVIDIA hardware by the means of OpenCL).

The OpenCL standard is actually composed of two distinct parts: a portable language to write compute kernels, and a standardized device management library. The OpenCL language does not transparently introduce vector parallelism in compute kernels. OpenCL language extensions instead propose a standardized alternative to the numerous non-portable vector intrinsics already available in most compilers and is somehow influenced by the syntax of CUDA as well. OpenCL's device management library provides a unified interface to manipulate accelerators and more generally all types of processing units capable of executing SIMD kernels. It is worth noting that such an attempt to provide a standard device abstraction has already been made with the VIA interface in the context of high performance networks . Similarly to VIA, OpenCL device abstraction is a low-level portable abstraction to perform common operations such as memory copies or launching a kernel, but it does not offer any high-level capabilities. Having such a standard interface often prevents vendors from exposing architecture-specific features. OpenCL however makes it possible to implement extensions to the actual standard to implement these features, even if this

does not really comply with OpenCL's portability goals. OpenCL only provides a portable interface and does not guarantee any portable performances: even though the same kernel may be executed on different types of accelerator, programmers still have to specifically tune their code for each architecture.

Programming multiple accelerators and hybrid machines (that combine accelerators with standard processing units) using these low-level vendor-provided toolkits usually requires to manipulate each accelerator separately. This is usually achieved by combining these low-level toolkits using the thread library provided by the operating system (*e.g.* pthreads on Linux). This typically implies an extensive knowledge of parallel programming, low-level programming and a significant understanding of the underlying architectures, which is not really compatible with the portability and the programmability required for end-user programmers. Having a runtime system, such as our proposal, StarPU, makes it possible to hide most of this complexity so that the user can concentrate on writing efficient kernels and designing efficient parallel algorithms instead of dealing with low-level concerns.

1.3.2 Era of libraries

Programmers cannot afford to continuously rewrite the same operations again and again. Instead of reimplementing algorithms for which there already exists heavily tuned implementations, programmers should invoke libraries whenever possible. Due to the complexity of modern architectures, the performance gap between naive implementations and fully optimized libraries indeed keeps increasing. According to Berkeley's classification of scientific computing, most applications can be classified under one of the 13 dwarfs, including sparse linear algebra, or n-body methods for instance [12]. Even though it is unclear whether such a categorization makes sense or there should not be more/less dwarfs, this classification underlines that having an efficient (standardized) library covering the common aspects of each dwarf should make it easy to design scientific applications. For instance, many spectral methods can rely on the FFTW library [69]; the OpenCV library helps designing visualization software; and dense linear algebra applications are typically programmed using libraries implementing BLAS or LAPACK kernels.

Investing into a non-standard type of hardware is a huge risk when developing industrial applications. When the lifetime of an architecture is too short to pay the price of rewriting kernels, the architecture remains unused by most programmers. However, relying on libraries allows domain-specific programmers to easily adapt their application to new platforms without having to actually understand the underlying machine. An interesting example is the Cell processor which did not really survive the HPC ecosystem not only because the entry cost was way too high for programmers, but also because of the total lack of library implementations. Standard libraries such as BLAS or FFTW algorithms were only very partially supported, and the quality of their implementation was not sufficient for industrial applications. On the other hand, NVIDIA CUDA devices – which are not fundamentally easier to program than Cell processors – are for instance shipped with efficient libraries such as CUBLAS, CUSPARSE, CUFFT or CURAND. Even though most people actually tend to avoid writing CUDA kernels, many programmers adopted CUDA because they simply had to replace their library calls by invocations of these libraries. Such an approach makes accelerators accessible to all kinds of scientists and all kinds of programmers, not only computer scientists and/or parallel programming experts. Numerous domain-specific libraries have therefore been adapted to support accelerators such as CUDA or OpenCL devices. The VSIPL signal processing library was ported on GPUs in VSIPL++ [36]. Libraries are not only

used in scientific computing but also by mainstream programmers. Various libraries targeting multimedia applications have for example also been ported on accelerators: GpuCV [63] and NPP [147] for instance respectively adapt the OpenCV and Intel's IPP library to CUDA devices.

While most libraries either target multicore processors or a certain type of accelerator, only few of them actually target an entire heterogeneous system mixing different types of processing units. Since hybrid computing is a solid trend, and the power of manycore CPUs cannot be neglected when combined to powerful GPUs, a significant challenge consists in adapting these libraries to such hybrid environments. Even though library programmers are usually programming experts, their expertise may not cover both their specific domain on the one hand, and parallel/low-level programming on the other hand. In order to let library designers concentrate on developing efficient kernels and efficient parallel algorithms, the portable abstractions provided by runtime systems like our proposal, StarPU, are necessary. For example, the PLASMA and the MAGMA libraries respectively implement state of the art LAPACK kernels for multicore CPUs or for a CUDA device. But neither targets both types of processing units at the same time. Using our StarPU runtime system as an intermediate layer, it was however possible to design a state-of-the-art library implementing LAPACK kernels on hybrid platforms, using PLASMA kernels on CPUs and MAGMA kernels on CUDA devices [AAD+11b, AAD+11a, AAD+10b]. Likewise, we have used StarPU to design a hybrid implementation of the FFTW library relying on kernels from the CUFFT library on CUDA devices and on FFTW kernels on CPUs.

1.3.3 Generating compute kernels for accelerators

Using libraries whenever possible ensures good performance and a low entry cost. However, not all codes can entirely be written as a succession of library invocations, either because there is no such library call to perform a specific treatment on data between various library invocations, or for performance purpose in case the library is not optimized for a specific input (*e.g.* when we have symmetric matrices and the library ignores it). In this case, programmers need to actually (re)write compute kernels which will be executed either on accelerators or on CPUs. In many cases, this actually consists in porting CPU kernels on accelerators.

Modifying industrial applications composed of millions of lines of code to introduce totally new programming paradigms is not always realistic, and would sometimes take longer than the actual lifetime of the accelerating technology. Instead of writing kernels using the low-level vendor-provided toolkits described in the previous section, programmers can rely on higher level environments such as domain specific languages (*e.g.* MATLAB using the Jacket tool which compiles MATLAB code for CUDA devices [167, 199]) or by the means of compilers automatically generating kernels from codes written with high-level constructs or even directly from mainstream languages such a C or Fortran when the code is simple enough and regular enough to be translated efficiently.

The hiCUDA directive-based language provides a set of directives to express CUDA computation and data attributes in a sequential program [87]. hiCUDA makes it possible to improve existing CUDA kernels with a source-to-source translation. Even though hiCUDA hides the CUDA language with a nicer looking programming interface, the actual programming complexity is not so different from CUDA programming because hiCUDA has the same programming paradigm as CUDA. For example, programmers are still responsible for manipulating CUDA specific mechanisms such as allocating data in shared memory.

Automatically translating existing codes into accelerated kernels drastically reduce the entry

cost to take advantage of accelerators in legacy codes. BASKARAN *et al.* for instance use polyhedral models to automatically convert C functions with affine loops into CUDA kernels [18, 19]. Similarly to the widely used F2C source-to-source translator which transforms Fortran code into C codes, the F2C-ACC tool automatically parallelizes Fortran codes to generate CUDA source codes [144]. Even though such a source-to-source translation does not generate fully optimized code, it can be a convenient first step that generates code which can be further optimized later on.

The OpenMP standard is naturally designed to expose vector parallelism by the means of code annotations in C or Fortran codes. The OpenMPC project therefore slightly extends OpenMP to generate efficient auto-tuned CUDA codes. OpenMPC also takes advantage of user-provided hints to generate even more optimized code. This incremental methodology allows programmers to easily start porting their code on accelerators, and gradually provide more information about the algorithm to help the compiler generating better code. This is especially interesting for industrial codes composed of millions of lines of code but which only have a few hot spots that actually need to be accelerated. The well-established PIPS compiler framework [178] is also able to automatically extract portions of code to be accelerated [85] and to transform these portions of C code into OpenMP, FPGA, CUDA or OpenCL kernels [5]. Such a static analysis and code generation would be very useful when combined with our StarPU runtime system because programmers would only supply – possibly annotated – sequential C code, and determine a suitable kernel granularity at compile time [6]. StarPU would thus provide the compiling environment with an portable and efficient abstraction which avoids reimplementing a runtime library to dispatch kernels and to move data. As a result, most of the work required to support a new type of architecture would consist in adding a backend to generate kernel code, instead of having to redesign the runtime library to support new types of interactions between the processing units.

1.3.4 Coordination languages

Besides generating efficient kernels, compiling environments can provide convenient languages to coordinate the different pieces of computation with a high-level interface.

Libraries and templates Some environments provide APIs which are simpler to manipulate than accelerators' native low-level toolkits. The PyCUDA scripting language for instance implements a wrapper library to invoke CUDA from a Python script [113]. Domain specific languages can also provide more or less transparent support for accelerators. The LibJacket library for instance allows MATLAB applications to launch computation on CUDA devices [167]. Hierarchically Tiled Arrays (HTA) are C++ data structures that facilitate data locality and permit to extract parallelism within compute intensive array computation thanks to their block-recursive data-parallel design which is well suited for hierarchical multicore architectures [8].

Annotation-based language extensions Language annotations are a common way to extend a standard language to guide the underlying language's runtime library or to cope with limitations of the original language. Similarly to the OpenMP standard which permits to extract data parallel sections out of sequential C/Fortran codes using *pragma* statements [24], a successful approach to isolate kernels that must be offloaded is to identify them using such pragma code annotations. HMPP [55], StarSs [22, 14] and PGI Accelerators [179] for instance extend C and/or Fortran with

OpenMP-like directives to isolate portions of code which are transformed into tasks which are dispatched between the different processing units at runtime.

HMPP and PGI Accelerator are also able to directly generate device kernels from annotated kernels written in C or Fortran. Considering that programmers can gradually annotate their code, these annotation-based environments, which also automatically generate device kernels, provide a very productive solution which enables a very low entry cost to start accelerating legacy applications. StarSs does not have such code generation capabilities, so that it only provides a programming interface that is more convenient than the low level task interface which we propose in our StarPU runtime system. This however requires a specific compiler to accommodate with the extra annotations. Noteworthy, a common denominator of these approaches is that they require programmers to explicitly specify which data are accessed by the tasks, and to provide the corresponding access mode. Such an extension is also being considered directly in the OpenMP standard [13]. This is conceptually equivalent to requiring that StarPU applications explicitly register data before they are accessed in the tasks.

POP and COHEN propose to extend OpenMP with streaming capabilities [158]. In addition to features available when using the previous annotation-based extensions, their approach permits to directly manipulate streams of data instead of mere blocks of data, so that it is not limited to task parallelism. Such an extension would typically avoid to manipulate numerous fine-grain tasks with a significant overhead when it is possible to simply deal with data streamed to/from a single *persistent task*. This is especially interesting on architectures which only feature little amounts of local memory but have significant bandwidth capabilities, such as the SPUs of the Cell processor.

High-level data parallel languages New languages were also designed to control accelerators with high-level data parallel paradigms. MARS [88], Merge [130, 131] and Hadoop [170] for instance all implement a map-reduce model on GPU-based platforms. The Merge framework compiler automatically converts map-reduce statements to standard C++ code which is interfaced with Merge's runtime library. Similarly to our *codelet* proposal in our StarPU runtime system, merge also lets the application supply different implementations for the different types of processing units supported by Merge (*e.g.* x86, Intel X3000 GPU, etc.). Multiple implementations can even be provided along with different predicates: one can for instance implement a sorting algorithm for vectors smaller than a certain size, and another implementation for vectors which are larger. Software components are also a common way to implement a streaming paradigm. Various projects such as the Scalp [143] multimedia framework therefore schedule kernels described using an XML-based specific coordination language.

The Sequoia [64, 20] data-parallel environment lets programmers declare when a loop can be parallelized by replacing C's `for` statements by `mappar` statements whenever possible. It is also possible to explicitly specify that a function should instantiate a divisible task by adding a `task<inner>` keyword in the function prototype. By providing a second implementation of the same function, and marking it with a `task<inner>` keyword, Sequoia can recursively replace inner tasks by multiple tasks that process subsets of the inner tasks' input data in parallel. At compile time, and given some user-provided configuration file to decide how to divide data, Sequoia therefore recursively partitions computation in a recursive way in order to feed each processing unit with tasks of a suitable granularity with respect to both heterogeneity and memory hierarchy. At runtime, Sequoia ensures that statically mapped data are transferred on the various processing units composing clusters of machines enhanced with accelerators. Given the hierarchical nature

of Sequoia, it could internally rely on our StarPU runtime system to take care of task scheduling at leaf level (*e.g.* at MPI node or at NUMA node level).

RAVI *et al.* also describe a data-parallel language based on explicitly parallel for loops which are marked with a `foreach` keyword [164]. Their environment divides problem data into multiple subsets which are dynamically dispatched between multiple CPUs and a GPU using a work-stealing scheme. When a GPU requests work, a certain number of data chunks are merged together to ensure that the GPU processes a large piece of data. This language is however specifically designed to provide support for the MapReduce paradigm, so that it is unclear whether their work-stealing mechanism will be sufficient to cope with data locality concerns for data-parallel applications which do not adopt the MapReduce paradigm.

1.3.5 Autotuning Frameworks

In the previous sections, we have shown that it is possible to write portable code using high-level tools such as compilers. Ensuring that an application runs everywhere however does not mean that it executes efficiently everywhere: this latter property is called *performance portability*. Hand-tuning is often not applicable anymore in a heterogeneous context, especially when the platform is not known in advance (or does not exist yet). As a result, auto-tuning techniques are required to automatically optimize codes or algorithms so that they fully take advantage of such complex platforms. Auto-tuning covers multiple aspects such as selecting the most suitable code optimization (*e.g.* loop unrolling, automatic tiling, etc.), or selecting the most efficient algorithmic variants when multiple implementations are available.

One of the goals of our StarPU runtime system consists in selecting the best processing unit to perform a piece of computation. StarPU is thus orthogonal and complementary to all auto-tuning efforts which ensure that the dynamically scheduled kernels are fully optimized. Automatic kernel tuning and automatic exploration of design space was already a serious concern on sequential architectures, and later on with any type of parallel architecture. It is even more critical to provide such automated tuning facilities when the number of architecture combinations increases. WILLIAMS *et al.* generate optimized lattice Boltzmann kernels (LBMHD) for very different types of architectures including Itanium, Cell and Sun Niagara processors [195]. The authors obtain significant performance improvements compared to the original code by the means of a script which automatically generates multiple variants of the same code that enable different combinations of optimization (*e.g.* TLB blocking, loop unrolling or reordering, data prefetching, etc.). While this approach automatically selects the best optimization among a list of pre-available techniques, advanced compilers are also able to automatically introduce common optimization methods in existing codes. For example, CUDA-lite is a directive-based approach that generates code for optimal tiling of global memory data [4]. It takes an existing CUDA code and performs a source-to-source transformation to introduce advanced tiling techniques that improve the performance of data accesses in global memory. Auto-tuning techniques are also useful to automatically determine the optimal parameters required to tune the numerous kernels that compose high-performance libraries. Libraries such as ATLAS [194], FFTW [69], OSKI [192] or SPIRAL [161] already obtained portable performance using such auto-tuning techniques on SMP and multicore processors.

Libraries typically use automatically generated code variants which correspond to different optimization (*e.g.* different levels of loop unrolling) or even to different algorithms (*e.g.* quick sort or bubble sort). Since there can be thousands of code variants, auto-tuning facilities are also useful to select the most efficient generated variant. Such auto-tuning techniques are typically

implemented by the means of precalibration runs. For example, the kernels implemented by the ATLAS library call the best variants depending on statically determined conditions [194] and similar techniques are for instance used in the context of GPU computing by the MAGMA project which automatically selects the most appropriate kernel parameters during offline precalibration runs [129]. Some libraries such as FFTW [69] however select the best variant at runtime, which provides more flexibility but requires that the decision overhead remains low. The HMPP compiler also provides a tool that permits to efficiently generate libraries by automatically selecting the best code variation among the different variants obtained when applying the different possible optimizations. This tool however relies on a costly benchmarking step which consists in measuring the performance of all variants on all input sizes. The SkePU [61, 52] and the PetaBricks [9] frameworks allow programmers to supply multiple implementations of the same kernel so that the best parallel implementation is selected at runtime (*e.g.* depending on problem size). Being able to automatically select the most appropriate code variant is thus useful on very different types of applications, ranging from heavily tuned hand-coded libraries to automatically generated compute kernels. A tight collaboration between our StarPU runtime system's performance feedback facilities and higher level tools permits to provide such tools with performance estimation which permits to take accurate decisions to select the best variants.

Besides selecting the best algorithmic parameters or the best code variant for a sequential algorithm, auto-tuning techniques are used to automatically determine how to efficiently parallelize an algorithm over a possibly heterogeneous accelerator-based machine. Another very difficult issue consists in selecting the optimal number of problem subdivision (*i.e.* the granularity) required to ensure that all processing units are kept busy at all time without too much parallelization overhead.

1.4 Schedulers

Dispatching work between the different processing units composing a accelerator-based platform is a problem that has been studied at different levels. Asking programmers to manually map computation is indeed a delicate issue because it requires them to have an idea of the relative performance of the various processing units for the different tasks. Expert programmers like TOMOV *et al.* can design accelerated libraries, but it represents a significant extra programming burden, and requires to deeply modify the code every time there is a new type of hardware available [182]. Besides productivity concerns, manually scheduling code in a portable way is difficult, and it is very hard to consider all parameters at the same time: for instance, it is hard to manually take data transfers into account on a machine with 2 GPUs on PCI-e 16x slots and a GPU on a PCI-e 8x slot. Even though it naturally introduces some overhead, dynamical scheduling however enables performance portability and greatly simplifies programming, especially when data management is automated as well. Another approach is to rely on a compiler which performs a static code to determine a suitable task mapping for an application written in OpenCL [82]. Static code analysis is indeed extremely promising, however it often assumes a certain regularity in the code so that the compiler can make meaningful prediction. In the case of applications invoking accelerated libraries, pure static code analysis is also insufficient.

Some environments also provide a mix of static and dynamic scheduling. Annotation-based languages such as HMPP or StarSs for instance assume that the application can specify which type of processing unit must execute a given task. In the case of a multicore system accelerated with a

single GPU, this typically means that GPU tasks are enqueued and that CPU tasks are scheduled using a simple load balancing scheme (*e.g.* CPU cores pick up tasks from a centralized queue).

At a coarser grain, some environments decide which resource should be assigned to the different applications running simultaneously in the system. JIMENEZ *et al.* for instance decide which resources should be assigned to an application based on the relative speedups measured during previous executions of the applications on a the entire machine or on a subset of the machine: in case there are several applications executed concurrently, their scheduler will assign processing resources to the application which can use them the most efficiently according to the performance history [106]. Likewise, GREGG *et al.* track contention within CPU-GPU based systems to detect which application should be GPU accelerated, and which ones should remain on the CPUs [80]. In both case, the application is told which resources are available, but the scheduler does not provide support for an application that would be spread over the entire hybrid platform. So that programmers must either implement a static scheduling scheme within their application, or rely on a third-party environment taking decisions at a lower granularity.

There are indeed schedulers which provide a finer control by actually deciding which processing unit should execute a specific piece of computation within an accelerated application. Qilin for instance determines the ratio of data which should be processed by the GPU by considering the relative speedups measured on previous kernel executions [133]. This is similar to the scheduling strategies based on performance models in our StarPU runtime system [ATN09, ATNW09], except that Qilin only schedules a single kernel across the entire machine while StarPU dispatches different number of tasks to the different types of processing units without actually determining tasks' granularity. The Anthill runtime environment provides a data-flow oriented framework in which applications are decomposed into a set of event-driven filters, where for each event, the runtime system can use either GPU or CPU for its processing. Anthill implements the Heterogeneous Earliest Finish Time [183] (HEFT) scheduling policy and relies on a performance model database [177]. We had already implemented the same approach in StarPU [ATNW09]. GHIASI *et al.* also consider the case of heterogeneous platforms by ensuring that memory-bound tasks are scheduled on processing units running at a lower frequency, which minimizes the total performance loss by allowing compute intensive kernels to run on the fastest units [76].

Other environments provide support at an even smaller granularity by scheduling kernels concurrently within accelerators. For instance, CUDA indeed typically requires that programmers only submit coarse-grain kernels in order to fully utilize the entire GPU. Efficiently scheduling small tasks makes it possible to create much more parallelism, which is especially interesting for systems coupling GPU(s) with multi-core processors operating at a smaller granularity. TZENG *et al.* schedule irregular workloads within a GPU by executing persistent GPU threads that continuously fetch tasks submitted from the host directly into queues located in GPU memory [186]. Similarly, CHEN *et al.* provide a fine-grain intra-device scheduler which exploit the asynchronous kernel execution capabilities that allow each multi-processor of the device to execute different pieces of code [42]. Their experimental results with a single-GPU configuration show that such a fine-grained approach uses the hardware more efficiently than the CUDA scheduler for unbalanced workloads. This environment also provides Multi-GPU support thanks to work-stealing mechanisms between the different GPUs. However, stealing tasks from another GPU may lead to superfluous data transfers between the GPUs for applications which do not feature enough data locality. Such intra-device schedulers are therefore complementary to our approach because they permit to optimize irregular CUDA kernels which are scheduled between the different units at a

larger granularity by our StarPU runtime system. Relying on StarPU to schedule tasks between the different devices also ensures a certain data locality.

1.5 Data management support

Even though accelerators can provide impressive speedups, data movements are sometimes more costly than performing computation locally on a slow processing unit. Ensuring that data are available on time to keep all processing units busy within an accelerator-based platform is thus a critical issue. The actual added value of runtime systems like our StarPU proposal therefore consists in tightly combining task scheduling with an efficient data management.

1.5.1 Support for explicitly managed memory

A significant source of troubles when programming accelerator-based machines results from the fact that memory is typically distributed into separate memory banks located on the different accelerators. Providing a coherent memory subsystem throughout such a system is often too expensive, so that the low-level programming environments typically require that programmers should explicitly manage memory and implement data transfers by hand. On the Cell processor, data transfers are for instance performed using low-level DMA transactions. Programmers *must* also implement advanced data management techniques such as data prefetching or direct transfers between SPUs [22] to keep SPUs busy, and thus to get any decent performance. There are many potential approaches to deal with such complex explicitly managed memory, either through a high-level programming model or using a compiler that automates efficient data transfers.

Adopting a high-level streaming-based programming environment such as StreamIt [115] or Scalp [143, NBBA09] makes it possible to hide data management concerns from the programmer by automatically streaming data on the accelerators in an efficient way. The Sequoia programming environment also relies on a static user-guided data partitioning to automatically and efficiently transfer data on clusters of heterogeneous machines enhanced with accelerators [64].

On the Cell processor, EICHENBERG *et al.* extend IBM's XL compiler to automatically optimize memory access (*e.g.* to avoid unaligned data accesses) and enforce data caching to avoid continuously fetching the same data [58]. The CPU-GPU Communication Manager (CGCM) combines such a static analysis with a runtime library helping to manage and optimize CPU-GPU communication without strong code analysis [105]. The CGCM indeed tracks allocation units at compile time and invokes the runtime to determine the size and the shape of the data structures so that they can be transferred between the host and the accelerators. From a programming point of view, the CGCM gives the impression of a single coherent address space by transparently translating pointers. Such a technique is useful to design hybrid kernels (as found in the MAGMA library [182, 142]) which can be scheduled by our StarPU runtime system.

1.5.2 Virtually Distributed Shared memory (VDSM)

A classical approach to deal with distributed memory is to implement a *Distributed Shared Memory* (DSM) which provides programmers with a unified shared address space. The GMAC library [73], for instance, implements a DSM that allows CUDA kernels to directly manipulate data normally located in host memory. While a DSM offers a very high productivity, it is extremely hard to

guarantee good (or even reproducible) performance. There are indeed a large number of parameters that need to be tuned in order to ensure a limited overhead. Similarly to the virtual memory mechanisms implemented in operating systems, many DSMs divide the address space into regular blocks. Block size is an example of parameter which must be selected with care: too large blocks result in *false-sharing* issues, and too small blocks lead to a large overhead. The use of a DSM is also akin to the availability of the proper low-level hardware capabilities (or a software-based solution) which are not portable. Recent NVIDIA drivers allow programmers to map part of host's memory in the memory of the device [146], provided that the CUDA device is recent enough to support it. The coprocessors of the Cell processor (*i.e.* SPUs) however do not provide such a virtual memory. Implementing a software-based DSM on the Cell processor is a complex task which was for instance achieved by inserting data management code, either by the means of static code analysis [126, 105] or with dynamic binary code edition [162].

Combining a DSM like GMAC [74] with our StarPU runtime system is for instance useful in the case of large legacy codes which cannot be modified to explicitly register all pieces data. This would for instance avoid having to modify code outside the critical path where productivity matters more than performance. As a result, programmers could incrementally register data which must be accessed efficiently within code hot spots so that they are not managed by GMAC anymore.

1.6 Runtime systems for accelerator-based platforms

Runtime systems provide higher-level software layers with convenient abstractions which permit to design portable algorithms without having to deal with low-level concerns. Compared to most of the approaches presented in the previous sections, the typical added value of these runtime systems is to provide support for both data management and scheduling altogether.

In this section, we introduce some runtime systems which were designed or extended to support accelerator-based platforms. We first present Cell-specific runtime systems which provided support for early adopters of such accelerating technologies. High Performance Library with specific requirements typically integrate their own runtime library. We thus give a few examples of runtime systems that provide support for linear algebra libraries. Finally, we present different general purpose runtime systems which have goals similar to those of our runtime system proposal, StarPU.

1.6.1 Cell-specific runtime systems

Mercury System's MultiCore Framework (MCF) Mercury System's MultiCore Framework lets programmers submit tasks which are kernels that operate on blocks of data [27]. MCF is based on an improvement of the Parallel Acceleration System (PAS) [79] which was a commercial implementation of the Data Reorg standard [51]. The application submits tasks by injecting tiles in *tile channels*. The SPEs autonomously fetch work from these channels until they are empty. MCF automatically implements a pipeline which transfers data asynchronously between main memory (XDR) and the Local Stores on the SPEs. Host memory is divided into blocks which are explicitly dispatched between the different SPEs. Task scheduling is therefore directly derived from data mapping in this data centric environment.

Cell Run Time Library The Cell-RTL (Cell Run Time Library) is a C library that provides an interface to submit fine-grain tasks which are automatically dispatched between the different co-processors or the Cell processor, called SPEs [NBBA09]. Similarly to the Offload API used in Charm++ [116], a task is described by a function index (*i.e.* which kernel to apply) and by a list of buffers (*i.e.* address in host memory and buffer length) that need to be accessed, and their respective access modes (*i.e.* R, W or RW). The Cell-RTL implements an efficient pipeline which is capable of processing chains of tasks on each SPE while performing data transfers in the background. The application submits tasks in a pool of tasks stored in main memory. The various SPEs eagerly fetch tasks from this pool by the means of low-level DMA transfers and using SPEs' mailbox registers. The Cell-RTL is not based on a DSM, but the low-level mechanism required to implement the data transfers are totally hidden to the user. Contrary to StarPU which implements caching techniques that avoid transferring data continuously, the Cell-RTL does not cache coherent data replicates on SPEs' Local Stores because of their limited size of 256 KB. The StarPU port to the Cell [ATNN09] actually relies on the Cell-RTL to perform task offloading and manages data transfers between the main memory and Local Stores on the SPUs. In a way, StarPU leverages the Cell-RTL by adding scheduling facilities and providing with the high-level data management and task dependencies enforcement, permitting efficient task chaining. StarPU could leverage other backends like IBM's ALF [49] or CellSs' runtime [21].

Tagged Procedure Calls (TPC) The goal of the TPC library is to permit to offload tasks on SPUs with a very low overhead [185]. For example, TPC takes care of only creating in-chip traffic when scheduling a task, which improves scalability by reducing contention in main memory. This very low overhead allows TPC to implement a fine grain parallelism. Similarly to the Cell-RTL, TPC provides an asynchronous RPC programming-style and can only offload tasks on the SPEs. Task queues are located directly on the local store of the SPUs and are managed by the means of atomic DMA transactions. TPC however have limited scheduling capabilities, so that TZENAKIS *et al.* for instance show a benchmark implementing a "static load balancing scheme to ensure that all SPEs execute the same number of tasks". Similarly to Cell-RTL, StarPU could leverage TPC with scheduling capabilities by using TPC within a driver for SPUs that would benefits from TPC's excellent latency.

1.6.2 Runtime systems specifically designed for Linear Algebra

DaGUE The DPLASMA project implements dense linear algebra kernels over clusters of machines equipped with accelerators [25]. DPLASMA algorithms are described using the JDF format which is similar to COSNARD and LOI's formalism to represent parametric DAGs [46]. DPLASMA relies on a static data distribution between the different MPI nodes. This partitioning is obtained from a static analysis of the parametric DAG very similar to the methodology previously designed by COSNARD *et al.* [47]. Intra-node task and data management is however performed by the DAGUE runtime system [26]. DAGUE dynamic schedules tasks within a node using a rather simple strategy based on work-stealing. In order to avoid having too many superfluous data transfers caused by work-stealing, programmers typically have to manually specify which type of processing unit should process the different classes of kernel, for instance by specifying that matrix multiplication should only be performed by GPUs which process most of the tasks. Relying on StarPU's scheduling engine within each MPI node would offer more flexibility. Likewise, we could use the techniques applied by DPLASMA to statically map data on a cluster. This would

be complementary to the systematic methodology used to map DAGs of StarPU tasks on a cluster depending on an initial data distribution (see Section 5.4.1 for more details).

TBLAS Similarly to DPLASMA, the TBLAS library implements BLAS and LAPACK kernels for machines accelerated with GPUs [172]. It automates data transfers and provides a simple interface to create dense linear algebra applications. TBLAS assumes that programmers should statically map data on the different processing units, but it supports heterogeneous tile sizes: for example, it is possible to only provide very small tasks to multicore CPUs and to provide large blocks of data to GPUs instead. The runtime system of the TBLAS library is specifically designed for dense linear algebra, but it is worth noting that such granularity concerns commonly occur when mixing different types of processing units which preferred granularity differs greatly, such as multicore CPUs and large SIMD units like GPUs. StarPU should therefore be flexible enough to allow applications to perform work at multiple granularity, for instance by the means of parallel tasks spread over multiple slow processing units at the same time, or by dividing large tasks into smaller independent tasks.

1.6.3 Generic runtime systems for hybrid platforms

Qilin Qilin provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of an heterogeneous machine [133]. This is achieved by training a model for each kernel to determine the amount of time required to process the kernel on the different processing units, depending on data input size. As a result, Qilin evenly dispatches data to ensure that all processing units should finish at the same time, with respect to heterogeneity. Qilin dynamically compiles code for both CPUs (by the means of TBB) and for GPUs, using CUDA. Similar to StarPU, Qilin builds similar performance models. Qilin only allows to execute a single kernel that is executed over the entire machine, while StarPU does not automatically divides an array into multiple sub-arrays of different sizes, instead, scheduling decisions are taken at task level. Generating StarPU tasks with a suitable granularity using Qilin's code generation framework could thus result in more scalable performance.

Charm++ Charm++ is a parallel C++ library that provides sophisticated load balancing and a large number of communication optimization mechanisms [108, 107]. Programs written in Charm++ are decomposed into a number of cooperating message-driven objects called *chares*. Charm++ has been extended to provide support for accelerators such as Cell processors [117] and GPUs [193]. Its low-level Offload API [116] provides an asynchronous RPC-like interface to offload computation with a task paradigm. Even though Charm++'s extensions to support are not a fundamental evolution of the original model, the various techniques implemented in Charm++ provide a real solution to the problem of heterogeneity. In case the different processing units do not share the same endianness, Charm++ for instance automatically converts data when they are transferred so that they can be used throughout the heterogeneous system [109].

KAAPI The KAAPI environment offers support for hybrid platforms mixing CPUs and GPUs [90]. It relies on the dataflow description language Athapascan [71]. Its data management is based on a DSM-like mechanism: each data block is associated with a bitmap that permits to determine whether there is already a local copy available or not [90], which is similar to the techniques used

in StarPU [AN08]. Task scheduling is based on work-stealing mechanisms or on graph partitioning. Thanks to the work-first principle stated by the Cilk project, one can provide a bound on the amount of work-stealing events for tree-based algorithms [70]. As a result, KAAPI is only guaranteed to provide an efficient scheduling for such algorithms. Data transfers and kernel launch overhead makes it hard to implement a straightforward work-stealing policy suitable to any type of algorithm. Applications need to provide extra locality hints, so that the KAAPI runtime system only steals tasks from queues that are close to the current worker. This avoids numerous superfluous data transfers across accelerators or NUMA nodes.

KAAPI is therefore typically designed for cache-oblivious hierarchical algorithms such as the SOFA physical library. Physical objects are partitioned at runtime using SCOTCH or METIS. This partitioning provides the scheduler with hints about data locality, so that SOFA obtains excellent speedup over hybrid multicore platforms accelerated with multiple GPUs [90].

Harmony Harmony is the runtime system at the basis of the Ocelot dynamic execution infrastructure. Ocelot permits to execute native PTX code on various types of architectures. PTX code (which is NVIDIA's virtual ISA) is obtained by compiling CUDA codes directly with NVIDIA's compiler. Ocelot can therefore generate code for either native CUDA GPU devices, x86 CPU cores (by the means of a PTX emulation layer) or on various OpenCL devices (thanks to a code transformation layer based on the LLVM compiler). The Harmony runtime system [54] is a simple yet efficient runtime system that permits to schedule the different generated pieces of code and to reimplement CUDA runtime libraries on a hybrid platform. Native CUDA applications can thus be executed directly on hybrid platforms that may even not feature any actual CUDA device. As a result, Ocelot is also a very powerful debugging platform [62, 54] which is part of the Keeneland project [190]. While Ocelot takes advantage of the parallelism available within PTX code to exploit all processing units, it is conceptually similar to the SOCL library which relies on StarPU to execute OpenCL kernels directly on a hybrid platform [89].

StarSs The StarSs project is actually an "umbrella term" that describes both the StarSs language extensions and a collection of runtime systems targeting different types of platforms. It is a follow-up of the GridSs project which provided support for computational grids [15]. As mentioned in Section 1.3.4, StarSs provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. In this section, we concentrate on the aspects related to runtime systems in StarSs.

Multiple implementation of StarSs are therefore available: GPUSs [14], CellSs [22, 21] and SMPSs [17] respectively target GPUs, Cell processors, and multicore/SMP processors. GPUSs was for instance used in the libflame [159] project which formerly used the SuperMatrix runtime system to schedule dense linear algebra kernels. CellSs implements advanced data management techniques which for instance permits to directly transfer cached data between SPUs: such a technique could be implemented using StarPU's data management library and was for instance applied to allow direct transfers between GPUs.

The main difference between StarPU and the various runtime implementations of StarSs is that StarPU really provides a unified abstraction of driver which makes it possible to deal with hybrid platforms. PLANAS *et al.* have shown use cases where CellSs tasks where nested within SMPSs function, which makes it possible to design tasks that can take advantage of an entire Cell processor, including the PPU [157]. However, tasks are scheduled in a hierarchical fashion

so that CellSs tasks are only scheduled once the corresponding SMPSs task has been assigned to a Cell processor. Having separate runtime systems thus does not however permit to actually schedule tasks between heterogeneous types of processing units unless the programmer explicitly selects the targeted platform, and therefore which runtime system should process the task. The OMPSs project attempts to unify all these runtime systems and to provide support for clusters of hybrid accelerator-based platforms, possibly featuring heterogeneous accelerators [34]. Similarly to StarPU, OMPSs tasks are non-preemptible [38]. OMPSs' scheduler is however *non-clairvoyant* which means that execution times are not known in advance, which prevents OMPSs from implementing scheduling strategies based on an estimation of tasks' termination time.

StarPU and StarSs/OMPSs are closely related, so that we also implemented an interface close to that of StarSs on top of StarPU using either the Mercurium source-to-source compiler used in StarSs, or alternatively by the use of a GCC plugin.

1.7 Discussion

Accelerating technologies have existed for a long time to address domain-specific problems. Due to the growing concerns in terms of energy consumption, their use is however becoming standard in mainstream technologies. More generally, heterogeneous processing units will be required at some point to overcome these physical limitations. We cannot just consider having hundreds of full fledged cores anymore: we need to design simpler processing units in order to reduce the overall energy consumption. However, we still do need to keep a few powerful CPU cores to perform tasks which are not suitable for accelerators such as dispatching the workload and preparing data. The use of accelerating boards have also greatly benefited from the re-use of existing technologies such as GPU devices which made it possible to introduce accelerators in mainstream computers. This hardware trend is also underlined by the fact that all major processor makers (*e.g.* Intel, AMD, IBM) are now designing heterogeneous manycore chips. Besides, accelerating boards such as GPU devices or Intel's Knight Ferry provide a convenient approach to enhance a standard multicore machine with huge parallel processing capabilities

Accelerators and heterogeneity introduce a significant number of challenges throughout the software stack. Numerous libraries have been ported on accelerators, and many languages are being adapted to support hybrid computing. Parallel compilers make it possible to automatically extract parallelism and to generate efficient kernels, for instance by gradually annotating legacy codes. Parallel machines are becoming harder and harder to program, so that efficient runtime libraries are now a key component to provide convenient abstractions for higher-level software layers. Due to their flexibility, tasks are becoming increasingly popular on manycore and accelerator-based platforms, and it is likely that most high-level environments will somehow rely on tasks, at least internally. As a result, most of the high-level environments previously mentioned implement their own runtime library to dynamically offload task-based computation and to implement data transfers. Instead of relying on existing runtime systems, these programmers therefore have to keep their application-specific runtime library up-to-date every time there is a new type of accelerator available, or when a new feature is added into existing accelerator technologies.

Since designing such runtime libraries requires a significant parallel programming and low-level expertise, most of these environments do not actually tap into the full potential of the machine. Such runtime libraries would indeed need to tightly integrate data management and task

scheduling supports to ensure an efficient load balancing without impacting performance with superfluous data transfers. Instead, we propose to design StarPU, a generic runtime system that provides higher-level software layers with a unified abstraction of the different processing units. Its generic and flexible interface should make it possible to adapt the different pieces of software previously mentionned so that they use StarPU instead of their own non-portable application-specific runtime library. Besides the portability ensured by the use of a unified abstraction, these higher-level software layers automatically obtain portable performance by relying on a generic runtime system that supports a variety of accelerator technologies and which keeps integrating the latest features available in the different vendor-provided drivers. By relieving programmers from such a burden, StarPU enables separation of concerns, so that compiling environments can concentrate on generating efficient code, and library designers can focus on designing scalable parallel algorithms and efficient kernels.

There is a huge potential in having all these software layers to actually cooperate. On the one hand, runtime systems should provide compilers and libraries with performance feedback that simplifies auto-tuning. On the other hand, runtime systems should expose abstractions which are expressive enough to let applications guide the scheduler whenever possible. This tight integration between libraries, compilers and the runtime systems provides an outline of a potential *ideal* software stack for hybrid accelerator-based and manycore platforms. Programmers should invoke parallel libraries to perform common operations. Parallel languages should allow to easily write or generate kernels which are not already available in libraries and to coordinate the different kernels. Additionally, annotation-based languages ensure a low entry cost by making it possible to gradually extend legacy codes with annotations. Registered data should be efficiently managed by the runtime system, and an additional software distributed shared memory would allow to access non-registered data throughout the platform.

Part I

Contribution

Chapter 2

A task-based paradigm for Accelerator-Based platforms

Chapter Abstract

In this chapter, we study the suitability of task parallelism for hybrid accelerator-based platforms and we introduce StarPU, a runtime system which permits to schedule tasks and to manage data over such machines in a portable and efficient way. We then describe StarPU's programming model in terms of data and task management. The design of the memory management is detailed, and we describe how StarPU makes it possible to access data with relaxed coherency modes, such as data reductions. We explain how StarPU handles the different types of dependencies, and we illustrate the flexibility and the simplicity of its execution model by showing how we added support for architectures such as GPU devices or Cell's SPUs.

2.1 A programming model based on tasks and explicit data registration

Selecting an appropriate paradigm is a crucial issue to propose a model that should be portable across multiple generations of accelerators. In the previous chapter, we have seen that the flexibility offered by task parallelism has become very successful on hybrid accelerator-based platforms. In this section, we first discuss whether the use of a task-based paradigm is a suitable approach for a runtime system. Data management is a crucial issue encountered when offloading computation between different processing units that do not share the same address spaces. Automatically tracking arbitrary data throughout a system equipped with accelerators is often not doable, and usually not efficient. We thus discuss how such a task-based model also requires programmers to explicitly describe and register all data to avoid having the runtime system to guess approximately what programmers can tell exactly most of the time.

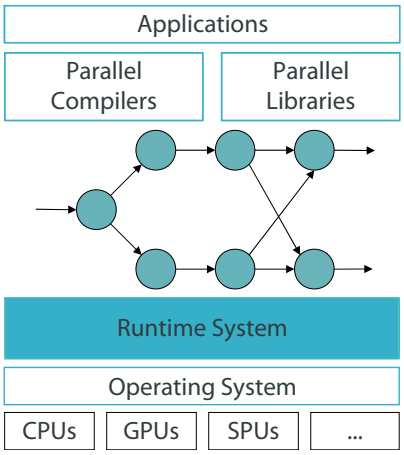


Figure 2.1: Runtime Systems play a central role in Hybrid Platforms

2.1.1 Task parallelism

Task parallelism consists in isolating the different pieces of computation into *tasks* that apply a computation kernel on a predefined data set. Tasks can be independent or organized into directed

graphs that express the dependencies between the different pieces of computation. These graphs are usually acyclic because a task should not depend on a task that already depends on it. Some extensions to this classic *Directed Acyclic Graph* (DAG) model are sometimes allowed in the case of applications that repeat the same sub-graph multiple times (*e.g.* when applying a set of kernels on a data flow). Our approach consists in having DAGs of tasks that can possibly be marked to be regenerated after their execution: the application describes an arbitrary graph of tasks, but the cycles are handled by regenerating new tasks that eventually permit to only consider DAGs of tasks.

Implicit task parallelism Even though brand new source codes are often written using task-parallelism, and many codes are being rewritten to rely on tasks, there exists a huge momentum in scientific programming. The need for perennity is indeed a critical concern for applications that need to run for decades, especially for codes that consists of hundreds of thousands or millions of lines of code. In spite of the deep evolution of the architectures, completely rewriting such applications is simply not realistic in many case. A common way to easily adapt to technological changes is to rely on high-level layers which automatically take care of providing performance portability. Libraries and compilation environments can indeed internally take advantage of task parallelism without impacting the original code. Contrary to actual applications, such environments need to be adapted to take advantage of new technologies anyway. The massive use of high-profile libraries (*e.g.* LAPACK, VSIPL, etc.) clearly indicates that such rewriting efforts are worthy. Another approach to reduce intrusiveness is to extend existing codes with annotations. Similarly to the OpenMP standard which enables parallelism within C/Fortran codes [24], some language extensions have been designed in order to easily offload parts of the computation in a sequential code. The StarSs [22, 14] and the HMPP [55] languages permit to easily modify existing codes, without significantly changing the overall design of the application. Both environments actually rely on tasks internally. It is therefore possible to take advantage of task parallelism in legacy codes without having to rewrite millions of lines of code. Instead, programmers can specifically select the real hot spots in the code that really need to be reworked.

Hybrid paradigms Finally, it must be noted that task parallelism is not always suitable. Besides the granularity considerations previously mentioned, there are codes which are naturally expressed using other parallel paradigms. The pivoting phase of an LU decomposition is an example of naturally synchronous data parallel algorithm that is hardly implemented by the means of dynamically scheduled tasks. This suggests that we must consider hybrid paradigms, which for instance allow to invoke data parallel kernels within tasks. Enabling parallelism at the task level by the means of standard parallel programming environments such as OpenMP, TBB, or even pthreads, is a promising approach to fully exploit the flexibility of task parallelism at a coarse-grain level without forcing programmers to rewrite optimized parallel kernels in a less efficient way with a non-suitable paradigm.

2.1.2 Explicit data registration

The problem of data management should not be overlooked on accelerator-based systems. While accelerators often feature very high processing capabilities with huge internal bandwidth, efficient data transfers between the different processing units are critical. Overall performance is indeed

often affected by the limited speed of the I/O buses which usually constitutes the bottleneck of the system. Still, data management is neglected in many environments which adopt a *pure offloading model* and concentrate on efficient load balancing without really taking care of the data transfer overhead. Instead, data management should be tightly integrated along with task management to ensure both an efficient load balancing and to minimize the impact of data transfers on overall performance.

A portable model should not let programmers manage data by hand. There are indeed too many low-level architecture-specific problems that occur to make it likely to obtain portable performances when taking care of data within a hybrid platform that is not even necessarily known in advance. Besides hybrid machines, the notoriously difficult transition from single-accelerator to multi-accelerator-based platforms illustrates the real complexity of managing data in such complex setups. Hiding the overhead of data transfers usually also requires to use low-level architecture-specific mechanisms such as asynchronous data transfers. Implementing such techniques in an efficient way requires a certain expertise that is not compatible with the goals of a system that attempts to make the use of accelerators more accessible. Another example of difficult issue is found when the problems to be solved do not necessarily fit into the memory embedded on the devices. The input size is therefore limited in many libraries that do not support this delicate situation: even well-optimized libraries such as MAGMA [182] or CULA [95] currently do not deal with arbitrarily large problems.

When considering the overhead and the portability issues related to DSMs, it is interesting to note that adapting an algorithm to task parallelism often requires to understand data layout anyway. If the programmers already knows which pieces of data are accessed by the different tasks, there is no need to guess this information, in a less precise way, and by the means of possibly expensive mechanisms that should be required only when such knowledge is not available. Instead of relying on DSMs, a common aspect found in most accelerator-related language extensions is to require that the programmers should explicitly specify the input and the output data of the different tasks [13, 55]. Knowing in advance which pieces of data are accessed by a task allows powerful optimizations (*e.g.* data prefetching) that are hardly doable with a generic DSM (even though it can sometimes be performed by a static analysis of the code).

Our model therefore assumes that the programmer explicitly registers the different pieces of data manipulated by the different tasks, and specifies which registered data are accessed by a specific task, along with the different types of access (*i.e.* read, write, or both). Another approach would consist in inserting instructions to explicitly upload or download a piece of data on/from an accelerator, but expressing data transfers instead of data accesses in this way has a certain number of drawbacks. It is then up to the application to ensure that all uploaded data can fit into an accelerator. Programmers are also responsible for ensuring that a piece of data is still available and valid prior to computation on a device, which can lead to a severe waste of bandwidth if the application does not manage data locality properly.

2.2 The StarPU runtime system from a user's point of view

The discussion of the previous section considered, we here introduce a new runtime system called StarPU. StarPU automatically schedules tasks among the different processing units of an accelerator-based machine. Applications using StarPU do not have to deal itself with low-level concerns such as data transfers or an efficient load balancing that takes the heterogeneous nature

of the underlying platform into account.

Indeed, StarPU is a C library that provides an API to describe applications' data, and to asynchronously submit tasks that are dispatched and executed transparently over the entire machine in an efficient way. Such a separation of concerns between writing efficient algorithms and mapping them on complex accelerator-based machines therefore makes it possible to reach portable performance and to fully tap into the potential of both accelerators and multi-core architectures.

In this section, we first give a brief overview of the programming model proposed by StarPU and will give more details in the following sections. Then we take a closer look at task and data management interfaces.

2.2.1 Programming model overview

Application first have to register their data to StarPU. Once a piece of data has been registered, its state is fully described by an opaque data structure, called *handle*. Programmers must then divide their applications into sets of possibly inter-dependant tasks. In order to obtain portable performances, programmers do not explicitly choose which processing units will process the different tasks.

Each task is described by a structure that contains the list of handles of the data that the task will manipulate, the corresponding access modes (*i.e.* read, write, etc.), and a multi-versioned kernel called *codelet*, which is a gathering of the various kernel implementations available on the different types of processing units (*e.g.* CPU, CUDA and/or OpenCL implementations). The different tasks are submitted asynchronously to StarPU, which automatically decides where to execute them. Thanks to the data description stored in the handle data structure, StarPU also ensures that a coherent replicate of the different pieces of data accessed by a task are automatically transferred to the appropriate processing unit. If StarPU selects a CUDA device to execute a task, the CUDA implementation of the corresponding codelet will be provided with pointers to local data replicates allocated in the embedded memory of the GPU.

Programmers are thus concerned neither by where the tasks are executed, nor how valid data replicates are available to these tasks. They simply need to register data, submit tasks with their implementations for the various processing units, and just wait for their termination, or simply rely on task dependencies. Appendix A contains a full example illustrating how StarPU was used to easily port a state-of-the-art Cholesky decomposition algorithm on top of hybrid accelerator-based platforms.

2.2.2 A tasking model enabling heterogeneous scheduling

A task is defined as a piece of computation that accesses (and possibly modifies) a predefined set of data handles. In order to facilitate data management and the design of scheduling policies, StarPU does not allow task preemption. The lack of preemption also avoids perturbing kernels that are very sensitive to rescheduling, such as BLAS kernels which are especially sensitive to cache perturbation. It is also much easier to take fine-grain scheduling decisions for well delimited tasks than with a single thread continuously offloading a flow of computation. Dealing with non-preemptible tasks also avoids concurrency issues when managing data (*e.g.* when a preempted task locks a piece of data that needs to be accessed from another flow of computation). This not only reduces the complexity of our model, but this also helps to lower the overhead of task and data management. Preemption also requires a specific support from the hardware or from the

```
void axpy_cpu(void *descr[], void *cl_arg)
{
  struct vector_interface *v_x = descr[0];
  struct vector_interface *v_y = descr[1];

  SAXPY(v_x->n, 1.0, v_x->ptr, 1, v_y->ptr, 1);
}

void axpy_gpu(void *descr[], void *cl_arg)
{
  struct vector_interface *v_x = descr[0];
  struct vector_interface *v_y = descr[1];

  cublasSaxpy(v_x->n, 1.0, v_x->ptr, 1, v_y->ptr, 1);
  cudaThreadSynchronize();
}

int main(int argc, char **argv)
{
  float vec_x[N], vec_y[N];

  (...)

  starpu_vector_data_register(&handle_x, 0, vec_x, N, sizeof(float));
  starpu_vector_data_register(&handle_y, 0, vec_y, N, sizeof(float));

  starpu_codelet axpy_cl = {
    .where = STARPU_CUDA|STARPU_CPU,
    .cpu_func = axpy_cpu,
    .cuda_func = axpy_gpu,
    .nbuffers = 2
  };

  struct starpu_task *task = starpu_task_create();

  task->cl = &axpy_cl;

  task->buffers[0].handle = handle_x;
  task->buffers[0].mode = STARPU_R;

  task->buffers[1].handle = handle_y;
  task->buffers[1].mode = STARPU_RW;

  starpu_task_submit(task);
  starpu_task_wait_for_all();

  (...)
}
```

Figure 2.2: Adding two vectors with StarPU.

operating system: kernels launched on CUDA devices are for instance typically non preemptible. A portable model should not assume that such capabilities are available. Likewise, StarPU guarantees that a processing unit is fully dedicated during the execution of a task, even though some minor external perturbation (*e.g.* OS noise) may sometimes occur.

Programmers do not decide which processing unit should process a given task *a priori*. Tasks are thus likely to be executed on different types of architectures in the case of multicore machines enhanced with accelerators. When defining a task, the application can therefore provide multiple implementations of the same kernel in order to let StarPU choose the most appropriate processing

unit for which an implementation is available. The *codelet* data structure describes such a multi-versioned kernel. It is a gathering of the different implementations available for this kernel (*i.e.* for a CPU core, for an OpenCL device, etc.). Besides pointers to the different kernel implementations available, the codelet data structure contains extra information such as the number of data handles accessed by the kernel. Finally, a StarPU task is defined as a codelet working on a set of data handles.

Figure 2.2 for instance shows the code of an application which submits a task that computes the sum of two vectors. Lines 27 to 32 illustrate how to define a codelet. The `.where` field on line 28 specifies that the kernel is available on both CUDA devices and CPU cores. The `.cpu_func` and `.cuda_func` fields respectively points to the CPU and CUDA implementations defined on lines 1 and 9. Kernel implementations always have the same prototype: the first argument is an array of pointers to the data interfaces that describe input data, and the second argument is a constant value that can be specified in the `cl_arg` field of the task data structure. The number of data handles accessed by the kernel (*i.e.* the size of the first array) is specified by the `.nbuffers` field of the codelet data structure on line 31. More details on the actual data management within the compute kernels are given in Section 2.2.3. Lines 34 to 44 illustrate how to create and submit a StarPU task. The task structure initialized on line 34 is mostly composed of the `.cl` and `.buffers` fields that respectively specify which codelet is implemented by the task, and which data handles are accessed by this codelet. This task structure is asynchronously submitted to StarPU on line 44. It is worth noting that the application does not take any scheduling decision, and simply waits for the completion of the task by issuing a barrier that ensures that all tasks are terminated on line 45.

In this example, we have used the `axpy` BLAS kernel that takes two vectors x and y, and computes $x = \alpha x + y$. This illustrates how StarPU can be used to provide an hybrid implementation of an algorithm that is composed of tasks for which there already exist optimized implementations for the different types of architectures. Programmers can therefore rely on StarPU in order to concentrate on providing efficient task-based algorithms and on writing optimized kernels, instead of dealing with low-level and portability concerns.

Expert programmers can also provide extra hints to guide the scheduling policy. It is for instance possible to define a priority level by setting the `.priority` field of the task data structure. One can also provide a performance model in the `.model` field of the codelet structure so that the scheduling policy can predict the duration of the task. More details on scheduling hints such as priorities and performance models are given in Section 3.3.

2.2.3 Registering data to StarPU

The first step to port an application on top of StarPU is to register the different pieces of data that need to be accessed by the different tasks. An opaque data structure, called *handle* is created during the registration of a piece of data.

This opaque data structure is a convenient mean to fully characterize a piece of data. Handles are used by the application to specify which piece of data are accessed by a task. The handle structure also contains a full description of the registered piece of data, which makes it possible for StarPU to transfer this piece of data between different parts of the machine (*e.g.* from host memory to the memory embedded on a GPU).

From the point of view of the application, registering a piece of data to StarPU ensures that each piece of data will be available to the different processing units that need it. The semantic proposed by StarPU indeed offers the impression of a central coherent memory: each piece of

data is loaded before the execution of a task, and is stored back in main memory afterward. For performance reasons, the actual implementation of StarPU memory subsystem of course avoids to perform unnecessary data transfers, and therefore maintains multiple coherent replicates of the same piece of data. To execute a task, StarPU will make each processing unit perform a request to allocate and to transfer the registered piece of data into a new local replicate. Besides, whenever the data replicate will be modified, all other replicates are invalidated.

Once a piece of data has been registered, its coherency is solely ensured by StarPU, so that an application cannot directly access registered data in host memory without informing StarPU prior to this access. StarPU provides programmers with an acquire/release semantic: a valid copy of a registered piece of data is put back in host memory when calling `starpu_data_acquire` on the corresponding data handle. An access mode is also specified during this acquire phase, so that StarPU invalidates the various data replicates when the application wants to modify the piece of data, which becomes available again when the application calls `starpu_data_release`.

Data Interface API

We still have not precisely defined what is meant by a *piece of data*. A common approach adopted in various programming environments for accelerator-based computing is to reduce all data types to arrays of scalar data types. Language extensions such as StarSs [14, 22], HMPP [55] or even the current propositions to extend the OpenMP standard to support accelerators [13] are typically limited to data types that are arrays of scalar types. An annotation-based approach for instance does not permit to describe more complex data types than what is available in the original language (*e.g.* C or Fortran).

Defining any piece of data by the pair composed of its address and its length is however not sufficient for a generic system such as StarPU. An application performing matrix-based computation may for instance want to register a non-contiguous subset of a matrix. In the context of image processing, one could for instance store a picture using three distinct layers in the RGB format. Higher level environments provide richer sets of data structures: the Intel Ct programming environment [137, 102] for instance relies on C++ Standard Template Library (STL), which provides data types which are much more complex than mere vectors of scalar types. The containers found in the SkePU skeleton programming library are also inspired by the STL. Finally, specific libraries often manipulate their own specific opaque data types. Modifying an application that use such a library is much easier when using library's native opaque data structures directly instead of reimplementing them by the means of vectors. For the sake of generality, defining a piece of data only by an address and a length is therefore not sufficient, a system like StarPU must capture all these situations.

In order to provide more flexibility than environments which only manipulate vectors of scalar types, the StarPU runtime system provides a data structure called *data interface*. A data interface is first defined by a C structure data type that can contain the description of a piece of data. Secondly, implementing a data interface requires to provide StarPU with the methods that permit to manipulate pieces of data stored in this format (*e.g.* allocate a new piece of data, transfer data between host and accelerators, etc.).

Figure 2.3 gives a few examples of C structures that describe different types of data layouts. Scalar vectors are managed by the means of the vector interface shown on Figure 2.3(a). Matrices are handled with the interface on Figure 2.3(b), which for instance contains a *leading dimension* field (`ld`) that makes it possible to describe non contiguous matrices using a semantic that is similar to

```
struct vector_interface {
    void *ptr;
    int n;
    size_t elemsize;
};
```

(a) Vector interface.

```
struct matrix_interface {
    void *ptr;
    int nx, ny, ld;
    size_t elemsize;
};
```

(b) Matrix interface.

```
struct csr_interface {
    int nlines, nrows;
    char *r, *g, *b;
};
```

(c) RGB image interface.

```
struct csr_interface {
    int nnz, nrows;
    float *nzval;
    int *colind, *rowptr;
};
```

(d) CSR matrix interface.

Figure 2.3: Examples of data interfaces.

that of the BLAS library. Figures 2.3(c) and 2.3(d) also show that StarPU is able to manipulate more complex data types. An algorithm processing an image with the red, green and blue colours stored in three different layers could take advantage of the RGB interface on Figure 2.3(c). The *Compressed Sparse Row* (CSR) format is also a classical way to store sparse matrices [171] which is for instance useful when implementing Krylov Methods such as Conjugate Gradients.

Table 2.1: Methods required to implement a new data interface.

Method name	Usage
`copy_methods`	Transfer data between the different memory nodes.
`allocate_data_on_node`	Allocate a piece of data on a memory node.
`free_data_on_node`	Free a piece of data on a memory node.
`register_data_handle`	Register a new piece of data with this interface.
`get_size`	Return data size.
`footprint`	Return a key uniquely identifying the data layout.
`compare`	Detect whether an unused allocated piece of data can be reused to allocate an other piece of data.

The different methods required to implement a new data interface are shown in Table 2.1. These methods provide StarPU with mechanisms to manipulate data stored according to the layout described by the data interface. For instance, the *copy methods* are used to transfer data between the different processing units. In the case of a vector, this is a simple call to a memory copy operation such as `memcpy` or `cudaMemcpy`. The copy methods of complex data types such as the CSR format actually transfer multiple pieces of data. StarPU also needs methods to *allocate* and *free* data on the memory banks associated to the different processing units.

A few other methods are also required to implement other advanced functionality in StarPU. The data filters which are used to automatically subdivide data need to internally register new data handles for the different data subsets (see Section 2.2.3). Scheduling policies may consider the size and the shape of the data layout to predict the performance of the kernel accessing registered data (see Section 3.5). Finally, the memory allocation cache needs a method to detect whether an unused memory region can be reused by another piece of data that needs to be allocated (see

Section 2.3.4).

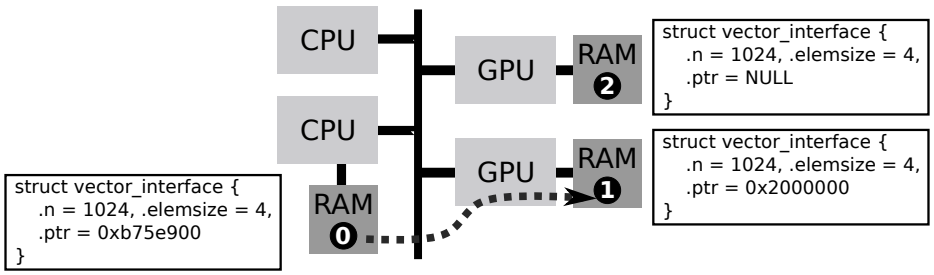


Figure 2.4: Each processing unit is associated to a memory node. An array of data interface structures contains a description of the data replicates in each memory node.

The example on Figure 2.2 illustrates how to manipulate data in StarPU. The `vec_x` and `vec_y` vectors defined on line 20 are registered on lines 24 and 25. `starpu_vector_data_register` is a wrapper function that fills an instance of the `vector_interface` structure described on Figure 2.3(a).

Figure 2.4 shows that each processing unit is attached to a *memory node*. When a task is executed on a processing unit, the different pieces of data are transferred to the memory node attached to the processing unit. CPU cores are for instance attached to the first memory node which stands for host memory (by convention), and the different GPU devices are attached to their embedded memory. When a piece of data is registered to StarPU, an array of structures describing the data interface on the different memory nodes is therefore stored in the data handle structure. The second argument on lines 24 and 25 thus indicates that the piece of data is registered in the first memory node, that is to say, the registered vectors are initially located in host memory.

Once the x and y vectors have been registered to StarPU, they are both associated to a data handle which is used to manipulate them. Lines 38 to 42 for instance specify that the task will access x and y in *read* and *read-write* mode, respectively. When the task is finally executed on the processing unit selected by the scheduler, the appropriate codelet implementation is called (*i.e.* `axpy_cpu` on a CPU core, or `axpy_gpu` on a CUDA device). The first argument passed to the kernel is actually an array of pointers to the structures describing the data interfaces of the different pieces of data on the processing unit's memory node. Assuming that the task is executed on the first GPU on Figure 2.4, the first argument of the kernel therefore contains an array with two pointers to the `vector_interface` structures that describe the local replicates of `vec_x` and `vec_y` located in the memory embedded on the first GPU. On lines 3, 4, 11 and 12, the entries of the array are therefore casted into pointers to `vector_interface` structures. On lines 6 and 14, the programmer can therefore directly retrieve the addresses of the two local vector replicates which were seamlessly transferred to the appropriate processing units prior to the execution of the task.

StarPU provides a set of predefined data interfaces such as scalar values, vectors, matrices, 3D blocks, or sparse matrices stored in the CSR format. In the future, we also plan to add support for the data structures used in mainstream libraries such as VSIPL or OpenCV for example. Expert programmers can embed their own data interfaces when registering a piece of data. This is for instance useful for domain-specific applications. The flexibility of this data interface API permits to register any kind of data structure, so that the application can naturally manipulates its application-specific data types.

Data Filters

In the previous section, we have shown that StarPU provides a flexible data interface API which makes it possible to naturally manipulate the various types of data structures used by the different applications. Some data are however naturally defined by subdividing other pieces of data. The blocked algorithms found in the context of matrix computations for instance consists of tasks that concurrently access sub-blocks of the initial matrix [39].

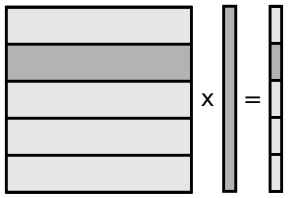


Figure 2.5: Example of a matrix-vector product using data filters: the input matrix and the output vector are partitioned in five sub-parts which can be computed independently. The data accessed to compute the second sub-vector are for instance shown in dark.

Instead of registering all data subsets independently, it is sometimes more convenient to only register a large piece of data and to recursively generate data sub-sets by applying a *data filter* with StarPU. Numerous data-parallel environments (*e.g.* HPF [68] or Rapidmind/Intel Ct [102]) have shown that partitioning data and explicitly accessing data sub-sets is also advantageous from an algorithmic point of view because many data-parallel algorithms are naturally parallelized by providing a suitable data partitioning. On Figure 2.5, a matrix-vector multiplication is parallelized by multiplying sub-sets of the initial matrix with the input vector to compute the different output sub-vectors. An image processing algorithm could also require that some task for example only access the red component of an image registered with the RGB data interface shown on Figure 2.3(c).

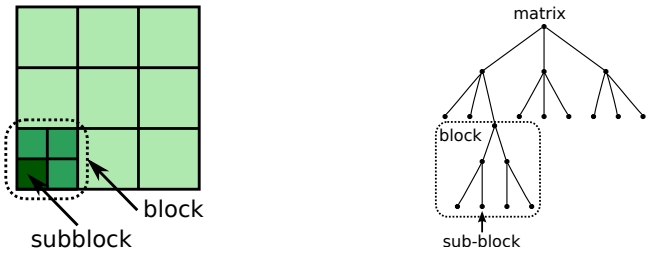
Data coherency is managed at the data handle level in StarPU: one cannot concurrently modify multiple subsets of the *same* data handle at the same time. Applying a filter on a data handle alleviates this problem by making it possible to manipulate independently the handles of the different data sub-sets. From the perspective of StarPU, each data sub-set thus becomes a standalone piece of data once a filter has been applied on the original piece of data. As a consequence, filters cannot divide a data handle into overlapping pieces of data, and it is not possible to directly manipulate a data handle on which a filter has been applied without unpartitioning the handle first.

Each data sub-set becomes a self-standing piece of data: it is therefore described using one of the available data interfaces. Applying a filter on a data handle thus requires to provide a function that fills each of the data interfaces that describe the different data sub-sets. Figure 2.6 for instance gives the code of a partitioning function that divides a vector into multiple vectors of the same size. This function is called for each memory node, and for each data sub-set. The first two arguments of this partitioning function are respectively, pointers to the interface describing the data handle to be partitioned, and a pointer to the data interface structure that must be filled by the function. The `nchunks` and the `id` arguments respectively specify the total number of data sub-sets, and the index of the sub-part that the partitioning function must describe. In this case, the partitioning consists in reading the size of the initial vector (lines 7 and 8), to compute the size

```
1  void vector_filter_func(void *father_interface, void *child_interface,
2            struct starpu_data_filter *f, unsigned id, unsigned nchunks)
3  {
4    struct vector_interface *vector_father = father_interface;
5    struct vector_interface *vector_child = child_interface;
6
7    int n = vector_father->n;
8    size_t elemsize = vector_father->elemsize;
9
10   int chunk_size = (n + nchunks - 1)/nchunks;
11   int child_n = STARPU_MIN(chunk_size, n - id*chunk_size);
12   size_t offset = id*chunk_size*elemsize;
13
14   vector_child->n = child_n;
15   vector_child->elemsize = elemsize;
16
17   if (vector_father->ptr)
18     vector_child->ptr = vector_father->ptr + offset;
19 }
```

Figure 2.6: Code of a filter partitioning a vector into multiple sub-vectors.

of each data sub-set (line 10), and to derive the address and the length of the resulting sub-vector (lines 11 and 12). The data interface describing the sub-vector is then filled with these parameters (lines 14 to 19).



(a) A partitioned piece of data. (b) Its tree representation.

```
1  /* Register the matrix to StarPU */
2  starpu_data_handle matrix_handle;
3  starpu_matrix_data_register(&matrix_handle, ptr, n, n, ...);
4
5  /* Divide the matrix into 3x3 blocks */
6  starpu_data_map_filters(matrix_handle, 2, filter_row, 3, filter_col, 3);
7
8  /* divide the bottom lower block (2,0) into 2x2 subblocks */
9  starpu_data_handle block_handle;
10 block_handle = starpu_data_get_sub_data(matrix_handle, 2, 2, 0);
11 starpu_data_map_filters(block_handle, 2, filter_row, 2, filter_col, 2);
12
13 /* bottom left sub-block (1,0) */
14 starpu_data_handle subblock_handle;
15 subblock_handle = starpu_data_get_sub_data(block_handle, 2, 1, 0);
```

Figure 2.7: An example of partitioned data, its tree representation and the corresponding StarPU code

Figure 2.7 gives an overview of the API that permits to manipulate filtered data in StarPU.

Once the large matrix has been registered (line 3), partitioning functions similar to the one shown on Figure 2.6 are applied recursively on the data handle. Any data sub-set becomes a self-standing piece of data that is associated to a data handle as well: the handle of the block shown on Figure 2.7(a) is for instance retrieved on line 10. Such data filtering can be done in a recursive fashion (lines 10 and 11), so that a registered piece of data can eventually be seen as a hierarchical data structure. The handle of a data sub-set is then characterized by the root data handle, and the path from the root and to the data sub-set (lines 10 and 15).

Applying a filter on a data handle is equivalent to registering each data sub-set. Applying a filter therefore does not modify the underlying data layout. The vector data interface shown on Figure 2.3(a) for instance cannot be used to register the data sub-sets obtained by partitioning a vector following a cyclic distribution. Such a distribution would indeed require to register interlaced non-contiguous sub-vectors than cannot be represented by the means of the vector interface. This could however be achieved by writing a data interface that takes an extra striding argument (*i.e.* the distance between two elements of the vector).

2.2.4 Expressing dependencies

In order to describe an application following a task-based paradigm, we must be able to express dependencies between the different tasks submitted by the application. StarPU provides multiple ways to express such dependencies, by the means of callbacks, with explicit dependencies between task structures, by the means of logical tags, or by relying on implicit data-driven dependencies.

Callbacks and barriers

One of the simplest forms of task dependencies are found in applications with a *fork-join* parallelism. Such algorithms can be implemented by asynchronously submitting the different tasks, and by waiting for each of the task independently, or by issuing a barrier that waits for the termination of all pending tasks. Not all parallel algorithms can however be implemented efficiently using a fork-join paradigm. The shift observed from the fork-join parallelism used in the LAPACK library to the dynamically scheduled tasks used in PLASMA suggests that the often limited scalability resulting from fork-join parallelism is not suitable for manycore platforms.

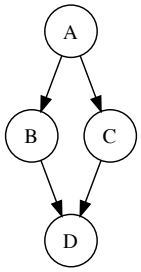


Figure 2.8: A simple task DAG.

Tree-based task graphs can also be implemented by the means of a continuation-passing programming style which consists in submitting tasks during the callback executed during the termination of another task. The entire tree is unfolded by recursively submitting all the children of a node in the tree during the callback executed at the end of this task. Similarly to the fork-join paradigm, the application typically submits the tasks that are immediately ready for execution (*i.e.* the root of the tree), and issues a global task barrier which is only unlocked after the termination of all the leaves in the tree. Using such callbacks is however not very practical for graphs that are not based on trees. On Figure 2.8, expressing the dependency between task D and tasks B and C by unlocking D during the callback of either B or C would typically require that the application maintains a reference count that permits to detect whether both tasks have finished or not. StarPU therefore provides explicit mechanisms to enforce the dependencies between the tasks that compose a DAG.

Explicit task dependencies

```
1  struct starpu_task *deps_taskD[2] = {taskB, taskC};
2
3  starpu_task_declare_deps_array(taskB, 1, &taskA);
4  starpu_task_declare_deps_array(taskC, 1, &taskA);
5  starpu_task_declare_deps_array(taskD, 2, deps_taskD);
6
7  taskD->detach = 0;
8
9  starpu_submit_task(taskA);
10 starpu_submit_task(taskB);
11 starpu_submit_task(taskC);
12 starpu_submit_task(taskD);
13
14 starpu_task_wait(taskD);
```

Figure 2.9: Describing the dependencies on Figure 2.8 with explicit dependencies between task structures.

StarPU provides a low-level interface to express dependencies between the different task structures. The code on Figure 2.9 for instance corresponds to the task graph on Figure 2.8. It is worth noting that dependencies must be expressed prior to task submission. Line 5 shows that it is possible to have a task depend on multiple other tasks, which allows programmers to describe any task DAG by using this generic API.

By default, dynamically allocated tasks are considered as *detached*, which means that once they have been submitted to StarPU, it is not possible to synchronize with the task anymore. This permits to automatically release the internal resources used by a task after its termination.

Contrary to the MPI standard that for instance requires that any asynchronous request should be completed by a wait or a test call, an application written on top of StarPU does not need to explicitly test the termination of each and every task. This avoids having to keep track of all the dynamically allocated task structures that would otherwise have to be destroyed at the termination of the algorithm. On line 7, task D is however marked as *non-detached* because the application explicitly waits for its termination on line 14.

Logical dependencies with tags

Directly expressing dependencies between task structures can be tedious when the tasks are dynamically allocated because the programmer has to keep track of the addresses of the different tasks that were possibly created by totally different pieces of code. Instead of recording such pointers all over the application, it is possible to express dependencies at a logical level with *tags*.

In the case of a library, decoupling dependencies from the actual task data structures for instance allows to transparently generate an arbitrary DAG and to notify that a piece of computation is terminated by unlocking a tag. Instead of having to synchronize with the last tasks internally generated by the library, the application would perceive this library as a black box which exposes a logical integer that can be used to synchronize with the library call.

```
1  #define tagA  0x15
2  #define tagB  0x32
3  #define tagC  0x52
4  #define tagD  0x1024
5
6  starpu_tag_declare_deps(tagB, 1, tagA);
7  starpu_tag_declare_deps(tagC, 1, tagA);
8  starpu_tag_declare_deps(tagD, 2, tagB, tagC);
9
10 taskA->tag_id = tagA;
11 taskB->tag_id = tagB;
12 taskC->tag_id = tagC;
13 taskD->tag_id = tagD;
14
15 starpu_submit_task(taskA);
16 starpu_submit_task(taskB);
17 starpu_submit_task(taskC);
18 starpu_submit_task(taskD);
19
20 starpu_tag_wait(tagD);
21 starpu_tag_remove(tagD);
```

Figure 2.10: Describing the dependencies on Figure 2.8 with tags.

An example of code using tags is shown on Figure 2.10. The logical integers are chosen arbitrarily by the application (lines 1 to 4), and the relationship between the tags and the tasks is established by setting the `.tag_id` field of the task structure (lines 10 to 13). Finally, the application waits for the termination of task D by synchronizing with the tag previously associated with D (line 20). The interesting point is that tag dependencies can be declared even before the actual tasks are created. As a result, the application can first describe the application as a logical DAG of tags, and map task structures on this graph later on.

There can be multiple tags that depend on a given tag (*e.g.* both B and C depends on A on Figure 2.8). Tasks and tags being decoupled, it is possible to declare a new dependency with a tag after the termination of the task associated with that tag. As a consequence, a tag remains usable until it has been explicitly disabled by the application (line 21). In the future, it should be possible to automatically specify the lifetime of a tag by providing StarPU with a method that sets an initial reference count for a given tag, and by automatically disabling tags for which this counter has reached a null value (*i.e.* when the last dependency with this tag has been fulfilled). Another possible extension would be to specify an extra namespace with the different tags: similarly to MPI communicators, a library could transparently use an entire 32-bits tag space without conflicting

with other libraries running in different *contexts*.

2.2.5 Implicit data-driven dependencies for sequentially consistent codes

In many source codes, task dependencies are directly derived from data input and output. A task indeed typically depends on the tasks that generate its input data. In this Section, we therefore show how StarPU allows programmers to boost their productivity by the means of implicit data-driven dependencies that are automatically inferred from data accesses in sequentially coherent codes.

Sequential consistency

In order to automatically build implicit data-driven dependencies, StarPU first requires that the application should unroll the DAG following a sequentially consistent order. LAMPORT defined this property by stating that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [123]

For instance denoting $T(d_i^r, d_j^{rw})$ the task T that accesses data d_i in read mode and d_j in read-write mode, let us consider the following task sequence:

$$A\left(d_1^{rw}\right); B\left(d_1^r, D_2^{rw}\right); C\left(d_1^r, d_3^{rw}\right); D\left(d_2^r, d_3^r, d_4^{rw}\right);$$

This results in having tasks B and C that depend on task A which generates data d_1, and task D to depend on tasks B and C which respectively generate d_2 and d_3. As long as the application submits tasks following the A, B, C, D or the A, C, B, D sequences, StarPU therefore products the DAG depicted on Figure 2.8.

Implicit data-driven dependencies are enabled by default in StarPU, even though it is possible to explicitly disable this property completely, or just for specific data handles. It is indeed sometimes not possible to ensure that the tasks are submitted following a sequentially consistent order. Some applications may also include tasks that have extra side-effects that require that the programmer manually takes care of explicit task dependencies in addition to implicit data-driven dependencies. Some pieces of data can also be updated in a commutative fashion: accessing an accumulator value should for instance not introduce dependencies between tasks that can be reordered without affecting correctness. Programmers should therefore disable implicit-data driven coherency for the data handles that describe accumulators.

Such a sequential consistency is also implicitly required by StarSs [14] and other language extensions that are not expressive enough to make it possible to express explicit dependencies between the different tasks. As shown in Chapter 8, providing implicit data dependencies in StarPU therefore made it much simpler to port these higher-level approaches on top of StarPU.

Function-call like task submission semantic

In sequentially coherent source code, task submission becomes very similar to function calls. In order to boost productivity, StarPU therefore provides a helper function that permits to asynchronously submit tasks with a semantic that is similar to a function call. This interface is directly inspired from the `QUARK_insert_task` function of the Quark scheduler used in the PLASMA library for multicore platforms [35].

```
1  int a = 42; float b = 3.14;
2
3  starpu_insert_task(&cl_A, STARPU_RW, handle1, 0);
4  starpu_insert_task(&cl_B, STARPU_R, handle1, STARPU_RW, handle2, 0);
5  starpu_insert_task(&cl_C, STARPU_R, handle1, STARPU_RW, handle3,
6                     STARPU_VALUE, &a, sizeof(int), STARPU_VALUE, &b, sizeof(float), 0);
7  starpu_insert_task(&cl_D, STARPU_R, handle2, STARPU_R, handle3, STARPU_RW, handle4, 0);
8
9  starpu_task_wait_for_all();
```

Figure 2.11: Example of code relying on implicit data-driven dependencies.

```
1  void codelet_C_func(void *buffers[], void *cl_arg)
2  {
3    struct matrix_interface *matrix = buffers[0];
4    struct vector_interface *vector = buffers[1];
5
6    int a; float b;
7    starpu_unpack_cl_args(cl_arg, &a, &b);
8
9    (...)
10 }
```

Figure 2.12: Accessing the arguments of task C in Figure 2.11.

Figure 2.11 illustrates the programmability improvement resulting from this helper on a sequentially consistent piece of code. StarPU tasks are characterized by their codelet, the data which are accessed (and the corresponding access modes), and optionally some constant arguments passed to the functions that implement the codelet. The first argument of the `starpu_insert_task` helper corresponds to the address of the codelet instantiated by the task. The end of arguments for this variable-arity function is marked by a null argument. The other arguments either correspond to data that were previously registered to StarPU, or to constant arguments that are directly passed to the codelet. For instance, the first task, executing `cl_A`, accesses data handle `handle1` in a read-write mode (line 3). StarPU automatically infers task dependencies by analyzing the access modes associated to the different data handles, so that the second and the third tasks for instance depend on the first one which is scheduled immediately. Constant arguments are passed with the `STARPU_VALUE` argument, followed by a pointer to the constant value, and finally by the size of the argument to be transmitted to the codelet (line 6).

Figure 2.12 gives a possible implementation of the codelet used for the third task of Figure 2.11. As usual, the first argument is an array of pointers to the data interfaces that describe the pieces of data that are registered to StarPU. The second argument of the codelet implementation is a pointer to a stack containing the various constant arguments which were passed to the `starpu_insert_task` helper. The *unpack* method is a convenient function that retrieves the different constant values from this stack. The a and b values that are passed on line 6 of Figure 2.11 are therefore copied into the a and b local variables declared on line 6 of Figure 2.12.

This function-call semantic is very convenient to implement portable libraries or to provide support for compiling environments that permit to offload some portions of code. In Section 5.4.2, we describe how this functionality was extended to exploit clusters of machine enhanced with accelerators. Chapter 8 illustrates how implicit data-driven dependencies were used to provide

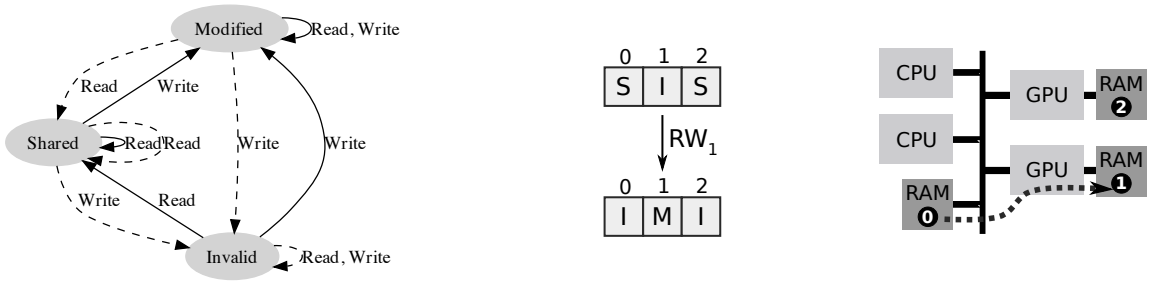
support for the StarSs, GCC and HMPP compiling environments. Finally, appendix A also gives a complete example relying of the `starpu_insert_task` helper to implement a state-of-the-art portable implementation of Cholesky decomposition for hybrid accelerator-based platforms.

2.3 Efficient asynchronous data management

In this section, we describe the different mechanisms used by StarPU to efficiently implement the flexible data model described in Section 2.2.3. Section 2.3.1 presents the coherency protocol that makes it possible to keep the different data replicates coherent. Section 2.3.2 explains how StarPU manages data transfers in an asynchronous fashion. Finally, Sections 2.3.3 and 2.3.4 respectively describe the memory allocation cache, and the memory reclaiming mechanisms.

2.3.1 MSI Coherency Protocol

Transferring data between host memory and the accelerators before and after each task would not be efficient. On Figure 2.4 in Section 2.2.3, we have described that StarPU keeps data replicates on the different memory nodes. When a task is assigned to a processing unit, the different pieces of data accessed by the task are first replicated into the local memory node if they were not already available there. Data replicates are therefore only updated in a lazy fashion when it is strictly needed in order to avoid wasting bandwidth.



(a) MSI coherency protocol automaton.

(b) Data state is updated.

(c) Data transfer between main memory and a GPU.

Figure 2.13: The MSI protocol maintains the state of each data on the different memory node. This state (Modified, Shared or Invalid) is updated accordingly to the access mode (Read or Write).

We must ensure that these replicates are kept coherent when a processing unit modifies one of them (*i.e.* a task that accesses the data handle in a `STARPU_RW` mode is executed). StarPU therefore implements a MSI cache coherency protocol which is illustrated on Figure 2.13. When a piece of data is registered to StarPU, an array describing the state of its replicates on the different memory nodes is stored in the handle data structure. Each entry of this array indicates whether the local data replicate is *modified* (M), *shared* (S) or *invalid* (I). A replicate is considered as modified when the local replicate is the only valid copy. It is marked as shared when the local copy is valid, and that there exists other valid replicates. A replicate is invalid when the local memory node does

not contain an up-to-date copy of the piece of data. The automaton describing the MSI protocol is given on Figure 2.13(a). When a task accesses a piece of data, the entries of the array are updated accordingly to the access mode (read-only or write). Full-line edges show the evolution of the status of the replicate on the memory node attached to the processing unit that executes the task. Dashed lines corresponds to the state modifications on the other memory nodes.

Figures 2.13(b) and 2.13(c) illustrate an instance of application of the MSI protocol. In the initial state of Figure 2.13(b), the piece of data is replicated on nodes 0 and 2, that is to say in main memory and on the second GPU. A task that accesses the data handle in a read-write fashion is scheduled on the first GPU which is attached to memory node 1. One of the valid replicates is selected and transferred to the appropriate memory node as shown on Figure 2.13(c).

Accordingly to the automaton on Figure 2.13(a), remote replicates are then invalidated on memory nodes 0 and 2, and memory node 1 which contains the only valid replicate is marked as modified.

It is interesting to note that the choice of the most appropriate source memory node can be done with respect to the actual topology of the machine. In Section 3.6.3, we will for instance show that the performance prediction mechanisms available in StarPU are used to detect which memory nodes are the closest to a processing unit that needs to access a piece of data that is not available locally.

StarPU completely decouples the problem of the cache coherency protocol from the actual implementation of the data transfers. Implementing a new data interface indeed requires providing the different methods which permit to transfer a piece of data between two memory nodes. When a processing unit needs to access a piece of data which is invalid on the local memory node, an abstract method is called to transfer a valid data replicate into the local memory node. As a result, the MSI coherency protocol is applied regardless of the underlying data interface.

2.3.2 Decentralized asynchronous data management

Maintaining the coherency of the various data replicates requires to be able to transfer coherent replicates between the different parts of the machine. Data transfers might thus be required between any combination of processing units (*e.g.* from a NVIDIA CUDA device to an AMD OpenCL device). The different devices however do not only have different programming interfaces, but also different capabilities (*e.g.* asynchronous transfers, direct transfers between multiple devices, etc.). Besides, constructors impose different constraints in their drivers, such as different levels of support of thread-safety.

Data requests

To cope with all these disparities, StarPU provides a unified view of the different types of processing units, and of the memory attached to the different devices. Each processing unit is controlled by a driver that is attached to a memory node which represents the local memory bank. Similarly to the message passing paradigm classically used to exchange data in distributed systems, data transfers are managed in a decentralized fashion by the means of *data requests*. A data request describes a data transfer that must be performed between two memory nodes. During its initialization phase, StarPU creates a list of data requests for each memory node. When a task ends on a processing unit, the driver in charge of this processing unit first checks whether there are pending data requests submitted to its local memory node before asking a new task to the scheduler.

```
int vector_cpy_cuda_to_ram(void *src_interface, unsigned src_node,
                           void *dst_interface, unsigned dst_node, cudaStream_t stream)
{
  struct vector_interface *src = src_interface;
  struct vector_interface *dst = dst_interface;
  size_t size = src->n*src->elemsize;

  cudaError_t cures;
  cures = cudaMemcpyAsync(dst->ptr, src->ptr, size, cudaMemcpyDeviceToHost, stream);
  if (!cures)
    return -EAGAIN;

  /* Synchronous fallback */
  cudaMemcpy(dst->ptr, src->ptr, size, cudaMemcpyDeviceToHost);
  return 0;
}
```

Figure 2.14: Example of method defining how to transfer a piece of data registered with the vector interface between a CUDA device and host memory.

Given the source and the destination memory nodes, the driver performs the data transfer by selecting the appropriate copy method in the data interface data structure. Figure 2.14 for instance shows the implementation of this method between a CUDA device and host memory in the case of the vector data interface. A data request is defined by the source and the destination memory nodes, as well as the data interface structures that describe the source data and the piece of memory where the piece of data must be copied. An optional callback function can also be provided along with the data request to signal the termination of the data transfer.

Asynchronous data transfers

Data transfers can be very long due to the main bus typically being a bottleneck. Nowadays most acceleration cards support asynchronous data transfers, so that these transfers can be overlapped with computations.

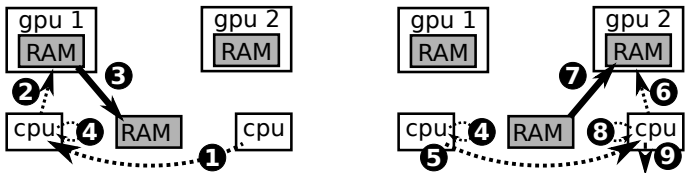
Transfers are not necessarily completed immediately when the drivers process the different data requests. Handling a data request on a processing unit that supports asynchronous data transfers indeed simply consists in initiating a data transfer that will be completed later. In addition to the list of new data requests attached to each memory node, StarPU therefore maintains a list of pending asynchronous data requests, which completion is tested regularly (*e.g.* after the termination of the different tasks or during scheduling holes).

In order to fully overlap data transfers with computation, transfers must be programmed early enough. StarPU therefore provides the scheduling policies with a data prefetch mechanism that consists in programming a data transfer by submitting a data request in advance. It should be noted that data prefetching permits to exploit scheduling holes even if the hardware does not support asynchronous transfers.

Indirect data transfers

Complex interaction may be required to implement data transfers within an accelerator-based machine possibly featuring different types of accelerators at the same time. However, not all devices are compatible, so that we cannot always implement data transfers by the means of a

single data copy between two devices. It is for instance currently not possible to transfer a piece of data directly between a NVIDIA CUDA device and an AMD OpenCL device.



(a) Step 1: GPU1 to RAM (b) Step 2: RAM to GPU2

Figure 2.15: Transfer between two GPUs are implemented by the means of chained requests when direct transfers are not allowed.

When a piece of data must be transferred between two processing units that cannot communicate directly, StarPU submits a chain of data requests, so that the piece of data can indirectly reach its destination. Figure 2.15 for example illustrates the case of two GPUs that do not support direct GPU-GPU transfers. On Figure 2.15(a), the GPU2 worker (running on a dedicated CPU), which needs some data D, first posts a data request to fetch D into main memory (Step 1). Once D is not busy any more, the GPU1 worker asks the GPU driver (*e.g.* CUDA) (Step 2) to asynchronously copy D to main memory (Step 3). On Figure 2.15(b), once the termination of the transfer is detected (Step 4), a callback function posts a second data request between main memory and GPU2 (Step 5). Steps 6, 7 and 8 are respectively equivalent to 2, 3 and 4. Finally, the GPU2 worker is notified once the data transfer is done (Step 9).

This mechanism can deal with different types of accelerators (*e.g.* AMD and NVIDIA at the same time). If one has both a synchronous GPU and an asynchronous GPU, the transfer between main memory and the asynchronous GPU will still be done asynchronously. Likewise, adding support for the direct GPU-GPU transfer capabilities introduced in the fourth release of the CUDA toolkit [146] only required to change the routing method used to detect whether an indirect request is needed to perform a transfer between two devices or not, and to use the appropriate functions in the copy methods to transfer data between two NVIDIA GPUs. In other words, an application already written on top of StarPU can take advantage of the latest driver improvements without any further modification. This illustrates the significant advantage of using a runtime system in terms of performance portability.

2.3.3 Memory Allocation Cache

Allocating a piece of memory can be an expensive operation, especially on accelerators which often have limited memory management capabilities[1]. Calling the `cudaMalloc` function on a CUDA device can for instance stall the GPU while the CUDA (host-controlled) driver allocates a piece of memory. This not only creates a significant latency overhead, but this also introduce extra synchronization barriers within the CUDA driver which therefore blocks until the end of the various pending asynchronous calls.

Considering that in many algorithms, the different pieces of data have similar sizes and similar layouts, a common approach to avoid memory allocation overhead consists in creating a memory

[1]This issue starts to be a concern for constructors: NVIDIA Fermi cards for instance feature a 64-bit address space controlled by the means of a virtual memory system.

allocation cache. Instead of systematically allocating new pieces of data which are freed when they become unused (*e.g.* the data handle was unregistered or the replicate was invalidated), StarPU puts unused data in a per memory node data cache that is queried when a new piece of data must be allocated. StarPU calls the `compare` method of the data interface data structure on each entry of the cache. This method permits to check whether an unused data replicates can be reused to store another piece of data with the same data interface (*e.g.* an unused vector can be reused to store a vector of the same size). If an entry of the cache is compatible (*cache hit*), the unused piece of memory is reused and removed from the cache. If no entry is compatible (*cache miss*), StarPU calls the `allocate` method of the data interface data structure to actually allocate a new piece of data. If this method fails, the cache is flushed and StarPU calls `allocate` again.

2.3.4 Memory reclaiming

Many hand-coded applications such as the highly efficient CULA [95] and MAGMA [96] LAPACK implementations are limited to problems that can fit in the memory of the accelerators. Host memory is however sometimes an order of magnitude larger than the memory embedded on the devices. Similarly to out-of-core algorithmic that deals with problems that cannot fit into main memory, it is therefore crucial that StarPU permits to take advantage of the entire memory available in a machine in order to fully exploit its processing capabilities. Some library designers manually address this delicate programming issue [84], but this is transparent with StarPU which prevents programmers from introducing complex non portable mechanisms into their algorithms.

Considering that algorithms which exhibit a reasonable amount of data locality typically only access a subset of data at a time, the entire problem needs not be stored in each device at all time. StarPU detects unused data replicates by maintaining a per-replicate reference count of the tasks that are currently using it. When a memory allocation failure occurs on the device, StarPU therefore uses a memory reclaiming mechanism which discards unused and invalidate data replicates from the memory of the device. Similarly to other data transfers, the transfers induced by this memory reclaiming mechanism are performed asynchronously.

Algorithm 1 summarizes the memory allocation procedure. When StarPU needs to allocate a piece of data on a memory node, the first step consists in looking into the cache for a matching entry. If none is found, the `allocate` method of the data interface is called. In case there is not enough memory, StarPU flushes the cache associated to the memory node and tries to allocate the data again. If this is still not sufficient, a memory reclaiming operation is performed to remove superfluous data replicates from the memory node. If the allocation method still fails the overall procedure is finally aborted.

Assuming that memory reclaiming is a rare event, its current implementation consists in evicting all unused data and to flush the entire cache, regardless of the amount of memory that needs to be allocated. We are investigating less aggressive policies that would for instance only discard the least recently used pieces of data. Another possible improvement would be to automatically discard such infrequently used pieces of data when the amount of memory allocated on a device reaches a certain threshold. In Section 3.6, we will show examples of scheduling strategies that minimize the amount of data transfers by penalizing scheduling decisions which incur superfluous data transfers. Likewise, we could also reduce the risk of filling up the memory of the different devices by penalizing the scheduling decisions that would lead to allocating new pieces of data which are already replicated on other memory nodes.

Algorithm 1: Memory allocation procedure.

```
entry ← CACHE_LOOKUP(data, node);
if (entry ≠ NULL) then
    Remove entry from Cache;
    return entry;
attempt_cnt ← 0;
while (attempt_cnt ≤ 2) do
    new_interface ← ALLOCATE(data, node);
    if (new_interface ≠ NULL) then
        return new_interface;
    switch (attempt_cnt) do
        case 1
            CACHE_FLUSH(node);
        case 2
            MEMORY_RECLAIM(node);
    attempt_cnt ← attempt_cnt + 1;
return FAILURE
```

2.4 Relaxing the data coherency model

In this section, we present new types of data access modes which permit to relax the data coherency model previously described. Combining read and write access modes is indeed sometimes not sufficient to properly capture all the situations which occur in parallel algorithms. The first extension consists in providing a *scratch* access mode which provides kernels with a local scratchpad memory. In order to deal with massively parallel machines, we then provide a *reduction* access mode which is a convenient and powerful mean to express high-level algorithms while saving a lot of parallelism.

2.4.1 Scratch access mode

Writing in-place computation kernels is sometimes more difficult and/or less efficient than out-of-place kernels. The TRMM BLAS3 routine of the CUBLAS library is for instance reported to perform three times faster in its out-of-place version on Fermi architectures [146]. Out-of-core kernels rely on the availability of extra local memory, here denoted as *scratchpads*, in addition to the different pieces of data accessed in a standard way using R, W or RW access modes. Many parallel programming environments therefore have a notion of *local memory* which permits to manipulate local temporary buffers within computation kernels. Contrary to data accessed using these regular modes, the lifetime of scratchpads is limited to the kernel execution. Even though a scratchpad is both readable and modifiable, they must not be manipulated with a RW mode because there is no need to maintain their coherency.

One could presumably allocate memory directly within computation kernels, but this is forbidden by StarPU which maintains a data allocation cache. A data allocation could therefore potentially fail because all memory is used to cache recently used data. Since tasks are non-preemptible in StarPU, we cannot tell StarPU to remove some entries from the cache in a safe way. This might

indeed introduce data transfers and other potential sources of deadlocks which we avoid by allocating and locking all pieces of data beforehand, in a specific order as detailed in Section 2.5.1. Allocating memory on an accelerator can be a very expensive operation: some CUDA devices are for instance sometimes stalled while invoking `cudaMalloc`. Programmers should thus rely on StarPU's data allocation cache facility instead of allocating and deallocating temporary memory within each kernel.

Another common approach consists in allocating a local scratchpad for each processing unit during the initialization phase and to release all resources at the end of the algorithm. Doing so is possible with StarPU by executing a codelet specifically on each worker. Explicitly allocating memory on each processing is not really in the spirit of our model which intends to automate data management and to hide low-level and non-portable concerns such as efficiently allocating local memory. Allocating large buffers on each device in advance may also cause more pressure on the data management library. Since there is less memory available for the remaining tasks, the memory allocation cache becomes less efficient, which potentially affects data locality and increase the likeliness of encountering – expensive – memory reclaiming mechanisms.

Since scratchpads cannot be efficiently implemented using existing access modes or by explicitly allocating beforehand or within compute kernels, we have introduced a new type of access mode which is selected using the `STARPU_SCRATCH` value. When a task accesses a data handle using this new access mode, StarPU automatically allocates a piece of data locally before the execution of the task, and releases this piece of memory when the task is terminated. For performance reasons, such temporary buffers are actually fetched directly from the data allocation cache before execution, and put back in the same cache after the termination of the task. This transparently provides programmers with good locality because different tasks can reuse the same buffer which is likely to be already in the cache, so that there are less cache misses and less TLB misses as well.

2.4.2 Reduction access mode

In some situations, maintaining data strictly coherent at all time is not efficient on a massively parallel architecture. For instance, algorithms modifying a common piece of data very often might suffer from a serious lack of parallelism if this piece of data is accessed in a RW mode. Not only all tasks modifying it would be serialized, but the actual content of the data would also have to be transferred throughout the system between each task. In case tasks actually modify this piece of data in a commutative way, data reductions offer a powerful mean to update the variable lazily without wasting parallelism.

A data reduction consists in maintaining multiple incoherent data replicates which are updated by the different tasks accessing the piece of data in such a reduction mode, and to finally compute a coherent variable by reducing all local data together. This reduction phase is typically performed with a tree-based parallel algorithm that makes reduction-based algorithms scalable. Updating an accumulator (*e.g.* with the += C operator) is a typical example of operation that can be performed with a reduction. Another example of reduction is found when computing the minimum or the maximum value of an array: after searching for local extrema, it is possible to compute global extrema by performing a reduction with the *min* and the *max* operators.

Noteworthy, compilers can sometimes also detect such reduction patterns and generate code accordingly [175]. Providing compilers with a runtime system offering such a data reduction abstraction therefore makes it easier to generate efficient code, without having to reimplement a fully asynchronous tree-based reduction algorithm relying on vendor-specific data management

APIs within the compiler.

```
1  void init_cpu_func(void *descr[], void *cl_arg)
2  {
3    double *dot = STARPU_VARIABLE_GET_PTR(descr[0]);
4    *dot = 0.0;
5  }
6
7  void redux_cpu_func(void *descr[], void *cl_arg)
8  {
9    double *dota = STARPU_VARIABLE_GET_PTR(descr[0]);
10   double *dotb = STARPU_VARIABLE_GET_PTR(descr[1]);
11   *dota = *dota + *dotb;
12 }
13
14 struct starpu_codelet_t init_codelet = {
15   .where = STARPU_CPU,
16   .cpu_func = init_cpu_func,
17   .nbuffers = 1
18 };
19
20 struct starpu_codelet_t redux_codelet = {
21   .where = STARPU_CPU,
22   .cpu_func = redux_cpu_func,
23   .nbuffers = 2
24 };
```

Figure 2.16: Codelets implementing the data accumulator used on Figure 2.17.

We have therefore added the `STARPU_REDUX` access mode to StarPU which permits to update a piece of data in a commutative way. In order to use this access mode, programmers must provide two codelets. Figure 2.16 for example shows the CPU implementation of these two codelets in the case of an accumulator. The role of the first codelet is to set the variable to the a neutral value for the operator (*e.g.* 0 for an accumulator, or $-\infty$ for the *max* operator). This initialization method is called when a new local variable is created prior to the execution of the first task accessing the data handle in a `STARPU_REDUX` mode on the local worker. The second codelet reduces two variables together and updates the first variable with the reduced value.

Figure 2.17 gives a complete example of code which computes the dot product of a vector by the means of a data reduction. The `dot` variable is registered as usual on line 35. The reduction operators associated to this handle are set on line 36. On lines 39-41, each task computes a local dot product of the two input vectors (lines 9-10) and adds it to the accumulator (line 12). When the `dot` variable is unregistered (or when we access it again using R or RW modes), a valid piece of data is constructed by the means of a reduction which is performed transparently to the programmer. We could have used a RW access mode when accessing the `dot` accumulator, but reductions allow to execute all tasks in parallel. As illustrated on this example, data reductions integrate nicely with implicit (and explicit) dependencies.

DURAN *et al.* also propose to extend OpenMP with user-defined reductions because reductions are currently limited to a set of base language operators applied on scalar types: this for instance prevents from having a double complex variable natively manipulated as an accumulator [57]. In Section 7.6 on 176, we illustrate the efficiency of data reductions with a highly scalable hybrid implementation of a Monte-Carlo algorithm. It must also be noted that commutative operators appears can be found at a high level too, much beyond the classic examples of data reductions of scalar types typically allowed by OpenMP (*e.g.* accumulators). For example, adding the different

```
void dot_cpu_func(void *descr[], void *cl_arg)
{
  float *local_x = STARPU_VECTOR_GET_PTR(descr[0]);
  float *local_y = STARPU_VECTOR_GET_PTR(descr[1]);
  double *dot = STARPU_VARIABLE_GET_PTR(descr[2]);
  double local_dot = 0.0;

  int n = STARPU_VECTOR_GET_N(descr[0]);
  for (int i = 0; i < n; i++)
    local_dot += local_x[i]*local_y[i];

  *dot = *dot + local_dot;
}

struct starpu_codelet_t dot_codelet = {
  .where = STARPU_CPU,
  .cpu_func = dot_cpu_func,
  .nbuffers = 3
};

double dot_product(float *x, float *y, int size, int N)
{
  starpu_data_handle x_handles[N], y_handles[N];

  /* Register input vector subsets. */
  for (int i = 0; i < N; i++)
  {
    starpu_vector_data_register(&x_handles[i], 0, &x[i*(size/N)], size/N, sizeof(float));
    starpu_vector_data_register(&y_handles[i], 0, &y[i*(size/N)], size/N, sizeof(float));
  }

  double dot = 0.0; starpu_data_handle dot_handle;

  /* Register the dot variable and define the reduction operators. */
  starpu_variable_data_register(&dot_handle, 0, &dot, sizeof(dot));
  starpu_data_set_reduction_methods(dot_handle, &redux_codelet, &init_codelet);

  /* Compute the local contribution of each pair of vector subsets */
  for (int i = 0; i < N; i++)
    starpu_insert_task(&dot_codelet,
        STARPU_R, x_handles[i], STARPU_R, y_handles[i], STARPU_REDUX, dot_handle, 0);

  /* Unregister data to StarPU so they are available to the user again */
  starpu_data_unregister(dot_handle);
  for (int i = 0; i < N; i++)
  {
    starpu_data_unregister(x_handles[i]);
    starpu_data_unregister(y_handles[i]);
  }

  return dot;
}
```

Figure 2.17: Dot product based on data reductions.

contributions of the input matrices is a commutative operation during a matrix multiplication. Data reductions should therefore be useful to design scalable dense and sparse linear algebra algorithms.

2.4.3 Elements of Implementation

When a processing unit accesses a piece of data using a standard R,W or RW mode, StarPU ensures that a valid data replicate is available in the memory node attached to the processing unit (*e.g.* host memory for a CPU core, or embedded RAM for a GPU device). Access modes with a relaxed coherency are not managed at memory-node level, but at processing unit level because there can be multiple workers accessing (and modifying) non coherent data replicates. When accessing a data handle in such way for the first time on a worker, StarPU allocates a local data interface corresponding to the data layout of the handle. The value of data replicates accessed in a `STARPU_REDUX` mode are also initialized as a neutral element using the user-provided specified initialization codelet. Scratchpad memory is not initialized.

After the termination of a task accessing a piece of data with a `STARPU_SCRATCH` mode, the local data replicate is freed. It should be noted that it is actually taken from (resp. put back) directly from (resp. into) the memory allocation cache to ensure a minimal overhead and to maximize data locality. Processing units can directly access already initialized local data replicates which are accessed in a `STARPU_REDUX` mode until the application accesses the corresponding data handle in a coherent way again (*e.g.* with a R, W or RW mode). When such a coherent access is made, StarPU transparently submits a tree of tasks which perform the reduction of the different local data replicates. Since all data replicates actually correspond to the same data handle, we cannot directly manipulate the various replicates using the original data handle. During the reduction phase, StarPU thus registers each local incoherent replicate as a standalone temporary piece of data. When the reduction phase is terminated (*i.e.* all tasks submitted internally have been processed and a coherent data replicate has been reconstructed), these temporary data handles are unregistered and the local data replicates are released too. At that time, the original data handle becomes accessible to the application again.

Instead of hardcoding a data reduction operation within StarPU, we therefore implement this reduction phase directly as a StarPU algorithm, transparently for the application. This requires much less code modification than if we had to manually implement an efficient hierarchical implementation of this reduction phase (*e.g.* to reimplement dependencies between the different tasks). Internally submitted tasks also take advantage of all the features available in StarPU, such as data prefetching and asynchronous task execution. This also means that the reduction tasks are scheduled just like any other StarPU tasks. Not only this allows to efficiently interleave tasks from the application and internal tasks, but it also makes it possible to automatically deal with some delicate situations that would have been hard to solve by hand. For example, it is possible that the user did not provide an implementation of the reduction codelet on each type of processing unit because the reduction phase is not suitable at all for a GPU device for instance. This is a common situation for StarPU's scheduler which simply has to deal with extra constraints. Finally, internal tasks are totally transparent to the application because StarPU automatically introduces implicit dependencies with tasks accessing the data in a coherent way.

2.5 Execution of a Task within StarPU

Figure 2.18 gives an overview of the journey of tasks within StarPU, thus providing a general idea of how StarPU modules are related. The application first submits tasks (Step 1). When a task becomes ready (Step 2), it is dispatched to one of the device drivers by the scheduler (Step 3). The

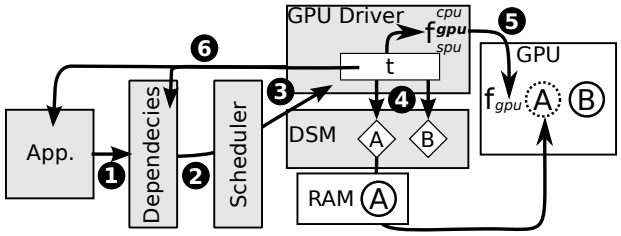


Figure 2.18: Overview of the path followed by a task within StarPU.

DSM ensures the availability of all pieces of data (Step 4). The driver then offloads the proper implementation for the task (Step 5). When the task completes, tasks depending on it are released and an application callback for the task is executed (Step 6).

The scheduling decision is taken between Steps 2 and 3 on Figure 2.18, after all dependencies have been successfully fulfilled. The model implemented by StarPU is very simple: a ready task is *pushed* into the scheduler (Step 2), and an idle processing grabs work from the scheduler by to *popping* scheduled tasks (Step 3). The scheduler thus appears as a generic black-box, and the actual implementation of these two operations is defined by a scheduling policy. Since there does not exist a single perfect scheduling policy which suits all algorithms, StarPU provides a flexible scheduling engine in which expert programmers can plug their own custom policy, or select one of the predefined scheduling policies. The design of the scheduling engine and the different policies are described in details in Chapter 3.

2.5.1 Enforcing explicit dependencies

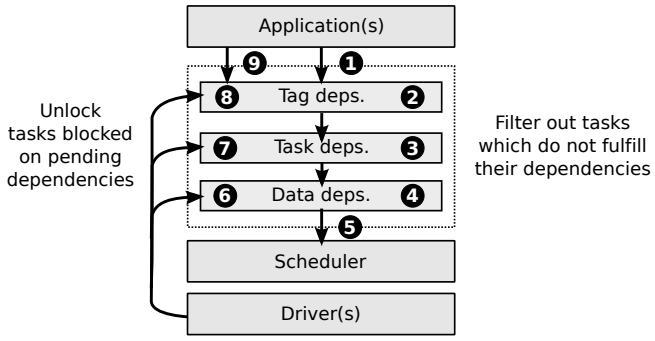


Figure 2.19: Detailed view of the different steps required to enforce dependencies.

Figure 2.19 gives a more detailed view of how dependencies are enforces by StarPU. Once the task has been submitted by the application (Step 1), all its dependencies are checked within the different stages that correspond to the different type of blocking resources. As soon as StarPU encounters a dependency that is not fulfilled, the task is blocked in the corresponding stage (Steps 2, 3 or 4). Blocked tasks are unlocked later on when the blocking resource is freed, either during the termination of another task (Steps 6, 7 or 8), or when the application explicitly release the resource (Step 9). When all its dependencies are fulfilled, a task becomes *ready* for the scheduler.

Tag dependencies

Tags are the first type of dependency that is enforced (Step 2). If no tag was specified in a task structure, this step is simply skipped when the task is submitted.

Tags are implemented by the means of a hierarchical table which entries store tags' current value, the list of tags that depend on this tag, and which task has been associated to this tag (if any). There are indeed five possible states for a tag. ***invalid***: nobody is using that tag yet. ***associated***: a task with that tag has been submitted by the application, and the address of the task structure is stored in the appropriate entry of the tag table. ***blocked***: the tag depends on one or multiple tags that are not in a *done* state. ***ready***: the task associated to this tag can be executed. ***done***: the task associated to that tag has been executed.

When a new task associated to a tag T is submitted, the state of T becomes *associated*. If a dependency was declared between T and one or multiple other tags which are still not all marked as *done*, the task is therefore blocked in Step 2 and the status of T is set to *blocked*. Otherwise, the T becomes *ready*, and the task is passed to the next stage of dependency checking (Step 3).

When the application explicitly unlocks a tag T (Step 9), or during the termination of a task associated to T (Step 8), the state of T is changed to *done*, and the tags that depend on T are notified. If all the tag dependencies of one of these tags are fulfilled, its state is set to *ready* and the associated tasks are unlocked if they are blocked at the tag dependency stage.

Task dependencies

The implementation of explicit dependencies between task structure is straightforward. The task structure contains a reference count that indicates the number of pending task dependencies, and a list of tasks that depend on the current task.

On Figure 2.19, Step 3 therefore consists in checking whether the reference count is null or not. In case there are still pending dependencies, the task is blocked in this stage, otherwise it passed to the stage that checks data dependencies. When a task terminates, the reference count fields of its dependencies are decreased. tasks for which this value becomes null are unlocked and submitted to the next stage.

Data dependencies

The goal of Step 4 is to allow multiple readers but only a single writer at any time. We first describe the RW-lock mechanism used to implement the single-writer/multiple-readers policy which protects data handles from incoherent accesses. Then, we show why the different data handles used by a task must be grabbed following a specific order to avoid deadlocks, and we define such a full-order over data handles.

Reordering data to prevent deadlocks A RW-lock is attached to each data handle to ensure that either multiple readers or a single writer are allowed to access the same piece of data. A naive way to ensure that a task is allowed to access all its data handles is to perform a loop that grabs the RW-lock of each data handle following the order specified by the application. However, dependencies are enforced in a decentralized way so that we must pay attention to potential concurrency issues. Let us for instance assume that the application submits two tasks: $T_1\,(A_{rw}, B_r)$ which locks A before B, and $T_2\,(B_{rw}, A_r)$ which locks B before A. If T_1 and T_2 respectively grab the RW-lock of

A and B at the same time, neither T_1 nor T_2 will be able to lock their second handle, so that we have a deadlock situation. This is a classical synchronization problem which is often illustrated by the dining philosophers problem [91]. A simple solution consists in assigning a *partial order* to the resources, which must be requested in order, and released in reverse order. We must therefore lock the different data handles following a specific order.

A piece of data is fully characterized by its handle in StarPU. In a first approximation, a possible order would be obtained by directly comparing the addresses of the different data handle structures in host memory (which have the same lifetime as registered data). However, StarPU provides a recursive data filtering functionality that makes it possible to subdivide a data handle in multiple data subsets, which are also described by their own data handles. It is not possible to access overlapping pieces of data in StarPU, so we need to take data hierarchy into account to design an order over data handles. When a filter is applied on a data handle, an array of handles is created to described each data subset.

Since filters can be applied recursively, a piece of data d is completely characterized by the data handle of its root r_d, and the path p_d which makes it possible to reach d starting from r_d. As shown on Figure 2.7 on page 70, this path contains the successive indexes of the different data subdivisions from r_d to d. StarPU therefore grabs the RW-lock of the different data handles of a task following the order defined on Equation 2.1, where the $(p_a < p_b)$ comparison follows the lexicographic order.

$$a < b \Leftrightarrow (r_a < r_b) \vee ((r_a = r_b) \wedge (p_a < p_b)) \tag{2.1}$$

2.5.2 Enforcing data-driven implicit dependencies

Since data dependencies are the third type of dependency to be enforced (see Figure 2.19), we cannot just rely on a rw-lock to implement implicit data dependencies. Let us for instance consider two tasks, T_1 and T_2 which both access data A, respectively in a read-write and in a read-only mode. If T_1 also depends on a tag that is not unlocked yet, and that T_1 is submitted before T_2, the first task is blocked on a tag dependency, but the second task can directly grab the rw-lock protecting data A. We would instead have expected that T_2 depends on T_1 because of their common dependency on data A and sequential consistency.

StarPU thus enforces implicit data dependencies directly during task submission. The implementation of this mechanism consists in transparently introducing extra task dependencies between the tasks which access the same piece of data with incompatible access modes. Similarly to the rw-lock which protects data handle from invalid concurrent accesses, the data handle structure describing a piece of data D contains a field that stores the access mode of the last submitted task using D (*last submitted mode*). StarPU also maintains a list of all submitted tasks that requested D in a read-only fashion (*last readers*), as well as a pointer to the last submitted task trying to modify D (*last writer*). When a new reader task is submitted, the task is appended to the list of readers, and a dependency between this task and the last writer is added by StarPU. When a new writer task is submitted, it becomes the new last writer and a dependency is automatically added either between this task and all the previous readers (which are remove from the list in the meantime) if any, or between the new task and the previous last writer otherwise.

Let us for instance reconsider the case mentioned at the beginning of this section: the address of T_1 would be recorded as the last writer of A during the submission of T_1, and a dependency between T_2 and the last writer (*i.e.* T_1) would be inserted by StarPU when the user submits T_2.

2.6 A generic execution model

In spite of the heterogeneous nature of accelerator-based platforms, StarPU relies on a unified task execution model. Each processing unit is controlled by a CPU thread, called *worker*, which role is to execute the tasks that were assigned to the processing unit.

Even though all accelerators are different, task parallelism is a generic enough paradigm that can be implemented efficiently on most architectures. Designing a driver for a new type of processing unit therefore mostly consists in implementing the various techniques usually required to explicitly offload a piece of computation, and to provide StarPU's data management library with an implementation of the memory transfer operations between the processing unit and the rest of the machine.

More precisely, the role of a driver is to grab tasks from the scheduler, to execute them, and to notify StarPU when they are completed. Drivers should rely on the data management library provided by StarPU to manipulate data (*i.e.* it prevents invalid concurrent accesses, implements data transfers, and keeps track of the location of valid data replicates). Besides, the driver controlling a processing unit must service external requests from other processing units which request a piece of data located on the local memory node. A driver may also help the scheduling engine and the application by giving performance feedback which permits to update auto-tuned performance models, or to update performance counters.

In the next sections, we illustrate how this generic model was applied to support fundamentally different types of architecture such as the Cell processors and GPUs.

2.6.1 Supporting CPU cores

```
1 while (machine_is_running())
2 {
3   handle_local_data_requests();
4   task = pop_task();
5   acquire_task_data(task);
6   task->cl->cpu_func(task->interface, task->cl_arg);
7   release_task_data(task);
8   handle_task_termination(task);
9 }
```

Figure 2.20: Driver for a CPU core

Implementing a driver that executes StarPU tasks on a CPU core is straightforward. Figure 2.20 shows a simplified version of the code running on CPU workers (error management was removed for the sake of clarity). Once the driver has been initialized, its main role consists in getting tasks from the scheduler (line 4) which appears as a black box to the driver. Prior to task execution, the driver ensures that every piece of data accessed by the task are locked in host memory by the means of a call to data management library (line 5). The `cpu_func` field of the codelet structure specified in the task structure (`task->cl`) is a function pointer that is directly called by the driver (line 6). The first argument passed to the CPU implementation of the codelet is an array of pointers to the data interface structures describing the different pieces of data. The second argument is the user-provided argument that was stored in the `cl_arg` field of the task structure. After the termination of the task, the driver informs the data management library that the different pieces

of data can be unlocked from host memory (line 7), and notifies StarPU that the task is terminated so that dependencies can be fulfilled (line 8). Besides executing tasks, the driver also services the different data requests coming from external sources (*e.g.* from the application or from another driver). On line 3, the driver therefore ensures that the CPU core spends some time to initiate pending data requests. Even though this is not shown on Figure 2.20 for the sake of conciseness, there are actually timing facilities around the function call (line 6), so that StarPU transparently updates auto-tuned performance models and performance feedback.

In Chapter 4, we will show that the flexibility of this very simple model permits to implement parallel CPU tasks which run simultaneously over multiple CPU workers at the same time.

2.6.2 Supporting GPU devices

Supporting GPU devices is also very simple with this execution model. Since both CUDA and OpenCL control devices directly from the host, we directly call the CUDA and the OpenCL implementation of the codelet on the host (`cl->cpu_func` would be replaced by `cl->cuda_func` or by `cl->opencl_func` on Figure 2.20). The CUDA and the OpenCL codelet implementations are indeed host code which typically offload kernels on the device by using the usual API provided by CUDA (*e.g.* with the triple-angle brackets syntax) or OpenCL (*e.g.* with calls to the `clSetKernelArg` and `clEnqueueNDRangeKernel` functions). In order to support the recent evolution of some GPU devices which support the concurrent execution of multiple kernels, the host-function which implements the codelet can directly submit multiple kernels simultaneously doing different types of computation. In the future, another solution to exploit this hardware feature would consist in popping multiple tasks from the scheduler and executing them simultaneously. Another possible approach would be to have multiple CPU threads share the same GPU device.

Similarly to the driver for CPU cores, drivers do not really take care of data transfers, and simply call the methods provided by the data management library to lock and unlock data in the memory embedded on the accelerator. Complex features such as asynchronous and direct GPU-GPU transfers are directly handled within the data management library, and not by the different drivers which only ensure that data transfers are progressing (as on line 3 of Figure 2.20).

Since GPUs are controlled from the host, CUDA and OpenCL workers are implemented by the means of a host thread that is dedicated to executing tasks on a specific accelerator. In order to guarantee a sufficient reactivity, a CPU core is therefore dedicated for each CUDA and OpenCL accelerator. Having a dedicated CPU core is often required to keep a GPU busy enough. This trade-off is also sensible because the number of CPU cores continuously keeps growing, and that this number is also usually significantly larger than the number of accelerators (typically less than 4). Highly efficient kernels are also often hybrid because of the specific nature of GPUs which cannot handle any type of computation efficiently: libraries such as MAGMA implement hybrid kernels that associate each GPU with a CPU core which deals with the non-GPU-friendly parts of computation [142]. The small performance loss caused by CPU cores dedicated to accelerators is therefore small, or even nonexistent in the case of hybrid kernels which also take advantage of the CPU core anyway. A possible improvement would also be to automatically transform CUDA and OpenCL drivers into CPU drivers whenever there is nothing but tasks that can only run on CPU cores.

2.6.3 Supporting the Cell processor

The Cell B/E processor is a heterogeneous multicore chip composed of a main hyper-threaded core, named PPU (Power Processing Unit), and 8 coprocessors called SPUs (Synergistic Processing Units). Each SPU only has 256 KB of memory, called local store, which is used for both data and code. This limited amount of memory only allows to execute very small tasks, which granularity is for instance significantly smaller than GPU tasks. Contrary to GPUs that are typically controlled from the host, SPUs are also full-fledged processors, so that the driver controlling SPUs here run directly on the SPUs, and not on the host-side processor (*i.e.* PPU) as it would be the case with GPUs.

The memory sub-system of the Cell processor is based on DMAs (direct memory accesses) which are naturally asynchronous, and fits our asynchronous distributed data management library very well, except that there is no shared memory on the Cell processor. Since the description of the different data replicates (*e.g.* MSI states) must be accessible from each processing unit, we must store the data handle structure in shared memory. Instead of relying on a piece of data located in shared memory, a possible solution is to use DMA and atomic DMA transfers to manipulate the data handle structure which is stored in main memory.

In order to illustrate the suitability of StarPU for the Cell processor, we have used the Cell Runtime Library (Cell-RTL) [NBBA09] to design a driver for the different SPUs [ATNN09]. The Cell-RTL is a runtime system which permits to efficiently offload tasks on a SPU. Its interface is similar to the Offload API proposed to extend the Charm++ environment on the Cell processor [117]. For the sake of simplicity, we also rely on the data management facilities available in the Cell-RTL to upload and download data between main memory and the local stores. In this case, the actual role of StarPU consists in assigning tasks to the different SPUs, so that they can be efficiently offloaded on the selected SPU by the Cell-RTL.

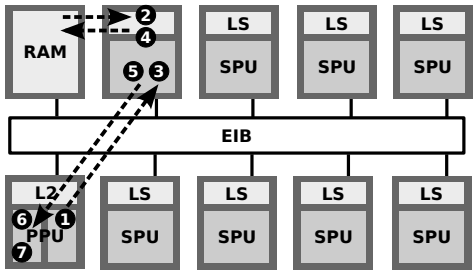
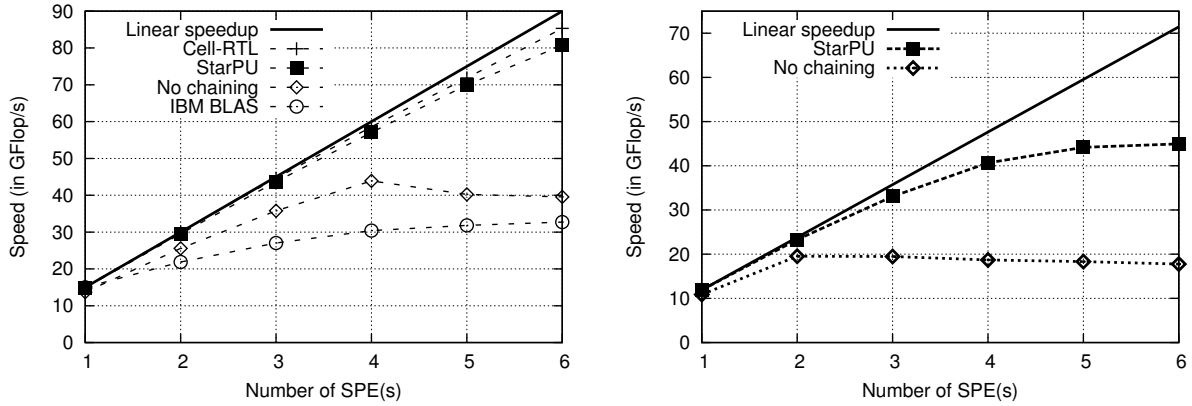


Figure 2.21: Offloading tasks with the Cell Runtime Library (Cell-RTL).

Figure 2.21 depicts how the Cell-RTL works: when a SPU task is submitted from the PPU, a message is sent from the PPU to the SPU (Step 1); an automaton running on each SPU reads this message, fetches data into the LS (Step 2), executes the corresponding task (Step 3), commits the output data back to the main memory (Step 4), and sends a signal to the PPU (Step 5); when the PPU detects this signal (Step 6), a termination callback is executed (Step 7). The Cell-RTL significantly reduces the resulting synchronization overhead by submitting chains of tasks to the SPUs [NBBA09]. To fully exploit the possibilities of the Cell-RTL, the StarPU Cell driver loop automatically builds such chains of tasks before submitting them. As most of this management is performed from the SPUs, which are almost full-fledged cores, only one thread is required to dispatch tasks between the different SPUs.

In a way, StarPU leverages the Cell-RTL by adding scheduling facilities and providing with the high-level data management and task dependencies enforcement, permitting efficient task chaining. StarPU could leverage other back-ends like IBM's ALF [49] or the runtime system used in CellSs [21]. The task chaining mechanism used in the Cell-RTL could also be useful with other types of accelerators when task granularity is too small to hide task management overhead. On NVIDIA Fermi devices this would for instance allow to execute multiple tasks concurrently on the same device [196].

Managing data in such a host-centric way is limited, but if the Cell had a better perspective, the data management library of StarPU could be extended to support environments with a distributed memory by ensuring that the data handle structure can be manipulated in a distributed fashion. This could for instance be achieved with atomic DMA transactions in the case of the Cell processor. Considering that only 25.6 GB/s of memory bandwidth are available between main memory and a Cell processor, transferring data directly between SPUs would significantly improve the overall memory bandwidth. Similarly to direct GPU-GPU transfers, data requests would permit to implement such direct DMA transfers between SPUs.



(a) Matrix multiplication.

(b) Cholesky decomposition.

Figure 2.22: Scalability of StarPU on the Cell processor.

The performance of a blocked matrix multiplication and a Cholesky decomposition are respectively shown on Figure 2.22(a) and 2.22(b). Since there is no comprehensive implementation of the BLAS kernels available for the SPUs, we use the SGEMM kernel from IBM SDK, and we use the SPOTRF and STRSM kernels written by KURZAK *et al.* [118].

On Figure 2.22(a), IBM's implementation is given as a reference. A manual implementation on top of Cell-RTL shows the best performance that could be obtained by StarPU, which uses Cell-RTL internally. Contrary to the manual implementation that requires to explicitly create chains of Cell-RTL tasks, StarPU automatically constructs such task chains. This introduces a small overhead when using StarPU instead of a manual implementation. When the chaining mechanism is disabled, the overhead caused by the small size of the tasks running on the SPUs becomes non negligible. Adapting StarPU's implementation of Cholesky decomposition for the Cell processor only required to provide the proper BLAS kernels. On Figure 2.22(b), StarPU not only has to deal with small tasks to provide enough parallelism, but it must also hide the resulting task management overhead by the means of task chaining. The limited amount of available memory however

impacts the amount of parallelism, and therefore limits scalability.

2.7 Discussion

In this chapter, we have studied the suitability of task parallelism for hybrid accelerator-based platforms. Tasks provide a flexible and generic representation of computation. It is especially convenient with regards to our portability concerns because it does not require to know the targeted platform in advance, and adapting an application to a new platform only requires to implement additional kernels. It also appears that data management is a crucial issue when dealing with accelerators; tasks are also appropriate there because they permit to explicitly specify which pieces of data are accessed, and how they are accessed. Tightly coupling data management with task management is crucial to ensure good performance on such machines which not only depend on the relative performance of the different processing units, but also on the activity on the I/O bus which traditionally constitutes a major bottleneck.

A significant choice in the design of StarPU was to ask programmers to actually describe the data layout instead of automatically detecting data accesses. This indeed gives StarPU much more opportunities for optimizing data accesses (*e.g.* data prefetching). Even though it requires an additional programming effort, it is worth noting that programmers usually need to figure out which pieces of data are going to be accessed in such environments with a distributed memory. One could also rely on a software-based Distributed Shared Memory (SDSM) which typically provides a better productivity but has a significant impact on the overhead of data management. Requiring programmers to explicitly register the various pieces of data when designing a task parallel algorithm is therefore a sensible trade-off, especially because we can implement a SDSM on top of StarPU.

We have also shown that the task paradigm is flexible enough to be enhanced with high-level abstractions such as data reductions. Reductions indeed allow programmers to design algorithms that scale beyond the limits encountered with programming environments which only provide the classical read/write data accesses. Considering the tremendous amount of parallelism required to exploit upcoming hybrid manycore platforms, such advanced algorithmic features will certainly become necessary.

Finally, supporting a new architecture only requires limited efforts in StarPU, which suggests that we meet our portability goals. Providing support for architectures which are as different as Cell SPUs and CUDA devices with the same execution model is also an indication that of the suitability our approach. Even though most efforts are currently being ported to accelerating applications with multiple accelerators coupled to a few CPU cores, the techniques designed to efficiently support nowadays accelerators will still be useful for future manycore architectures. StarPU therefore not only provides portable performance to higher-level software layers, it also constitutes a flexible environment to deal with new types of hardware.

Chapter 3

Scheduling Strategies

Chapter Abstract

This chapter introduces StarPU's scheduling engine and gives an overview of the different scheduling techniques implemented on top of StarPU. After showing that task scheduling is a critical concern on hybrid accelerator-based platforms, we describe the interface exposed by StarPU to easily implement portable scheduling strategies. We illustrate how scheduling hints can improve the quality of the scheduling, and we detail how user-provided or auto-tuned performance models help StarPU to actually take advantage of heterogeneity. We then consider the significant impact of data transfers and data locality on performance, and we give examples of scheduling strategies that take into account both load balancing, data locality and possibly other criteria such as energy consumption. Finally, we consider the problem of scheduler composition which is a possible approach to design flexible and scalable scheduling strategies suitable to compose parallel libraries on manycore platforms.

3.1 Scheduling tasks in heterogeneous accelerator-based environments

Nowadays architectures have gotten so complex that it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive. Even though such a static scheduling is sometimes possible, it requires significant efforts and a great knowledge of both the entire software stack and of the underlying hardware. While HPC applications tend to assume they are alone on a machine which is perfectly known in advance, machines may actually not be fully dedicated, and the amount of allocated resource may even evolve dynamically (*e.g.* when there are multiple parallel libraries running concurrently). Programmers might not even know which will be the target platform when designing third-party libraries or applications.

Writing code that is portable across all existing platforms is a delicate problem. Writing code that can be easily adapted to follow the future evolution of the architectures is even more complex. When a new type of hardware is available (or when the software undergoes a significant evolution), it is crucial that programmers do not need to rethink their entire application. Rewriting the computation kernels is often a necessity, but applications relying on libraries or on compilation environments can expect that these environments will be upgraded to support such a new platform. An application which statically maps computation, and thus data transfers, would however certainly have to go through a major redesign to support new architecture features such as asynchronous or direct data transfers between accelerators, since these totally change the way accelerators interact. Machines may be upgraded gradually, so that they could for instance eventually contain a mix of synchronous and asynchronous cards. Supporting such heterogeneous multi-accelerator platforms without too much effort is a serious concern when mapping computation by hand. Finally, adapting an application to a brand new type of architecture (*e.g.* from a Cell

processor to a multi-GPU setup) also requires a significant programming effort for programmers who manually deal with low-level issues such as data management.

To face the complexity of current platforms, and to ensure that applications will be ready to support tomorrow's architectures, it is therefore crucial that our system is seamlessly able to dynamically dispatch computation and take care of managing data efficiently with respect to the underlying hardware. Programmers should only have to design their application once (*e.g.* as a task graph): when a new platform is available, the only programming effort should consist in re-implementing the various computation kernels instead of entirely rethinking how the different processing units should interact.

3.1.1 Dealing with heterogeneous processing capabilities

In order to illustrate why attention must be paid to scheduling tasks properly on hybrid platform, let us now consider a problem that is usually regarded as very simple: matrix multiplication. This problem is parallelized by dividing the input and output matrices into multiple blocks with an identical size. In this case, the algorithm is directly implemented by submitting a set of identical tasks, each performing a small matrix multiplication.

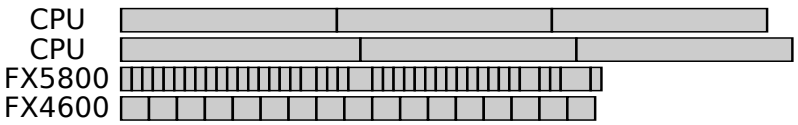


Figure 3.1: A pathological case: Gantt diagram of a blocked matrix multiplication with a greedy scheduling strategy.

Figure 3.1 shows the Gantt diagram observed when executing this algorithm naively on a hybrid platform composed of two GPUs and two CPU cores (two other cores are actually devoted to the two GPUs). The scheduling strategy used for this experiment consists in putting all tasks in a shared queue, and having all processing units to eagerly grab tasks from the queue. Even though both CPU and GPU kernels are fully optimized, the resulting performance is much lower than the optimal performance one could expect: the two last tasks were indeed executed on CPU cores, which are significantly slower than the GPUs.

The lesson learned from this simple experiment is that, even for one of the simplest type of problem, having fully-optimized kernels is not sufficient to fully exploit heterogeneous machines: one must also really pay attention to scheduling the different tasks properly. Another intuitive conclusion that we can draw from this experiment is that one should distribute tasks with respect to the relative speedups of the different processing units. Somebody statically mapping tasks on such a machine would indeed naturally put the two last tasks on GPUs so that computation ends earlier. In Section 3.4.3, we will show that following this intuition leads to very good load-balancing capabilities on hybrid platforms, through the use of extra performance modeling.

3.1.2 Impact of data transfers

For non-trivial parallelism, communication is usually needed between the different processing units, to *e.g.* exchange intermediate results. This not only means data transfers between CPUs and

GPUs, but also potentially between GPUs themselves, or even between CPUs, GPUs, and other machines on the network in the case of clusters.

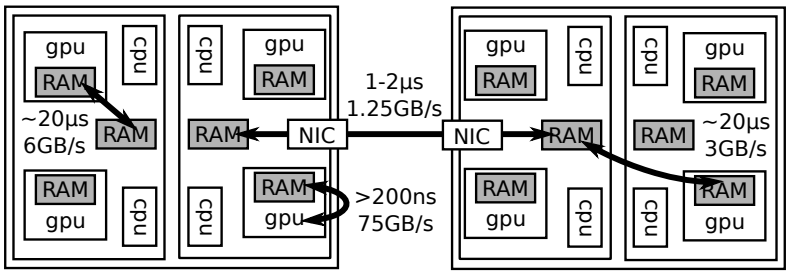


Figure 3.2: Typical performance of the different types of memory interconnects.

Figure 3.2 shows typical latencies and bandwidths that can be measured in a machine equipped with a few GPUs. The RAM embedded in GPUs usually provides a trade-off of a very high bandwidth but a non-negligible latency. The main I/O bus of the machine typically has a much lower bandwidth, but also a very high latency, due to quite huge overheads in software stacks like CUDA. NUMA factors also affect transfers, mostly their bandwidth, which can be seen cut by half, while the latency penalty is negligible compared to the software overhead. Eventually, cluster network interface cards (NICs) have a quite good latency, but their bandwidth is yet lower. As a result, with the increasing number of processing units and their increasing performance, data transfers become a critical performance concern since the memory bus can easily be a bottleneck.

StarPU already keeps track of where data have already been transferred to avoid spuriously consuming memory bandwidth by sending them again. In section 3.6, we will explain how, to further optimize memory transfers, we not only take benefit from asynchronous transfers supported by recent accelerators, but also save yet more memory transfers by extending scheduling policies so as to improve their task placement decisions according to data locality and transfer costs. We can also improve the efficiency of the unavoidable data transfers by automatically trying to overlap them with computations.

3.2 A generic scheduling engine

In this section, we first explain that there does not exist a perfect scheduling policy which fits any type of problem. We then describe the design of StarPU's generic scheduling engine and how it permits to create custom scheduling strategies which can be plugged into the scheduling engine at runtime.

3.2.1 No single perfect scheduling strategy exists

The tremendous amount of literature dealing with task scheduling [59] illustrates the fact that there does not exist a single perfect scheduling solution that would solve any type of problem. Instead of looking for such an ultimate scheduling algorithm, an other approach is to provide programmers with multiple scheduling strategies that can be selected according to the actual characteristics of the application. Designing such a flexible scheduling engine is already a concern on multicore platforms ; for example, the ForestGomp [31] OpenMP implementation relies on the

```
struct starpu_sched_policy_s {
  /* Initialize the scheduling policy. */
  void (*init_sched)(struct starpu_machine_topology_s *, struct starpu_sched_policy_s *);

  /* Cleanup the scheduling policy. */
  void (*deinit_sched)(struct starpu_machine_topology_s *, struct starpu_sched_policy_s *);

  /* Insert a task into the scheduler. */
  int (*push_task)(struct starpu_task *);

  /* Notify the scheduler that a task was directly pushed to the worker without going
   * through the scheduler. This method is called when a task is explicitly assigned to a
   * worker. This method therefore permits to keep the state of the scheduler coherent even
   * when StarPU bypasses the scheduling strategy. */
  void (*push_task_notify)(struct starpu_task *, int workerid);

  /* Get a task from the scheduler. The mutex associated to the worker is
   * already taken when this method is called. */
  struct starpu_task *(*pop_task)(void);

  /* This method is called every time a task has been executed. (optional) */
  void (*post_exec_hook)(struct starpu_task *);

  /* Name of the policy (optional) */
  const char *policy_name;
};
```

Figure 3.3: Data Structure describing a scheduling strategy in StarPU

BubbleSched [181] user-level thread scheduler which provides a flexible API to design portable scheduling strategies that can for instance take advantage of data locality, or optimize data bandwidth. On hybrid platforms, the low-level concerns encountered by the programmers are different (*e.g.* efficient data transfers), but selecting the best scheduling strategy still depends on the application.

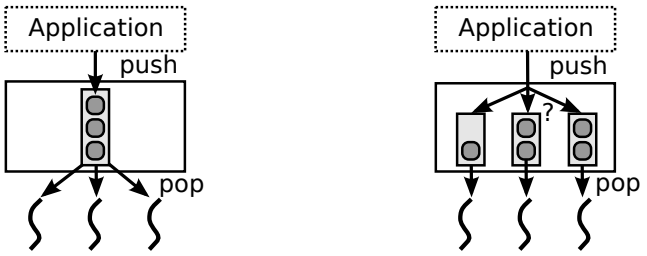
In this chapter, we present some strategies that rely on performance models. Some problems are however totally unpredictable and therefore need other types of load-balancing mechanisms. Work stealing is for instance well suited for tree-based algorithms [70], but it would be totally inefficient in some situations. This illustrates that a generic environment like StarPU must provide an API to design flexible scheduling strategies to cope with the various constraints met on the different types of algorithms.

3.2.2 A Flexible API to design portable Scheduling Strategy as plug-ins

Our approach consists in providing a scheduling API which permits to implement scheduling policies by the means of scheduling plug-ins that can be embedded with the applications. Applications can also select one of the predefined scheduling strategies. A predefined scheduling strategy can be selected by specifying the name of the strategy in an environment variable or when invoking `starpu_init()`. Custom strategies can be embedded in the application by passing a pointer to a C structure that describes a scheduling strategy (see Figure 3.3) when calling `starpu_init()`.

Designing dynamic scheduling strategies

The general idea behind StarPU's scheduling plug-ins is to consider the scheduling engine as a black-box. The C structure describing a scheduling strategy is shown on Figure 3.3. When a task becomes ready (*i.e.* all dependencies are fulfilled), the task is *pushed* into the scheduler by calling the `push_task` method (line 9). On the other end of the scheduler, each processing unit gets tasks from the scheduler by calling the `pop_task` method (line 19).



(a) A portable interface.

(b) An example of push-based strategy.

Figure 3.4: All scheduling strategies implement the same queue-based interface.

As illustrated on Figure 3.4, these are the two fundamental functions that are used to define a scheduling strategy in StarPU. The actual scheduling decision can be taken at any time between the *push* and the *pop* steps. On Figure 3.4(a), tasks are assigned at the last moment when idle processing units pop tasks from the scheduler. Figure 3.4(b) shows a totally different strategy which assigns tasks as soon as they become ready. This very simple API allows programmers to design extremely different types of scheduling strategies independently from the applications. Since all strategies implement the same interface, this provides a convenient experimental framework for experts in the field of scheduling theory who can implement state-of-the-art scheduling strategies that can be used transparently within actual applications.

The `init_sched` method (line 3) is called to initialize the scheduling policy when StarPU is launched. This method typically permits to create the intermediate queues that are used to store pushed tasks until they are popped by one of the processing units. The initialization method typically relies on the hwloc library [30] to detect the topology of the machine in order to build an appropriate set of task queues. Any type of data structure can be used for this purpose (*e.g.* FIFOs, stacks, double-ended queues, priority queues). One can also improve the scalability of the scheduling policy by using lock-free data structures [187]. Conversely, the `deinit_sched` method (line 6) is called to free all the resources allocated by the scheduling policy.

Since scheduling decisions are taken only when tasks become ready, such scheduling strategies are naturally dynamic, even though it is possible to integrate static knowledge (obtained at compile-time or user-provided) within the different methods implementing the strategy. Such late decision may also limit the visibility of the scheduler (*e.g.* to detect that the amount of parallelism is dropping), but it makes the design of scheduling strategies much easier because there is no obligation to parse the inherently complex structure of tasks and data dependencies. Anyway, nothing prevents a strategy from inspecting the internal state of StarPU to consider the future arrival of non-ready tasks when scheduling tasks that have already been pushed into the scheduler.

Decentralized strategies

```
void starpu_worker_set_sched_condition(int workerid,
                        pthread_cond_t *sched_cond, pthread_mutex_t *sched_mutex);
```

Figure 3.5: Method used to associate a worker with a condition variable used by StarPU to wake the worker when activity is detected on the worker.

For the sake of scalability, there is no concurrency limitation concerning these scheduling methods. *Push* and *pop* methods can be invoked concurrently on different threads for different tasks, thus allowing to implement decentralized scheduling strategies. In order to avoid wasting energy or simply to reduce the contention on the various data structures, StarPU permits to block a processing unit when there is no on-going activity. In order to keep StarPU and the scheduling strategies properly synchronized, the method shown on Figure 3.5 is used to associate each worker with a condition variable. When StarPU detects activity on an idle processing unit (*e.g.* when a task is assigned to this worker), the corresponding blocked worker is awoken by signaling the proper condition variable which notifies the worker that an event occurred. Concurrency problems are solved very differently by the various types of scheduling strategies. In the strategy depicted by Figure 3.4(a), a single task queue is used: in this case, all workers share the same mutex and the same condition variable. In the strategy shown on Figure 3.4(b), all workers have their own local queue which is protected independently with a local mutex and a local condition variable that is signaled when a task is assigned to the queue. It is up to the designer of the scheduling strategy to decide how to protect the task queues created during by the `init_sched` method. The function shown on Figure 3.5 must therefore be called once for each processing unit during the initialization of the scheduler.

Implicit pop method

Very often, the scheduling decision is taken exclusively during the push method. In such *push-based* strategies, each worker is typically associated to a local queue in which the pop method grabs its tasks. This is for instance illustrated on the strategy depicted on Figure 3.4(b).

StarPU provides a flag in the task structure to let the user specify explicitly which worker should execute the task, thus bypassing the scheduler. This is implemented by the means of local queues that StarPU creates during its initialization phases: when a task that is explicitly assigned to a worker becomes runnable, it is put directly in the local queue associated to the selected worker.

To avoid reinventing the wheel by always reimplementing a pop method that grabs tasks from a local queue, StarPU allows the `pop_task` method being undefined (*i.e.* set to `NULL`). In this case, the scheduling engine assumes that all tasks are put directly in the existing StarPU-provided local queues, so that the pop method implicitly consists in grabbing task from the local queue. Scheduling strategies then do not need to create task queues, they can directly inject tasks to a specific worker by using the `starpu_push_local_task` function.

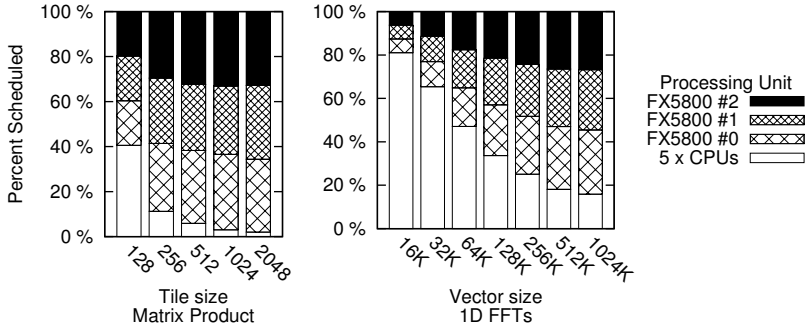
```
int starpu_push_local_task(int workerid, struct starpu_task *task, int prio);
```

Even though these local queues implement a FIFO ordering, the last parameter of the function indicates whether the task should be put at the head or at the tail of the queue, so that prioritized tasks are put directly at the end where the workers grab their tasks.

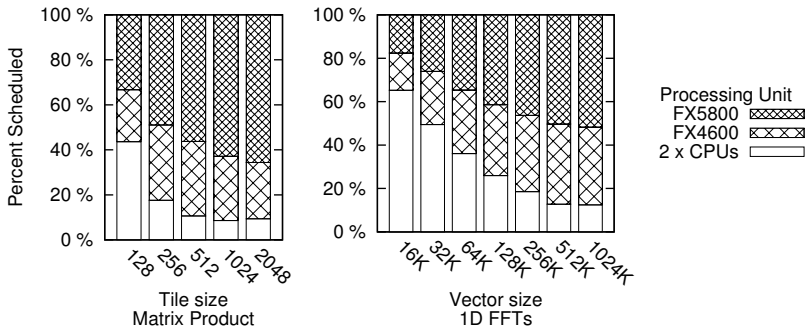
3.2.3 Use case: implementing the greedy strategy

In the previous section, we have already described the most simple strategy, called **greedy**, illustrated on Figure 3.4(a), which consists in a single task queue shared by all workers. While this example of centralized scheduling strategy naturally suffers from potential scalability issues, it illustrates the simplicity of our API to design scheduling strategies, and already provides interesting results.

The global task queue is respectively created and destroyed by the `init_sched` and the `deinit_sched` methods. `push_task` and `pop_task` just insert and remove tasks from the global queue, and are thus both straightforward in this strategy. A FIFO ordering is ensured, except for prioritized tasks that are directly put where the workers grab tasks first.



(a) Homogeneous multi-GPU system.



(b) Heterogeneous multi-GPU system.

Figure 3.6: Workload distribution in a hybrid environment.

In spite of its scalability limitations, this strategy dispatches tasks with respect to the actual speed of the different processing units. Indeed, various forms of heterogeneity appear in accelerator-based platforms. Different types of processing units may be available within the same machine (*e.g.* CPUs and GPUs): the codelet structure encapsulates the implementations for the different types of architectures. Another type of heterogeneity consists in having different models of the same category of processing unit. The machine used for Figure 3.6(b) for instance contains both an NVIDIA QUADRO FX4600, and an NVIDIA QUADRO FX5800 which is faster. Such heterogeneous platforms are typically found when machines are gradually upgraded. The load balancing capabilities of the **greedy** strategy permit StarPU to handle both types of heterogeneity

because the different processing units only grab tasks when they are ready: a slow CPU core will therefore retrieve less tasks from the scheduler than a fast GPU.

On the left-hand side of Figure 3.6, we have a blocked matrix-product, and on the right-hand side, we have a band-pass filter implemented using FFTW and CUFFT. In both benchmarks, the GPUs become relatively more efficient than the CPUs and thus get attributed more tasks when the granularity increases on Figures 3.6(a) and 3.6(b). Figure 3.6(b) also illustrates that StarPU is able to distribute tasks onto different models of GPUs with respect to their respective speed: a QUADRO FX5800 is given more tasks than a QUADRO FX4600 (which is much less powerful).

3.3 Scheduling hints: a precious help from the application

In order to provide performance portability, runtime systems can apply a wide range of optimizations to make sure that the application is executed as efficiently as possible. Aggressive optimizations are sometimes made possible by having a better understanding of the application's algorithm. While they rely on runtime systems to transparently implement efficient low-level optimizations in a portable fashion, programmers are not ignorant. Instead of wasting a lot of resource to try to guess approximately what is sometimes well-known for the programmers, the runtime system should take advantage of the knowledge of the algorithms they are running.

In this Section, we thus illustrate how programmers can guide the runtime system with hints that can be useful to the scheduler.

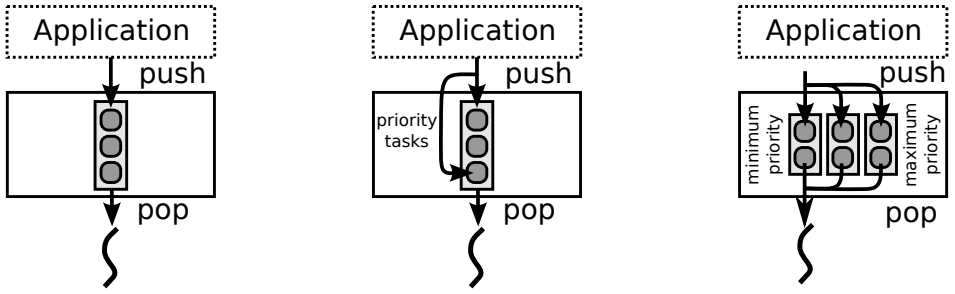
3.3.1 Task priorities

In order to exhibit as much parallelism as possible, a common method is to make sure that the critical path of an application is executed as fast as possible. *Look-ahead* techniques are for instance widely used in dense linear algebra, either on multicore machines [121] or on accelerator-based platforms [132]. This means that a scheduler should sometimes be able to select some tasks in priority, and to defer the execution of less critical tasks.

While in theory, it is possible to statically infer the critical path out of the task graph, it usually requires very costly computation. The suitability of the different heuristics to predict such priorities also heavily depends on the type of graph. Automatically detecting the critical path and which tasks should be prioritized is therefore a complex problem. On the other hand, programmers often have an idea of what the critical path is, and which tasks should be done in priority. We therefore take advantage of programmers' knowledge by the means of a priority level field in the task data structure proposed by StarPU.

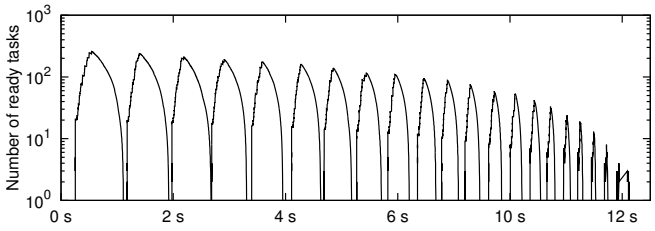
As shown on Figure 3.7, priorities are enforced accordingly to the capabilities of the selected scheduling strategy. Some strategies just ignore them (Figure 3.7(a)), others only allow a binary choice (Figure 3.7(b)), and others let programmers select a priority level within a whole range (Figure 3.7(c)). Applications can query which set of priorities is supported by calling the `starpu_sched_get_min_priority` and `starpu_sched_get_max_priority` functions. Since this range depends on the design of the strategy, it must be set during its initialization phase using the `starpu_sched_set_min_priority` and `starpu_sched_set_max_priority` functions.

LU, QR and Cholesky decomposition algorithms are for instance known to easily suffer from lack of parallelism if some critical tasks are not executed as soon as possible. Figure 3.8 shows

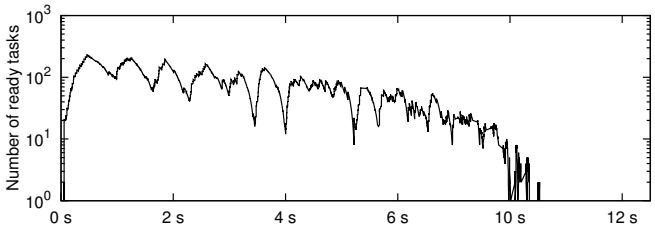


(a) No priority. (b) Binary priorities. (c) Range of priorities.

Figure 3.7: Examples of scheduling strategies offering different level of support for task priorities.



(a) Without task priorities (485 GFlop/s).



(b) With task priorities (550 GFlop/s).

Figure 3.8: Impact of priorities on Cholesky decomposition.

the evolution of the number of ready tasks during the execution of a Cholesky decomposition running on a multi-gpu platform equipped with 3 C2050 Fermi GPUs and 12 Nehalem CPU cores. In order to provide a meaningful comparison, we use the **heft-tmdp-pr** scheduling policy which is the best available policy for this benchmark (see Section 3.6.4 for more details). While all tasks have the same priority in the top curve, we put a maximum priority for the critical tasks in the bottom curve. Priority-aware scheduling here prevents substantial loss of parallelism, so that we observe a 15 % speed improvement when taking advantage of scheduling hints.

3.3.2 Performance Models

In the case of matrix multiplication, we have seen that somebody statically mapping tasks on a hybrid machine would intuitively rely on the relative speedups of the different processing units. A GPU would for instance certainly be assigned much more tasks than a CPU core. Assuming the scheduling policy can use such information, StarPU allows the application to give some hints

about the expected performance of the different tasks. Depending on programmers' knowledge and on the amount of work they are able to spend to guide the scheduler, there are different types of performance hints that the application can provide to StarPU.

Peak and sustained performance are among the most easily available parameters that can guide the scheduling policies. Such information can sometimes be derived from the operating system or from the libraries provided by constructors. They can also be provided once for all by the administrator of a machine who can for instance run a set of reference benchmarks to measure the sustained performance of the different processing units. Besides pure processing capabilities, there are other characteristics which could be helpful to scheduling strategies which support them. One could for instance guide an energy-aware scheduler by providing the peak and the base power consumption of the different devices.

A very useful type of scheduling hint is to predict the duration of the tasks. Programmers can provide StarPU with explicit functions that return the expected duration of the tasks depending on input data and on the selected worker. In order to construct such explicit models, this approach requires that programmers have a significant knowledge of both the application and the underlying hardware. This is for instance applicable for well-known kernels (*e.g.* BLAS) for which extensive performance analysis are sometimes available.

In many cases, an algorithmic analysis of the kernel permits to extract a parametric model such as $\mathcal{O}(n^3)$ for a matrix multiplication, or $\mathcal{O}(nln(n))$ for a sorting kernel. This still requires a significant amount of work to manually extract the performance of the different kernels in order to tune the different model parameters for each and every processing unit. The use of tracing tools in StarPU however facilitates this exercise by making it possible to directly measure tasks length during a real execution.

Higher-level layers that rely on StarPU can also use these performance feedback capabilities to build their own performance models. The SkePU framework is for instance able to decide automatically which code variant to select by the means of performance models [52]. In return, such environments can use their internal predictions to construct explicit performance models that are directly used by StarPU.

Before detailing how StarPU uses its performance feedback capabilities to automatically tune history-based and parametric performance models in section 3.5, we first explain how StarPU actually takes advantage of these performance models to better exploit heterogeneous platforms.

3.4 Scheduling strategies relying on performance models

In Section 3.1.1, we have seen that task scheduling must take into account the heterogeneous nature of the different processing units that compose an accelerator-based machine. We have also shown that StarPU allows programmers to give scheduling hints such as performance models, either at the processing-unit level or directly at the task level. We now give examples of scheduling strategies that actually take advantage of such hints.

3.4.1 Strategies based on the sustained speed of the processing units

A naive – yet common – way to avoid scheduling tasks on inefficient workers is to simply ignore slow processing units: this is obviously not suitable for a generic approach. Always assigning

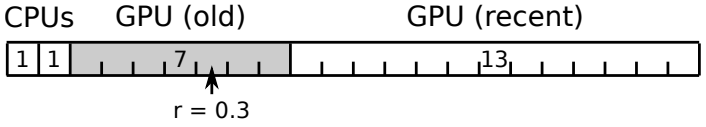


Figure 3.9: Practical example of the Weighted-Random Strategy. The numbers give the relative speedup of the different workers. The shaded area indicates which worker is selected for a random value $r = 0.3$.

tasks on the worker which has the best relative speedup is also not sufficient either because some relatively inefficient kernels may never get any tasks to execute (*e.g.* when all tasks are identical).

A general idea to deal with heterogeneous processing capabilities is to dispatch tasks proportionally to the relative speed of the different workers. A GPU that is ten times faster than a CPU core should for instance be given ten times more tasks than the latter. This principle was applied in the *weighted-random* strategy (also denoted as **w-rand**). The key idea behind this policy is that the probability of scheduling a task on a processing unit is equal to the ratio between the speed of this processing unit, and the speed of the entire machine with all processing units put together.

The first step to schedule a tasks with the **w-rand** strategy consists in computing a random number r that is comprised between 0 and 1. Assuming there are n workers, and that the relative speed of the i-th processing unit is denoted as s_i, the task is assigned to the worker with an index k which maximizes the following equation :

$$\max_k \left(\sum_{i=0}^{k} s_i \leq r \sum_{i=0}^{n-1} s_i \right) \tag{3.1}$$

Figure 3.9 gives a practical example of this algorithm. In a system that consists of two CPU cores and two heterogeneous GPUs, respectively going 7 and 13 times faster than a CPU core, if the computed random number is 0.3 the task will be assigned to the first GPU.

Compared to the **greedy** strategy, **w-rand** makes is less *likely* to be subject to pathological cases such found at the end of Figure 3.1. The probability to have the last tasks attributed to CPU cores instead of faster GPU devices would indeed be relatively low. This **w-rand** is also potentially much more scalable than the **greedy** one because all decisions are taken in a decentralized way, and workers only pick tasks from their local queue, so that there is no contention in the scheduler.

On the one hand, an obvious drawback of the **w-rand** policy is that it relies on a good statistical distribution of the tasks. According to the law of large numbers, this strategy is thus only guaranteed to provide a good load balancing if there is a sufficient amount of tasks. On the other hand, it must be noted that in the **greedy** strategy load imbalance only occurs when the amount of parallelism becomes too low to keep all processing units busy. In case the number of tasks is not large enough to ensure a good statistical distribution, the advantage of the **w-rand** strategy over the **greedy** one therefore seems quite limited. Enhancing this strategy with an efficient work-stealing mechanism however makes it possible to take advantage of its good scalability without causing too much load imbalance.

3.4.2 Predicting performance using per-task performance modesl

A certain number of signs indicate that we should consider per-task indications rather than processing unit-wide performance models. Scheduling tasks with respect to a constant per-worker

accelerating factor neglects the fact that some tasks behave much better than others on accelerators. Contrary to BLAS3 kernels (*e.g.* dense matrix multiplication) which almost reach the theoretical performance peak, some other kernels are relatively less efficient on accelerators (*e.g.* sparse matrix multiplication on GPUs). Considering a single accelerating factor per device is therefore quite irrelevant for applications that are composed of a variety of kernels which behave very differently. An important goal of the **w-rand** strategy is to make sure that every processing unit has a chance to execute tasks: not only very fast GPUs, but also slow CPU cores are eligible to schedule tasks with a probability that is uniquely defined by their speed. All tasks are however not equally important in terms of scheduling: a task that takes 80% of the overall computation time would better be executed on a fast GPU than on a slow CPU which would introduce a serious bottleneck. Tasks in the critical path should be scheduled on the fastest workers whenever possible to ensure that the amount of parallelism remains high. The optimal scheduling decision must therefore not only be taken with respect to the speed of the different processors, but also depending on the overall runtime status (*e.g.* load imbalance, amount of parallelism, etc.).

3.4.3 HEFT: Minimizing termination time

We now present an example of scheduling strategy which assumes that performance hints are available at task level: instead of predicting the relative speed of the different processing units, we can directly predict the duration of the tasks, as will be described in section 3.5.

The speed of accelerators often comes at the price of a significant kernel-launch overhead. A very common strategy thus consists in determining a size threshold and to assign tasks that are smaller than the threshold on CPU cores, while larger tasks are put on accelerators. While this prevents assigning very long tasks to slow processing units, this does not provide a solution to the problem of balancing load over multicore machines enhanced with multiple accelerators that are possibly heterogeneous.

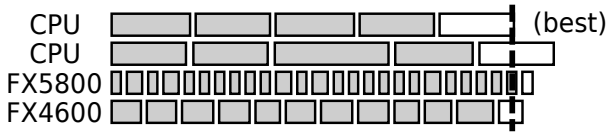


Figure 3.10: The Heterogeneous Earliest Finish Time Strategy.

The HEFT scheduling algorithm (Heterogeneous Earliest Finish Time [183]), implemented in the **heft-tm** StarPU policy, is illustrated by a real-case execution trace on Figure 3.10. The general idea of this strategy is to always assign tasks to the processing unit on which the task is expected to terminate earliest, according to the performance models supplied by the application. It is implemented by keeping track of the expected dates $Avail\,(P_i)$ at which each processing unit will become available (after all the tasks already assigned to it complete). A new task T is then assigned to the processing unit P_i that minimizes the new termination time with respect to the expected duration $\underset{P_i}{Est}\,(T)$ of the task on the unit *i.e.*

$$\min_{P_i} \left(Avail\,(P_i) + \underset{P_i}{Est}\,(T) \right) \tag{3.2}$$

Implementation of the heft-tm strategy

The **heft-tm** strategy is a push-based strategy, which means that the scheduling decision is taken by its *push* method. Implementing this method is actually straightforward with the API to design scheduling policies that was presented in Section 3.2.2.

```
1  /* Current predictions */
2  static double exp_start[STARPU_NMAXWORKERS];  /* of the first queued task */
3  static double exp_length[STARPU_NMAXWORKERS]; /* of the set of queued tasks */
4  static double exp_end[STARPU_NMAXWORKERS];    /* of the last queued task */
5
6  int heft_push_task(struct starpu_task *task)
7  {
8    int best_id = -1;
9    double best_exp_end, best_length;
10
11   for (int id = 0; id < nworkers; id++)
12   {
13     /* Which type of worker is this? */
14     enum starpu_perf_archtype arch = starpu_worker_get_perf_archtype(id);
15
16     /* Ask StarPU to predict the length of the task */
17     double length = starpu_task_expected_length(task, arch);
18     double exp_end = exp_end[i] + length;
19
20     if (best_id == -1 || exp_end < best_exp_end)
21     {
22       /* A better solution was found */
23       best_id = id;
24       best_exp_end = exp_end;
25       best_length = length;
26     }
27   }
28
29   /* Update predictions */
30   task->predicted = best_length;
31
32   pthread_mutex_lock(&sched_mutex[best_id]);
33   exp_length[best_id] += best_length;
34   exp_end[best_id] += best_length;
35   pthread_mutex_unlock(&sched_mutex[best_id]);
36
37   /* Prefetch input data */
38   unsigned memory_node = starpu_worker_get_memory_node(best_id);
39   starpu_prefetch_task_input_on_node(task, memory_node);
40
41   /* Put the task on the local queue of the selected worker */
42   int is_prio = (task->priority > 0);
43   return starpu_push_local_task(best_id, task, is_prio);
44 }
```

Figure 3.11: Simplified code of the push method used in the **heft-tm** strategy

A simplified version of the corresponding code is given by Figure 3.11. For the sake of clarity, we removed the code dealing with unavailable performance models and with tasks that can only be executed on a subset of the different workers (*e.g.* tasks without a CPU implementation). This method is an implementation of the minimization function given by equation 3.2. The expected availability dates of the different workers ($Avail\,(P_i)$) is stored in the `exp_end` array defined on line 4. These values are initialized to the current date by the initialization method of the **heft-tm**

strategy. For each worker, the expected length of the task ($\underset{P_i}{Est}\ (T)$) is also computed between lines 16 to 18.

As a result, we determine which worker is going to terminate the first, and we schedule the task on this worker on line 43. Lines 37 to 39 illustrate how StarPU prefetches task's data once a scheduling decision has been taken. More details on data prefetching will be given in Section 3.6.1.

Similarly to the strategy depicted on Figure 3.7(b) on page 104, **heft-tm** provides binary priorities: on lines 42 and 43, tasks with a high priority are indeed put directly at the head of the local queue from which the selected worker picks up tasks.

3.4.4 Dealing with inaccurate or missing performance models

There is sometimes no available performance model, either because the behaviour of the tasks is totally unpredictable (or unknown), or simply because the performance model is not calibrated yet. This for instance happens in the case of internal control tasks (*e.g.* tasks allocating pinned memory for CUDA) as their performance often depend on the state of the underlying operating system.

```
1 void heft_post_exec_hook(struct starpu_task *task)
2 {
3   int id = starpu_worker_get_id();
4   pthread_mutex_lock(&sched_mutex[id]);
5   exp_len[id] -= task->predicted;
6   exp_start[id] = starpu_timing_now() + task->predicted;
7   exp_end[id] = exp_start[id] + exp_len[id];
8   pthread_mutex_unlock(&sched_mutex[id]);
9 }
```

Figure 3.12: Hook called after the execution of a task to avoid drifts in the predictions used by the **heft-tm** strategy

Scheduling strategies based on performance models must therefore also deal with this situation. The simplest approach to handle tasks without a model is to assume that their length is null, and to correct erroneous predictions by the means of a callback function called after the execution of the task. Even though the code of **heft-tm** on Figure 3.11 is simplified and does not deal with unavailable performance models, a hook is therefore called after the execution of a task to update the values used by the *push* method to predict the best worker. This hook is automatically called by defining the `post_exec_hook` method of the strategy as the `heft_post_exec_hook` function shown on Figure 3.12. That way, the error on the $Avail\ (P_i)$ value used by equation 3.2 is kept under control, even if the predictions are not perfectly accurate.

In addition to this feedback loop, it is also possible to take a scheduling decision that does not depend on performance predictions. The actual implementation of the **heft-tm** policy (and its variations) for instance considers the speed of the different processing units, similarly to the **w-rand** strategy presented in Section 3.4.1.

3.5 Auto-tuned performance models

We have just illustrated that performance models are very precious hints for the scheduler. An extensive knowledge of both the algorithm and of the underlying hardware is generally required to predict the performance of a kernel on a specific piece of hardware. Explicitly providing manually tuned performance models is therefore a tedious task that is hardly compatible with our portability and productivity goals.

Regular applications are often characterized by the repetition of a limited number of task types. In this case, we show that we can simply predict performance by keeping track of the average performance previously measured on this small set of task types. Many performance models boil down to parametric models which have to be tuned for each and every processing unit. In this section, we explain the techniques used by StarPU to automatically perform this tuning.

3.5.1 History-based models

Sometimes the algorithm is regular enough that providing a complex performance model which predicts the performance of the different kernels for any input size is useless. Many regular algorithms are composed of tasks that almost always access pieces of data which have the same size and the same shape (or a very limited number of sizes/shapes). The tiled algorithms used in dense linear algebra for instance divide the input matrices into square blocks of equal size [39]. Even though it is still possible to dynamically adapt granularity, only a couple of sizes are used when the size of the input matrix is a multiple of the size of the blocks, which is often the case.

When the input size of the different tasks is always the same, it is not even possible to provide a meaningful regression-based model calibrated by the means of performance feedback because all measurements are concentrated on the same input size. Instead, we can take advantage of this regularity to rely on history-based performance models. The rationale behind history-based performance models is to assume that the performance of a task should be very similar to the performance previously measured on identical tasks (*i.e.* same data layout and same input size). A significant advantage is that such performance models do not require any knowledge of the application or of the hardware. The runtime system indeed only has to record a history containing the performance of the tasks previously executed, and to match incoming tasks with corresponding entries in this history.

However, history-based models assume that the performance of a task is independent from the actual content of the different pieces of data, and just depends on their layout. This limits the suitability of this approach for kernel which do not have a static flow of control. Contrary to matrix multiplication which is an example of especially regular kernel, the duration of the pivoting phase found in LU decomposition is usually unpredictable. Nevertheless, taking the average performance is often suitable, especially when the number of unpredictable tasks is low compared to the number of extremely regular tasks (*e.g.* matrix multiplication kernels in a LU decomposition). Even though it is not supposed to happen very often, another limitation of history-based models is also that no information is available outside the set of sizes for which a measurement was already performed. Combined with regression-based models built with the available entries of the history, we can use an interpolation to predict the performance of a task which does not match any entry in the history.

In a first approximation, the history could be indexed by the size of the tasks, but this does not capture all possible configurations. Applying a kernel on a (1024×1024) matrix often does not

take the same time as the kernel applied on a (256×4096) matrix. Instead, we take advantage of the flexibility of data interface API that we have described in Section 2.2.3. The data structure that describes a piece of data to StarPU indeed contains the different parameters that characterize the data handle. The matrix data interface for instance contains the number of lines, the number of rows and the size of the elements in the matrix. The CSR sparse matrix format contains other types of information such as the number of non-zero elements in the matrix. For each data interface, we can therefore chose a number of parameters that fully characterize the data layout.

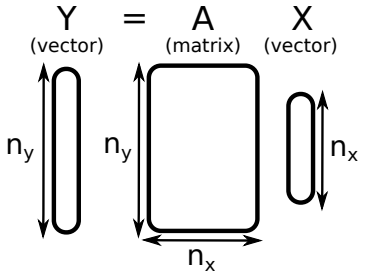


Figure 3.13: Parameters describing the data layout of the different piece of data that are used during a matrix-vector multiplication.

On Figure 3.13, in addition to the size of the different numbers, the two vectors are respectively characterized by the n_y and the n_x values, and the layout of the matrix is described by the (n_x, n_y) pair. The index used in a history based on total size would be $(n_x + n_y + n_x n_y)\, s_e$ where s_e denotes element size. Instead, the key used by StarPU is a presumably unique *hash* of the different parameters as shown on Equation 3.3. The CRC notation here stands for a *Cyclic Redundancy Check* which is a widely used type of hash function. More generally, the key identifying a task is computed by taking the hash of the different values obtained when hashing the set of parameters that describe each piece of data. We also denote this key as the *footprint* of a task. Per-handle hashes are actually computed by a method that is defined when implementing a new data interface.

$$key = CRC\left(CRC(n_y), CRC(n_x, n_y), CRC(n_x)\right) \tag{3.3}$$

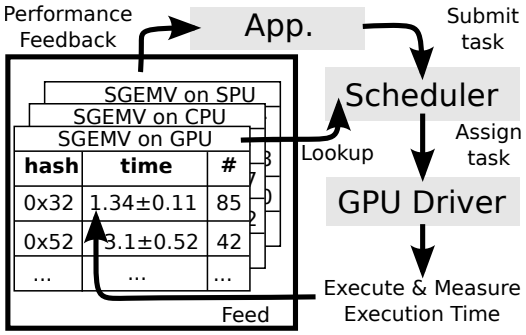


Figure 3.14: Performance feedback loop.

As illustrated by Figure 3.14, history-based performance models are therefore naturally implemented by the means of a hash table that is indexed by the footprint computed using this very simple – yet generic – algorithm. One hash table is created for each history-based performance

model (*i.e.* for each kernel), and for each type of architecture (*e.g.* CPU cores, first CUDA device, etc.). Each table is saved on the disk when StarPU is shut down, and it is loaded when a task using this performance model is scheduled for the first time. The proper table is accessed when a scheduling decision has to be taken, and it is updated when a new measurement is available after task termination. Collisions in the hash tables can be detected by comparing history entries with the sequence of parameters used to compute the key. CRC functions being really fast on modern processors, the overhead to compute the footprint of a task is also limited.

In the case of non-contiguous data types, the distance between elements (often called *striding*, or *leading dimension* in the case of BLAS kernels) is usually not part of the parameters that are used to compute the footprint of a data handle. This might lead to inappropriate performance predictions for an application that would mix task accessing data allocated continuously (*i.e.* with a very high locality) and data spread in multiple parts (*i.e.* with a poor locality, and possibly causing extra TLB misses). In order to avoid tampering with the history entries associated to a contiguous piece of data, one could for instance compute a different footprint depending on the contiguity of the data handle.

3.5.2 Regression-based models

Algorithms which are not regular enough to use history-based performance models because there are too many different input sizes should use models that can extrapolate the performance of a task by considering previous executions of the kernel applied on different input sizes.

Regressions are commonly used to tune parametric models given a set of actual performance measurements for a variety of input sizes. In addition to explicit and history-based performance models, StarPU therefore also provide performance models which are based on linear or non-linear regressions. The models currently implemented only have a single parameter, which corresponds to data size by default. Programmers can for instance guide StarPU by specifying that the relationship between data input size s and the duration of a kernel should be an affine law ($\alpha \times s + \beta$). It is also possible to specify an exponential law (αs^β) which can be extended to add an extra constant overhead into account to model kernel start latency on accelerators ($\alpha s^\beta + \gamma$).

When an affine model ($\alpha s + \beta$) is selected in a codelet, the scheduling engine simply predicts task duration by computing the sum of the sizes of the different input data and by applying the affine law given the current values of α and β on the different processing units. No prediction is returned if the number of samples is not sufficient to make an interpolation. Similarly to the performance feedback loop used to maintain history-based models on Figure 3.14, StarPU automatically dynamically updates the α and β terms every time a new measurement is available. In Appendix B, we detail the different techniques used by StarPU to automatically tune the different terms of these parametric models (*i.e.* α, β, etc.), either online by the means of the Least Square method for linear-regression, or offline for non-linear regressions (*e.g.* of the form $\alpha s^\beta + \gamma$).

More advanced programmers can also provide a custom method called by StarPU to compute the size injected into the parametric model. An affine law of the form $\alpha p + \beta$ can for instance be used to predict the performance of a matrix multiplication kernel by specifying that the input of the parametric model p should be equal to n^3 where n is the size of the square matrices.

In Appendix B, we detail the algorithms used by StarPU to calibrate linear and non-linear regression-based models by the means of the Least Square Method.

3.5.3 How to select the most appropriate model?

In the previous sections, we have shown that StarPU provides different types of auto-tuned performance models which address different types of problems. Depending on the regularity of the application, and on the regularity of the kernels themselves, it is indeed important to select an appropriate performance model. Using a history-based model is not applicable if each and every kernel has a different type of input. History-based models are thus typically intended for regular algorithms such as found in dense linear algebra for instance. In order to decide whether a history-based model was appropriate, StarPU provides a tool which displays the number of entries in the history, including the number of measurements per entry and the standard deviation of the measurements.

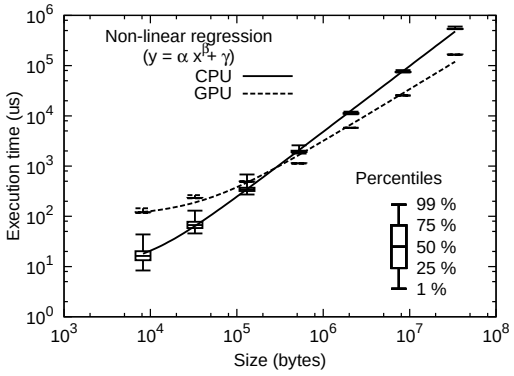


Figure 3.15: Performance and regularity of an STRSM BLAS3 kernel depending on granularity.

When dealing with irregular problems (*e.g.* sparse linear algebra, unstructured meshes, etc.), programmers should prefer performance models based on interpolations. As shown on Figure 3.15, StarPU provides a tool to display models based on linear and non-linear regressions. In this case, we see that the performance predicted by the means of a non-linear regression is fairly accurate for a STRSM kernel applied on square blocks, both on CPUs and GPUs.

There are still many improvements which could be made to provide better performance models. Based on the correlation coefficient of the regression obtained previous measurement, StarPU could automatically decide whether a linear or non-linear regression-based model is suitable or not. In case it is not, StarPU could automatically decide to select an history-based model. Such history-based models could also be improved by constructing piecewise models based on linear regressions between the different measurement available in the history.

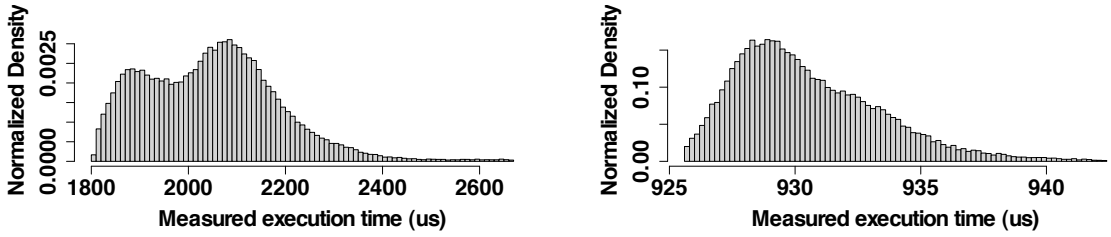
StarPU's auto-tuned performance models assume that the performance of a kernel should only depend on data layout (*e.g.* multiplying matrices of the same size should always take the same time). This assumption is sometimes violated: in case we have a non-static control-flow, the performance of the kernel also depends on the actual data content. Taking the average measurement is sometimes sufficient to estimate the performance of an irregular kernels, for instance during the pivoting phase of a LU decomposition which is not really predictable but only accounts for a limited portion of the total computation.

When computing a Mandelbrot fractal, the amount of computation per pixel is however totally unknown in advance. Depending on the output of the tools previously mentioned, programmers can decide which model is appropriate. If tasks are too unpredictable to find a meaningful model,

programmers should consider scheduling strategies that are not necessarily based on performance models. Policies based on work-stealing for instance provide a good load-balancing within a homogeneous system (*e.g.* with multiple GPUs but no CPUs) in spite of really unpredictable execution times if there is a sufficient amount of parallelism to avoid having too many work stealing events. They also require that the algorithm has enough data locality to avoid having data bouncing throughout the distributed memory.

3.5.4 Sharpness of the performance prediction

In the previous sections, we have presented very simple examples of performance models that are easily tunable. Providing an accurate model which perfectly predicts the performance of a kernel for any input size is however a very delicate problem.



(a) CPU implementation (GotoBLAS 2).

(b) GPU implementation (CUBLAS).

Figure 3.16: Distribution of the execution times of a STRSM BLAS3 kernel measured for tiles of size (512×512).

Instead of building complex models that analyze the behaviour of the kernel to determine its length, StarPU simply assumes that two instances of the same kernel with the same input size should have similar performance. Figure 3.16 for instance gives the distribution of the execution time measured during multiple invocations of the same BLAS3 kernel. It is worth noting that even if all tasks are identical, the performance is not exactly constant: making an exact prediction of the performance, solely based on the task and its input data is therefore deemed to failure. Even though we do not measure a perfectly constant time, the deviation of the measurements is somehow limited. Complex and often unpredictable interactions occur on a multicore processor (*e.g.* cache trashing, noise from the OS, internal copies within the CUDA driver, etc.). The distribution on Figure 3.16(a) typically illustrates that the performance of the kernel depends on data contention and data locality, which are hardly predictable when scheduling tasks ahead from their actual execution. While the GPU on Figure 3.16(b) is simple enough to have particularly regular performance, it is worth noting that GPUs tend to integrate complex mechanisms as well (*e.g.* concurrent kernel execution), which will tend to impact regularity. All in all, our empirical approach to transparently estimate the duration of future kernels simply consists in taking the average execution time measured during the previous invocations of the same kernel.

While it is theoretically possible to take extra parameters into account in order to characterize the overall state of the platform to provide a better estimation (*e.g.* with respect to the contention reported by hardware performance counters), given the complexity of nowadays architectures, and to avoid facing even more challenging problems in the future, we have concentrated on making sure that our scheduling algorithms are robust to performance prediction inaccuracies instead

(see Section 3.4.4). Such a pragmatic approach also appears in the theoretical scheduling literature: CANON and JEANNOT for instance study the robustness of DAG scheduling in such heterogeneous environments with performance prediction inaccuracies [37]. What really matters is indeed the scheduling decision and not the performance prediction which was used to take this decision.

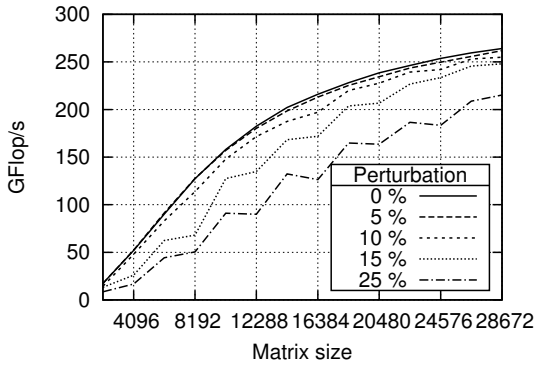


Figure 3.17: Impact of performance model inaccuracies on the performance of an LU decomposition.

Figure 3.17 shows the impact of performance prediction inaccuracies on the overall performance of an LU decomposition running on a machine equipped with an NVIDIA QUADRO FX4600 GPU and a quad-core Intel processor (BARRACUDA). We have used the **heft-tm** scheduling policy which is solely based on the estimations of the duration of the different tasks. We applied a random perturbation of the performance prediction to show that our scheduling is robust to model inaccuracies. When a task is scheduled, we do not consider its actual predicted length P, but $e^{\alpha ln(P)}$ with α being a random variable selected uniformly from $[1-\eta; 1+\eta]$ where η is the perturbation. A 5% perturbation here results in very low performance degradation, and really important miss-predictions are required before **heft-tm** is outperformed by strategies that do not use performance models. We obtain such a robust scheduling because the **heft-tm** policy implements a feedback loop which avoids drift in the overall estimation of the amount of work already assigned to each worker. As shown on Figure 3.12 in Section 3.4.4, this loop updates the amount of work associated to a processing unit according to the actual duration of the task that was just executed.

3.6 Integrating data management and task scheduling

The time to transfer data in an accelerator-based platform is actually far from negligible compared to task execution times. Considering that CUDA and other software stacks also introduce a non-negligible latency, taking data transfers into account while scheduling tasks is a crucial problem.

In this Section, we present various techniques to hide this overhead by overlapping data transfers and computation thanks to data prefetching mechanisms. Similarly to Section 3.4, we show that predicting the time required to transfer data throughout the machine makes it possible to design scheduling policies that provide better data locality, and to take NUMA effects into account.

3.6.1 Data prefetching

A classical approach to hide data transfer overhead is to overlap computation with communications whenever possible. The key to achieve that is to submit data transfer requests ahead enough to complete before the start of tasks which need them. This can be particularly important when using multiple accelerators which do not support direct GPU-GPU transfer, in which case the data has to be written back to memory first, hence at least doubling the transfer time.

In the **heft-tm** strategy (presented in Section 3.4.3), scheduling decisions are taken ahead of the actual execution of the tasks. As soon as they become ready to be executed, tasks are indeed dispatched between the various different processing units. In StarPU, a task is defined as *ready* (and thus handed off to the scheduler) when all its dependencies are fulfilled, and that all data handles can be used directly: a task accessing a data handle in a read-mode is guaranteed that nobody is going to modify the data until it terminates. The **heft-tm-pr** strategy therefore extends **heft-tm** by requesting StarPU to *prefetch* the input data of a task on the memory node that is attached to the processing unit selected by the HEFT algorithm. **heft-tm-pr** thus takes advantage of hardware's asynchronous hardware capabilities.

A potential drawback of this strategy is to increase the memory footprint: since data transfers have to be performed ahead of time, extra memory is required to store the input data of tasks that are not being executed yet. Data prefetching is thus not guaranteed to succeed, so that the execution of already scheduled tasks cannot fail due to a lack of memory caused by excessive data prefetching. In case such a failure occurs, StarPU will anyway ensure that tasks' data are available to the processing unit prior to its execution. In the future, we could improve the **heft-tm-pr** strategy by keeping track of the memory that was already allocated, and of the total amount of data currently being prefetched so as to avoid prefetching more data than what can fit in memory.

Another optimization would be to put some recently unused replicates of data back into host memory in order to make room for incoming data, which has to be allocated on the device later on anyway. A possible implementation of this strategy would be to enforce the *Least Recently Used* (LRU) or the *Least Frequently Used* (LFU) policies to select the most appropriate candidates for cache eviction. More generally, we could exploit scheduling holes to asynchronously put cached data back in host memory, so that it can be accessed sooner by other devices, or just save some room to avoid memory allocation failures.

Data prefetching is a common technique to hide a significant part of data transfer overhead. It is however not a mean to reduce the overall amount of data transfers. In the following sections, we detail complementary techniques which goal is to improve data locality, and thereby to reduce the activity on the I/O bus.

3.6.2 Predicting data transfer time

Getting an estimation of the time required to perform a data transfer is also important in order to decide whether is it better to move data or to migrate computation to another processing unit. Since StarPU keeps track of the different data replicates, it knows whether accessing some data from some processing unit requires a transfer or not.

When StarPU is initialized for the first time on a machine, a *sampling* procedure is used to evaluate the performance of the bus, so that we can estimate the time required to perform a data transfer when taking scheduling decisions. This sampling phase consists in a set of *ping-pong* benchmarks which permit to measure both the bandwidth and the latency between each pair of

processing units (*e.g.* from a GPU to a CPU core).

The numbers collected during these offline benchmarks are stored in a file that is loaded everytime StarPU is initialized. A rough estimation of data transfer times can then be derived from the bandwidth ($B_{j \to i}$) and the latency ($\lambda_{j \to i}$) measured between workers i and j:

$$\mathcal{T}_{j \to i} = B_{j \to i} \times size + \lambda_{j \to i} \tag{3.4}$$

Modeling data transfers accurately is a difficult problem because a lot of interactions occur within the I/O buses and the entire memory subsystem. Contention is an example of global factor that is especially difficult to model in a decentralized model where scheduling decisions are taken dynamically at the processing-unit level. The I/O subsystem is however a potential bottleneck that we cannot afford to ignore, especially in the context of multi-accelerator platforms where data transfers must be initiated carefully to avoid saturating the bus. For the sake of simplicity, we therefore assume that each accelerator consumes a similar portion of I/O resources: the bandwidth used in equation 3.4 is thus divided by the number of accelerators in the system:

$$\mathcal{T}_{j \to i} = \frac{B_{j \to i}}{n_{accel.}} \times size + \lambda_{j \to i} \tag{3.5}$$

More accurate transfer time estimations are possible. Instead of directly using the asymptotic bandwidth (*i.e.* measured for huge messages during the sampling phase), one could for instance consider the actual impact of message size on the predicted bandwidth. Just like with the prediction of tasks' execution time, correct scheduling decision is however what really matters, compared to prediction accuracy. Detecting that a task would take twice as much time because of an extra data transfer is more important that evaluating the exact time to perform such a data transfer which is likely to be overlapped anyway. StarPU therefore relies on the model presented on equation 3.5, which gives satisfactory prediction in spite of its simplicity.

Scheduling policies can thus now estimate the overhead introduced by data movements when assigning a task to some processing unit. Combined with execution time predictions, we can for instance decide whether it takes more time to execute a task on a slow CPU core, or on a faster processing unit that would however require data transfers. This also permits to select the least expensive transfer when multiple replicates are available over a non-uniform machine. Such sampling techniques are also used in other environments which optimize the efficiency of data transfers. The NEWMADELEINE multicore-enabled communication engine for instance divides messages into multiple chunks that are put on different high-speed network cards: the size of different chunks are proportional to the latencies and bandwidths of the different cards [32].

3.6.3 Non-Uniform Memory and I/O Access on hierarchical machines

Data transfers actually occur between the memory of the different devices and/or host memory, which is often divided into multiple NUMA nodes. When measuring the performance of the bus between a CPU core and an accelerator, we therefore have to consider the bandwidth between the accelerator and each NUMA node. We accelerate the sampling process by only evaluating the performance of the bus between a NUMA node and an accelerator only once, instead of doing this measurement once for each of the CPU cores attached to the same NUMA node.

Sampling the performance of the I/O bus also permits to detect the position of accelerators within a hierarchical machine. Similarly to the well-known *Non-Uniform Memory Access* (NUMA)

	CPU cores #0 to #5	CPU cores #6 to #11
GPU #0	H → D: 5.6 GB/s D → H: 5.1 GB/s	H → D: 6.4 GB/s D → H: 7.3 GB/s
GPU #1	H → D: 5.6 GB/s D → H: 5.1 GB/s	H → D: 6.4 GB/s D → H: 7.3 GB/s
GPU #2	H → D: 6.4 GB/s D → H: 7.4 GB/s	H → D: 5.6 GB/s D → H: 5.1 GB/s

Figure 3.18: Example of output of the sampling procedure on a hierarchical machine with NUIOA effects. The machine composed of 3 GPUs and 12 CPU cores separated between 2 NUMA nodes. H → D (resp. D → H) indicates the bandwidth measured from Host to Device (resp. from Device to Host). The measurements indicate that GPUs #0 and #1 are close to CPU cores #6 to #11, while GPU #2 is close to CPU cores #0 to #5.

effects that occur when CPU cores access remote NUMA memory banks, the performance of data transfers are limited when an accelerator performs a transfer between its local memory and a remote NUMA node. The sampling procedure therefore makes it possible to directly measure which NUMA nodes are the closest to the different devices. The CPU cores that are devoted to the management of the different accelerators are thus selected among the CPUs that are directly attached to these NUMA nodes. Table 3.18 gives an example of sampling output for a hierarchical machine equipped with 3 GPUs. This effect is sometimes called *Non-Uniform I/O Access* (NUIOA). It has also been observed on other types of devices that generate activity on the I/O bus such as network cards or with Intel's IO A/T technology. In the future, such effects are likely to be generalized on manycore and accelerator-based platforms [173, 139].

3.6.4 Using data transfer time prediction to improve data locality

While the **heft-tm** policy provides good load balancing, it does not take data transfers into account. Even though data prefetching makes it possible to hide a significant part of the data transfer overhead, the **heft-tm-pr** policy does not consider avoiding to transfer data.

Considering that the main I/O bus of the machine typically has a much lower bandwidth than the aggregated bandwidth of the different processing units, this becomes a real problem when it comes to multi-accelerators platforms, or when the number of CPU cores becomes large. Ignoring data locality when taking scheduling decisions therefore results in serious contention issues, and puts scalability at stake.

We therefore extended the **heft-tm** policy into the **heft-tmdp** policy which takes data locality into account thanks to the tight collaboration between the scheduler and the data management library. Contrary to the **heft-tm** strategy which solely considers execution time, **heft-tmdp** makes a trade-off between the load-balancing requirements, and enforcing data locality to preserve the I/O bus.

In addition to the computation time, the scheduler computes a penalty based on the times $\mathcal{T}_{j \to i}(d)$ required to move each data d from P_j (where a valid copy of d resides) to P_i. Such penalty of course reduces to 0 if the target unit already holds the data, *i.e.* $j = i$. We can therefore attribute

a task to the worker that minimizes the following metric :

$$\min_{P_i} \Big(\underbrace{Avail\left(P_i\right) + \underset{P_i}{Est}\left(T\right)}_{\text{termination time}} + \underbrace{\sum_{data} \min_{P_j} \Big(\mathcal{T}_{j \to i}\left(data\right) \Big)}_{\text{transfer time}} \Big) \tag{3.6}$$

Similarly to the execution time already computed by the **heft-tm** policy, we are able to explicitly evaluate this data transfer time. StarPU indeed keeps track of the different valid replicates of a piece of data, so that it detects whether a transfer would be required or not. In case a transfer is actually needed, we directly predict its duration by using the model described in the previous section. If the piece of data is already being transferred to the memory node, we consider that the penalty is null. This avoids penalizing the same data transfer multiple time, which makes data prefetching even more useful.

In Section 3.4.4, we have shown that the **heft-tm** strategy is robust to reasonable execution time prediction inaccuracies. The **heft-tmdp** strategy is also robust to data transfer time prediction inaccuracies. While these estimations could be improved in many ways (*e.g.* by monitoring the activity on the I/O bus), they already allow StarPU to take scheduling decisions that ensure a good data locality, and therefore minimize the impact of data transfers on performance.

As will be shown in the case of a state-of-the-art QR decomposition in Chapter 7, this results in massive reduction of the amount of data transfers. When combined with data prefetching in the **heft-tmdp-pr** strategy, the impact on the overall performance is particularly significant, especially if the number of processing units grows (along with the pressure on the I/O bus).

3.7 Taking other criteria into account

The **heft-tmdp-pr** strategy performs a trade-off between load-balancing and data locality. The optimal scheduling decision is obtained by computing the sum of the *penalty* associated to load imbalance and to the *penalty* associated to superfluous data transfers. It is possible to extend this strategy by penalizing other criteria, such as superfluous energy consumption, too large memory footprints, or contention on the memory bus. Depending on the context, one can even reinforce the impact of a specific parameter. For example, energy efficiency is critical in embedded systems which may not even really suffer from a limited load imbalance.

3.7.1 Reducing Power consumption

Power consumption has become a major concern nowadays. Similarly to the bandwidth consumption or to the execution time of the different tasks, it is possible to maintain models of the power consumption too. We have therefore extended the **heft-tmdp-pr** strategy to consider user-provided energy consumption models.

However, such energy consumption models could also be obtained automatically from hardware sensors, provided they are available at a reasonable granularity. Unfortunately, such sensors often provide only machine-wide power consumption estimations, which makes it difficult to automatically detect the actual energy consumption of a specific task. Power consumption can still be measured by the means of manual offline energy benchmarks of the different kernels, by relying on accurate hardware simulators (see Section 8.5.1), or even by using tools which perform a static code analysis to predict energy consumption [44, 93].

In addition to the energy required to execute a task on a specific processing unit, one must consider the overall energy consumed while processing units are idle. Indeed, scheduling a task on an extremely power-efficient processing unit does not guarantee that the total energy consumption would be smaller than when executing the task on a faster processing unit that would keep other processing units idle for less time. The energy consumption of the idle machine must currently be provided as an environment variable. It can for instance be measured once for all by the administrator of the machine using a machine-wide power-meter.

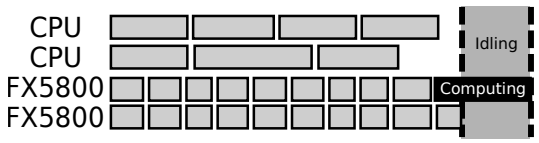


Figure 3.19: Extending the HEFT strategy to minimize energy consumption.

As illustrated on Figure 3.19, the energy penalty is computed by summing the energy consumption required to execute a task on the selected processing unit, and the energy required to keep all workers idle between the previous expected end of the algorithm, and the new expected end. This not only prevents scheduling a task on a worker that is extremely power inefficient, but it also avoids scheduling task on a slow worker that would cause all processing unit to consume energy while being idle.

A possible improvement of this strategy would consist in measuring precisely the extra energy consumption resulting from data transfers which are power consuming in embedded platforms. In this case, optimizing data locality is also a way to improve energy consumption.

3.7.2 Optimizing memory footprint and data bandwidth

The **heft-tmdp-pr** simply avoids data transfers, but it does not minimize the actual memory footprint. If we have a problem that is larger than the size of the memory on the devices, the memory reclaiming mechanism presented in Section 2.3.4 is likely to be used in order to put back unused data back into host memory, so that new pieces of data can be allocated on the device. In order to provide a better support for such out-of-core problems, one could penalize superfluous data allocations to avoid filling the memory allocation cache. Even though it may result in a less optimal scheduling at a certain time, it avoids having to undergo expensive memory reclaiming mechanisms later on. Besides strongly penalizing new data allocations on a device which is almost full, the scheduling strategy could also reduce the overhead of memory reclaiming by asynchronously putting back not recently used data into host memory when the ratio of allocated memory exceeds a certain ratio.

Some applications feature memory bandwidth-bound kernels which should not be scheduled concurrently. In case multiple memory intensive kernels are scheduled together on the same NUMA node, the overall performance can be much lower than when serializing these kernels. The scheduling policy should make its best to co-schedule memory-bound kernels with computation intensive kernels which do not consume much memory bandwidth and which are not suffering too much from a limited memory bandwidth. This could be achieved by associating a memory bandwidth consumption model to each task (*e.g.* maximum or average memory bandwidth requirement). The scheduling strategy should ensure that the total bandwidth consumption per

memory bank is smaller than the actual capacity of each memory bank. This could possibly involve characterizing tasks into multiple categories, for instance indicating whether the kernel is memory bound or computation bound. Once a the capacity of a memory bank is almost reached, the scheduler would penalize memory bound kernels and to favor computation bound kernels instead. Such models or categorization could either be provided by the programmers or obtained by the means of hardware performance counters (*e.g.* by counting memory accesses with PAPI).

These techniques could also be extended to ensure that concurrent kernels do not consume too much shared resources, such as the different levels of cache or I/O.

3.8 Confining applications within restricted scheduling domains

StarPU may be used by different parallel libraries that can be executed concurrently. The different libraries can either share the entire machine, or be executed independently on different machine subsets. In order to efficiently mix concurrent parallel libraries with very different requirements, we must therefore be able to confine tasks within specific *scheduling domains*. These scheduling domains are similar to the MPI communicators.

Since there does not exist a single perfect scheduling strategy that would be suitable for any library, the various scheduling domain are possibly managed using distinct scheduling policies to fulfill the specific requirements of the different libraries. The various mechanisms used to create separate scheduling domains have been implemented during the master thesis of ANDRA HUGO.

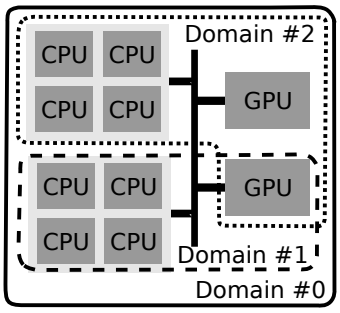


Figure 3.20: Example of overlapping scheduling domains.

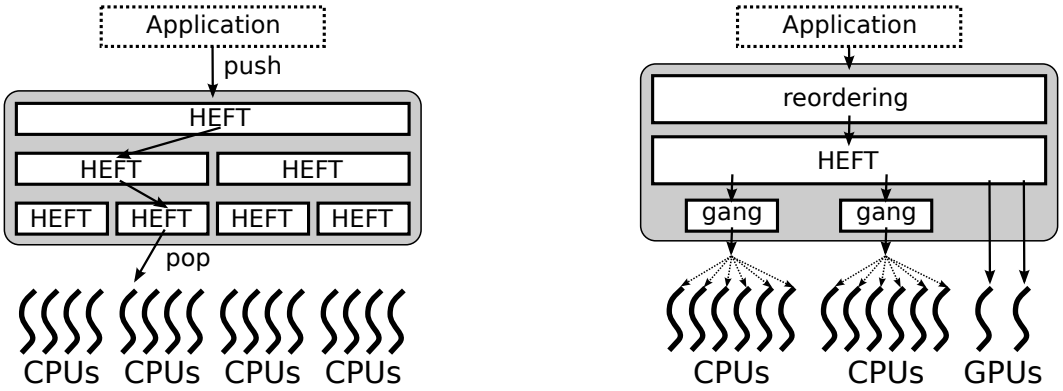
As shown on Figure 3.20, the same processing units can be part of different scheduling domains. Such overlapping scheduling domains are for instance useful when a library is not fully using the entire machine subset at all time. Instead of having a GPU underutilized by the library restricted to Domain #1 on Figure 3.20, it is shared between Domains #1 and #2 to avoid wasting processing resources. As a result, the StarPU driver controlling this GPU device can receive work from both scheduling domains, which are managed independently by different scheduling policies. Scheduling policies must therefore also deal with external scheduling decisions, either taken by the application when mapping a task by hand (*i.e.* bypassing the scheduling policy), or when a task has been scheduled by another scheduling policy when there are such overlapping scheduling domains. Runtime systems cannot solve this issue without coordination between the different instances of the runtime systems.To support this constraint, we have thus added a new method to the data structure describing a scheduling strategy (see line 15 on Figure 3.3). This method notifies the scheduling policy that a task was assigned to a worker. In the case of the **greedy** policy,

this method is let undefined because the worker will simply not try to pick tasks while there is already a task assigned to it. For strategies based on performance models, this handler typically updates the expected amount of work already assigned to the worker, so that the strategy properly keeps track of the availability of the different processing units in spite of concurrent scheduling decisions.

Libraries are not necessarily all initialized at the same time, and they may not be used during the entire application. StarPU therefore allows programmers to dynamically create and destroy scheduling domains, similarly to MPI2 which extends MPI1 with dynamic communicators [72]. Besides supporting dynamic invocations of parallel libraries, this also permits a better management of processing resources. In case there are two scheduling domains each sharing half of a machine, it is reasonable to re-assign the entire machine to a domain when the other domain is not used anymore. In the future, we expect that StarPU should be able to automatically dispatch processing resources between the different contexts when a new context is created (or destroyed), or when some resources are underutilized within a domain.

The S_GPU library allows applications to share GPU devices between various processes by the means of per-device mutexes [75]. While this is suitable for greedy scheduling strategies, it is harder to implement strategies based on performance models without knowing how much time the process should wait for these mutexes. The notification mechanism used to share resources between multiple domains could also be implemented on top of a message passing paradigm. This would make it possible to efficiently share processing units between multiple MPI processes running concurrently on the same machine, without loosing StarPU's flexible scheduling capabilities.

3.9 Toward composable scheduling policies



(a) Combining HEFT scheduling policies in a hierarchical fashion to avoid scalability issues within HEFT.

(b) Enhancing the HEFT scheduling policy to support task reordering and generate parallel tasks on CPU cores.

Figure 3.21: Composing scheduling policies.

Design perfectly scalable scheduling strategies is a delicate issue. Parallel machines are becoming more and more hierarchical, so that an efficient scheduling is *often* obtained by scheduling tightly coupled tasks together within a machine. The BUBBLESCHED [181] library provides an in-

terface to design scheduling policies for the MARCEL [141] user-level thread library. It allows to design hierarchical scheduling policies that map trees of threads on top of trees of hierarchical processors. As shown on Figure 3.21(a), it similarly makes sense to decentralize StarPU's scheduling policies as well. Instead of designing a scheduling strategy which implements the *Heterogeneous Ealiest Finish Time* scheduling algorithm (HEFT) [183] on a flat parallel machine, we can compose multiple strategies in a hierarchical fashion. Each instance of the scheduling algorithm either recursively assign tasks to one of the underlying scheduling strategies, or to a processing unit when the strategy considers a machine subset that is small enough to prevent scalability issues.

Another application of multi-level scheduling is to compose scheduling policies with complementary capabilities. This avoids designing extremely complex strategies with all features at the same time. On Figure 3.21(b), the HEFT scheduling strategy is enhanced with a reordering strategy that simply changes the order of the tasks, for instance depending on priorities or implementing a more complex heuristic (*e.g.* largest tasks first). Such strategies can also be combined with a strategy that implements a scheduling window to avoid scheduling too many tasks in advance to reduce the risk of misprediction. In Chapter 4, we will show that StarPU can execute a task in parallel over multiple processing units by submitting the same task simultaneously to multiple processing units. We can therefore improve the HEFT scheduling strategy to support parallel tasks by composing HEFT with a **gang** scheduling policy that duplicates tasks over the different processing units within the domain. As a result, Figure 3.21(b) depicts a scheduling strategy that reorders tasks (*e.g.* accordingly to priorities) before dispatching them between the different processing units, which can possibly execute them in a parallel way. Directly implementing such a complex strategy is possible, but requires a significant engineering effort that would here be limited when designing all these features within different composable strategies.

Composing scheduling strategies still remains a very delicate problem in general. Some strategies are for instance not directly compatible for composition. In the specific case of strategies which take scheduling decisions solely during the *push* phase, it is possible to combine strategies in a hierarchical way. Still, we need to provide a flexible and easy-to-use abstraction which would allow to schedule tasks either on a processing unit (*i.e.* a worker) or to another scheduling policy, possibly modeled by the means of a scheduling domain. Providing scheduling policies with such an abstraction of processing resource which can either be a physical processor or a scheduling domain is really challenging. While current strategies typically predict the availability of the different processing units based on performance models, it is for instance not clear whether the performance model associated to an entire scheduling domain should be the speed of the fastest device or the sum of the speeds of the different processing units within the domain.

3.10 Discussion

In this chapter, we have shown why task scheduling is a crucial issue to fully exploit hybrid accelerator-based machines, even for algorithms which seem very simple to execute. While many programmers are looking for the perfect scheduling strategy that would fit any problem, we have found that there does not exist such an ultimate strategy. Instead, we have presented the design of StarPU's flexible scheduling engine, which provide programmers and third-party scheduling experts with a portable interface to easily design advanced scheduling policies.

We have shown that programmers can greatly improve the performance by providing hints to the scheduler. This avoids requiring the scheduler to *guess* what programmers *know*, as this is

often the case with schedulers which are not providing a flexible enough interface. Considering the huge impact of data transfers on accelerator-based platforms, it is also worth noting that we have illustrated how an efficient data-aware scheduling helps to drastically reduce the pressure on the I/O bus. Depending on the algorithm, and on the machine, one should select an appropriate strategy. Having a uniform interface makes it easy for the application to directly test which are the most efficient strategies without modifying the application. This task is also simplified by using the performance analysis tools described in Chapter 6 and more precisely with the automatic estimation of the optimal execution time detailed in Section 6.4.

Besides the numerous improvements that can be made to the different scheduling strategies currently available in StarPU, a major challenge consists in being able to efficiently design composable strategies to cope with scalability concerns and to provide an efficient support for concurrent parallel libraries. Another significant challenge would be to allow scheduling algorithmic experts to express their scheduling algorithms in a high-level formalism, for example by describing the policy as a set of constraints to be optimized in Prolog.

Chapter 4

Granularity considerations

Chapter Abstract

In this chapter, we discuss about the crucial problem which consists in selecting the most appropriate task granularity to take advantage of manycore and accelerator-based platforms. We show that granularity is a critical issue that must be considered to efficiently target such platforms. Multiple approaches make it possible to design scalable algorithms with granularities that are suitable to deal with heterogeneity concerns. We first consider the benefits of parallel tasks and we describe how they are implemented in StarPU. We will then describe strategies to schedule parallel tasks. Finally, we consider another challenging approach which consists in dynamically adapting the granularity by dividing or merging tasks so that they can be efficiently executed on the different types of processing units.

4.1 Finding a suitable granularity

In this section, we show that manycore and hybrid platforms introduce challenging issues to select an appropriate task granularity. We then present various approaches to extend pure task parallelism in order to address these concerns with hybrid paradigms.

4.1.1 Dealing with embarrassingly parallel machines

As the number of cores keeps continuously growing, programmers must really focus on generating enough parallelism to maximize core occupancy. With upcoming architectures expected to exhibit hundreds of cores per processor, keeping all cores busy on such embarrassingly parallel machines becomes a great challenge.

Specific libraries such as PLASMA, dealing with large dense data sets, manage to generate large graphs of tasks by using the smallest possible granularity per task [35]. However, in the general case, it is not always efficient to extract an arbitrary degree of parallelism from a given algorithm: tiny tasks cannot amortize the overhead of task creation and destruction, dealing with a large amount of tasks can incur a significant scheduling overhead, and, finally, increasing the number of tasks impairs some algorithms by generating more communication, more synchronization, etc. (*e.g.* stencil or unstructured grid methods).

In such situations where the number of tasks is smaller than the number of cores, finding additional sources of parallelism inside tasks themselves can solve the problem of core occupancy. The idea is to split some tasks over multiple processing units, using nested parallelism, that is, *parallel tasks*. In many cases, such a nested parallelism can be obtained indirectly by simply taking advantage of existing parallel libraries [152].

4.1.2 Dealing with computation power imbalance

Using multicore machines equipped with GPU accelerators is obviously even more challenging, not only because GPUs require to be programmed using specific languages and APIs, but also because they exhibit a significant computing power imbalance compared to CPUs.

Until recently, this problem has been neglected by most programmers who decided to give up CPU power and concentrate on exploiting GPUs efficiently. But since the number of cores has been constantly increasing, several projects have started to investigate how to exploit heterogeneous sets of processing units [157, ATNW09]. In particular, developers of specific parallel libraries (e.g. MAGMA [182], FFT [149]) have proposed algorithm-specific solutions where the problem is statically partitioned in such a way that each processing unit is assigned a workload proportional to its power. Although such approaches are usually quite efficient, they are hardly portable. Moreover, statically dividing the work can be impractical to implement in some cases, either because the hardware features too many kinds of heterogeneous processing units, or because the initial problem is not flexible enough to be divided arbitrarily. By using parallel tasks that may spread across multiple cores, we can however use a coarser task granularity that will better match the power of GPUs. The idea is to schedule such tasks over a single GPU or over multiple CPUs. This way, there is no need to decompose the initial problem in tasks of various sizes.

Designing a generic methodology to dispatch computation between the heterogeneous processing units is however a real problem. A first approach consists in dividing the work into mul-

tiple parts of equal size, and to distribute the different pieces of computation to the various processing units. If the number of parts is large enough, faster processing units simply process more parts than the slower ones, and most of the computational power is used. When the problem is too small to be divided in a really large number of sub-problems (*e.g.* when considering weak scalability), this greedy strategy is not sufficient. It indeed happens that a single GPU may process the entire problem faster than the time required to process a single sub-problem on a CPU core. In that case, the optimal scheduling consists in assigning all tasks to fast GPUs and to ignore CPU cores, regardless of their number. Considering a machine with a single accelerator and hundreds of slow CPU cores, this is of course not an acceptable solution. On the other hand, **parallelizing tasks over multiple CPU cores** makes it possible to take advantage of both CPUs and GPUs even when the amount of parallelism is limited. Another possible approach would either consist in further **dividing tasks that were assigned to CPUs into smaller tasks** with a granularity that would be suitable for multicore CPUs even though it would have been too small for accelerators such as GPUs. Finally, KURZAK *et al.* have investigated the reverse solution which is to create tasks with a small granularity from the beginning, and to **merge small tasks into larger tasks when they are assigned on a GPU device**. While their approach is interesting, it is not really generic as it assumes that kernels can easily be merged into a single kernel. In the specific case of BLAS kernels, this was achieved by reimplementing new kernels dealing with data in a non standard layout, which represents a huge programming effort, even for a simple matrix multiplication kernel.

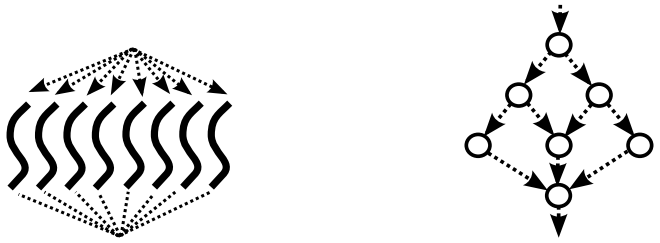
4.2 Parallel tasks

In this section, we describe how StarPU can combine tasks and SPMD parallelisms to execute DAGs of parallel tasks.

4.2.1 Beyond flat parallelism

Many environments and languages have been designed to ease programming of shared-memory multiprocessor machines. For long, the most dominant parallel scheme used when developing applications over such machines was SPMD (*Single Program Multiple Data*), where all the processors execute the same code on different data, in a loosely-coupled manner. The *Fork-Join* model exemplified by OpenMP illustrates well how such an approach can lead to easy-to-understand programs. Likewise, invoking parallel libraries (*e.g.* LAPACK, FFTW, etc.) within sequential code is a typical way of seamlessly getting the benefits of parallel computing. While there exists a tremendous amount of parallel codes, the overall performance of applications is directly limited by the scalability of the parallel libraries that they use. Such a **flat parallelism**, depicted by Figure 4.1(a), is therefore limited on large parallel platforms.

The emergence of multicore processors has introduced new issues compared to previous small-scale multiprocessor machines. Many existing parallel applications do not behave so well on multicore machines, because the underlying architecture has changed: memory access times are not uniform any more, cores are sharing a complex hierarchy of caches, etc. Indeed, exploiting the full computational power of always deeper hierarchical multiprocessor machines requires a very careful distribution of threads and data among the processing units. This certainly accounts in the recent interest of many programmers for task-based execution models, where applications are represented by directed acyclic graphs of tasks which only share data in a producer-consumer in-



(a) Fork-join parallelism. (b) Task parallelism.

Figure 4.1: Parallel programming paradigms.

teraction as illustrated on Figure 4.1(b). The developers of the PLASMA linear algebra library for multicore machines, for instance, have completely redesigned their algorithms to generate large graphs of different types of tasks, getting rid of the former fork-join execution model. As previously discussed in Section 2.1.1, task parallelism has become especially successful on accelerator-based platforms [21, 119, 117, 193].

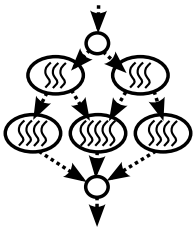


Figure 4.2: Hybrid DAG with parallel tasks.

One would however naturally like to mix the flexibility of task parallelism with the simplicity of parallel libraries. This is illustrated on Figure 4.2 which combines the two paradigms shown on Figure 4.1. In the past, the huge performance improvements observed on accelerator-based platforms very often relied on having large enough input data. This behaviour is usually called *weak scalability*, as opposed to *strong scalability* that considers the performance of a parallel system for a fixed problem size. Unfortunately, the high number of CPUs cores, combined with the imbalance between the performance of accelerators and those of the CPUs is now becoming a real concern.

We have thus modified StarPU to combine the advantages of task-parallelism and those of flat-tree parallelism in a hybrid model with tasks that run on multiple processing units at the same time. We will show that such a hybrid model allows significant scalability improvements. Another benefit is to enforce a better separation of concerns: the application is designed as a set of tasks, each task being possibly parallelized using the best parallel libraries available or thanks to parallel programming languages (*e.g.* TBB or OpenMP). It is worth noting that the Lithe environment has indeed proved that such a composition of parallel libraries is possible without too much impact on the code [152].

4.2.2 Supporting parallel tasks in StarPU

The key problem in implementing parallel CPU tasks is to ensure that multiple CPU workers (collectively called a *parallel CPU worker*) are given the same task simultaneously. In order to assign

```
1  while (machine_is_running())
2  {
3    handle_local_data_requests();
4    task = pop_task();
5
6
7
8    acquire_task_data(task);
9
10
11
12
13   task->cl->cpu_func(task->interface,
14                      task->cl_arg);
15
16
17
18   release_task_data(task);
19   handle_task_termination(task);
20
21 }
```

Figure 4.3: Original CPU driver.

```
1  while (machine_is_running())
2  {
3    handle_local_data_requests();
4    task = pop_task();
5    rank = task->alias_count++; /* Atomic op */
6
7    if (rank == 0)
8      acquire_task_data(task);
9
10   barrier_wait(task->barrier);
11   if ((rank == 0) ||
12       (task->cl->type != FORKJOIN))
13     task->cl->cpu_func(task->interface,
14                        task->cl_arg);
15   barrier_wait(task->barrier);
16
17   if (rank == 0) {
18     release_task_data(task);
19     handle_task_termination(task);
20   }
21 }
```

Figure 4.4: CPU driver supporting parallel tasks.

a task to a parallel CPU worker, the scheduler therefore submits the same task to the workers. A barrier structure and a counter variable are initialized in the task structure to ensure synchronization. Figure 4.4 shows the pseudo-code of the driver controlling a CPU core which may execute parallel tasks. When the worker retrieves a task from the scheduler (line 4), its rank is computed by atomically incrementing the counter variable (line 5).

All CPU workers access their data from host's memory which is shared between all CPUs. The master worker (*i.e.* which gets rank zero) is therefore the only one that calls the data management API to enforce data consistency before and after the execution of the parallel task (lines 8 and 16). Workers are synchronized before and after the execution of the codelet, so that the different processing units are fully devoted to the parallel task (lines 10 to 15).

A choice between two paradigms is provided in the parallel codelet structure: a parallel task is either executed in a *Fork-join* or in an *SPMD* mode. In Fork-Join mode, the codelet is only executed by the master worker while the other CPU cores (slaves) are blocked on a barrier until the master ends (line 11). The master is therefore allowed to launch new threads or to call a parallel library that will run on the dedicated set of cores. In SPMD mode, each worker executes the codelet. The rank of the worker and the size of the parallel section are respectively returned by the `starpu_combined_worker_get_rank()` and the `starpu_combined_worker_get_size()` functions.

4.2.3 Parallelizing applications and libraries

Little code modifications are required to make existing StarPU applications benefit from parallel tasks. Data registration and task definition are mostly unchanged, except that the kernel CPU implementation may be parallelized when possible. The choice between SPMD and Fork-Join modes is made when designing the parallel kernels, depending on which is more natural or suitable for the kernel developer. The Fork-Join mode is typically suited to codelets that call library functions that are already parallelized internally. As detailed above, StarPU ensures that processing units

are devoted to such parallel library call by having all slave workers wait on the barrier (line 15 of Figure 4.4). The SPMD mode is convenient when the codelet designer does not want to create extra threads to perform the computation, either because this can reveal itself expensive to do it on each kernel execution, or simply because it makes the code easier to express.

StarPU therefore provides a generic way to execute parallel kernels coming from various environments. In the future, we plan to make it possible to call tasks implemented in standard parallel environments such as OpenMP or TBB. Such environments however often assume that they are called only from sequential codes, so that they take control of the entire machine, and do not support concurrent instances. Such environments will have to permit to create parallel sections concurrently from several system threads, and to restrict the execution of such parallel sections to the sub-set of the CPU cores specifically devoted to the parallel task. PAN, HINDMAN and ASANOVIĆ have already achieved [152] such improvement without too invasive changes.

Such constraints also apply to parallel libraries which must be able to be confined and thread-safe so that they can be used within concurrent parallel tasks, i.e. *composable*. For instance, the FFTW library, which creates a pool of threads to perform jobs of an FFT computation, can be modified to create several pools of threads which can work concurrently. It is also worth noting that libraries that are written on top of standard parallel programming environments (*e.g.* OpenMP, TBB, etc.) would automatically inherit such properties from the underlying programming environments. As already exposed by Pan *et al* [152], making sure such parallel environments are *composable* is a crucial problem in the manycore era.

4.2.4 A practical example: matrix multiplication

Matrix product is one of the most common kernel in HPC. Tiled algorithms are new classes of kernels that have been redesigned to deal with matrices that are stored with a different layout in order to take advantage of the architectural features found on modern processors. Instead of storing matrices as contiguous arrays (*i.e.* as in LAPACK), tiled matrices are divided into sub-blocks, called tiles, that are stored contiguously. This indeed provides a much better data locality and greatly reduces the overhead of data transfers between main memory and accelerators.

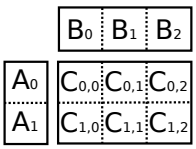


Figure 4.5: Product of two tiled matrices. The result is a 2×3 tiled matrix which is computed with only 6 tasks. Each tile (*e.g.* A_1 or $C_{0,2}$) is stored contiguously in memory.

The product kernel typically appears in many other dense linear algebra problems (*e.g.* during Cholesky decomposition). As shown on Figure 4.5, the number of tasks that compose a matrix product is given by the number of tiles in the target matrix, which is a parameter of the algorithm. Tiles have to be large enough to ensure that the kernels are efficient. It is therefore important to be able to deal with a limited amount of parallelism, even for such a simple algorithm, when the matrix size is small.

Parallel matrix multiplication kernel

```
1  void dgemm_cpu(void *descr[], void *cl_arg)
2  {
3    /* Get pointers to local data replicates */
4    double *A = STARPU_MATRIX_GET_PTR(descr[0]);
5    double *B = STARPU_MATRIX_GET_PTR(descr[1]);
6    double *C = STARPU_MATRIX_GET_PTR(descr[2]);
7
8    /* Get the dimensions of the matrices */
9    unsigned mC = STARPU_MATRIX_GET_M(descr[2]);
10   unsigned nC = STARPU_MATRIX_GET_N(descr[2]);
11   unsigned nA = STARPU_MATRIX_GET_N(descr[0]);
12
13   /* Rank and size of the parallel section */
14   int size = starpu_combined_worker_get_size();
15   int rank = starpu_combined_worker_get_rank();
16
17   /* Compute the size of the sub-block */
18   int bsize = (nC + size - 1)/size;
19   int bnC = MIN(nC, bsize*(rank+1)) - bsize*
         rank;
20   double *bB = &B[bsize*rank];
21   double *bC = &C[bsize*rank];
22
23   /* Do the actual BLAS call */
24   DGEMM(mC, sub_nC, nA, 1.0, A, bB, 0.0, bC);
25 }
```

Figure 4.6: Code of the Parallel Matrix Product kernel in SPMD mode.

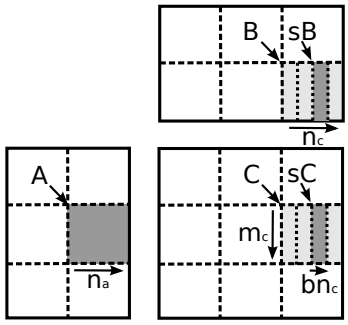


Figure 4.7: Parallel Matrix Product Algorithm.

Matrix multiplication is an example of kernel that can easily be parallelized. Figures 4.6 and 4.7 respectively show the code and the algorithm of a matrix multiplication kernel parallelized in an SPMD fashion. StarPU ensures that the A, B, and C matrices are available in main memory when the task starts. The description of the matrices is given by the `STARPU_MATRIX_GET_*` macros. We here read the addresses of the local matrices (`PTR`), as well as their number of rows and columns (`M` and `N`).

Lines 14 and 15 show how the application gets the size and the rank of the codelet. These values are used to determine which sub-parts of the input matrices should actually be computed by this worker (lines 18 to 21). The `DGEMM` BLAS function is finally called to compute the sub-matrix that starts at address `bC`.

Scalability of a Tiled Matrix Product with low parallelism

Figure 4.8 shows the strong scalability of the multiplication of two 2048×2048 matrices, which is a relatively small problem, on 12 CPU cores. This operation is split into a fixed number of tasks which is shown on the x-axis (it corresponds to the number of tiles of the C matrix on Figure 4.5). The number of tasks is rather small: there can be less tasks than there are CPU cores. The y-axis gives the speed measured in GFlop/s. Sequential and parallel tasks are dynamically scheduled using the HEFT and the Parallel-HEFT scheduling strategies, respectively (Parallel-HEFT will be described in Section 4.3.2).

When running sequential tasks, the performance of the multiplication is only optimal when

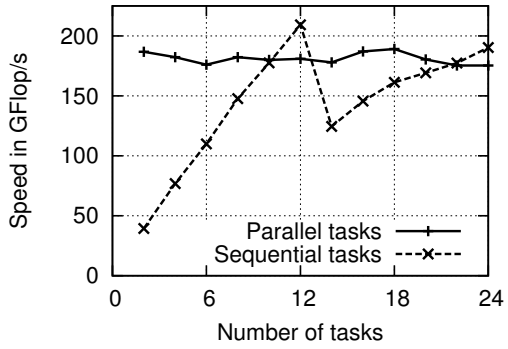


Figure 4.8: Product of two 2048×2048 single precision matrices with a small number of tasks running on 12 CPU cores.

the number of tasks exactly fits the number of cores. The machine indeed remains under-used as long as there are less tasks than the number of CPU cores. If the number of tasks is not exactly a multiple of the number of CPU cores, some cores become idle when the last tasks are processed sequentially.

Much more stable performance are however obtained when using parallel kernels, regardless of the number of tasks. The 10 % performance loss obtained by parallel tasks compared to 12 sequential tasks is explained by the limited strong-scalability of the parallel GEMM kernel. Such performance loss might be undesirable when it is possible to make sure to match the number of tasks with a multiple of the number of cores. A flexible scheduling policy however helps when the amount of parallelism (*i.e.* the number of tasks) evolves with time. The number of tasks is also a parameter that may not be easily tunable: some algorithms can for instance only be split in a quadratic number of tasks.

4.3 Scheduling parallel tasks

In this section, we show how to design scheduling policies, and how they can support parallel tasks. We also give some insights about the difficult problem of selecting the size of the parallel sections, and which workers should be combined together to ensure the best performance.

4.3.1 Taking machine hierarchy into account

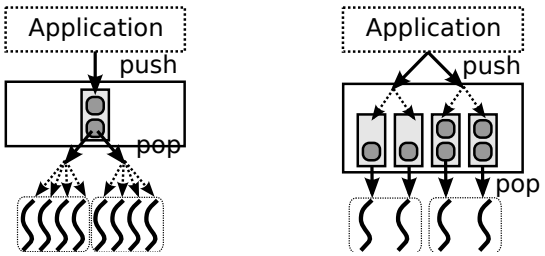
In order to fully take advantage of parallel tasks, one needs to consider how tasks get mapped on the actual hardware. In particular, taking hierarchy into account is a crucial problem to ensure good performance. For instance, it is preferable to combine workers that share the same cache or which are on the same NUMA nodes, instead of creating teams of unrelated processing units.

StarPU uses the hwloc [30] library to detect the topology of the machine in a portable way. This ensures that parallel tasks are only scheduled on workers that are close enough within a hierarchical machine.

It must however be noted that finding the best combinations of processors is a hard problem in general. On the one hand, a parallel computation intensive kernel (*e.g.* BLAS3) only gets the best performance if processors are close enough to keep data within the caches. On the other hand,

a parallel memory intensive kernel (*e.g.* BLAS1) may benefit from a better aggregated memory bandwidth when the processors are located on different NUMA nodes.

4.3.2 Scheduling strategies for parallel tasks



(a) Parallel greedy. (b) Parallel HEFT.

Figure 4.9: Scheduling strategies support parallel tasks by submitting tasks simultaneously to multiple workers. The dotted arrows indicate that the task is duplicated when the scheduling decision is taken.

We have extended the **greedy** (resp. **heft**) scheduling policy into the **parallel-greedy** (resp. **parallel-heft**) which supports parallel tasks. In Section 4.2.2, we have explained that parallel tasks are implemented by submitting the same task multiple times to different workers. Multiple replicates of the parallel task are created when the policy decides to assign the task to a parallel CPU worker (*i.e.* in the *pop* method of the **greedy** policy, or in the *push* method of the **heft** policies). This task duplication is illustrated by the dotted arrows on Figure 4.9.

On Figure 4.9(a), the different workers are partitioned into groups of fixed size (here, two groups of size 4) during the initialization of the scheduler. One worker is elected for each group: when this *master* grabs a task from the shared queue, a copy of this task structure is passed to each of the different slave workers that belong to the same group. This variation of the greedy strategy, called **parallel-greedy** is therefore able to execute tasks that are parallelized between the workers of the different groups. It should be noted that it is possible to have parallel sections which are smaller than the size of the worker group by submitting task aliases only to a subset of the slave workers. This also permits to mix sequential and parallel tasks by having only the master worker execute sequential codelets.

The strategy shown on Figure 4.9(b) is a modified version of the **heft** strategy described in Section 3.4.3. When deciding which worker minimizes the termination time of a task, this strategy considers the performance model of the parallel codelets as well. The predicted execution time, and therefore the predicted termination time, thus now also depends on the number of workers in the parallel section. When a task is assigned to a parallel worker, aliases of the task are submitted to each of the underlying workers. A special care is taken to avoid deadlocks with prioritized tasks: the ordering of the task aliases is the same within all local queues.

Both strategies are examples of multi-level scheduling policies. They could be obtained as the composition of a strategy (*e.g.***greedy** or **heft**) with a simple strategy that merely dispatches aliases of the tasks between the different workers of the group. In the future, we expect to use such composition to automatically add support for parallel tasks into some of the other existing strategies. However, this is not directly applicable to strategies that cannot ensure that the ordering of the

tasks will not be changed, or that aliases of the same task are assigned to the same worker (*e.g.* because of work-stealing mechanisms).

4.3.3 Dimensioning parallel tasks

Finding the optimal size of parallel workers is also a difficult problem. Parallel tasks may not be effective on a too small parallel team of workers. Too large parallel tasks might not be scalable enough. A pragmatic approach usually used for hybrid programming models [163] consists in mapping parallel tasks directly on a resource shared by a subset of the processors (*e.g.* one task per NUMA node, or one task per processor socket).

Let us now consider for instance the specific case of an application with a fixed amount of identical tasks running on a hybrid platform composed of k_{gpu} GPUs, and k_{cpu} CPU cores. Let us assume that there is a total of N tasks, and that the relative speedup between the GPU and the CPU implementations is s (*i.e.*. the kernel runs s times faster on a GPU than on a single CPU core). In the remaining of this section, we now try to determine what the minimum value of N is for all processing units to be used.

Assuming the duration of the kernel is 1 on a GPU, the total time required to execute the whole problem entirely on GPUs is $\lceil \frac{N}{k_{gpu}} \rceil$. Since the execution time of a single CPU task is s, it is never useful to execute one of the N tasks on a sequential CPU core if this time s is longer than the time required to compute the entire problem on GPUs.

This forbids the use of CPUs, regardless of the number of CPU cores, as long as:

$$\left\lceil \frac{N}{k_{gpu}} \right\rceil \leq s \tag{4.1}$$

Using parallel tasks permits us to relax this constraint: if we assume that we have a perfect scaling of the parallel CPU task, we can create a parallel CPU worker that combines all CPUs. The relative speedup between the GPU and the parallel CPU implementation becomes $\frac{s}{k_{cpu}}$. The previous constraint is thus relaxed:

$$\left\lceil \frac{N}{k_{gpu}} \right\rceil \leq \frac{s}{k_{cpus}} \tag{4.2}$$

For example, if one CPU is 4 times slower than one GPU, and there are 10 tasks, if there are 12 CPU cores along with 3 GPUs, the first constraint is not met because $\frac{10}{3} \leq 4$. However, if we combine the 12 cores, we can take advantage of CPUs because $\frac{10}{3} > \frac{4}{12}$. This shows how parallel tasks can be a very convenient way to overcome the imbalance of processing power between the different types of processing units in a hybrid platform.

It should be noted that if the number of CPU cores gets high, parallel CPU workers may actually have to wait for the GPUs to finish their computation if executing a single task on a GPU is slower than with a parallel CPU worker. In that case, the CPU workers should try to execute multiple parallel tasks, either serially, or more likely using smaller parallel sections to prevent scalability concerns within the parallel CPU tasks.

As a consequence, we can determine the minimum number of tasks required to take advantage of a hybrid system. If the tasks do not have a parallel CPU implementation, we need to have enough tasks to keep all GPUs executing tasks while executing a single task on each CPU core:

$$N \geq k_{gpu} \times s + k_{cpu} \tag{4.3}$$

If tasks have a parallel CPU implementation, we can execute parallel tasks on all the CPUs, which thus execute about k_{cpu} times faster than on a single CPU. Either the GPUs remain faster than such parallel worker ($\frac{s}{k_{cpu}} \geq 1$), in which case the minimum number of tasks to make use of all processing units is obtained by creating a single parallel CPUs task and keeping GPUs busy during this task, the minimum is thus

$$N \geq k_{gpu} \times \left\lceil \frac{s}{k_{cpu}} \right\rceil + 1 \tag{4.4}$$

, or this parallel worker becomes faster than a GPU ($\frac{s}{k_{cpu}} \leq 1$), and the minimum number of tasks is obtained by putting one task on each GPU and having multiple parallel CPU tasks, the minimum is then

$$N \geq k_{gpu} \times 1 + \left\lceil \frac{k_{cpu}}{s} \right\rceil \tag{4.5}$$

Since $1 \geq \lceil \frac{k_{cpu}}{s} \rceil$ in the first case and $1 \geq \lceil \frac{s}{k_{cpu}} \rceil$ in the second case, equations 4.4 and 4.5 can be generalized as

$$N \geq k_{gpu} \times \left\lceil \frac{s}{k_{cpu}} \right\rceil + \left\lceil \frac{k_{cpu}}{s} \right\rceil \tag{4.6}$$

Which is indeed better than equation 4.3.

This very simple model permits us to determine how many tasks should be created to take advantage of a hybrid platform. It also confirms that parallel tasks make it easier to fully exploit a hybrid platform when the amount of parallelism is limited. Its practical use is shown in the case of a CFD kernel parallelized on a hybrid platform in Section 7.7.2.

4.4 Toward divisible tasks

Parallel tasks permit to extract parallelism without modifying the task graph. Sometimes, it is possible to actually modify the task graph to subdivide the tasks into smaller tasks which are scheduled independently on multiple processing units. In the case of algorithms which do not expose enough parallelism, this makes it possible to reach a better scalability by ensuring that tasks which constitute a bottleneck in the algorithm are spread over multiple processing units.

The amount of parallelism may also vary during the execution of the algorithm: during a Cholesky decomposition which recursively processes subsets of the input matrix, the amount of parallelism keeps decreasing. As a result, the optimal block size depends on the input problem size: large problems can be solved using a large granularity which ensures that the matrix multiplications are performed efficiently, but small matrices should be factorized with a small block size to provide a sufficient amount of tasks. Since the last steps of the decomposition of a large matrix consists in decomposition a small subset of the matrix, this indicates that the granularity should be adapted during the execution of the algorithm. Designing such an algorithm with varying block size is not an easy programming task. It is however easier to design an algorithm with a fixed block size and to subdivide large tasks later on during the execution.

From a programming point of view, we first need to provide a convenient mechanism to manipulate divisible tasks. A low-level interface could consist in making it possible to provide a

callback function along with the task structure: when StarPU decides (or is told) to divide the task, it calls this callback function which can for instance replace the task by a set of tasks doing the same work at a smaller granularity. Such a low-level interface could also be hidden in case StarPU is used as a back-end for a compiling environment. A compiler like PIPS could for instance generate multiple variants of code by extracting tasks at different granularities and provide StarPU with a callback function to replace a kernel operating on a large piece of data by smaller kernels.

A difficult problem is to decide when tasks should be divided. A first approach consists in relying on specific algorithmic knowledge to manually decide when granularity needs to be adapted. In the case of the Cholesky decomposition previously mentioned, the amount of parallelism is for instance known to systematically drop throughout the algorithm, so that a numerical kernel designer could decide that tasks should be divided when the submatrix currently being processed is smaller than a fixed ratio. This decision can also be made more dynamically, either from the application or directly from an advanced scheduling policy. For example one can monitor performance feedback information to determine whether there is too little parallelism and that ratio between StarPU's overhead and the actual work is low enough to justify smaller tasks. Considering that granularity concerns often result from heterogeneity (*e.g.* GPUs typically process much larger tasks than CPU cores), another possible approach is to create multi-level schedulers. The scheduling policy would first decide which type of processing unit should execute a given task, and split it into multiple tasks in case the targeted architecture requires smaller tasks. This is conceptually similar to the reverse approach we adopted on the Cell processor which consists in merging small tasks together to construct tasks that are large enough to accommodate with the latency of an accelerator.

Besides scheduling divisible tasks which is a challenging issue of its own, it must be noted that managing data simultaneously at multiple scales is a very difficult problem because StarPU maintains data coherency per data handle. This means that if we apply a filter on a registered piece of data, it is not possible to access simultaneously a data subset and the entire filtered data in a coherent way. As a result, tasks with a varying granularity may need to filter data to allow concurrent accesses on the different data subsets, but there cannot be an undivided task that accesses the entire unfiltered data at the same time. We are currently investigating how to remove this constraint in StarPU, at least to allow concurrent read accesses on overlapping pieces of data. Similarly to the approach used to create multiple incoherent data replicates when accessing a piece of data in a reduction mode (see Section 2.4.2 for more details), we could explicitly (re)register data subsets accessed concurrently so that StarPU would not figure out that multiple tasks are accessing the same piece of data at multiple scales.

4.5 Discussion

Granularity concerns do occur in many situations when porting algorithms on heterogeneous platforms because we need to fulfill contradictory goals which are to provide large enough tasks to SIMD accelerators and small enough tasks to cope with the high degree of parallelism found in manycore processors. Determining beforehand which type of processing unit should process the different types of tasks is a way to determine in advance which should be the granularity of the different tasks, but it violates our assumption which is that the user should not have to manually schedule tasks.

Runtime systems like StarPU must therefore provide convenient mechanisms to allow programmers to design algorithms with a variable granularity to incline them to actually consider this delicate issue which must be addressed to fully exploit heterogeneous multicore platforms enhanced with accelerators. Divisible tasks permit to design algorithms which an adaptive granularity, but there remains a large number of challenges which must be solved to make this approach realistic. It is not only hard to provide StarPU with a convenient representation of divisible tasks, but automatically deciding when to modify the granularity is an even more difficult problem.

Parallel tasks are a very convenient way to obtain extra parallelism without modifying the initial algorithm. Considering that parallel libraries are also becoming widespread, providing StarPU with parallel kernels instead of sequential ones is not necessarily a difficult issue. However, while HPC libraries traditionally assume that the entire machine is dedicated, it is now crucial that library designers consider the possibility of concurrently invoking parallel routines within the application. Most LAPACK implementations for instance do not allow the application to simultaneously invoke multiple parallel kernels from different OS threads. In case the parallel libraries are implemented on top of other programming environments such as TBB or OpenMP, it is crucial that the implementation of these programming environments offers support for concurrency. Implementations of the OpenMP standard should for instance make it possible to execute multiple independent parallel sections at the same time to allow parallel libraries written in OpenMP to be invoked from parallel StarPU tasks. Providing an efficient support to efficiently handle concurrency between parallel environments is therefore a critical concern which must be addressed in the different layers of the software stacks.

Chapter 5

Toward clusters of machines enhanced with accelerators

Chapter Abstract

In this chapter, we present how StarPU integrates in a cluster MPI-based environment. We first discuss about the suitability of our task-based paradigm in a distributed environment. Then, we describe some challenges which appear when introducing accelerators in MPI applications, and we present a small library that permits to provide an MPI-like semantic to exchange data between multiple instances of StarPU running on different nodes, which makes it easy to accelerate existing MPI applications with StarPU. In case it is possible to rewrite the application using a task paradigm, we show a systematic methodology to easily map DAGs of StarPU tasks on clusters of machines equipped with accelerators. Finally, we describe how this methodology was implemented to provide application programmers with a convenient sequential function-call like semantic to seamlessly submit StarPU tasks throughout a cluster.

5.1 Adapting our task-based paradigm to a cluster environment

StarPU is intended to provide task scheduling and data management facilities within a multicore machine enhanced with accelerators. In spite of the tremendous amount of processing power now available within such machines, the problem sizes encountered in many HPC applications make the simultaneous use of multiple machines necessary. In this chapter, we therefore present extensions of the StarPU model so that applications can fully exploit clusters of multicore machines enhanced with accelerators. MPI is by far the most common programming environment used to develop HPC applications on clusters. We therefore need to find a way to provide MPI support for applications within StarPU. The integration of StarPU and MPI can take different aspects, depending whether we accelerate existing MPI codes, or whether we distribute existing StarPU applications over clusters.

One one hand, provided the huge amount of MPI applications, we need to make it possible to accelerate existing MPI applications so that they can take fully advantage of accelerators thanks to StarPU. Considering that StarPU would typically be used inside libraries, it may not be possible to modify the entire application so that it fully relies on StarPU to take care of both data management and load balancing. Instead of having a single instance of StarPU distributed over the entire cluster, our approach is to have an instance of StarPU initialized on each MPI node. The flexibility of such hybrid programming models has already been illustrated in the case of MPI applications which call libraries written in OpenMP or TBB for instance. It is therefore up to the application to decide which MPI process should submit a given task to its local instance of StarPU. Even though data management is still performed by StarPU within a MPI node, a message-passing paradigm implies that the different nodes should be able to exchange data managed locally by StarPU. In Section 5.2, we therefore present a small library implemented on top of StarPU which provides an MPI-like semantic to transfer the piece of data described by a data handle into another data handle located in a different MPI process.

On the other hand, we must provide a convenient way to write new applications, or to extend StarPU applications so that they can exploit clusters. There is indeed a real opportunity for applications that are not already too large or too complex to be rewritten using a task-based paradigm which provides a portable abstraction of the different algorithms that can be efficiently mapped on very different types of platforms, going from single-node multicore machines to large clusters of multicore machines equipped with accelerators. In Section 5.4, we therefore show the distributed extension of the function-call like semantic used in Section 2.2.5 to automatically generate DAGs of tasks which are derived from implicit data-driven dependencies.

5.2 Managing data in an MPI world enhanced with accelerators

In order to accelerate MPI applications with StarPU, we must be able to exchange registered data between the different instances of StarPU. There are various issues which must be addressed to efficiently couple StarPU and MPI. The first challenge is to mix the asynchronous task paradigm used in StarPU with the SPMD (*Single Program Multiple Data*) paradigm used in MPI. When transferring a piece of data that was registered to StarPU on MPI, we must also consider that there might be no valid data replicates in the main memory, so we sometimes need to fetch a valid replicate from one of the accelerators at first.

A certain number of low-level technical issues typically appear when mixing MPI and accel-

erators. Besides the fact that we need to keep track of the location of valid data replicates, we usually cannot transfer data directly between an accelerator and the network, so that multiple intermediate transfers may be required to transfer a piece of data over the network. Thread safety is also a serious concern for most MPI implementations and for most accelerator programming environments. CUDA (before its fourth release) and OpenCL implementations, for example, are not thread safe. Considering the case of a non-thread safe MPI library initialized by the application, transferring a piece of data located on a CUDA device that is controlled by another thread would first imply that the thread controlling the CUDA device should issue a memory copy from the device to the host memory by the means of CUDA's specific API. When this transfer is done, the thread which has previously initialized the MPI library is responsible for sending data from the host memory to the network. In addition to the severe portability issues which result from the simultaneous use of various vendor-specific APIs combined with MPI, a significant expertise is required to manually implement this protocol in an asynchronous fashion in order to hide most data transfer overhead.

The data management API provided by StarPU provides a convenient way to hide this low-level problems. Its acquire and release functions indeed consists in asking StarPU to lock a valid copy of the data handle in host memory (acquire), and to unlock this data replicate from host memory (release). When a registered piece of data must be sent over the network, the main thread simply has to keep a valid data replicate locked in main memory by the means of the acquire/release semantic during the entire MPI transfer. Since the application thread is blocked until transfer completion, this method is somehow limited to very synchronous MPI applications, for instance with a fork-join parallelism. In the case of loosely coupled MPI applications which for instance only use MPI to scatter and gather data at the beginning and at the end of computation, and which uses accelerators in between, such an approach is a suitable way to easily integrate existing MPI codes because the application only performs MPI operations on data which is guaranteed to be located in main memory. Deeply integrating synchronous MPI transfers with a fully asynchronous task paradigm however potentially reduces the amount of parallelism potentially unlocked when reordering tasks (*e.g.* when executing the critical path as soon as possible by the means of user-provided task priorities).

To cope with the asynchronous nature of StarPU, we therefore extended the acquire/release semantic with a non-blocking acquire method that asynchronously fetches a piece of data in host memory. A user-provided callback is executed when the data handle is available to the application An application can thus perform an operation over a piece of data (*e.g.* here send it over MPI) without blocking the main application thread. In a way, submitting a non-blocking acquire operation is similar to submitting an asynchronous task that operates on a single piece of data.

5.3 A library providing an MPI-like semantic to StarPU applications

Even though the non-blocking acquire/release semantic provides a powerful mechanism which can be used to exchange registered data between different MPI processes, using such a low-level facility is tedious. We therefore implemented a small library using this technique internally to provide an MPI-like semantic on top of StarPU to facilitate the integration of StarPU into existing codes and to easily parallelize StarPU applications with MPI.

5.3.1 Main API features

Table 5.1: Functions provided by our MPI-like library

Type of call	Examples of functions
Blocking	`starpu_mpi_{send,recv}` `starpu_mpi_{wait,barrier}`
Non-Blocking	`starpu_mpi_{isend,irecv}` `starpu_mpi_test`
Array	`starpu_mpi_array_{isend,irecv}`
Detached	`starpu_mpi_{send,recv}_detached`

The goal of this library is to provide an effective interface that can easily be used to implement message passing between multiple instances of StarPU. Instead of manipulating data addresses and data lengths, programmers can transfer data handles (which description is internally transformed into an address and an MPI data type).

Table 5.1 gives an overview of the different types of functions implemented by this small library. The send and receive operations are used to synchronously transfer the content from a data handle to another data handle located in a different MPI process. The asynchronous send and receive operations are very similar to the asynchronous data transfer API provided by MPI. Asynchronous transfers must therefore be completed either by explicitly waiting for transfer completion or by periodically testing whether the transfer has come to completion or not. Array calls simply extend the previous functions and allow to simultaneously transfer multiple data handles with a single call to our library.

In the case of a DAG of asynchronously submitted tasks, it is not practical to explicitly wait for the completion of each an every data transfer. Moreover, data and task dependencies are actually sufficient to find out that a data transfer needs to be terminated before we can use the corresponding data handle in a task. Similarly to the detached tasks (see Section 2.2.4) which are automatically cleaned up after their completion, the detached functions provided by our MPI-like library are asynchronous data transfers which need not be completed by calling test or wait functions. Thanks to an internal progression mechanism implemented within the library, the data handle is automatically released when the completion of the data transfer is detected. An optional user-provided callback also makes it possible to receive a notification when a detached transfer is terminated.

Figure 5.1 illustrates the simplicity to transfer data between multiple instances of StarPU by the means of our MPI-like library, and more precisely thanks to detached calls. Each process gets a token from its previous neighbour (line 8), increments it using a StarPU task (line 11), and sends the token to the next neighbour (line 14). The absence of explicit dependencies between these different operations shows that our MPI-like is nicely integrated with implicit dependencies which not only apply to tasks but also to other types of data accesses (*e.g.* acquiring, releasing or unregistering a piece of data). To summarize, the for loop merely submits all MPI communications requests and tasks with implicit dependencies to StarPU, without blocking, and then simply waits for StarPU to complete them in an optimized way. It is worth noting that even though the token was registered by every MPI process (line 3), the coherency of these tokens is managed independently by the different instances of StarPU. Indeed, the MPI-like library does not implement a Distributed Shared Memory (DSM) that would manage handles in a distributed fashion. What it

```
unsigned token = 0;
starpu_data_handle token_handle;
starpu_variable_data_register(&token_handle, 0, &token, sizeof(token));

for (unsigned loop = 0; loop < NLOOPS; loop++)
{
  if ((loop > 0) || (rank > 0))
    starpu_mpi_recv_detached(token_handle, prev_rank, TAG, MPI_COMM_WORLD, NULL, NULL);

  /* The increment_cl codelet increments the variable described by the handle */
  starpu_insert_task(&increment_cl, STARPU_RW, token_handle, 0);

  if ((loop < NLOOPS) && (rank < size))
    starpu_mpi_send_detached(token_handle, next_rank, TAG, MPI_COMM_WORLD, NULL, NULL);
}

starpu_task_wait_for_all();
```

Figure 5.1: Code of a MPI Ring using detached calls to increment a variable distributed over multiple machines.

really does instead is to provide a convenient API to transfer the content of a handle into another handle which was registered by another process.

5.3.2 Implementation overview

The implementation of this library is based on the non-blocking acquire functionality presented in Section 5.2. The general idea consists in using a thread dedicated to MPI operations: when the application wants to submit an MPI request that transfers a data handle, a non-blocking acquire operation is performed on the data handle, and the user-provided callback sends a signal to the MPI thread so that it performs the actual MPI transfer of the piece of data that is then guaranteed to be locked in main memory until it is explicitly released.

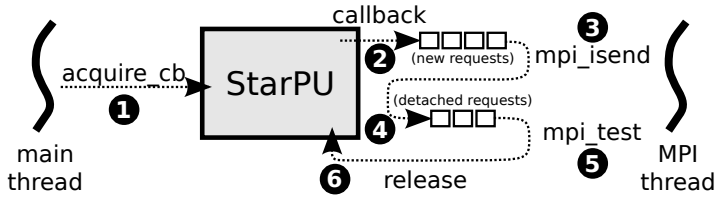


Figure 5.2: Implementation of the detached send operation.

Figure 5.2 describes the implementation of the `starpu_mpi_send_detached` function in more details. During the first step, the thread calling `starpu_mpi_send_detached` requests that the handle should be put back into host memory. When this asynchronous operation succeeds (*i.e.* when the handle is available in host memory), StarPU executes a callback function that enqueues a request in a list of new requests (Step 2). When the thread that is dedicated to MPI detects a new request, the actual MPI call is performed to transfer the handle that was previously put in host memory (Step 3). Detached requests should be automatically completed by StarPU: we therefore put the request in a list of detached requests that need to be completed (Step 4). The MPI thread periodically checks whether the different requests are terminated or not (Step 5). The

handle associated to a request that has come to completion is then released so that it is available again to the application (Step 6).

This protocol clearly incurs a small latency penalty, but it is designed to avoid all thread-safety issues, both in MPI implementations and in the drivers of the different accelerators. A thread-safe MPI implementation would for instance allows to skip Step 2 by directly performing MPI calls as soon as the data is available in host memory. Besides, native support for direct transfers between accelerators and network cards would also make it possible to avoid having to fetch data back into host memory: given such *peer transfers*, this library could for instance request that the handle should be locked on an accelerator (using a similar acquire/release semantic), and directly transfer data between the accelerator and the network card, thus saving a significant amount of bandwidth and greatly reducing the overhead of the current protocol. Such peer transfers are for instance possible between some NVIDIA GPUs and InfiniBand network cards by the means of the GPU Direct technology [176]. Unfortunately, most MPI implementations still provide a limited support for thread safety [184], and the direct transfer of data between accelerators and network card is an even less common capability.

The overhead introduced by the thread dedicated to MPI operations might also be a concern. Since there is no equivalent of the `select` system call in MPI, checking the completion of the different detached requests is a relatively expensive operation because we have to continuously poll the different requests. This is actually a common problem not only for threaded applications that use asynchronous MPI calls, but also to ensure the progression of communication within MPI implementations. The PIOMan library was for instance specifically designed to address this problem in the NewMadeleine communication engine [32]. PIOMan automatically selects the most suitable strategy to ensure that all communications progress and to detect their termination with an optimal reactivity [184]. For the sake of simplicity, the current solution adopted by StarPU consists in polling the MPI library continuously in order to ensure a good reactivity: considering that the number of CPU cores keeps growing, wasting a little amount of processing power is a sensible trade-off anyway.

Even though there are some efforts to introduce them [92], the MPI standard still does not provide support for asynchronous collective communications. Such an extension of the MPI standard has however already been successfully implemented in the OpenMPI in the context of the DPLASMA project to improve the scalability of dense linear algebra kernels [25]. Our MPI-like library only provides a limited support for collective communications because it relies on the features available in the current MPI standard. Broadcast is not supported, and the scatter and gather operations are simply implemented by the means of multiple independent asynchronous MPI transfers.

5.4 Mapping DAGs of tasks on clusters

One of the motivations for adopting a task-based model is that describing an application as a graph of tasks is generic enough to allow a runtime system to map the graph on a variety of platforms. Task graphs are indeed a convenient and portable representation which is not only suited to hybrid accelerator-based machines, but also to clusters of nodes enhanced with accelerators.

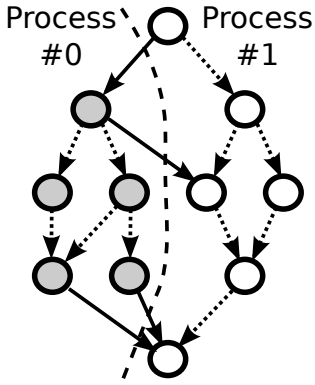


Figure 5.3: Example of task DAG divided in two processes: MPI communications are shown in full arrays, intra-node data dependencies are shown in dotted arrays.

5.4.1 A systematic methodology to map DAGs of tasks on clusters

Figure 5.3 illustrates how to map a DAG of tasks on a two-node cluster. An instance of StarPU is launched in each MPI process. The first step consists in partitioning the graph into multiple subgraphs which correspond to the tasks that are submitted to the different instances of StarPU. Assuming data-driven dependencies, dependencies that cross the boundary between two-processes are enforced by replacing the dependency by a data transfer that is performed by the means of our MPI-like library. A node that generates a piece of data required by its neighbour(s) issues detached send calls. Similarly, a node that needs a piece of data that was generated on another node posts a detached receive request to its local instance of StarPU. Intra-node dependencies are directly managed by the instance of StarPU that schedules tasks on the corresponding MPI node. Provided an initial partitioning of the DAG, this very simple methodology therefore demonstrates that our task-based paradigm is also suited for clusters of multicore nodes enhanced with accelerators.

5.4.2 The starpu_mpi_insert_task helper

Combined with implicit data-driven dependencies, the `starpu_insert_task` helper that we have presented in Section 2.2.5 is a convenient way to submit tasks using a paradigm that looks like a sequential code with function calls. Given the methodology we have just described, we can extend this approach to provide a similar semantic in a distributed environment.

The first step of our method was to partition the DAG into multiple parts. Our approach to facilitate this step is to select an *owner* for each data registered to StarPU: a task that modifies this handle must then be executed on its owner. Task partitioning therefore directly results from an initial data mapping. Interestingly, this approach is commonly used to dispatch computation in a distributed environment such as MPI. A classic strategy to implement dense linear algebra kernel over MPI (*e.g.* ScaLAPACK [23]) for instance often consists in determining an initial data partitioning (*e.g.* 2D cyclic mapping of the blocks). Sequential algorithm initially designed for shared memory (*e.g.* LAPACK [7]) are then ported to a distributed environment by following this data mapping, and exchanging the various data contributions produced by the different MPI nodes during the algorithm.

We have seen that, in order to execute a DAG using the `starpu_insert_task` function, the application only has to insert a task for each node in the DAG following an order that is sequentially valid. Actually, executing the same DAG on a cluster with the `starpu_mpi_insert_task` is similar. The application first has to assign an owner process to each data handle. All MPI processes then submit the entire DAG, but a task is actually executed only on the node that owns the data that it modifies. The other nodes only automatically call our MPI-like library to send a valid copy of the needed data, while the execution node automatically calls it to receive them. Since every MPI process unrolls the entire DAG, we actually keep track of which MPI node already holds a valid copy of the data, so that it is not necessary to send the same piece of data multiple times unless it has been modified by another process.

5.4.3 Example of a five-point stencil kernel automatically distributed over MPI

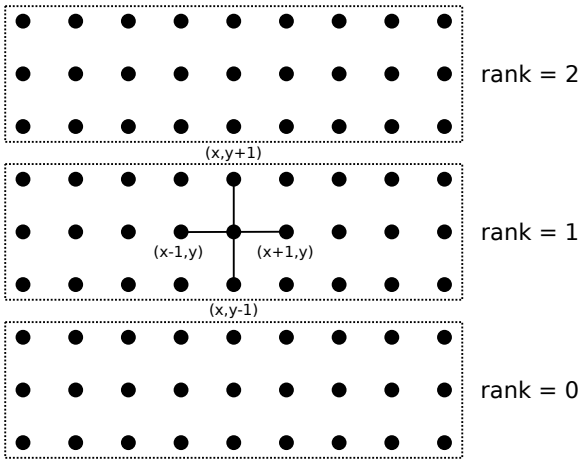


Figure 5.4: Five-point stencil kernel distributed over MPI. Each node of the grid is updated accordingly to the values of its four neighbours. The grid is divided into multiple blocks of equal size to optimize load-balancing and minimize the amount of data transfers.

Five-point stencil kernels are often used to approximate derivatives in finite differences schemes. As illustrated on Figure 5.4, they consist in iterating over each node of a 2D grid and update it according to the values of its four neighbours. A good load balancing is obtained by dividing the entire grid into multiple parts of the same size. The actual challenge of this data-intensive kernel is however to ensure that data transfers are performed efficiently to avoid wasting time on the numerous data dependencies. Partitioning the grid into multiple blocks also minimizes the size of the interface between the MPI processes, and thus the amount of data transfers between each iteration.

Figure 5.5 gives a very simple example of a five-point stencil kernel distributed over MPI using StarPU. The first step consists in registering the different entries of the 2D grid (lines 4 to 12). It must be noted that this code must be executed by each MPI process, so that all entries are registered in every instance of StarPU. The `starpu_data_set_rank` function specifies which MPI process owns a data handle, and therefore executes any task that modifies this matrix entry (line 11). In this case, the 2D grid is divided into blocks along the y-axis. The second step is to asynchronously submit all the tasks that constitute the DAG, following an order that would be a

```
1  unsigned matrix[X][Y];
2  starpu_data_handle data_handles[X][Y];
3
4  for (int x = 0; x < X; x++)
5  for (int y = 0; y < Y; y++) {
6    /* Register the matrix entry to StarPU */
7    starpu_variable_data_register(&data_handles[x][y], 0, &(matrix[x][y]), sizeof(unsigned));
8
9    /* Define which MPI process is the owner of this handle */
10   int data_rank = y / ((Y + size - 1)/size);
11   starpu_data_set_rank(data_handles[x][y], data_rank);
12 }
13
14 for (int loop = 0 ; loop < niter; loop++)
15 for (int x = 1; x < X-1; x++)
16 for (int y = 1; y < Y-1; y++) {
17    starpu_mpi_insert_task(MPI_COMM_WORLD, &stencil5_cl, STARPU_RW, data_handles[x][y],
18                STARPU_R, data_handles[x-1][y], STARPU_R, data_handles[x+1][y],
19                STARPU_R, data_handles[x][y-1], STARPU_R, data_handles[x][y+1], 0);
20 }
21
22 starpu_task_wait_for_all();
```

Figure 5.5: Implementation of a five-point stencil kernel over MPI.

valid sequential execution (lines 14 to 20). On line 17, the first argument specifies the MPI communicator in which transfers must be done. The second argument gives the codelet associated to the task: in this case, `&stencil5_cl` is a five-stencil kernel that updates the current matrix entry (`data_handles[x][y]`) according to the values of its four neighbours. The trailing null argument indicates that there is no more arguments for this helper.Finally, the barrier on line 22 ensures that all tasks have been executed within the local MPI process. It should be noted that the entire DAG is unrolled regardless of the underlying data mapping. Such a separation of concerns between having a suitable data mapping and describing the application as a graph of tasks therefore enhances both productivity and portability.

5.4.4 Implementation overview

Each MPI process unrolls the entire DAG described by the means of task insertion facility. Since all tasks are visible to each MPI process, and that the order of task submission is the same for all processes, there is no need for control messages between the different MPI nodes (which behave in a deterministic way). When a task modifies a piece of data managed by the local MPI node, task insertion results in the submission of an actual StarPU task in the local MPI process. On the one hand, if the data owner detects that another MPI process needs a valid data replicate, the data owner issues an asynchronous MPI send operation. On the other hand, the MPI process which actually needs a piece of data which is not managed by the local process issues an MPI receive operation before executing tasks that access this data replicate.

Each process keeps track of the validity of its local data replicates. The MPI process which is the owner of a piece of data is always guaranteed to have a valid copy. Other processes receive a copy of the data replicate from the owner. In order to avoid transferring the same piece of data multiple times, data replicates are kept as valid. Since every MPI process unrolls the entire task DAG, it is possible to perform replicate invalidation in a deterministic fashion without exchanging control

messages. After receiving a valid copy from the data owner, this cache is indeed considered valid until the data owner modifies this piece of data.

There are still corner cases that need to be solved because the mapping of the tasks is directly derived from that of the data. A specific decision must for instance be taken for tasks modifying several pieces of data that are owned by different processes. A choice must also be made for a task that does not modify any data (*i.e.* which only has side effects). In such cases, we can break the tie by adding a few additional rules. Tasks modifying multiple data can be assigned to the owner of the first data that is modified, and results are immediately sent back to the owners of other data. Tasks which do not modify data can for instance be assigned in a deterministic fashion by the means of a round robin policy. An application can also explicitly require that a task should be scheduled by a specific MPI process.

5.4.5 Scalability concerns and future improvements

"MPI-zing" StarPU applications that were written using the `starpu_insert_task` function is therefore straightforward. Registering all data and having the entire DAG unrolled by each MPI process however introduces a potential scalability pitfall. We first need to ensure that the overhead is very limited when dealing with tasks that are not executed locally and which do not introduce any data transfer. Expert programmers can also be helpful by pruning parts of the DAG by hand. In case the programmer can determine the exact subset of data handle that will be accessed by a MPI node, there is no need to unroll the parts of the DAG that process other data handles. Such pruning is also sometimes doable from high-level tools that can take advantage of a phase of static analysis.

This approach can still be improved in many ways. The reduction access mode presented in the previous chapter can be supported by creating a local temporary data handle on each MPI node, and to perform a reduction of the different temporary handles on top of MPI. In order to deal with irregular workloads, it should be possible to dynamically reassign a handle to a different owner.

In some situations, it is even possible to automatically detect the optimal data mapping. In particular, COSNARD *et al.* have proposed algorithms which use a static analysis phase to automatically provide an efficient data mapping for parameterized task graphs. [46, 47]. The same techniques have also been used in the DPLASMA project in the context of accelerator-based computing. Unlike StarPU, DPLASMA therefore requires that algorithms should be expressed in a new language which makes it possible to describe parameterized task graphs [25].

5.5 Discussion

Even though StarPU is designed to manage computation and data transfers within a compute node, real HPC applications usually run on clusters of machines potentially enhanced with accelerators. In order to address real-life HPC applications, adding support for MPI is therefore crucial. Some upcoming manycore architectures also tackle scalability issues by mixing MPI and other programming paradigms (*e.g.* TBB, OpenMP, etc.). For example, an instance of the OS should run on each core of the Intel SCC chip, and the different cores should communicate using a message passing paradigm. Having a proper support for message passing within StarPU makes it possible to exploit these large manycore processors. In this chapter, we have shown that such support can

take various forms: StarPU can either be used to accelerate legacy MPI codes by offloading part of computation on the accelerators available locally; or tasks can be used as first-class citizen by mapping DAGs of interdependent tasks and the corresponding data on the different MPI nodes.

In order to accelerate legacy MPI codes without having to rewrite the entire application using a non-standard programming model, we have implemented a MPI compatibility layer to exchange data between different instances of StarPU. Latency and bus contention being major concerns on a cluster, we need specific capabilities such as direct transfers between GPUs and NICs, and better integration of low-level toolkits within MPI. Besides, we need to improve StarPU's scheduling engine to consider network activity. Similarly to the techniques shown in Chapter 3 to reduce intra-node data contention, we could model the performance of the network to predict when data should be available on the different MPI nodes.

We have also shown that the task paradigm is flexible enough to deal with many levels of hierarchy, including multiple MPI nodes. As a result, StarPU tasks can be used as first-class citizen throughout a cluster even though each MPI node runs a separate instance of StarPU. This permits to extend existing StarPU applications so that they exploit clusters of machines enhanced with accelerators. Even though modifying large MPI codes remains a real concern, StarPU can be used transparently within parallel libraries or higher-level programming environments which can be rewritten using a task-based paradigm internally. In order to cope with the asynchronous nature of task parallelism, the MPI standard needs to evolve to fit applications that are not written in a synchronous SPMD style. For example, we would need asynchronous collective operations (*e.g.* a non-blocking scatter/gather MPI call) to efficiently dispatch the output of a task to all MPI nodes which may all need it at a different time.

Mapping data and/or tasks over a cluster enhanced with accelerators is a significant research issue. This can be achieved statically thanks to a tight collaboration with compiling environments for regular enough algorithms. Dynamic load balancing over a cluster is a delicate issue. Similarly to out-of-core algorithms, we for instance need to ensure that the memory footprint of the application is not too high when migrating tasks which access too many different pieces of data on the same MPI node.

Chapter 6

Debugging and Performance analysis tools

Chapter Abstract

This chapter gives an overview of some of the performance debugging facilities available in StarPU. Having performance analysis tools is critical to ensure that applications perform as expected and to detect algorithmic issues on such complex architectures. We first describe StarPU's tracing facilities which for example permit to generate either offline or on-the-fly Gantt diagram of the activity of/between the different processing units. Then we present the performance counters exposed by StarPU, for instance to properly analyze the behaviour of the different kernels or the activity on the I/O bus. After illustrating how these performance debugging tools were actually useful to improve a cosmological data analysis benchmark, we explain how StarPU is capable of automatically providing the application with a theoretical execution time upper-bound estimation, which makes it possible to quickly detect whether an algorithm is parallelized efficiently over the machine or not.

6.1 Performance analysis tools

Modern platforms have become so complex that analyzing the actual behaviour of a parallel application to ensure that there is no performance issue is a delicate problem. Performance analysis

tools are therefore required to understand and improve the performance obtained on such machines. They can be used to detect code hot-spots, which are the kernels which should be further optimized, and whether further algorithmic improvements are still required (*e.g.* to generate more parallelism or to improve data locality).

6.1.1 Offline tools

The internal events occurring in StarPU (*e.g.* data transfer, kernel execution, etc.) during the execution of a StarPU-enabled application can be recorded transparently using the low-overhead FxT library [50].

The resulting execution trace contains a lot of useful information which are typically well summarized with a Gantt diagram showing the overall activity within StarPU over the time. The application can therefore automatically generate a Gantt diagram described in the Pajé trace format [110] which can be visualized with the Vite open-source trace visualizer [48]. The usefulness of this tool is illustrated on a real use-case in Section 6.3.

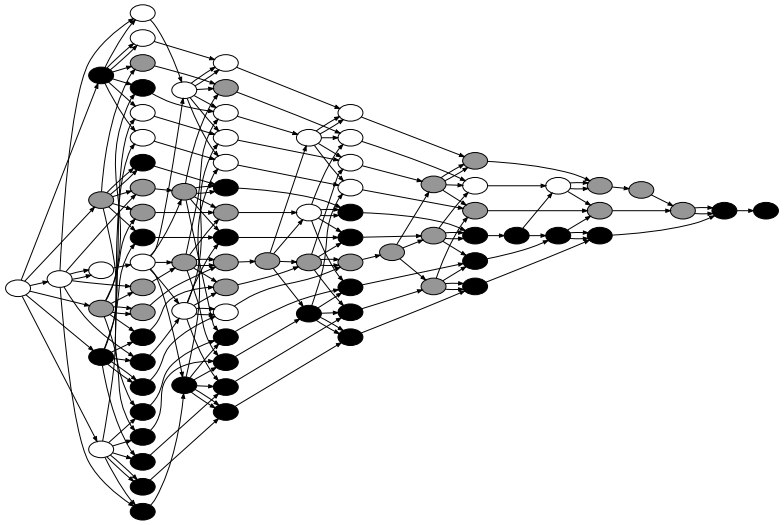


Figure 6.1: DAG obtained after the execution of a Cholesky decomposition on a machine with 3 GPUs. The different colours indicate which GPU has processed the task.

Task dependencies are also recorded by FxT during the execution of the application. As shown on Figure 6.1, this permits to display the graph of tasks that were actually executed. The task graph is generated by the means of the graphviz [60] library. The overall aspect of the graph gives a quick overview of the amount of parallelism available during the different phases of the algorithm. For example, the beginning and the end of the algorithm depicted on Figure 6.1 may not feature enough parallelism. In addition to the edges which indicate task dependencies, the colour of the vertices can be meaningful. In our example, the colours of the vertices for instance show which processing unit has executed a task. This permits to visualize data locality because neighbouring vertices tend to share the same colour, which means that tightly coupled tasks are executed within the same processing unit.

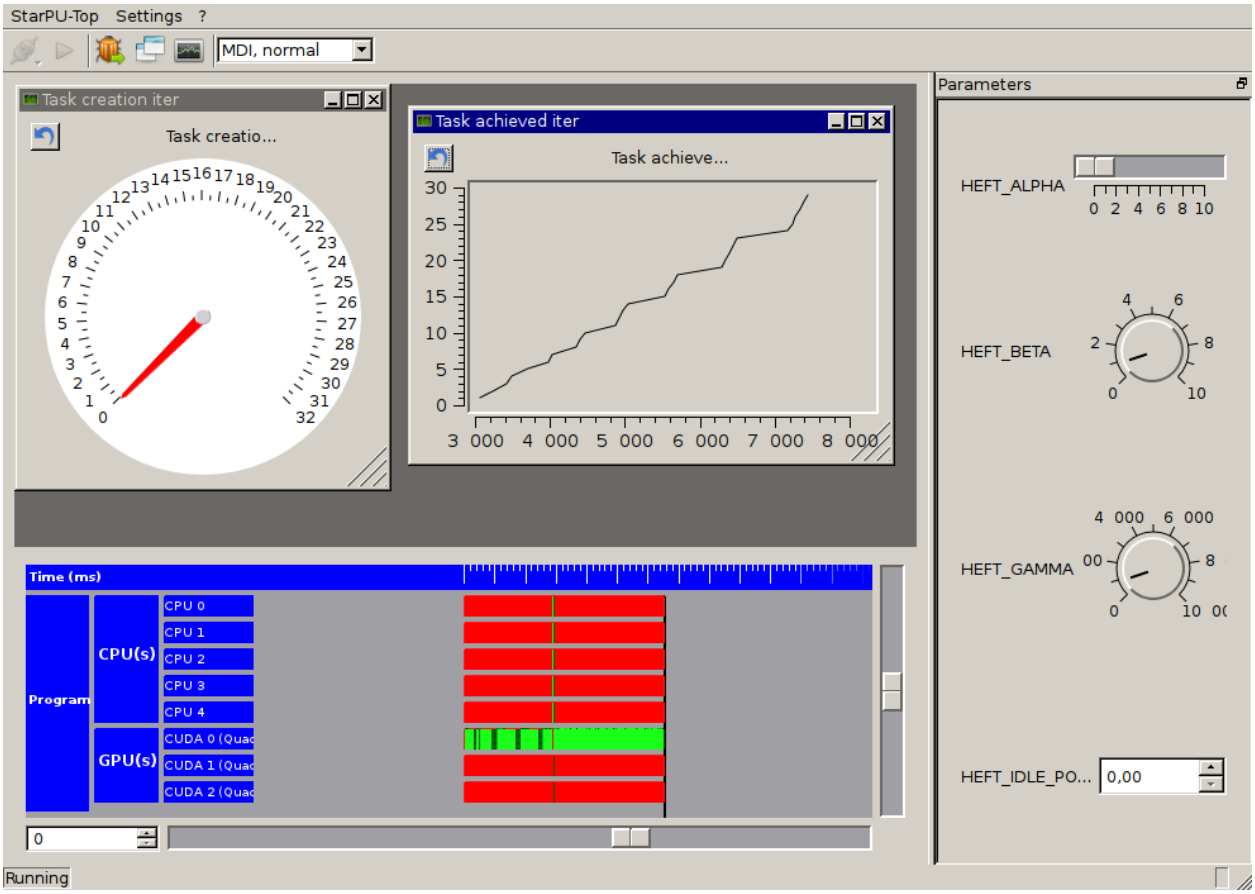


Figure 6.2: StarPU-Top controlling interface.

6.1.2 Online tools

Post-mortem performance analysis tools are not practical for real HPC applications that may run for days (or even for months). Besides the fact that waiting for days before detecting a performance issue is unpleasant, the total volume of traces generated during the execution becomes gigantic. We have therefore enhanced StarPU with steering capabilities. StarPU-Top[1], shown in Figure 6.2, is an online tool that dynamically generates Gantt diagrams. StarPU-Top not only allows to remotely monitor the activity of an instance of StarPU; it also permits to dynamically modify parameters within the application, or within StarPU itself. If poor data locality is observed, it is for instance possible to give more impact to the data overhead prediction used by the HEFT scheduling strategy introduced in Section 3.4.3.

6.2 Performance counters

StarPU also provides programmers with performance counters that are directly accessible to the applications. Various types of counters are available. It is possible to monitor the activity on the

[1]StarPU-Top was implemented by William BRAIK, Yann COURTOIS, Jean-Marie COUTEYEN and Anthony ROY.

bus (*e.g.* average bandwidth consumption), per-device information (*e.g.* predicted energy consumption, ratio of idle time, etc.) or per-task counters (*e.g.* kernel duration, scheduling overhead, etc.).

Per-task performance counters typically provide a convenient mechanism to easily measure the speed of a kernel used by an application. Performance feedback is also useful to higher-level environments relying on StarPU. The SkePU skeleton-based library [61, 52] (see Section 8.4.3) for example selects the most efficient algorithm within the different variants of the same kernel, accordingly to the performance reported by StarPU.

Observing the average scheduling overhead combined with the ratio of idle time of the different processing units indicates whether the application submits tasks with a suitable granularity. Indeed, a high scheduling overhead appears when tasks are too small, and a limited amount of parallelism may lead to idle processing units. The scheduling overhead measurements are also useful to evaluate the scalability of a scheduling strategy, either when selecting the most appropriate strategy, or simply when designing a strategy.

6.3 Case Study: Optimizing the TPACF cosmological data analysis benchmark

In this section, we give a concrete example of performance issue that was easily resolved by analyzing the output of the performance feedback tools. We here consider the TPACF (Two-Point Angular Correlation Function) benchmark from the PARBOIL benchmark suite [98]. TPACF is used to statistically analyze the spatial distribution of observed astronomical bodies. The algorithm computes a distance between all pairs of input, and generates a histogram summary of the observed distances. Besides its CPU implementation, this reference benchmark is available on both GPUs [166] and FPGAs [112]. In its original version, the TPACF benchmark relies on MPI to use multiple GPUs. Each MPI process is statically assigned a subset of the input files. The algorithm implemented by each MPI process consists in computing the correlation between each pair of files: once the first input file has been loaded from the disk, each of the files assigned to the MPI process are sequentially loaded from the disk, divided in multiple subsets which are all compared with the first input file, and the distance histogram associated to the first input file is updated when all subsets have been processed.

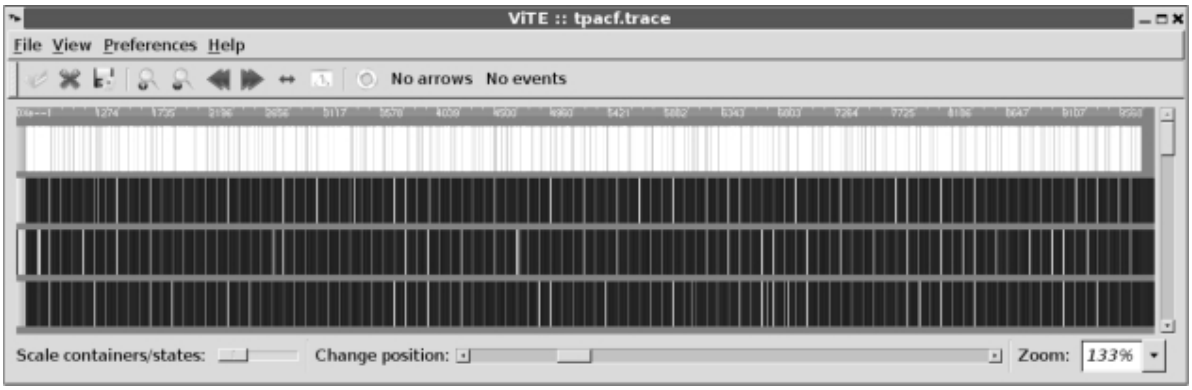
When using StarPU, there is no need for MPI within a single-node machine. We initialize a single instance of StarPU per node, and we used StarPU tasks instead of directly invoking CUDA kernels to perform the comparison of a pair of input files. For each pair of input files, the StarPU-enabled version of the TPACF benchmark is therefore able to dynamically dispatch the processing of the different subsets of the second input file on the different GPUs.

Table 6.1: Performance of the TPACF benchmark on HANNIBAL.

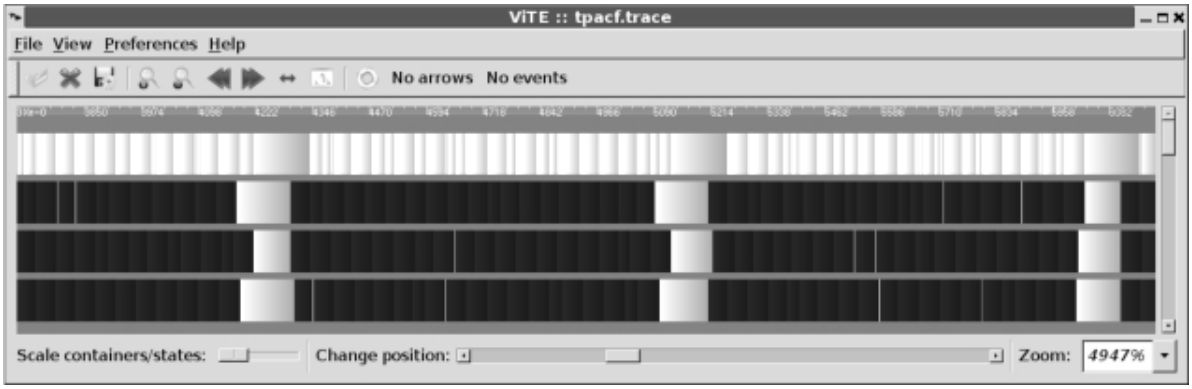
Version	#process	#GPUs/process	Processing Time	Overhead
Original	1	3	85.4 seconds	
StarPU	3	1	88.4 seconds	+3.5%
StarPU	1	3	91.5 seconds	+7.2%

Table 6.1 shows the performance obtained for the TPACF benchmark on a machine equipped with 3 GPUs. The original version of TPACF takes 85.4 seconds on a dataset containing 100 files

of 97179 bodies. In the case of the second line, 3 MPI processes are created, but CUDA kernel invocations are replaced by StarPU tasks which ultimately perform these kernel invocations. As one could expect, StarPU introduces a small overhead in this case. The last line shows the case of a single instance of StarPU controlling the 3 GPUs of the machine. Instead of improving performance thanks to a better load balancing, we observe an even larger overhead of 7.2% compared to the original MPI version.



(a) Entire execution.



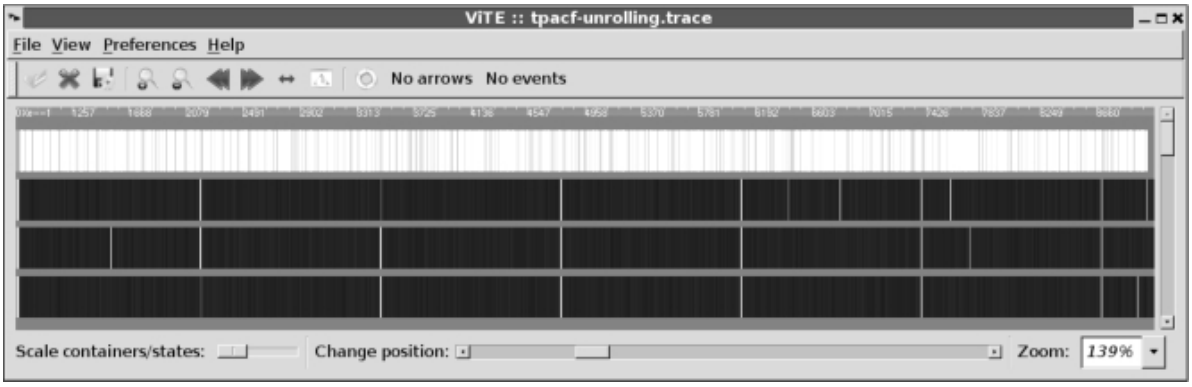
(b) Zoom on a few iterations.

Figure 6.3: Vite Trace obtained with a naive port of the TPACF benchmark on StarPU.

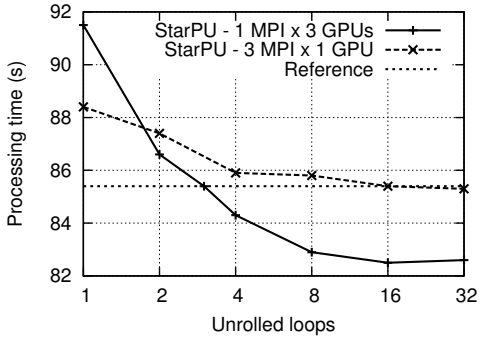
Figure 6.3 shows the trace produced during the execution of the TPACF benchmark when a single instance of StarPU controls the 3 GPUs. The first line of this Gantt diagram describes the status of the only CPU core used during this experiments. Since CPUs are only used to update the histograms, this CPU core is almost always idle: it appears in white on the diagram. The three bottom lines correspond to the status of the three different GPUs: the black colour indicates that they are almost always busy. Figure 6.3(b) shows a detailed diagram obtained by zooming in the trace shown on Figure 6.3(a). It appears that the GPUs are regularly idle at the same time, which suggests that there is a bottleneck in the algorithm.

It quickly appears that this corresponds to the preprocessing of the input file read from the disk. When a single process is used, there is indeed a single application thread that is in charge of preprocessing the input files for the entire machine and submitting StarPU tasks. In the original MPI-enabled version, this sequential section is relatively smaller because each MPI process

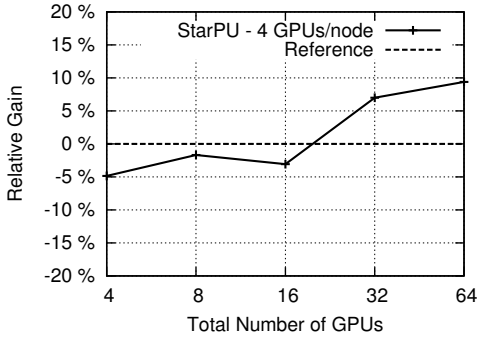
concurrently reads the different input files. According to AMDHAL's law, one can indeed expect a limited scalability in case the sequential portion of the code is non negligible. Contrary to the original version that is completely synchronous, task submission can be done in an asynchronous way in StarPU. We can therefore fix the performance issue by processing multiple files at the same time within the same process in StarPU which performs the preprocessing on on a CPU core in the background.



(a) Gantt diagram visualized with Vite.



(b) Processing time on a single machine (HANNIBAL).



(c) Gain on the AC cluster.

Figure 6.4: Impact of loop unrolling on the TPACF benchmark.

The benefits of this unrolling technique indeed appears on Figure 6.4(a) on which the amount of idle times has significantly dropped on GPUs. While GPUs were only active 86.5% of the time on Figure 6.3, an average of 97.8% of activity is measured on Figure 6.4(a), according to the performance counters described in Section 6.2. This has a direct impact on the performance depicted on Figures 6.4(b) and 6.4(c) which respectively show the impact of loop unrolling on the HANNIBAL machine and on multiple nodes of the AC cluster. Unrolling more than 3 loops permits to have at least 3 files processed at the same time, which is equivalent to behaviour of the reference implementation where the 3 MPI processes all read their input file in parallel. In the end, the use of StarPU combined with a proper loop unrolling to avoid the side-effects of AMDHAL's law permits to outperform the reference implementation. In addition to its dynamic flexible load balancing capabilities, StarPU takes advantage of loop unrolling to preprocess multiple files on CPUs in the background. This example is an illustration of the necessity of performance analysis facilities to

understand the actual behaviour of the applications that run on such complex machines.

6.4 Automatically Predicting theoretical execution time upper-bounds

Performance analysis tools are very useful to understand the performance of an application. Even though it is important to determine whether or not it is worth further optimizing a piece of code, using execution traces to determine if an algorithm is sufficiently optimized requires a significant expertise. In this section, we show how StarPU uses Linear Programming techniques to automatically determine how close we are from the optimal execution of an algorithm without requiring any application-specific knowledge.

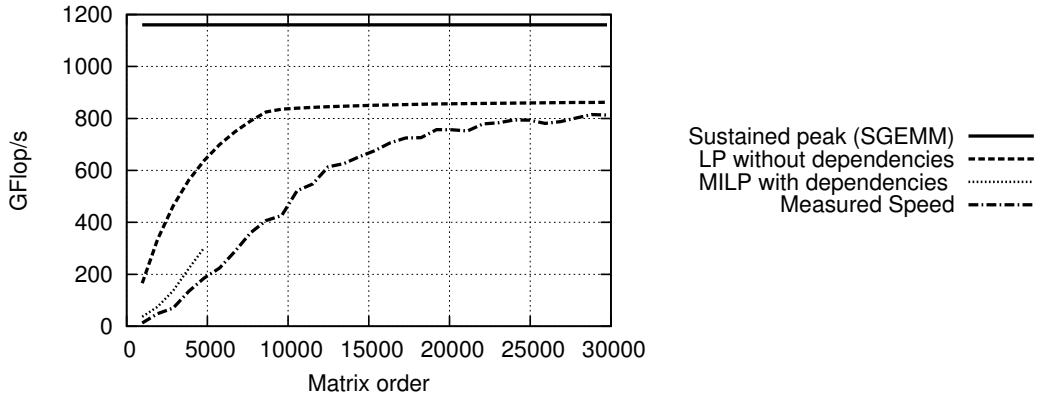


Figure 6.5: Comparison between the actual performance and the theoretical boundaries for a QR decomposition on HANNIBAL.

The speed of an algorithm is often compared with the speed of the fastest of its kernels. The speed of a matrix factorization (*e.g.* Cholesky, LU or QR) is for instance usually compared to the sustained speed of the matrix multiplication kernel on the different types of processing unit. A Cholesky decomposition is typically considered to be efficient when the speed of the overall algorithm reaches about 85 % of this sustained peak performance. Figure 6.5 shows the performance obtained with a single precision QR decomposition algorithm on the HANNIBAL machine (more details on this benchmark are given in Section 7.3). The sustained peak performance of SGEMM is 1160 GFlop/s on this platform. The asymptotic speed measured on the QR decomposition however reaches a ceiling around 800 GFlop/s. This suggests that the algorithm only obtains about 70 % of the optimal speed for large problems, and much less for smaller matrices.

Comparing measured speed with the speed of the fastest kernel however has various drawbacks. First, it requires some application specific knowledge because we need to know which is the fastest kernel in an arbitrary algorithm. This is also a very coarse grain estimation because most applications are composed of different types of kernels with varying GPU/CPU relative performance. In the case of a QR decomposition, the factorization of diagonal blocks for instance performs much slower than a matrix multiplication. Considering that the ratio of operations that are matrix multiplication get lower when the problem size decreases, this explains why it is not possible to reach the sustained peak performance, especially for small problems.

To properly and easily estimate a better upper bound of the execution time of an algorithm,

we have added to StarPU an optional mode that records the number of executed tasks according to their type (the kernel they run and their operand size). This can then be combined with the respective performances of the kernels, which StarPU can provide for each type of processing unit thanks to performance models. Estimating an upper bound for the execution time resorts to solving the corresponding Linear Programming problem given by Equation 6.1. It must be noted that the correctness of this technique relies on the accuracy of the performance models.

$$\min_{t_{max}} \left\{ \left(\forall w \in W, \sum_{t \in T} n_{t,w} t_{t,w} \leq t_{max} \right) \wedge \left(\forall t \in T, \sum_{w \in W} n_{t,w} = n_t \right) \right\} \tag{6.1}$$

Here, W is the set of workers (*i.e.* CPUs, GPUs, etc.), T is the set of the various task types, $n_{t,w}$ is the number of tasks of type t performed by worker w, $t_{t,w}$ is the duration estimation of a task of type t on worker w, and n_t is the number of tasks of type t encountered during the execution. Equation 6.1 expresses that each worker should have finished its assigned tasks before the total execution time, and each type of tasks was distributed over the workers.

StarPU was extended to optionally compute this bound so it can be easily printed along with other timing information. Internally, this is performed by relaxing the initial linear programming problem, since the difference with integer resolution is negligible for non-tiny sizes. This provides the second curve of the figure, which shows a much better upper bound since it is optimal according to the heterogeneity of both task types and workers. Here, solving the linear programming problem actually means optimizing the distribution of tasks on workers according to their respective performance.

```
1 starpu_bound_start(deps, prio);
2 (...) /* StarPU code to be profiled */
3 starpu_task_wait_for_all();
4 starpu_bound_stop();
5
6 double min; /* Theoretical Minimum Time */
7 starpu_bound_compute(&min, NULL, 0);
```

Figure 6.6: Comparing the actual execution time with the theoretical bound.

From the point of view of the application, the programmer just has to insert function calls to start and stop recording information about the current execution (lines 1 and 4 on Figure 6.6). Once the various tasks have been executed, the application can request what would have been the minimum execution time of the portion of code that was profiled (line 7). By comparing this limit to the wall clock time, the application seamlessly gets an estimation of the ratio between actual and optimal performance.

The bound described on Equation 6.1 only expresses that there is a sufficient amount of time to execute all tasks independently on a heterogeneous platform. To get an even more precise upper bound, we need to take task dependencies into account. We thus use a Mixed-Integer Linear Programming (MILP) problem, in which we distinguish all tasks independently to integrate dependency constraints. We introduce $a_{t,w}$ binary variables which express whether a given task t is run on worker w. The problem then expresses that for each task exactly one worker executes it, a task can begin only after all its dependencies have finished, and no two tasks can be executed by the same worker at the same time. We have extended StarPU to optionally emit such a problem automatically from the actual execution of any application. The application can select whether to

enable taking such dependencies into account. For a 5x5-blocked QR decomposition, *i.e.* 60 tasks, this typically results in about 1400 variables (of which 1250 are binaries) and 13000 constraints (most of which are Mixed-Integer). This is the biggest size for which the resolution is possible in a reasonable amount of time. The third curve shows that thanks to taking dependencies into account, this provides a much better upper bound, as dependencies are what typically reduce parallelism and possibilities for optimizations. Even if few values are available, this already provides a good indication that the performance obtained by our scheduling optimizations is already very close to the optimum that could be achieved with linear programming. This bound could be further refined by also modeling data transfers, but this would produce yet more variables and constraints while most transfers are actually overlapped with computation.

6.5 Discussion

Machines have become so complex that performance analysis tools are crucial to understand and improve the performance of parallel algorithms. An environment without debugging capabilities is not usable in real situations. Performance bugs are usually much harder to fix than usual bugs, especially because the programmer is neither supposed to be an expert of parallel programming nor an expert of the underlying architecture. The example of the TPACF benchmark illustrates how easy it becomes, with proper tools, to quickly realize that there is a performance issue, and what can be the origin of this issue. Doing so by hand is always possible, but requires a significant expertise.

Performance feedback capabilities also allow users and higher-level tools to detect kernel inefficiencies, or to select the most appropriate implementation when several of them are available. Feedback also makes it possible to determine the hot spots in the application. As a result, programmers need not spend too much time optimizing kernels with a little impact on the overall performance. Instead, being able to detect that a specific kernel takes too long or creates a bottleneck in the parallel algorithm is a crucial information to improve the algorithm. Having an automatic tool which gives an estimation of the optimal execution time is also very convenient to determine whether a parallel algorithm is suitable or not, or if the only way to further improve performance would be to improve the speed of the kernels.

In order to provide a meaningful support for higher-level software layers and end-user programmers, it is therefore almost as important to be able to report how the algorithm was executed as to actually execute it efficiently.

Part II

Evaluation

Chapter 7

Experimental Validation

Chapter Abstract

This chapter gives an experimental validation of the model implemented by StarPU. We first list the different machines that were used to carry out the experiments shown in this document. Microbenchmarks giving an estimation of task management and task scheduling overheads are shown. We then study the impact of the different scheduling optimizations implemented in StarPU on a QR decomposition. We also illustrate the programmability improvements enabled by StarPU by evaluating an implementation of a hybrid Communication-Avoiding QR decomposition which is often considered as a relatively complex algorithm. A computation-intensive Cholesky decomposition and a memory-bound stencil kernel are used to evaluate the efficiency of StarPU within a cluster environment. We illustrate the benefits of providing relaxed coherency data access modes by showing the strong

scalability of a Monte Carlo kernel based on data reductions. Finally, we study the performance of a Computational Fluid Dynamic (CFD) kernel that takes advantage of parallel tasks to fully exploit manycore platforms and hybrid accelerator-based machines.

7.1 Experimental platforms

Table 7.1: List of experimental setups.

Machine	#cores	Freq. (GHz)	Memory (GB)	SP/DP peak (Gflop/s)	Power (Watts)
ATTILA					
Intel Nehalem X5650 CPU	2×6	2.67	48 (2×24)	$2 \times (153.6/76.8)$	2×95
NVIDIA Fermi C2050 GPU	3×448	1.15	9 (3×3)	$3 \times (1030/515)$	3×238
BARRACUDA					
Intel Xeon E5410 CPU	1×4	2.33	4 (1×24)	37.28/18.64	80
NVIDIA Quadro FX4600 GPU	1×96	0.5	768 MB	518 / no double	134
NVIDIA Quadro FX5800 GPU	1×240	1.3	4 (1×4)	622/78	189
BERTHA					
Intel Xeon X7460 CPU	16×6	2.67	192 (4×48)	$16 \times (55.92/27.96)$	6×130
HANNIBAL					
Intel Nehalem X5550 CPU	2×4	2.67	48 (2×24)	$2 \times (85.4/42.6)$	2×95
NVIDIA Quadro FX5800 GPU	3×240	1.3	12 (3×4)	$3 \times (622/78)$	3×189
MORDOR					
AMD Opteron 8358 SE CPU	4×4	2.4	32 (4×8)	$4 \times (76.8/38.4)$	4×105
NVIDIA Tesla S1070 GPU	4×240	1.3	16 (4×4)	$4 \times (690/86)$	4×188
PS3					
Cell B/E's PPE	1	3.2	256 MB	25.6/6.4	80
Cell B/E's SPE	6	3.2	256 KB	$6 \times (25.6/1.8)$	
AC cluster ($\times 32$ nodes with an Infiniband QDR 40 Gb/s network)					
AMD Opteron 2216 CPU	2×2	2.4	8 (4×2)	$2 \times (19.2/9.6)$	2×95
NVIDIA Tesla S1070 GPU	4×240	1.3	16 (4×4)	$4 \times (690/86)$	4×188
PLAFRIM cluster ($\times 8$ nodes with an Infiniband QDR 40 Gb/s network)					
Intel Nehalem X5650 CPU	2×6	2.67	36 (2×18)	$2 \times (153.6/76.8)$	2×95
NVIDIA Fermi M2070 GPU	3×448	1.15	18 (3×6)	$3 \times (1030/515)$	3×225

Table 7.1 gives an overview of the different machines which have been used to evaluate our model. It contains various types of machines which are intended to be representative of the different types of accelerator-based platforms supported by StarPU.

The first column gives the name of the different types of processing units in the machines, and the second column contains the number of processing cores of each type. The ATTILA machine for instance contains two hexacore CPUs and three NVIDIA FERMI GPUs. The last two columns respectively indicate the vendor estimation of peak performance (in single and double precision),

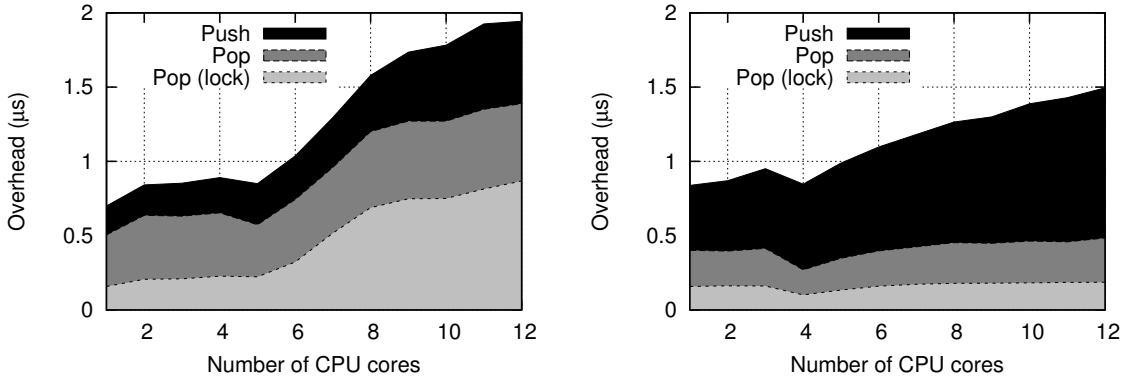
and the estimated power consumption.

The variety of accelerators (and their respective performance) gives an indication of the fast pace followed by constructors in the recent years. While the Cell processor used in the PS3 machine provides about 180 GFlop/s, the ATTILA machine which is only about four years more recent virtually delivers more than 3 TFlop/s. Over the years, the number of CPU cores has significantly increased. Besides a real gap in terms of programmability, accelerators have been improved to better support powerful features such as computing in double precision, asynchronous data transfers, or concurrent execution of multiple kernels. Power consumption has however become a real concern too: a single machine like ATTILA now requires almost one kilo-watt.

Apart from the PS3 machine which is based on the Cell B/E processor, and the BERTHA machine which is a manycore platform that does not feature any accelerator, all platforms are based on NVIDIA GPUs. This is mostly explained by the wide adoption of CUDA, which is by far the most widely used programming environment for accelerators. Since real HPC test-cases are frequently too large for a single machine, we also experiment with clusters of machines enhanced with accelerators (AC and PLAFRIM).

All the machines of table 7.1 run on LINUX, but we have successfully tested StarPU on WINDOWS and on MAC OS/X. Unless explicitly specified, experiments were carried on with CUDA 3.2 on all platforms based on NVIDIA GPUs. We have also used the CUBLAS 3.2, CUFFT 3.2, MKL 11.1 and FFTW 3 libraries.

7.2 Task scheduling overhead



(a) **greedy** strategy.

(b) **heft** strategy.

Figure 7.1: Task scheduling overhead on ATTILA.

Figure 7.1 shows the average scheduling overhead to execute a task. The experiment consists in launching 65536 tasks sleeping for $20\,\mu s$. The total overhead is divided in three distinct parts. The two upper parts give the average overhead of the push and the pop methods. The lower part indicates the average time required to take the mutex associated to a processing unit prior to calling the pop method. These numbers were directly obtained from the average values reported by the performance counters associated to tasks.

The **greedy** strategy simply implements a central queue where the push and the pop methods

respectively insert and retrieve tasks. This strategy obviously has a limited scalability because all processing units continuously compete for the lock protecting the shared queue. The regular increase of the time required to take the lock before calling the pop method on Figure 7.1(a) is a good indication of this contention.

The **heft** strategy is a *push-based* strategy, which means that all scheduling decisions are taken by the push method which eventually inserts each task directly in the local queue associated to the selected processing unit. The pop method simply consists in taking a task in this local queue, and to keep performance predictions up to date. The overhead of the pop method is almost constant because each processing unit simply locks its local queue, which is likely to be available except when there is a task currently being assigned to the worker. The push method cost is however not constant. This strategy first has to compute a performance prediction for the different types of processing units, which explains why on a single CPU core the push method takes more time than with the **greedy** strategy. The overhead of the push method also increases regularly with the number of processing units because the **heft** strategy has to compute the minimum termination time on each processing unit individually.

We could optimize the scalability of the various scheduling strategies. The cost of the push method implemented in the **heft** strategy could be drastically reduced by maintaining lists of *equivalent*[1] processing units sorted with respect to their available date. It would indeed make it possible to prune a significant number of processing units which have no chance of being eligible to schedule the task anyway. Another way to reduce the overhead of task scheduling is to use efficient data structures. AGHELMALEKI *et al.* have for instance extended StarPU with a scheduling strategy based on work-stealing mechanisms which relies on lock-free data structures [187]. Finally, the use of multiple scheduling domains organized in a hierarchical fashion is a possible approach to cope with strategies that do not scale efficiently. As discussed in Section 3.9, this however requires to design composable scheduling policies in order to properly distribute the load between the different scheduling domains.

In spite of their currently limited scalability, these raw scheduling overhead measurements must be considered with respect to the actual gain resulting from a good task scheduling. Even though an engineering effort would still be required to efficiently deal with a really small grain size, it is worth noting that this overhead does not exceed $2\,\mu s$ while any call to the CUDA runtime library typically takes at least a couple of micro-seconds [191].

7.3 QR decomposition

In this section, we explain how StarPU was used to combine the QR decomposition implementations of the PLASMA and MAGMA libraries to design a state-of-the-art hybrid QR decomposition.

After introducing PLASMA and MAGMA, we provide a detailed analysis of the impact of the different optimizations on the performance of the hybrid Tile-QR decomposition. We first improve the original Tile-QR algorithm so that it fits hybrid accelerator-based platforms well. Then, we analyze the impact of the scheduling policies, and we finally study the scalability of this algorithm with respect to the number of processing units to demonstrate that StarPU fully takes advantage of the heterogeneous nature of a hybrid platform.

[1] With the same architecture and attached to the same memory node.

7.3.1 The PLASMA and the MAGMA libraries

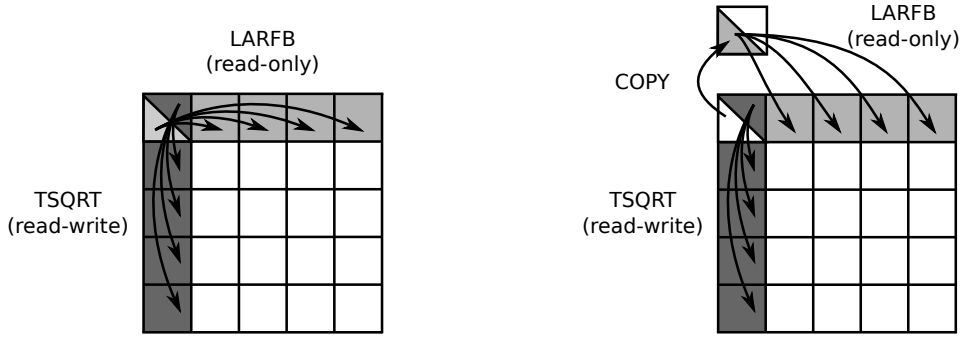
PLASMA [97] and MAGMA [96] are state-of-the-art implementations of the LAPACK library, respectively for multicore processors and for NVIDIA CUDA devices. In order to fully exploit modern parallel architectures with a sufficient amount of parallelism, they both implement dense linear algebra algorithms which are said to be *tiled* because the matrices are subdivided into small contiguous blocks called *tiles*.

PLASMA is based on the Quark dynamic task scheduler: our approach to mix PLASMA and MAGMA therefore consists in taking the tiled algorithms from PLASMA, and to generate tasks for StarPU instead of Quark. These StarPU tasks are naturally implemented by invoking PLASMA kernels on CPU cores, and MAGMA kernels on CUDA devices. In addition to that, the StarPU runtime system requires that the different tiles should be registered before they are used for computation. This simple methodology was used to design an hybrid implementation of the different one-sided factorizations available in PLASMA and MAGMA, which are Cholesky, QR and LU decompositions. Contrary to the MAGMA kernels which are limited to problems that can fit into the memory of a single GPU, StarPU can deal with arbitrarily large problems, provided each of the task can fit independently in the different GPUs. In spite of the architecture-specific optimizations within the kernels, we also had to ensure that the kernels of PLASMA and MAGMA produce exactly the same output, so that each task can be computed either on a CPU core or on a GPU device. Even though these three algorithms are rather similar, each of them contains specific challenges that must be addressed. Cholesky decomposition is the simplest of the three algorithms. As shown on Figure 3.8 on page 104, we improved the performance of the Cholesky algorithm by guiding the scheduler with priorities that indicate tasks which are on the critical path. An overview of the implementation of the Cholesky decomposition based on PLASMA and MAGMA kernels is given in Appendix A. During an LU decomposition, the pivoting step required for numerical reasons also impacts the amount of available parallelism. More details about the performance of this kernel are given in a previous study [AAD+11a]. The data accesses of the QR decomposition introduce even more complex issues to ensure that enough parallelism is available. A detailed analysis of the performance of the QR decomposition is given in the remaining of this section.

7.3.2 Improvement of the Tile-QR algorithm

The tile-QR algorithm of the PLASMA library was designed for shared-memory architectures and does not fit well on platforms with a distributed memory. Figure 7.2(a) indeed shows that in its original form, the tile-QR algorithm suffers from a severe loss of parallelism which results from the fact that all LARFB tasks try to access the lower-triangular part of the diagonal tile while the various TSQRT tasks modify the upper-triangular part of the same diagonal block. Data are managed at the tile level in this algorithm, so that it is not possible to have multiple tasks concurrently access the upper and the lower parts of this diagonal block. While all LARFB could be performed in parallel, they are serialized with TSQRT tasks which modify the upper-part of the block.

A solution to this problem could consist in registering independently the upper and the lower parts of the diagonal blocks (*e.g.* by the means of a data filter), but it would be highly inefficient to transfer such a non-contiguous piece of data with a non-constant striding between host memory and the accelerators. Another problem of having LARFB tasks and TSQRT access the same block concurrently is that the TSQRT kernel is likely to create false-sharing cache issues in the LARFB kernel. We therefore modified the tile-QR algorithm of the PLASMA library so that it is more



(a) Original Tile-QR Algorithm

(b) Tile-QR Algorithm with a Copy of Diagonal Blocks

Figure 7.2: Modification of the Tile-QR Algorithm to increase the amount of parallelism. LARFB and TSQRT kernels are serialized in the original Algorithm. Duplicating the diagonal blocks permits to execute LARFB and TSQRT kernels concurrently.

adapted to a distributed memory environment and to prevent such false-sharing cache effects. As shown on Figure 7.2(b), we added a task which copies the diagonal block into a temporary piece of data that is passed to the LARFB tasks which can thereby run concurrently. This removes the false-sharing issue because the TSQRT tasks only modify the upper-part of the original diagonal block, and not that of the temporary copy which is used by LARFB tasks.

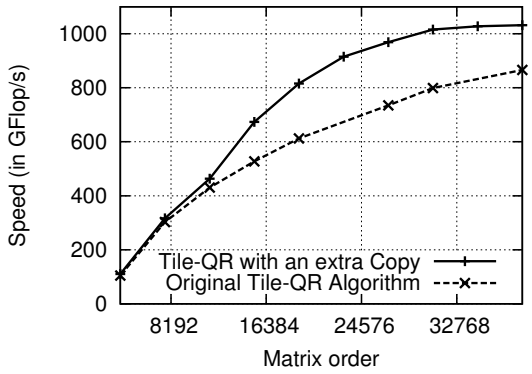
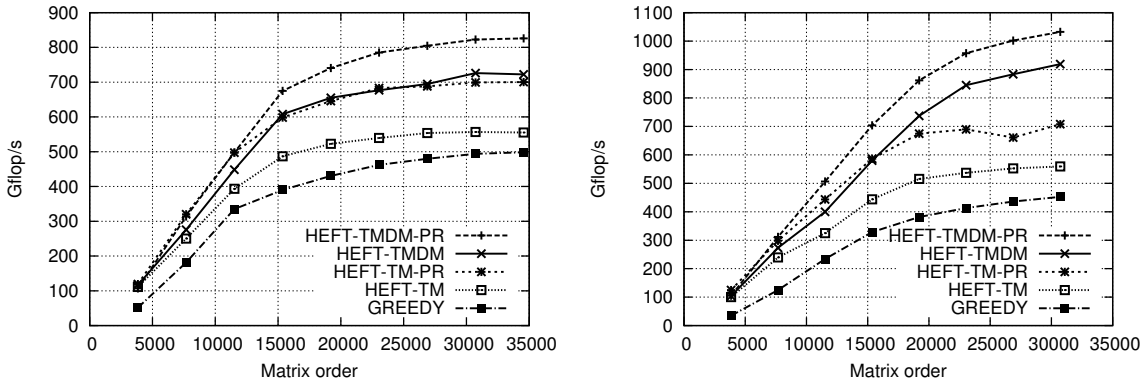


Figure 7.3: Duplicating the diagonal blocks to save parallelism on MORDOR.

The performance impact resulting from this optimization is given on Figure 7.3: on a machine equipped with four GPUs, we measure up to 30% of improvement compared to the speed of the original tile-QR algorithm used by the PLASMA library. This minor algorithmic improvement which only required to add the code that inserts a new task clearly illustrates the productivity gain resulting from the use of a runtime system like StarPU. Instead of dealing with technical concerns, it was indeed possible to concentrate on the algorithmic aspect of the problem and to seamlessly obtain a full-fledged implementation of the improved algorithm.

7.3.3 Impact of the scheduling policy



(a) SGEQRF on HANNIBAL

(b) SGEQRF on MORDOR

Figure 7.4: Impact of the scheduling policy on the performance of a QR decomposition on different platforms.

Figure 7.4 shows the impact of the different scheduling policies on the performance of the QR decomposition for two different platforms. Both platforms are multicore machines enhanced by multiple GPUs. The machine used on Figure 7.4(a) is composed of 8 CPU cores and 3 GPUs, the one on Figure 7.4(b) contains 16 CPU cores and 4 GPUs.

In both case, the **greedy** scheduling strategy constitutes the performance baseline. In this strategy, as described in more details in section 3.2.3, all tasks are put in a common queue that is shared by all workers, regardless of the heterogeneous processing capabilities of the different workers, and without any care for avoiding data transfers. The asymptotic speed measured with this simple strategy therefore only reaches about 500 GFlop/s on both platforms.

The **heft-tm** strategy, described in section 3.4.3, consists in dispatching tasks in a way that minimizes their termination time, according to auto-tuned performance models. This typically permits to ensure that tasks in the critical path are scheduled on GPUs when the amount of parallelism is very low. In both machines, this strategy results in a 100 GFlop/s performance improvement. This relatively modest performance gain is explained by the fact that this strategy does not provide any support for data locality, which is a crucial issue in such multi-GPU platforms.

Considering that data transfers are critical for the performance, the **heft-tm-pr** strategy extends the previous policy by issuing data transfers requests as soon as possible to avoid wasting time. This prefetching mechanism not only exploits the scheduling holes to transfer data, but it also takes advantage of the asynchronous capabilities of the GPUs that can overlap computation with communication. **heft-tm-pr** thus outperforms the **heft-tm** strategy and reaches about 700 GFlop/s on both platforms. On Figure 7.4(b), we can however notice that the performance ceases to improve for large problem, especially on the machine with the larger number of processing units, which is the sign of a scalability issue.

Even though the previous strategies provide a good load balancing and do their best to hide data transfer overhead, they are not designed to avoid data transfers. As shown on Table 7.2, the total amount of data transfers measured during the execution of a QR decomposition is very significant. The **heft-tm-pr** indeed does not consider the impact of data movements when taking

Table 7.2: Impact of the scheduling policy on the total amount of data transfers during a QR decomposition on HANNIBAL

Matrix order	9600	24960	30720	34560
heft-tm-pr	3.8 GB	57.2 GB	105.6 GB	154.7 GB
heft-tmdp-pr	1.9 GB	16.3 GB	25.4 GB	41.6 GB

a scheduling decision. As a result, the different processing units are assigned tasks that access pieces of data which are often not available locally, and permanently have to invalidate other processing units' data replicates. The **heft-tmdm** and **heft-tmdm-pr** respectively improve the **heft-tm** and **heft-tm-pr** strategies by considering the overhead of moving data when dispatching tasks. Table 7.2 indicates that the amount of data transfers is therefore greatly reduced when taking data locality into account. Given the huge impact of data transfers on the overall performance of such accelerator-based machines, we finally measure about 150 GFlop/s (resp. 300 GFlop/s) of extra improvement on HANNIBAL (resp. on MORDOR) compared to the strategies which do not consider data locality.

The respective impacts of data prefetching and of penalizing superfluous data transfers are illustrated by the difference of speed between the **heft-tmdm** and the **heft-tm-pr** strategies. Both implement a different mechanism that improves data management, but the prefetching mechanism is not sufficient to hide the tremendous amount of data transfers which results from a bad data locality: the curve representing **heft-tmdm** on Figure 7.4(b) reaches more than 900 GFlop/s, which is 200 GFlop/s faster than the **heft-tm-pr** strategy. Data locality is therefore a key for the scalability of such machines, especially when the number of accelerators increases. Finally, the performance of the **heft-tmdm-pr** strategy illustrates that both data prefetching and data locality enhancing are quite orthogonal optimizations which can efficiently be combined to obtain an efficient data management.

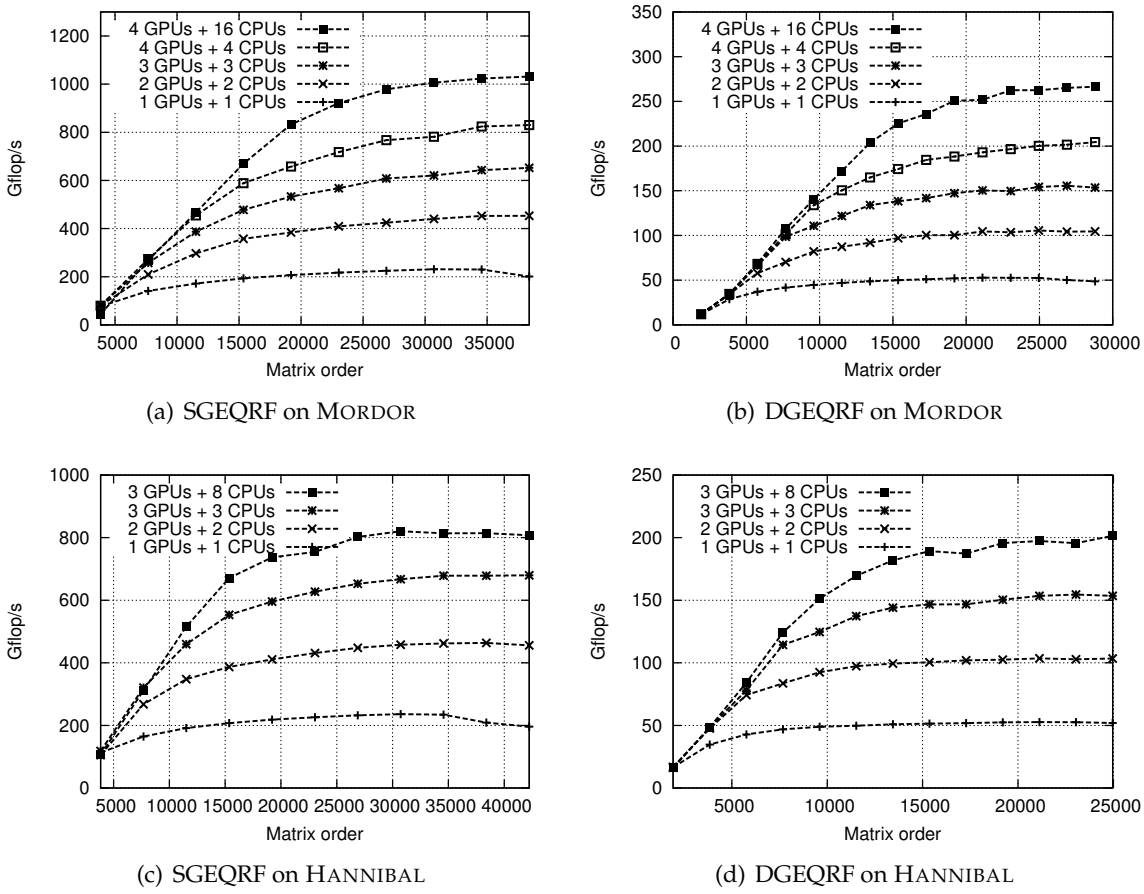
Scalability on hybrid platforms

In the previous section, we have shown that selecting the most appropriate scheduling policy has a huge impact on performance. We now consider the scalability of the QR decomposition with respect to the number of processing units. All scalability measurements are performed with the **heft-tmdm-pr** scheduling strategy which is the one that performs the best for this problem.

Figure 7.5 gives the performance of the QR decomposition benchmark on two platforms with a different number of processing units. Results are shown both in single and double precision in order to give evidence that StarPU provides an efficient scheduling even for double precision kernels, that are two times slower than their single precision counterparts on CPUs on the one hand, but typically eight times slower on GPUs on the other hand.

The GPU kernels are taken from the MAGMA library. These kernels are hybrid in the sense that they not only use a CUDA device, but also a CPU core which is tightly coupled to the GPU. On the lower curves, StarPU only schedules hybrid tasks between the different GPUs, so that the number of CPU cores is equal to the number of CUDA devices. In all cases, we almost obtain a linear scaling with respect to the number of CUDA devices used by StarPU. We even measure a super-linear efficiency when for problems that can fit in the memory distributed over multiple devices, but that are too large for a single GPU.

The top curve of each subfigure indicates the performance obtained when using kernels from



(a) SGEQRF on MORDOR

(b) DGEQRF on MORDOR

(c) SGEQRF on HANNIBAL

(d) DGEQRF on HANNIBAL

Figure 7.5: Scalability of the QR decomposition with respect to the number of processing units on two different platforms.

the PLASMA library on the remaining CPU cores. On Figures 7.5(a) and 7.5(b), there are 4 CUDA devices and 16 CPU cores, so that StarPU uses PLASMA kernels on 12 CPU cores, and MAGMA kernels on the 4 cores dedicated to 4 GPUs. On Figures 7.5(c) and 7.5(d), there are only 8 CPU cores and 3 CUDA devices, so that StarPU only executes PLASMA kernels on 5 CPU cores. In spite of the common belief that CPU cores are useless compared to very efficient GPUs that are especially efficient on such BLAS3 kernels, the significant performance improvement which results from the use of CPU cores in combination with GPU devices actually shows the strength of our hybrid approach.

When observing the performance obtained on a hybrid platform carefully, one can notice that the processing speed improvement resulting from the use of extra CPUs is really large. On 7.5(a), the use of the 12 remaining CPU cores in addition to the 4 GPU/CPU pairs for instance brings an extra 200 GFlop/s, while the total theoretical peak performance of these 12 CPU cores is approximately 150 GFlop/s. In our *heterogeneous* context, we define the **efficiency** as the ratio between the sum of the computation powers obtained separately on each architecture and the computation power obtained while using all architectures at the same time. This indeed expresses how well we manage to add up the powers of the different architectures. In the case of homogeneous proces-

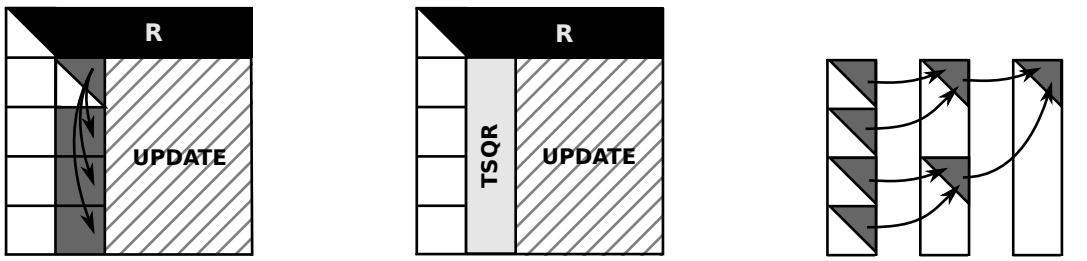
sors, the total power should usually not exceed the sum of the powers obtained by the different processing units. In the heterogeneous case there can however be some computation affinities: a GPU may be perfectly suited for some type of task while another type of task will hardly be as efficient as on a CPU, and the efficiency can then become greater than 1.

Table 7.3: Relation between speedup and task distribution for SGEQRF on MORDOR.

Kernel	CPU Speed	GPU Speed	Relative Speedup	Task balancing	
				16 CPUs	4 GPUs
SGETRF	9 GFlop/s	30 GFlop/s	×3.3	80 %	20 %
SSSMQR	10 GFlop/s	285 GFlop/s	×28.5	7.5 %	92.5 %

The last column of Table 7.3 for instance shows how StarPU dispatches two of the kernels that compose the Tile-QR algorithm. Not all kernels perform equally: the SSSMQR kernel, which is similar to a matrix product, is especially efficient on GPUs, but the SGETRF which takes care of decomposing the diagonal tile is not really suited to CUDA devices. Thanks to its performance models, StarPU tends to put almost all SGETRF kernels on the 12 CPU cores, while scheduling most SSSMQR kernels on the 4 GPUs. This illustrates how the **heft-tmdp-pr** scheduling strategy seamlessly takes advantage of the heterogeneous nature of the machine. *Each processing unit tends to be assigned tasks for which is very efficient, and relatively inefficient tasks tends to be assigned to other types of processing units*.

7.3.4 Communication-avoiding QR decomposition



(a) Original Tile-QR (b) Communication-Avoiding QR (CAQR) (c) Tall Skinny QR (TSQR)

Figure 7.6: Communication-Avoiding QR (CAQR) algorithm. Instead of serializing the TSQRT kernels during panel factorization, a reduction-based Tall Skinny QR (TSQR) is invoked to factorized the panel.

In the previous section, we have shown that the Tile-QR algorithm is very efficient in the case of square matrices. The panel factorization phase of the original Tile-QR algorithm given on Figure 7.6(a) illustrates that all TSQRT kernels are serialized when factorizing the panel. In the case of tall-and-skinny matrices, that is to say matrices which have many more rows than columns, the amount of parallelism available in this algorithm becomes extremely low in case there are only few columns. Such tall and skinny matrices are frequent in iterative methods, for instance when solving linear systems with multiple right-hand side (*e.g.* GMRES, CG, etc.). They are also commonly found in the context of iterative eigensolvers [53].

The CAQR algorithm depicted on Figure 7.6(b) however provides a scalable alternative to the original Tile-QR algorithm well suited for tall and skinny matrices [53, 3]. Instead of performing panel factorization in a sequential way, the Tall Skinny QR (TSQR) reduction-based algorithm is invoked to factorize the panel in a parallel fashion. Compared to the original Tile-QR algorithm, this higher scalability however comes at the price of a much more complex implementation. Implementing the CAQR algorithm on a multicore platform is already a challenging problem [3], but providing an efficient hybrid implementation without sacrificing code portability is a real problem.

Contrary to a hand-coded approach, implementing CAQR on top of StarPU only required to modify the Tile-QR algorithm described in the previous section to replace the panel factorization by a TSQR algorithm. Since task scheduling and data transfers do not have to be performed explicitly by the programmer, this task mostly required only to implement the additional kernels used by the TSQR algorithm. It is worth noting that there was previously no other hybrid CAQR implementation that takes advantage of both an arbitrary number of CPU cores and of multiple GPU processors, to the best of our knowledge. Again, this illustrates the significant programmability improvement which results from the use of runtime systems like StarPU.

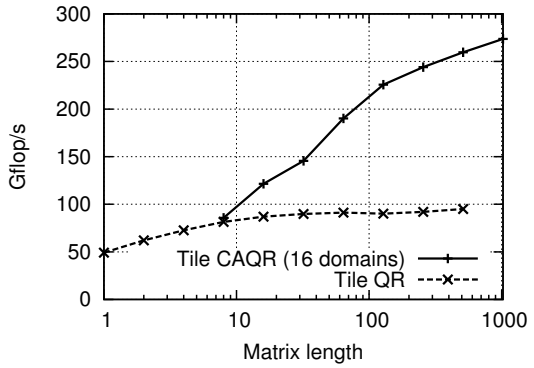


Figure 7.7: Performance of Tile CAQR for tall and skinny matrices with two block-columns on HANNIBAL.

Figure 7.7 shows the benefits of using the Tile CAQR algorithm on the HANNIBAL platform when processing tall and skinny matrices. The considered matrices have indeed a small fixed number of columns (two blocks) and a large varying number of rows (x-axis). Since Tile QR performs the panel factorization in sequence, parallelism is very limited and the performance remains low even when matrices have a large number of rows (right-most part of Figure 7.7). On the other hand, with Tile CAQR algorithm, the panel is divided into multiple domains (16 here) that can be processed concurrently. When the matrix has a large number of rows, this latter approach enables us to extract parallelism from the panel factorization and achieves a much higher performance than standard Tile QR on our hybrid platform.

7.4 Cholesky decomposition over MPI

In this section, we extend the single-node Cholesky decomposition detailed in Appendix A on a cluster of multicore machines enhanced with accelerators. This was achieved by following the

methodology presented in Section 5.4. The original Cholesky decomposition code is based on the `starpu_insert_task` helper function which permits to implement algorithms using a function-call semantic (see Section 2.2.5). We simply replaced this insertion function by its MPI-enabled counterpart (`starpu_mpi_insert_task`). In order to distribute the tasks over the different MPI nodes, we first have to assign an owner node to each registered data handle, so that each instance of StarPU can decide whether or not to execute a task submitted in the local MPI process. This is also used to automatically exchange data contributions between the various instances of StarPU running in the different MPI processes. Similarly to the reference SCALAPACK implementation, we distributed the different blocks according to a 2D-cyclic layout in order to minimize the total amount of communication.

Table 7.4: Total Physical Source Lines of Code used to implement Cholesky decomposition with StarPU.

	Sequential	Distributed
Kernels	159 lines	
Performance Models	13 lines	
Task submission	134 lines	159 lines

The productivity of our approach is illustrated on Table 7.4 which gives the total number of source lines of code of the actual implementation of the Cholesky decomposition algorithm, measured using DAVID A. WHEELER'S SLOCCount tool. The first line indicates the number of lines required to implement the CPU and GPU kernels used in the Cholesky decomposition. The conciseness of the implementation of these kernels mostly consists in invoking CPU and GPU BLAS kernels (*i.e.* from MKL, CUBLAS and MAGMA). Since the BLAS functions are very regular, we provided performance models for this kernels by adding only 13 lines of code to specify that StarPU should use history-based performance models. Registering data and unrolling the entire DAG describing the Cholesky decomposition algorithm only required 134 lines of code in the sequential version. Extending this sequential implementation to support MPI was achieved by assigning a owner MPI process to each registered data handle, by replacing the `starpu_insert_task` helper function by `starpu_mpi_insert_task`, and by adding code to initialize and deinitialize the MPI-like library described in Section 5.2.

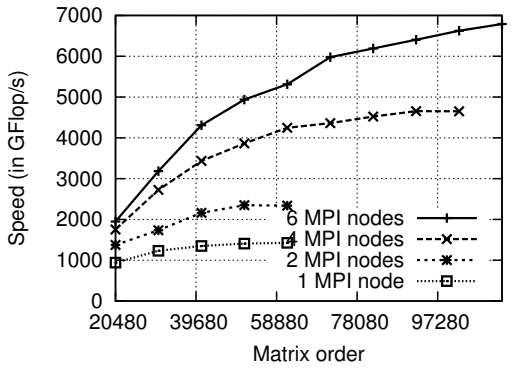
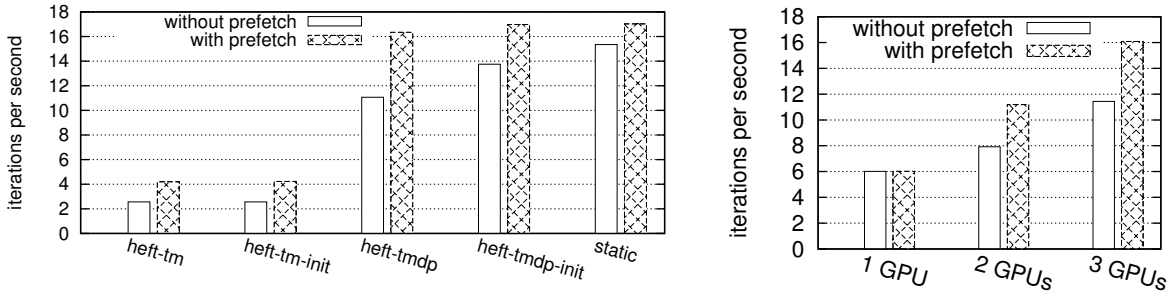


Figure 7.8: Strong scalability of a Cholesky decomposition over a Cluster of machines accelerated with 3 GPUs and 12 CPU cores per MPI node (PLAFRIM).

Figure 7.8 shows the strong scalability obtained by this implementation on a cluster of machines enhanced with accelerators. Similarly to the sequential implementation, we use the **heft-tmdp-pr** scheduling policy which obtains the best performance. Even though the difference between the sequential and the distributed implementations only consists in 25 lines of code, our distributed Cholesky decomposition almost reaches 7 TFlop/s (in single precision) on a 6-node cluster with 3 GPUs and 12 CPU cores per node. Such a speedup of approximately 5 on 6 MPI nodes is reasonable considering that the network is a serious bottleneck. It could be improved by extending StarPU's MPI-like library to support the features added in the fourth release of the CUDA driver: the thread safety improvements should be useful to reduce the latency of our communication protocol, and the direct transfer capabilities introduced between CUDA devices and network cards would significantly reduce the bandwidth consumption.

7.5 3D Stencil kernel



(a) Impact of scheduling on one node.

(b) Scalability within one node

Figure 7.9: Performance of a Stencil kernel over multiple GPUs.

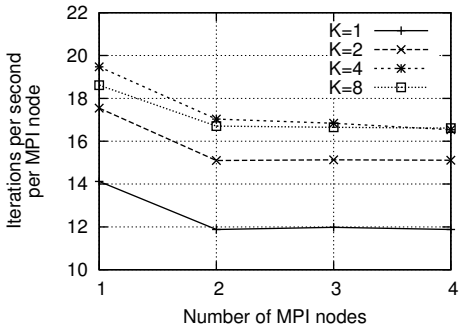


Figure 7.10: Throughput of a Stencil kernel over a cluster of machines with multiple GPUs (AC).

A stencil application puts a lot of pressure on data management because it is basically a BLAS1 operation, that *a priori* needs intermediate results transfer between Processing Units for each domain iteration. To get good performance, it is thus essential to properly overlap communication with computation and avoid the former as much as possible. This is also a good stress-test for the dynamic schedulers of StarPU since just statically binding all computations should *a priori* give

the best performance. We implemented [ACOTN10] a simple 3D 256x256x4096 stencil split into 64 blocks along the z axis.

Figure 7.9(a) shows the performance obtained with the 3 GPUs of HANNIBAL using various schedulers. The *static* case uses a trivial static allocation and thus gets the best performance of course. The **heft-tm** case does not use Data transfer Penalty and thus gets the worst performance, because it basically schedules all tasks quite randomly. The **heft-tm-init** case, which binds the tasks like in the *static* case for the first stencil iteration but does not use Data transfer Penalty, does not bring any benefit. The **heft-tmdp** case does use Data transfer Penalties, without any particular initial placement. The achieved performance is already very close to the *static* case. We observed that the penalties actually tend to guide the scheduler into assigning adjacent tasks to the same GPU and keeping that assignment quite stable over iterations. This means that StarPU permits to just submit tasks without having to care how they could ideally be distributed, since the scheduler can dynamically find a distribution which is already very good. In the **heft-tmdp-init** case, the initial placement permits to achieve the same performance as the *static* case, thanks to data penalties guiding the scheduler into keeping that initial placement most of the time. We additionally observed that if for some external reason a task lags more than expected, the scheduler dynamically shifts the placement a bit to compensate the lag, which is actually a benefit over static allocation.

Figure 7.9(b) shows the scalability of the performance obtained by the completely dynamic **heft-tmdp** scheduler: it scales quite linearly, even if the third GPU is on an 8x PCIe slot while the two first are on a 16x PCIe slot.

It can also be noticed that the prefetch heuristic does provide a fairly good improvement, except of course when using only a single GPU since in that case data just always remains automatically inside that GPU.

Figure 7.10 shows how the stencil application scales over 4 machines, using the *heft-tmdp-pr* scheduling policy. K is the size of the overlapping border that is replicated between domain blocks. It needs to be big enough to facilitate overlapping communication with computation, without incurring too much duplicate computation due to the border. One can note that performance drops quite a bit between 1 MPI node and 2 nodes, due to the limitation of the network bandwidth, but using more nodes does not really reduce the performance[2]. This shows that the StarPU execution runtime does not seem to have bad effects on the efficiency of the MPI library.

7.6 Computing π with a Monte-Carlo Method

Monte Carlo methods are widely used in numerical simulations [86]. They consist in repeatedly generating random numbers to measure the fraction of numbers which obey some properties. Figure 7.11 for instance illustrates how the π number can be computed using an algorithm based on the Monte Carlo method. The general idea is to take a random point in a unit square, and to detect which ratio of the points are located within the quarter of circle which has an area of $\frac{\pi}{4}$. Assuming that the random numbers are properly distributed over the unit square, this permits to compute an estimation of the π number. It is worth noting that from a numerical point of view, this algorithm only converges toward π very slowly as it requires about ten times more shots to obtain any additional significant digit. This benchmark is however very representative of the different algorithms based on Monte Carlo methods.

[2]Thanks to the 1D distribution, communication happens only between consecutive nodes, and the network switch is not (yet) saturated with the 4 available nodes.

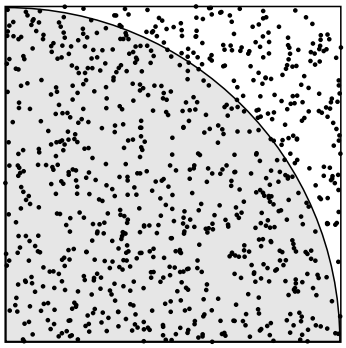


Figure 7.11: Computing π with Monte Carlo method. Points are randomly shot over a unit square. The shaded area delimits a quarter of circle with a unit radius, centered at (0,0). The probability that a point should hit the shaded is equal to $\frac{\pi}{4}$.

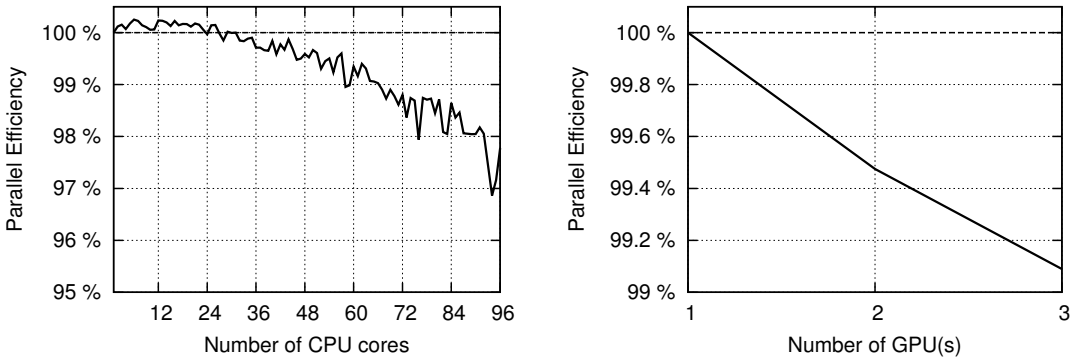
Algorithm 2: Implementation of Monte Carlo method.

```
1 begin
2     cnt ← 0;
3     for i = 1 → N do
4         x ← rand([0, 1]);
5         y ← rand([0, 1]);
6         if (x² + y²) < 1 then
7             cnt ← cnt + 1;
8     return 4×cnt / N;
```

The simplicity of this algorithm makes it a very good candidate for a benchmark. It should illustrate the behaviour of StarPU on an algorithm that can be considered as trivially parallel. A naive implementation would however consist in registering the cnt variable found in Algorithm 2, and to create N tasks that all perform the same test. Not only task management overhead, but also the vector nature of both GPUs and CPU processors would make it totally inefficient to create a task for each and every test. Since the number of iterations N is really large, we actually perform a few thousand tests per task, which avoids to schedule billions of tiny tasks.

Each task would also have to access the cnt variable in a *read-write* mode, which prevents multiple tasks from running concurrently. A manual implementation of this algorithm would consist in creating a local cnt variable for each processing unit, and to sum their values at the end. The reduction access mode presented in section 2.4.2 actually provides a portable and efficient way to manipulate the cnt variable as an accumulator which can be accessed concurrently.

Figures 7.12 and 7.13 respectively show the parallel efficiency and the speedup measured when running this benchmark either on a manycore machine equipped with 96 AMD CPU cores, or on a multi-GPU machine with 3 NVIDIA C2050 GPUs. It should be noted that no hybrid measurement is shown because of the huge performance gap observed between CPUs and GPUs on this benchmark which is perfectly suited to take advantage of the massive vector parallelism available in GPUs. We almost obtain a perfect scalability as we measure a 93.9 speedup over the 96 CPU cores of the manycore platform, and a 2.97 speedup when taking the 3 GPUs of the multi-GPU



(a) On a Many-core platform (96 AMD cores). (b) On a multi-GPU platform (3 C2050 GPUs).

Figure 7.12: Parallel efficiency of the Monte-Carlo method implemented with reductions.

Platform	Speed (in GShot/s)	Speedup
1 core	3.35×10^{-2}	$1\times$
24 cores	8.05×10^{-1}	$23.9\times$
48 cores	1.60	$47.8\times$
96 cores	3.15	$93.9\times$

(a) On a Many-core platform with 96 cores (BERTHA).

Platform	Speed (in GShot/s)	Speedup
1 GPU	4.76	$1\times$
2 GPUs	9.47	$1.99\times$
3 GPUs	14.2	$2.97\times$

(b) On a multi-GPU platform with 3 C2050 GPUs (ATTILA).

Figure 7.13: Speedup of the Monte-Carlo method implemented with Reductions.

machine.

Besides showing the internal scalability of the StarPU runtime system, this good parallel efficiency illustrates the significant impact of access modes with a relaxed coherency on scalability, especially on large machines. One of the challenges to reach portable performance indeed consists in designing scalable algorithms which take into account the fact that the amount of parallelism available in the algorithms may not grow as large as the number of processing units. Relying on runtime systems such as StarPU is therefore a convenient approach to develop sustainable algorithms which can benefit from powerful mechanisms which can hardly be re-implemented by hand every time there is a new platform, or when the algorithm has to be slightly modified.

7.7 Computational Fluid Dynamics : Euler 3D equation

We parallelized the Computational Fluid Dynamic (CFD) benchmark [2] from the Rodinia benchmark suite [41] to support hybrid multicore machines accelerated with multiple CUDA GPUs. This code implements a solver for the three-dimensional Euler equations for compressible flow. Such a scheme is very representative of unstructured grid problems, which are a very important class of application in scientific computing.

In order to parallelize the reference code between multiple processing units, we used the Scotch library [154] to decompose the original problem into sub-domains that are computed separately as shown on Figure 7.14. Boundary values (which are delimited by dotted lines) are kept coherent by the means of redundant computations and by exchanging up-to-date values between

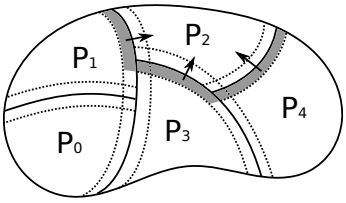


Figure 7.14: Parallelizing the CFD benchmark by dividing into sub-domains. The shadowed part indicates the redundant cells that are exchanged between each iteration to compute part P_2.

each iteration.

Three different kernels were used to parallelize the original CFD benchmark. The main kernel updates the cells of a sub-domain, it is directly based on the original benchmark, and typically takes more than 95 % of the processing time. The two other kernels are used respectively to save the content of a sub-domain's boundary values into an intermediate buffer, and to load these values into a neighbouring sub-domain.

These three kernel themselves were naturally enough parallelized in the SPMD fashion explained in section 4.2, by distributing the different cells that compose a sub-domain between the different CPU cores.

7.7.1 Scalability of the CFD benchmark on a manycore platform

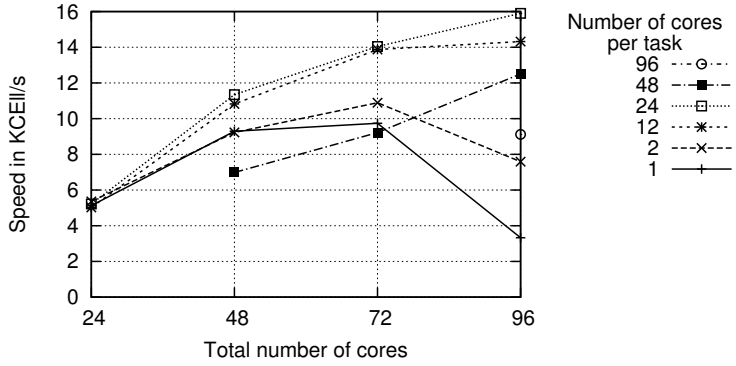


Figure 7.15: Strong scalability of the CFD benchmark (with 800K cells). The number of cores per task indicates the size of the parallel sections in each task: a single core per task stands for sequential tasks, and 96 cores per task means that a single task is spread over the entire machine

Figure 7.15 shows the strong scalability of the CFD benchmark on a machine with 96 cores. The different curves indicate how the size of the parallel sections (*i.e.* number of cores per task) impacts performance. When the number of task is the same as the number of cores (which was the optimal situation for matrix multiplication), the performance drops when the number of task gets too large. Speed improves as the number of cores per task increases, but drops again above 24 cores per task (*i.e.* when tasks are spread on workers larger than a NUMA node).

The more cores, the harder it is to scale. The parallelization of the CFD benchmark indeed relies on partitioning the initial domain into sub-domains that are kept coherent thanks to redun-

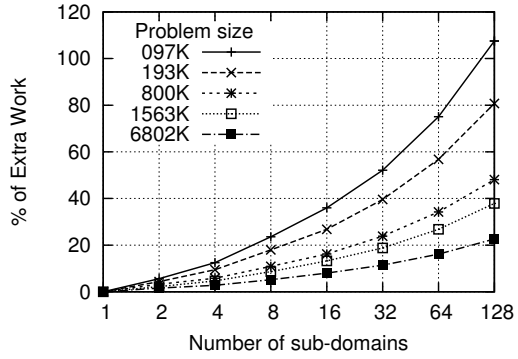


Figure 7.16: Parallelization overhead of the CFD benchmark.

dant computation. The amount of extra work introduced by this redundancy mechanism directly depends on the number of sub-domains, and on the initial problem size. Figure 7.16 shows the exponential growth of the extra amount of work with respect to the number of sub-domains. Partitioning a problem of $800K$ cells into 96 pieces for instance requires to perform 40% additional computation, compared to less than 10% when there are only 4 pieces.

Using parallel tasks is an easy way to keep the number of sub-domains under control. Apart from the parallelization overhead of the kernel itself, the amount of extra work required to parallelize the CFD algorithm therefore remains quite low. This explains why we observe better performance on Figure 7.15 when mutualizing cores instead of cutting the domain into numerous sub-parts.

Too large parallel sections however result in lower performance on Figure 7.15. This machine is indeed composed of 4 NUMA nodes with 24 cores each. Compared to its high number of cores, this machine has a *very* limited memory bandwidth between NUMA nodes (10 Gb/s). The performance of the parallel kernel therefore collapses as soon as the different processing units are not within the same NUMA node, so that we get poor overall performance with tasks of size 48 or 96 on Figure 7.15. As already mentioned, StarPU uses the hwloc library to ensure that it combines workers with respect to the actual machine hierarchy, here NUMA nodes, instead of randomly grouping unrelated CPU cores.

Scheduling more tasks also incurs more overhead on the runtime system. When the different sub-domains get smaller, the redundant cells are more likely to overlap with a high number of neighbouring sub-domains. With the $800K$ cell input, the average number of neighbours of the different sub-domains is about 10 when there are 96 parts. Having a smaller number of sub-domains not only results in less dependencies, but it also avoids to exchange a large amount of small boundaries, which is relatively inefficient compared to few large ones. Parallel tasks therefore permit to reduce the burden on the runtime system, so that it can deal with larger machines more easily. In the current manycore era, this is crucial for runtime environments which will hardly scale indefinitely without some hybrid programming model (*i.e.* DAGs of parallel tasks in our case).

7.7.2 Efficiency of the CFD benchmark on a Hybrid platform

In many cases, the CFD algorithm does not benefit from a hybrid platform because the GPU kernel completely outperforms its sequential CPU counterpart which is 30 to 40 times slower. As discussed in Section 4.3.3, this would require to divide the problem into at least a hundred sub-domains of equal size to make sure CPUs can actually become useful. Taking a dozen CPU cores in parallel however results in a relative speedup of only 4 between a single GPU and a parallel CPU workers. We therefore applied the model described in Section 4.3.3 in order to select the most appropriate number of tasks to run the CFD benchmark as efficiently as possible on the hybrid platform. For each iteration, we map a single parallel task on the different CPUs, and put 4 tasks on each GPU. This only requires to create about a dozen sub-domains, which considerably reduces the overhead compared to a hundred sub-domains.

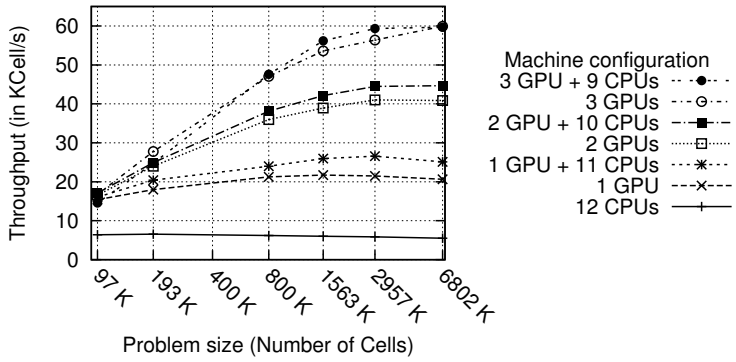


Figure 7.17: Throughput of the CFD kernel on a Hybrid machine (12 CPUs + 3 GPUs).

Figure 7.17 shows the speed of the CFD benchmark on various machine configurations. A static scheduling was used to ensure that the results are not perturbed by scheduling artifacts, so that we do measure the actual benefits of parallel tasks. The bottom curve gives the throughput of the algorithm on a multicore machine without GPUs. The other curves show how this benchmark performs on a GPU-based machine. We observe a linear scaling with respect to the number of GPUs, except for very small problem sizes which are penalized by data transfers that can be almost as long as doing the entire computation on a single GPU in case the amount of computation is really low.

Three hybrid configurations with parallel CPU tasks are however shown on Figure 7.17. Up to three GPUs are taken, and all remaining CPU cores are combined into a single parallel worker. The difference measured between GPU-only configurations and their hybrid counterparts is close to that of the CPU-only configuration. This indicates that we can fully take advantage of each of the processing units within an hybrid platform thanks to parallel tasks.

7.8 Discussion

In this chapter, we have shown the efficiency of StarPU on various benchmarks, and the impact of the scheduling policies on performance and on bus contention. While most programmers tend to consider that using both CPUs and accelerators at the same time has little impact or is simply

too complicated to be worth the effort, we have for example shown that the QR decomposition kernel actually takes advantage of heterogeneity by avoiding to offload non-GPU friendly kernels on CUDA devices and vice-versa. As a result, processing units are only assigned tasks for which they are relatively efficient. The overall sustained power obtained with a hybrid CPU-GPU system exceeds the added sustained powers of the CPUs and the GPUs taken separately. Obtaining such a result in a hand-coded application is possible, but it requires a certain knowledge of both the algorithm (*i.e.* of the different kernels) and the underlying architecture (*i.e.* which processing unit is the most suitable for a kernel). On the other hand, this is transparent for applications written on top of StarPU. This illustrates how StarPU actually provides performance portability.

The performance gain resulting from a better scheduling is therefore very significant, so that the overhead of dynamic task scheduling is completely amortized when compared to the programming efforts and the performance of manually scheduled application. Also, the latency to manipulate an accelerator (*e.g.* to initiate a memory transfer or a kernel on a CUDA device) is typically orders of magnitude larger than the overhead of a CPU function call. Paying a few extra micro-seconds to obtain a much more efficient scheduling that actually reduces the frequency of costly operations such as data transfers is therefore a sensible trade-off. Thanks to an engineering effort, we could also improve the implementation of StarPU to reduce this overhead, for instance by using more scalable internal data structures such as lock-free task queues.

The results shown in the chapter also confirm that the high-level constructs offered by StarPU also help designing scalable algorithms suited for upcoming manycore architectures. Relaxing the data coherency model by providing users with data reductions for instance improves scalability by lazily updating frequently accessed accumulators. In the case of the CFD kernel, we have also shown that parallel tasks are required to fully exploit manycore platforms, unless we use tasks with variable granularity. Parallel programmers should therefore really consider using such high-level constructs whenever possible to design algorithms that will scale on future manycore and/or hybrid platforms.

Chapter 8

Diffusion

Chapter Abstract

This chapter gives actual examples of applications relying on StarPU to take advantage of accelerator-based platforms. Runtime systems being a central part of the computing ecosystem, we then also show how StarPU was used to provide support for the different bricks composing the typical HPC software stack. For instance, we illustrate how StarPU permitted to easily develop an hybrid implementation of the LAPACK library based on the PLASMA and the MAGMA libraries. The we show examples of compilation environments which were modified to use StarPU as a back-end which provides portable performances and transparently solves most low-level concerns. Finally, we discuss about the relationship between StarPU and the OpenCL standard, and we show how OpenCL support was used to transparently provide support for a new type of architecture.

8.1 Integration of StarPU within the computing ecosystem

Dwarf	Example(s) of benchmarks
Dense Linear Algebra	Cholesky, QR, LU, CAQR (see Section 8.3.1)
Sparse Linear Algebra	Sparse Conjugate Gradient
Spectral Methods	Fast Fourier Transforms (see Section 8.3.2)
N-Body Methods	N-Body problems
Structured Grids	3D Stencil kernel (see Section 7.5)
Unstructured Grids	Euler 3D CFD kernel (see Section 7.7)
MapReduce	Monte Carlo methods (see Section 7.6)

Figure 8.1: Berkeley's classification of scientific computing problems into dwarfs

In the previous chapters, we have shown the efficiency of StarPU on a wide set of benchmarks over various types of machines. Table 8.1 recalls the different types of compute workload identified in Berkeley's classification of scientific computing problems [12]. Even though it is not necessarily realistic to provide a comprehensive list of dwarves that should capture any type of scientific computation, it is worth noting that we have experimented StarPU on a wide set of benchmarks covering most of the dwarfs; which suggests the suitability of our model on a large spectrum of application domains.

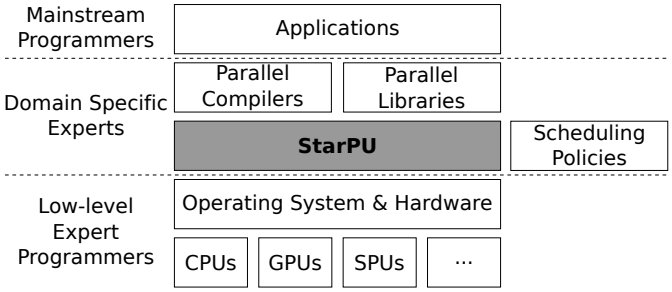


Figure 8.2: Integration of StarPU within the computing ecosystem.

Figure 8.2 depicts the central role played by runtime systems such as StarPU in the HPC computing ecosystem. Besides applications directly using StarPU, it is also important to note that runtime systems provide performance portability for higher-level layers which need both portability and performance portability such as high-performance libraries or parallel compilers possibly targeting hybrid accelerators-based platforms.

8.2 Real-Life Applications enhanced with StarPU

In this section, we illustrate how StarPU was used to enhance real-life applications to fully exploit hybrid accelerator-based platforms in a portable way. The first example is a medical imaging application that detects the contour of vertebrae in X-ray images. The second example is a multimedia application that recognizes *hummed* songs within an audio database. Both cases illustrate how StarPU made it possible to reach performance portability without having to deal with low-level concerns, so that actual programmers could fully concentrate on the design of efficient kernels.

8.2.1 Vertebra Detection and Segmentation in X-Ray images

MAHMOUDI *et al.* have developed an application that automatically detects the vertebra contour on X-ray images of cervical vertebrae [134, 125]. This permits to automate part of the hospital diagnosis that consists in analyzing vertebral mobility [160]. Segmenting regular radiography of the spine is the cheapest and fastest way of detecting vertebral abnormalities, but it is very challenging because of the poor contrasts obtained on X-ray images. Time being a crucial factor in the medical world, detecting the contour of cervical vertebrae must be done as fast as possible. Given the very important amount of data to process, the use of clusters and/or of accelerators such as GPUs is promising. The original GPU-enabled implementation only allowed to use a single GPU at the same time, and data management was up to the programmer [134]. The authors of this initial version have therefore modified their code so that StarPU automatically decides where to execute the different tasks with respect to the heterogeneous nature of the platform [125]. Figure 8.3[1] gives an overview of the different steps of the segmentation algorithm which details are totally out of the scope of this document. The interested reader will however find a detailed study of this framework in the original publication presenting the implementation that relies on StarPU [125].

Figures 8.4 gives the performance of the StarPU-based implementation applied on 200 images with different resolutions. This implementation only uses hybrid resources for the edge detection step, which is the most time consuming. It should be noted that the time scale was truncated for the sake of readability on Figure 8.4(a) because the time required to detect the edges on 200 images with a 3936×3936 resolution takes 538 seconds on a single GPU. On Figure 8.4(a), we measure very significant reduction of the computing time when the number of processing units increases. Besides the use of multiple GPUs, it is worth noting that using the CPU cores in addition to GPU devices actually bring a real performance improvement. In this real-life case, the relative speedup between the CPU and the GPU kernels are indeed limited, so that the performance gain obtained with extra CPU cores cannot be neglected. Higher image resolutions result in better speedups on Figure 8.4(b) because of the higher communication/computation ratio. This is typically an example of application that fully exploits hybrid platforms by the means of StarPU without having to deal with low-level concerns. Manually dispatching work between the different CPU cores and the different GPU devices would have been a tedious task, while it is done automatically and in a portable way by StarPU. This permits the application authors to concentrate on the design of efficient image processing kernels instead of complex low-level issues.

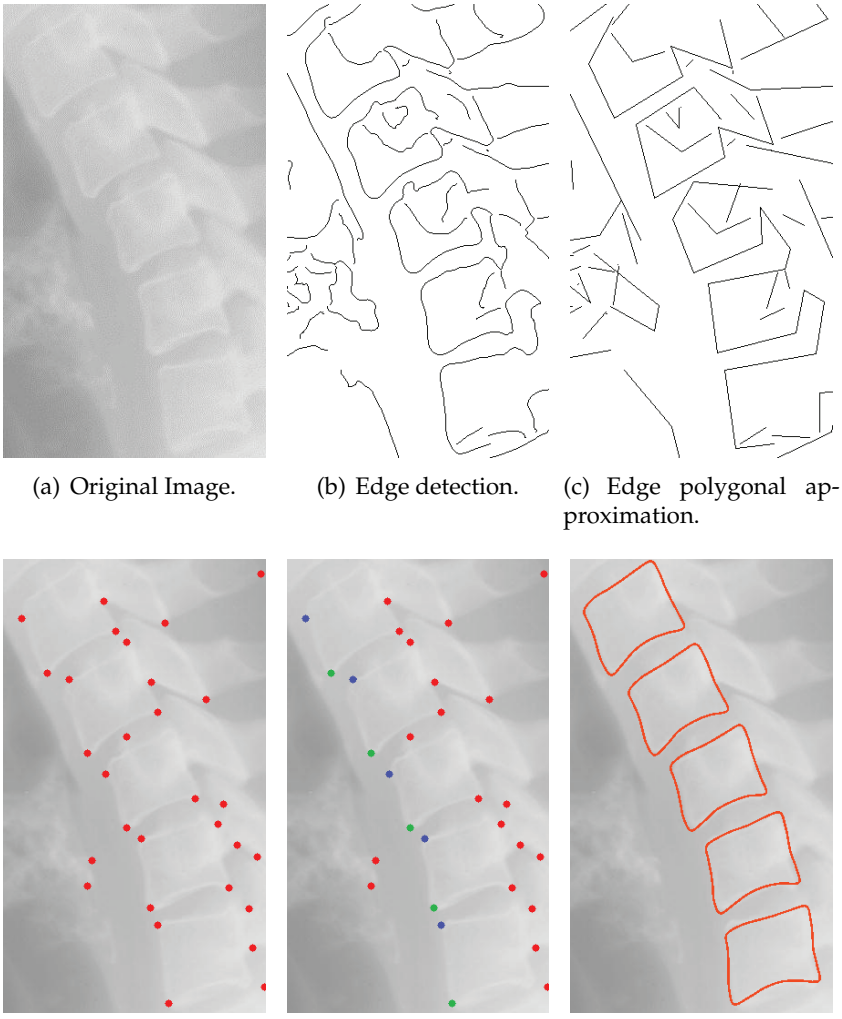
8.2.2 Accelerating a query-by-humming music recognition application

Table 8.1: Speedup obtained on SIMBALS with StarPU on HANNIBAL

Number of GPU(s)	Time	Speedup
1	79.5s	1x
2	40.1s	1.98x
3	26.7s	2.97x

SIMBALS (SImilarity Between Audio signaLS) is an application that takes a *hummed* audio input file and searches for the closest song(s) within a large database of monotonic audio files

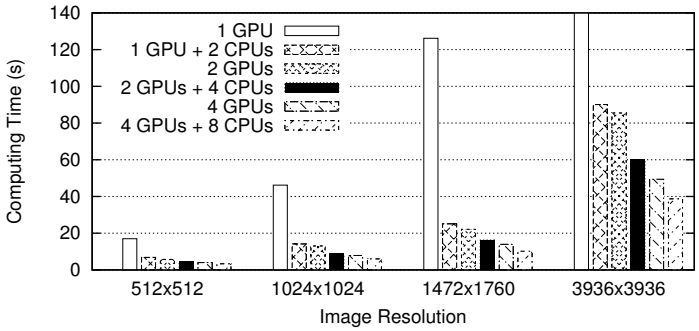
[1] Image courtesy of Sidi Ahmed MAHMOUDI from the University of Mons.



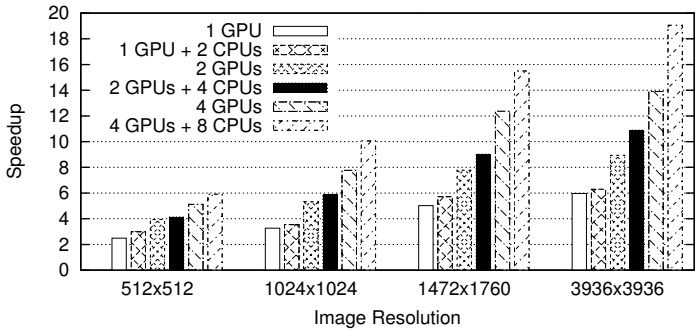
(a) Original Image. (b) Edge detection. (c) Edge polygonal approximation.

(d) Points of interest detection. (e) Vertebra detection. (f) Segmentation result.

Figure 8.3: Illustration of the whole segmentation framework.



(a) Computing time of edge detection step.



(b) Speedup obtained for edge detection step.

Figure 8.4: Performance of recursive edge detection on hybrid platforms.

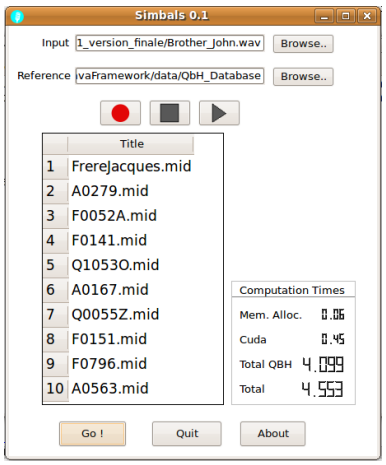


Figure 8.5: Screenshot of the SIMBALS music recognition library.

stored in the MIDI format [66]. Contrary to the popular SHAZAM application that looks for an exact match in its database, audio queries are hummed in SIMBALS which therefore needs to consider possible imperfections in the pitch and in the length of the different notes, because users are not all supposed to be professional singers. On Figure 8.5, SIMBALS for instance detects that the database entry that is the closest from the `Brother_John.wav` input file is the

`FrereJacques.mid` song which is indeed its french equivalent.

The distance defining songs similarity is obtained by counting the minimum number of transpositions between two songs. Such transpositions for instance include pitch modification, note deletion or note duration modification [66]. A Smith-Waterman algorithm is used to compare the audio input with the different database entries. SIMBALS considers local transpositions rather than global ones because the *hummed* input is usually shorter than the original song. This method returns the optimal local alignment, but they are very time consuming. The tremendous processing capabilities available in modern GPUs however significantly improve the usability of such an algorithm for musical applications.

The authors of the SIMBALS application have therefore ported their algorithm so that it exploits CUDA, and measured a significant speedup compared to an optimized CPU implementation [66]. This initial hand-coded implementation was however limited to a single CUDA device and could not deal with arbitrarily large databases. We[2] therefore modified the SIMBALS application and replaced explicit data management and kernel submission by calls to the StarPU runtime system. As a result, SIMBALS is now able fully exploit hybrid platforms with multiple accelerators without having to worry about data management and task scheduling. Table 8.1 for instance gives the impact of the number of GPUs on the duration of a query in a database composed of 78000 midi files. Compared to the initial hand-coded implementation that could not use more than one GPU, our implementation almost reaches a perfect speedup over the three GPUs of the HANNIBAL machine. It should be noted that CPU cores were not used during this experiment because of the huge speedup that exists between the CPU and the GPU implementations [66].

8.3 Libraries

Libraries are the basic building blocks of numerous applications, especially in the context of scientific computing. They are a convenient mean to write efficient code in a productive way. Performance portability is also ensured by using libraries that are available on various platforms, and which are progressively improved to support new architectures. Offering support for libraries with a high-impact (*e.g.* LAPACK, FFTW, etc.) therefore permits to seamlessly get the benefits of StarPU in a very large number of real applications.

8.3.1 A hybrid implementation of LAPACK mixing PLASMA and MAGMA

PLASMA [97] and MAGMA [96] are state-of-the-art implementations of the LAPACK library, respectively for multicore processors and for NVIDIA CUDA devices. Compared with previous LAPACK implementations, the novelty of the PLASMA library consists in using tiled algorithms which provide much more parallelism than the classic algorithms based on a fork-join parallelism. These algorithms are implemented by the means of the Quark dynamic task scheduler, which is designed to address multicore architectures with a shared memory. Compared to StarPU, Quark therefore provides a very limited support for data management, and no support for heterogeneous resources.

Similarly to the CUBLAS library that implements BLAS kernels on NVIDIA CUDA devices, the MAGMA library provides a fully optimized implementation of the LAPACK library for CUDA.

[2]The port on top of StarPU was performed in the context of an internship by Mathieu ACCOT, Evans BOHL, Eddy CHEN and Mehdi JUHOOR.

The algorithms implemented in the MAGMA library are limited to the size of a single GPU, and the different pieces of computation are scheduled by hand. Significant work was made to extend the MAGMA library in order to support multi-GPU systems with almost a perfect scalability [182]. In spite of its very good performance, the portability of this multi-GPU implementation is however very limited. Supporting a new optimization or modifying the underlying algorithms would both require a significant amount of work.

Even though both projects implement the similar algorithms, PLASMA and MAGMA target completely different types of hardware, and mixing these two libraries is a real challenge. Implementing a portable and efficient hybrid LAPACK library without the support of a runtime system such as StarPU would represent a huge effort. Given the very fast pace followed by accelerator constructors, such a library might actually be obsolete by the time it is available. KURZAK *et al.* however attempted to extend the PLASMA library in order to offload some kernels on a GPU [122], but their ad-hoc approach required that new BLAS kernels should be generated to deal with a specific memory layout, which might represent a gigantic effort.

We are therefore currently designing a full-fledged LAPACK library that fully takes advantage of the entire processing power available in multicore platforms enhanced with accelerators. This library will be based on the algorithms used in the PLASMA library and extends it to exploit CUDA GPUs by the means of kernels from the MAGMA library. Provided the significant gains in terms of maintainability and of portability, it should actually supersede the previous efforts to support multi-GPU in the MAGMA library. In addition to the one-sided factorizations previously mentioned (*i.e.* Cholesky, QR and LU decomposition), we are going to support for other LAPACK functions such as two-sided factorizations (*e.g.* Hessenberg factorization).

8.3.2 StarPU-FFT

Fourier transforms are extremely common in the context of signal processing. The FFTW library [69] (Fastest Fourier Transform in the West) is a library that provides a generic interface to compute discrete Fourier transforms (DFTs). Given the huge impact of FFTW, NVIDIA provides a similar interface to perform these operations on CUDA devices by the means of the CUFFT library [100]. We[3] have therefore implemented a library with an interface similar to that of FFTW and CUFFT, which implements DFTs on top of multi-GPU and hybrid platforms. This library currently provides support for 1D and 2D transforms in real and complex precision. It is based on parallel DFT task-based algorithms which use FFTW and CUFFT kernels internally. While GU *et al.* propose advanced data management techniques to support out-of-card FFT computations [84], this is transparently supported with StarPU.

Figure 8.6 shows an example of code implementing a 2D FFT with StarPU's FFT library. The similarity with FFTW's interface allows a very low entry cost for programmers who want to adapt existing codes to multi-GPU and hybrid platforms.

8.4 Support for compilers and programming environments

StarPU was not directly designed to offer support for the end-user. Instead, it is intended to help library programmers or to provide a convenient runtime system for higher-level programming environment targeting hybrid accelerator-based platforms. Even though libraries are a convenient

[3]The implementation of this library was done by Samuel THIBAULT.

```
STARPUFFT(plan) plan;
STARPUFFT(complex) *in, *out;

in = STARPUFFT(malloc)(size * sizeof(*in));
out = STARPUFFT(malloc)(size * sizeof(*out));

/* Fill with random values */
srand48(0);
for (i = 0; i < size; i++)
  in[i] = drand48() + I * drand48();

plan = STARPUFFT(plan_dft_2d)(n, m, SIGN, 0);
STARPUFFT(execute)(plan, in, out);

STARPUFFT(destroy_plan)(plan);
STARPUFFT(free)(in);
STARPUFFT(free)(out);
```

Figure 8.6: Example of 2D FFT performed with StarPU's FFT library.

way to seamlessly exploit complex machines, not all computation kernels can be rewritten as invocations of standard libraries. In addition to libraries, we must therefore also provide support for compilers.

While task parallelism is a powerful approach, it is not always accessible to end-users who cannot afford learning a new paradigm, or simply rewriting legacy codes. Industrial codes composed of millions of lines of code are typical examples of applications that cannot be completely rewritten every time there is a new technological breakthrough.

We therefore modified several compilation environments to use StarPU as a back-end. This permits to exploit the capabilities of StarPU with a minimal code intrusiveness.

8.4.1 Adding StarPU back-ends for annotation-based language extensions

Annotation-based approaches are a convenient way to adapt legacy codes. The *pragma*-based extensions proposed by the StarSs projects for instance permit to easily extend existing C and Fortran codes. It permits to exploit complex architectures with a simple-to-use function-call semantic.

The different implementation of the StarSs languages actually rely on the Mercurium source-to-source compiler [16]: SMPSs schedules tasks on SMP processors [17], CellSs dispatches work between the SPUs of Cell processors [22, 21] and GPUSs deals with CUDA devices [14]. Contrary to StarPU which provides a unified environment that dispatches work within a hybrid platform, each implementation of StarSs therefore separately provides its own specific back-end for the Mercurium compiler and a task-based runtime system implementing the low-level platform-specific mechanisms, even though multiple implementations can be used altogether [157]. We[4] therefore added another back-end for StarSs which generates StarPU tasks and relies on StarPU to manage data transfers. This provides a convenient interface to easily extend legacy sequential C code with tasks that are automatically generated. Even though it does not provide the same level of control as StarPU's low level interface (*e.g.* to synchronize with an arbitrary task or explicit task dependencies), this makes it possible to benefit from most of StarPU's capabilities with a low entry cost.

[4]Mercurium's StarPU back-end was designed during the master thesis of Sylvain GAULT.

```
1  #pragma css task input (A, B) inout (C)
2  void matmul(float *A, float *B, float *C, size_t nx , size_t ny , size_t nz);
3  #pragma css target device (smp) implements (matmul)
4  void matmul_cpu(float *A, float *B, float *C, size_t nx , size_t ny , size_t nz);
5  #pragma css target device (cuda) implements (matmul)
6  void matmul_cuda(float *A, float *B, float *C, size_t nx , size_t ny , size_t nz);
7
8  static float A[1024*1024], B[1024*1024], C[1024*1024];
9
10 int main(int argc, char **argv)
11 {
12   #pragma css start
13   #pragma css register variable(A:1024*1024*sizeof(float))
14   #pragma css register variable(B:1024*1024*sizeof(float))
15   #pragma css register variable(C:1024*1024*sizeof(float))
16   matmul(A, B, C, 1024, 1024, 1024);
17   #pragma css barrier
18   #pragma css finish
19   return 0;
20 }
```

Figure 8.7: Example of code using the Mercurium source-to-source compiler to automatically generate StarPU tasks.

Figure 8.7 gives an example of code using StarPU's Mercurium back-end. Lines 1 to 6 show the prototypes of different implementations of a matrix multiplication kernel which is implemented on both CPUs (lines 4 and 5) and CUDA devices (lines 6 and 7). These lines are statically transformed into a StarPU codelet structure by Mercurium. StarPU is initialized and deinitialized on lines 12 and 18. The three pieces of data are registered to StarPU on lines 13 to 15. The starting address and the length of each vector is stored in a hash table that permits to automatically retrieve the StarPU data handle describing the vector from the starting address only. The function call on line 16 is finally transformed into a task submission. This task implements the codelet previously generated, and the data handles describing the A, B and C matrices are taken from the hash table. The function-call semantic of StarSs naturally ensures that we have a sequentially consistent code, so that task dependencies are implicitly derived from data accesses.

This approach does not only apply to the Mercurium source-to-source compiler: we[5] are currently implementing a similar interface based on GCC plugins [77]. Besides removing a dependency to a third-party source-to-source compiler, this allows a better integration with the compiler. For instance, it is possible to automatically infer access types from the prototype. A pointer with a `const void *` type is automatically transformed into a data handle accessed in a read-only mode. Statically allocated data can also be automatically registered to StarPU. Such a tight integration with the compiler should also allow powerful optimizations based on the analysis of the control-flow graph (*e.g.* merging multiple function calls into a single task). While such optimizations are also possible in theory with a source-to-source compiler such as Mercurium, they are much more realistic in an environment such as GCC.

While these source-to-source transformation provide an easy-to-use coordination language, they do not provide any support to generate kernels. As shown on lines 4 and 6 of Figure 8.7, the application indeed still needs to provide both CPU and CUDA kernel implementations.

[5]The GCC plugin implementing the annotations required to automatically generate StarPU tasks is being designed by Ludovic COURTS.

8.4.2 Automatic kernel generation with HMPP

CAPS' HMPP [55] (Hybrid Multicore Parallel Programming) is a compilation environment which goes one step further than the previous approaches: it not only provides languages extensions to automatically offload pieces of computation, but it also transforms annotated C or Fortran codes into CUDA or OpenCL kernels. HMPP is thus complementary to StarPU which does not intend to generate kernels while HMPP's support for scheduling is very limited. We can use its coordination language (or those described in the previous section) to transparently generate StarPU tasks, and take advantage of HMPP's code generation capabilities to automatically generate the different implementations of the codelets. It must be noted that we could also rely on the PIPS compiler to take care of the generation of OpenCL and CUDA kernels [5].

All in all, programmers would have to write annotated C or Fortran codes that are automatically transformed into CUDA or OpenCL kernels by HMPP. The resulting multi-versioned codelets would be used by the tasks which are efficiently scheduled by StarPU. This approach is especially promising to accelerate legacy industrial applications for which it is sometimes not even realistic to rewrite the computation kernels.

In case rewriting such kernels is not illusory, another option, to avoid having to redevelop the same kernels multiple times, is to directly use the OpenCL standard to implement the kernels. In theory, OpenCL kernels should be portable across all devices supporting OpenCL. In practice, it is still not clear whether it is always possible to obtain portable performance without specifically tuning the OpenCL kernel for each target architecture [114]. Similarly, it is not clear whether annotation-based language can ensure performance portability without requiring architecture-specific annotations.

8.4.3 The SkePU skeleton library

SkePU is an example of programming environment based on skeletons [61]. Similarly to template-based libraries (*e.g.* NVIDIA Thrust [146] or Rapidmind/Ct [102]), it provides a portable interface that allows programmers to express their applications as high-level data-parallel computations. The conception of the algorithms implementing these skeletons is however up to the designers of SkePU. Still, such skeleton-based libraries internally face the performance and portability issues that they are actually trying to hide to the end-users.

The original implementation of SkePU is able to statically distributed the workload on multi-GPU platforms [61]. In order to easily obtain portable performance, new skeletons have been implemented by the means of tasks that are dynamically scheduled by StarPU. This approach thus makes it much easier to fully exploit complex hybrid platforms without having to determine a new static schedule every time there is a new skeleton or a new evolution of the underlying hardware. Adding a new back-end of SkePU only requires to implement the various computation kernels used in the skeletons: low-level issues such as an efficient data management are directly solved by StarPU.

SkePU also has auto-tuning capabilities which for instance permit to automatically select the most suitable algorithm and the most suitable granularity, depending on the input size of the skeleton invocation. The best implementation of a parallel sorting skeleton can for example be either obtained with an algorithm based on quicksort or on bubble sort, depending on the input parameters. SkePU actually selects the optimal implementation among the available algorithmic variants by considering the actual performance measured during the previous invocations of the

skeleton [52]. Instead of redesigning an experimental protocol to properly benchmark the performance of each new skeleton, we now simply rely on the performance feedback directly provided by StarPU.

While the scheduling strategies presented in Chapter 3 are able to select to most suitable type of processing unit, this permits to choose the most efficient algorithmic variant once the task has been assigned to a processing unit. Other skeleton-based frameworks with auto-tuning capabilities such as PetaBricks [9] could benefit from this technique as well. It is worth noting that we are also extending StarPU's codelet interface to support multiple implementations for the same type of architecture, *e.g.* by providing an array of CPU functions. It will then be possible to select the most efficient variant according to StarPU's performance models.

8.5 Relationship between StarPU and the OpenCL standard

The OpenCL standard is an effort to provide a unified programming interface for the various types of modern architectures and for a wide spectrum of applications [83]. OpenCL covers two very different aspects of hybrid computing. First, it defines a portable device abstraction which permits to implement the different low-level mechanisms required to manipulate accelerators (*e.g.* data transfers or kernel offloading). It is worth noting that in the past, similar standardization efforts have already been attempted (without much success) to provide a uniform device abstraction for high-speed network: the Virtual Interface Architecture (VIA) [56]. The second aspect covered by the OpenCL standard is to define a portable language permitting to easily write computation kernels for SIMD architectures. On the one hand, this unifies the various SIMD extensions already available in most compilers. On the other hand, such a standard language is an answer to the growing number of vendor-specific interfaces trying to become *de facto* standards, such as NVIDIA CUDA which was even ported on multi-core CPU architectures [174, 180].

Since the goal of StarPU is also to provide a portable abstraction for hybrid platforms, it is crucial to understand how StarPU and OpenCL relate to each other. The most obvious way to combine StarPU and OpenCL is to have an OpenCL back-end which permits to execute kernels written in the OpenCL language (*e.g.* to support AMD GPUs). In such case, the role of OpenCL is to provide a portable device interface, but not to offer any specific support for task and data scheduling. Another possible interaction thus consists in using StarPU internally within an OpenCL implementation, to benefit from its scheduling capabilities without having to explicitly use its non-standard interface.

In the remaining of this section, we will show how we combined StarPU and OpenCL, either by providing support for a new type of accelerator that implements the OpenCL device interface, or by actually integrating StarPU with an OpenCL implementation with scheduling capabilities.

8.5.1 Exploiting the power of an embedded processor with an OpenCL back-end

An important goal of OpenCL is to provide a portable device interface that can be implemented on the different types of accelerators. Movidius is a fabless semiconductor company which designs chips with advanced multimedia capabilities (*e.g.* High Definition 3D Video) [140]. Instead of creating yet another vendor-specific programming API for their chips, they have decided to implement the OpenCL standard on top of their new multicore chip prototype, called MYRIAD, which is composed of 8 processing cores called SHAVE.

We have therefore used the OpenCL driver available in StarPU to control a prototype of the Myriad chip running in a simulator. Besides offloading code on the SHAVE processors, StarPU has been successfully used to provide programmers with feedback about the energy consumption of the application running on the simulator. Energy is a critical concern on such embedded architectures. We are therefore developing energy-aware scheduling strategies which could for instance take into account the fact that moving data within this chip is sometimes much more expensive than performing computation on a sub-optimal processing core. The combination of StarPU with such a simulator predicting energy consumption with an accuracy below 10% (compared to actual silicon) is a real opportunity for the programmers of embedded applications to ensure that their code is ready by the time the actual chip is available on silicon.

8.5.2 StarPU as an OpenCL device: SOCL

In the OpenCL programming paradigm, an OpenCL kernel is directly assigned to the specified OpenCL device. An OpenCL device can actually consist of multiple physical sub-devices, in which case a scheduling decision has to be taken by the OpenCL implementation during kernel submission. From a conceptual point of view, launching a kernel on an abstract OpenCL device is therefore not so different from submitting a task to StarPU. This semantic similarity is the basis of the *StarPU OpenCL Library* [89] (SOCL) which implements an OpenCL device on top of StarPU. SOCL provides a fully compliant implementation of the OpenCL standard. Invocations of the OpenCL API are internally translated into calls to StarPU. This permits to transparently run legacy OpenCL applications on hybrid accelerator-based platforms featuring multicore CPUs and multiple GPUs. KIM *et al.* provide a similar approach which permit to transparently execute native OpenCL codes over multiple OpenCL devices [111]. While SOCL transforms OpenCL kernels into StarPU tasks, their approach benefits from compile-time and sampled information which permit to divide each OpenCL kernel into multiple sub-kernels that are dispatched between the various OpenCL devices.

Kernels written in OpenCL are transformed into StarPU codelets. When the OpenCL application enqueues a new kernel to the StarPU device, a StarPU task is scheduled on one of the underlying processing unit. Similarly to any task-based application, SOCL still encounters granularity issues when the OpenCL kernels are not large enough compared to the overhead of task management. Such problems could typically be alleviated by relying on a static analysis phase that could merge multiple OpenCL kernel into a single StarPU task [6].

Data allocated in OpenCL are automatically registered to StarPU. When a piece of data is explicitly copied in OpenCL, a new data handle is created because there is no guarantee that an OpenCL application will maintain the different copies coherent (except for read-only buffers). Optimal performance are obtained when the application does not explicitly transfer data, but use mapped memory which is automatically managed by StarPU. This permits StarPU to fully exploit its data caching capabilities and to avoid superfluous data copies. Most OpenCL applications still being written in a synchronous fashion, SOCL is also able to expose multiple identical devices which provide the application with an impression of a homogeneous platform.

Table 8.2 shows the performance of a matrix multiplication implemented in OpenCL. Two 16384×16384 single-precision matrices are given as input. This is a naive implementation which consists in dividing the output matrix into 64 regular blocks which are dispatched between 3 OpenCL devices. This straightforward implementation achieves $435\,GFlop/s$ on 3 GPUs with the OpenCL implementation provided by NVIDIA. An extra $20\,GFlop/s$ speed improvement is mea-

Table 8.2: Performance of a single-precision matrix multiplication on HANNIBAL with SOCL.

Platform	Native OpenCL	SOCL
3 GPUs	435 GFlop/s	456 GFlop/s
3 GPUs + 5 CPUs	23 GFlop/s	545 GFlop/s

sured when using SOCL. This can be explained by the fact that even though we have a presumably homogeneous setup, the buses between host memory and the three GPUs are not necessarily identical, which adds to the performance irregularities caused by NUMA effects. In this machine, two GPUs are for instance connected with a 16x PCI-e link while the third GPU only has an 8x PCI-e link. The dynamic load balancing capabilities enabled by SOCL allow to transparently counterbalance these irregularities. SOCL also benefits from the advanced data management mechanisms available in StarPU such as data prefetching.

On a heterogeneous system, mapping the blocks in a round-robin fashion obviously performs extremely bad because CPU cores become a bottleneck. Contrary to the standard OpenCL implementation, SOCL is able to dynamically dispatch the different blocks with respect to the actual processing power of the different OpenCL devices. It must be noted that SOCL not only performs better than the native implementation, but it also performs better than without the help of CPU cores.

While the initial goal of OpenCL is to provide a portable device abstraction, the previous example illustrates that it is possible to obtain portable performance too. With a proper scheduling, naive OpenCL codes are indeed able to take advantage of a hybrid platform. The current OpenCL standard only allows to attach a single device to each context, and defines data allocation at context level. Modifying the OpenCL standard to allow applications to create a single context shared between multiple devices would make it much easier to exploit multi-accelerator and heterogeneous platforms, provided the support of a runtime system such as StarPU. This however requires to redefine the scope of data allocation at device-level. Finally, encouraging applications to use context-wide mapped memory instead of per-device explicit data transfers would allow numerous optimizations which are hardly doable by hand (*e.g.* data prefetching or manipulating data sets larger than devices' memory).

8.6 Discussion

In this chapter, we have shown that our approach is applicable to a wide spectrum of real-life problems. StarPU has indeed permitted programmers to easily port their applications on complex hybrid accelerator-based machines without having to deal with complex data transfers or to decide where, when and how to offload computation. Instead, StarPU only required them to port their kernels and to describe the different pieces of data accessed by the algorithm. Loops of pure function calls can also be translated into asynchronous submissions of StarPU tasks.

Through the various examples, we have depicted the central role played by runtime systems like StarPU in the overall computing ecosystem. While many programming environments and many libraries are being adapted to accelerators, having such portable and flexible abstractions prevent them from dealing with technical concerns. This for instance permits compilers targeting accelerators and/or hybrid systems to concentrate on efficient code generation/extraction. We have also shown how we designed a full-fledge implementation of the LAPACK library by com-

bining the PLASMA algorithms and its CPU kernels with MAGMA's CUDA kernels. Various other libraries and applications have been or are being hybridized in a similar way using StarPU. StarPU's flexible scheduling engine provides a convenient framework to easily implement prototypes of scheduling algorithms for manycore and hybrid platforms.

Better collaboration between the different software layers is however desirable. On the one hand, performance feedback obtained by gathering profiling information at runtime would help higher-level layers such as iterative compilers to generate better code or to take better auto-tuning decisions. On the other hand, libraries and application can provide StarPU with algorithmic knowledge (*e.g.* performance models or tasks critical path) which can be useful to help the scheduler. Such scheduling hints can also be obtained by the means of static code analysis. In order to ensure that all software layers collaborate efficiently, standardization efforts are also required. We have for instance shown how complementary StarPU and the OpenCL standard are. Even though numerous runtime systems and compilation environments attempt at providing their own specific interface, providing higher-level tools with a standard task and data management abstraction is still a widely open problem. In this context, the actual advantages of StarPU when compared to other similar systems result from StarPU's ability to provide higher-level tools with performance feedback, and to take advantage of user-provided hints. A standard interface *must* therefore allow such a tight collaboration between the different software layers.

Conclusion and Future Challenges

AFTER hitting the *frequency wall* which led to the multicore era, architects are now facing the *energy wall* which prevents them from indefinitely replicating full-fledged processing units which would have an excessive power consumption. As a result, the manycore revolution is likely to be marked by the combination of numerous simplified processing units along with a few full-fledged cores. Even though they have existed for a long time under various forms, accelerators are therefore becoming a solid trend when designing parallel architectures. These accelerators can either be external accelerating boards typically connected to the host via a PCI-e bus, or they can be tightly integrated within the die of a heterogeneous multicore processor.

While a significant attention has been paid to providing support to efficiently offload computation on accelerating boards as a replacement for standard CPUs, the relative processing power coming from multicore processors and the number of processors has kept increasing in the meantime. In order to fully exploit such heterogeneous architectures, programmers need to step away from this pure offloading model to adopt a true hybrid paradigm that provides a unified support for heterogeneous processing units.

Moreover, programmers cannot afford to follow the unprecedented pace of hardware evolution anymore as the lifetime of an accelerating technology is sometime shorter than the time it takes to port applications. Programmers therefore need support to design portable applications that can fully exploit accelerators without necessarily being parallel programming or parallel architecture experts. Runtime systems can provide portable abstractions that permit to design scalable parallel algorithms without worrying about low-level non-portable concerns which are addressed seamlessly and efficiently at runtime. By relying on such runtime systems, programmers are not only able to design portable parallel algorithms, but they also obtain performance portability, which is the guarantee that the runtime system will always provide the best possible performance delivered by the various hybrid accelerator-based platforms.

Contributions

Throughout this thesis, we have shown that task parallelism provides a flexible paradigm which allows programmers to describe their parallel algorithms in a portable way. We have also shown that this paradigm is generic enough to be extended to a cluster environment as well. In spite of the hardware revolution which implies a programming model shift, all programmers are not strictly required to explicitly adopt such a task paradigm. End-users should indeed rely on high-level parallel compilers and/or parallel hybrid libraries whenever possible.

Efficient data management is a serious issue on such accelerator-based platforms because the I/O bus is typically a bottleneck, and data transfers are usually implemented by the means of

highly non-portable architecture-specific mechanisms (*e.g.* DMA on the Cell processor). Data management and task scheduling should therefore be performed jointly by the runtime system to obtain portable performance. Combining task parallelism with explicit data registration indeed enables aggressive optimization which would be difficult to implement efficiently by hand (*e.g.* asynchronous background data prefetching). Letting the runtime system in charge of data management also makes it possible to manipulate arbitrarily large data sets which do not necessarily fit within accelerators' memory.

Since there does not exist a single perfect scheduling strategy, we have designed a flexible interface to design scheduling policies and we have developed a set of efficient policies. Thanks to auto-tuned performance models and user-provided hints, we have shown that we can fully exploit a heterogeneous platform by optimizing tasks' termination time and minimizing data transfer overhead. The overhead of dynamic scheduling is amortized by the significant gains, both in terms of performance and with respect to programmability concerns because programmers are not required to determine a suitable task mapping by hand anymore. While heterogeneity usually appears to be an issue when programming accelerators by hand, our results indicate that we can actually take advantage of the heterogeneous nature of the machine, thanks to a suitable scheduling. For instance, we have shown that for a given task graph, the overall sustained speed of a hybrid CPU-GPU platform is sometimes higher than the sum of the sustained speeds measured independently either on a CPU-only platform or when solely using GPUs. StarPU's scheduler was indeed able to only assign tasks to units which could process them efficiently, and ensure that critical tasks were assigned to the fastest units. This confirms that using CPUs in combination with accelerators does makes sense, and that a runtime system exposing a uniform abstraction of processing unit is required.

The runtime system must collaborate with the application to determine a suitable granularity, with respect to heterogeneity and scalability concerns. Exposing high-level abstractions such as data reductions should also allow programmers to design highly scalable algorithms which are ready for the upcoming manycore revolution. More generally, it is important that runtime systems feature expressive interfaces that allow programmers to guide the scheduling with application-specific knowledge. In return, runtime systems should give performance feedback to feed auto-tuning frameworks and to implement powerful debugging and performance analysis tools which are a necessity for real-life applications. By providing a unified interface which hides the inner complexity, runtime systems relieve compilers and libraries from the burden of low-level programming. They avoid fully reimplementing a new backend every time there is a new architecture available. Besides portability concerns, runtime systems enable separation of concerns. Compilers can thus concentrate on generating efficient code and on applying powerful optimization which are out of the scope of a runtime system. By collecting information about the execution, runtime systems are also able to dynamically take the most appropriate decision which may have been difficult to take statically, without a global view of the machine's state. Runtime systems not only ensure that applications fully exploit the capabilities of the underlying hardware without any architecture specific knowledge. They also guarantee that codes will always transparently benefit from the latest features available, long after the code was written.

Perspectives

The model described in this thesis has raised many open questions, and there are numerous research opportunities to be explored.

Improving data management Adding a DSM mechanism (such as implemented by GMAC) in addition to StarPU's explicit data registration mechanism would boost productivity by making it possible to access unregistered data throughout the system when we cannot afford to entirely modify legacy codes. Combined with an annotation-based language using StarPU as a back-end, this would typically allow to gradually adapt such legacy codes so that they take advantage of StarPU's advanced features with a reduced entry cost. Due to the heterogeneity of the different processing units, the optimal data layout may depend on the underlying architectures: SIMD architectures may favor structures of arrays, but hierarchical cache-based multicore processors may perform better with arrays of structures. StarPU's data management should therefore be able to automatically convert a registered piece of data into another format, after deciding whether this is more efficient than to use the original format (*e.g.* based on performance models). Such a multi-layout data management would be especially useful for memory-bound kernels which performance highly depends on the efficiency of data accesses (*e.g.* sparse linear algebra kernels or unstructured grid problems).

Improving the scheduling We must implement more scheduling policies to address the limits of existing strategies. For instance, we could implement strategies that consider power consumption or that schedule memory bound kernels on units with a high bandwidth. Instead of merely improving data locality, scheduling policies should actually schedule data transfers. Aggressive data prefetching would indeed make it possible to avoid scheduling holes due to data access dependencies, and to minimize the memory footprint by evicting unused cached data as early as possible. We should also use more scalable data structures (*e.g.* lock-free queues) within the scheduling policies, and within StarPU directly to ensure a minimal management overhead on large manycore platforms. In case StarPU tasks are not first class citizens, and that some computation is performed directly from the application without calling StarPU, we need to ensure that the application can temporarily use some processing resources which are normally controlled by StarPU. This would for instance be doable by creating scheduling domains which can dynamically be disabled so that StarPU can only use a subset of the processing resources until the domain is enabled again.

In order to cope with the high degree of parallelism found in manycore architectures, it should be possible to design composable scheduling policies. This would permit to describe scalable strategies by recursively assigning tasks to subsets of the machine, following different strategies at the different level of machine's hierarchy. This requires to provide a generalized abstraction of processing unit, so that the scheduler can either assign tasks to a specific unit or to a subset of the machine controlled by another strategy. Another significant improvement would be to allow scheduling experts to describe (composable) strategies in a high-level language such as Prolog.

Providing better support for clusters In order to fully take advantage of the capabilities of the hardware, StarPU should exploit the low-level zero-copy mechanisms which avoid superfluous contention. Direct transfers between GPUs or even between a GPU and other types of devices such

as disks or network cards would also help to reduce latency and contention in clusters of machines enhanced with accelerators. Data-aware scheduling policies should also take into account dependencies with external events such as MPI transfers. To improve latency, a strategy should typically first schedule the tasks that are likely to produce data to be sent in order to rapidly unlock remote dependencies. We have shown that provided an initial data mapping between the MPI nodes, we can automatically map a StarPU application over a cluster. Such a data mapping could be obtained automatically through static analysis techniques in collaboration with compilation environments. We could also decentralize StarPU's scheduling engine to allow dynamic load balancing between the different instances of StarPU on the various MPI nodes.

Extending our task model We should provide the application with mechanisms to automatically take granularity-related decisions. For example, we could use StarPU's performance feedback to determine whether there is a sufficient amount of parallelism and if the different processing units are not overwhelmed with scheduling overhead. This would allow applications or advanced scheduling policies to detect when tasks should be divided, or provide indications that larger tasks should be generated. The current implementation of StarPU is mostly oriented toward HPC; in order to provide support for soft real-time applications, we may have to be able to run parallel algorithms in a degraded mode when it is not possible to fulfill the real time requirements. Similarly to divisible tasks which would consist in replacing a task by a set of tasks performing the same work, it should be possible to replace a whole set of task by an alternative set of tasks that can be executed timely.

Unification and standardization efforts Various environments offer support for accelerator-based platforms. Most of them are internally based on a task paradigm, and rely on a scheduler to some extent. There is however no consensus for a standard runtime system interface yet. Designing such a unified abstraction of runtime system is particularly delicate because it needs to be expressive enough to allow the runtime system to collaborate with higher-level software layers, not only by providing execution feedback, but also by letting programmers inject algorithm-specific knowledge into the runtime system (*e.g.* priorities or performance models). We should also take part to the existing standardization efforts in OpenCL and as well as the attempts to integrate accelerators in the OpenMP standard. For instance, we have constantly shown that a pure offloading model is not sufficient anymore, so that a programming standard should not expose data transfers but represent data access modes instead. In other words, all copy directives should be replaced by *read/write* access modes, which do not imply that there actually needs to be a transfer. Parallel libraries becoming mainstream, it is crucial that we can invoke OpenMP parallel sections within tasks. OpenMP implementations should be reentrant, and the standard should provide more control to dynamically confine the execution of a parallel section on a specific subset of the machine. The OpenCL standard currently provides a standard device interface and a standard language to write vectorized kernels. Each processing device is controlled by sending OpenCL commands into per-device contexts. We could extend OpenCL with a notion of global context and submit tasks in a global context controlling multiple devices at the same time. By introducing a runtime system like StarPU inside OpenCL implementations, this would permit to enhance OpenCL with load balancing capabilities. Noteworthy, this also requires to replace explicit data transfer commands with explicit data access modes, to avoid having to know in advance which processing would execute the kernels in advance.

Toward exascale computing and beyond

As parallel machines keep getting larger, programmability and scalability issues become even more serious. Exascale platforms will contain millions of probably heterogeneous processing units, and introduce new challenges which need to be (at least partially) addressed by runtime systems. Runtime systems should automatically enforce power capping by dynamically determining which processing units should be switched off and by dynamically adapting processors' frequency, for instance to reduce the overhead of memory latency for memory-bound kernels.

To avoid wasting processing power, the amount of resource allocated to each parallel algorithm may also evolve dynamically in such large systems that are shared between numerous users. Multiple parallel libraries executed concurrently could also cooperate to exchange processing units. Processing resource management will become much more dynamic. The average time between hardware failure will also significantly drop, so that processors may constantly appear or disappear. Resilient runtime systems should therefore be able to detect hardware faults to respawn failing tasks on different processing units. At such a large scale, data and load balancing will have to be performed in a fully decentralized fashion. To reduce latency and bandwidth requirements, and to fully take advantage of machines' massive processing capabilities, the same tasks may be replicated on multiple nodes in case the data transfer overhead is higher than the extra computing overhead. Distributing data management would also make it possible to reconstruct data in case of such a hardware failure.

To facilitate scheduling (*e.g.* with a better understanding of data locality), the amount of hints provided to the runtime system should gradually increase, either through static analysis or with performance aware languages which let programmers specify performance hints. As a result, the amount of information available for each task may explode, which indicates that we will need to design programming languages which are expressive enough to generate these tasks and decorate them with scheduling hints. These languages should expose high-level parallel constructs that permit to design highly scalable algorithms, such as data reductions or other common parallel patterns.

While the number of processing units keeps growing rapidly, the total amount of memory does not increase as fast. The memory available on each node will thus become very limited. Data management techniques should therefore include out-of-core algorithms. Runtime systems should transparently deal with the increased depth of memory hierarchy to provide applications with a simplified memory abstraction. Such a deep memory hierarchy will also significantly increase latency and bandwidth requirements. Thanks to an explicit data registration, the scheduler should maintain the lowest possible memory footprint in addition to ensuring an optimal data locality.

Similarly to accelerators which are programmed in a SIMD fashion even if they contain dozens of processing units, a possible approach to avoid generating too many tasks will also consist in implementing a SIMD model which permits to gather groups of processing units in larger logical processing units. Such a hybrid hierarchical model will reduce the pressure on the scheduler by dividing the number of logical processing units by several orders of magnitudes. Mixing task parallelism with a SIMD paradigm also avoids asking programmers to extract millions of concurrent parallel sections. In the meantime, intra-kernel parallelism should be obtained by the means of parallel libraries and parallel compilers which will generalize. In order to implement such hierarchical programming paradigms, we need composable runtime systems. Programmers should be able to write task-parallel algorithms which invoke parallel tasks and combine these with calls

to existing fully-optimized parallel libraries, the result being concurrent nested parallelism levels which nicely share the whole system.

Appendix A

Full implementation of Cholesky decomposition

This appendix gives the complete example of algorithm written on top of StarPU. We unravel the different steps of the methodology used to implement Algorithm 3. Tile Cholesky decomposition is indeed a very concise example of code that performs very well and which permit to demonstrate various features available in StarPU. This example relies on the `starpu_insert_task` helper function that enables a function-call semantic thanks to implicit data dependencies.

Algorithm 3: Tile Cholesky Algorithm.

```
1 for k ← 0 to N_t − 1 do
2 |   POTRF(A^rw_kk)
3 |   for m ← k + 1 to N_t − 1 do
4 |   |   TRSM(A^r_kk, A^rw_mk)
5 |   |   for n ← k + 1 to m − 1 do
6 |   |   |   GEMM(A^r_mk, A^r_nk, A^rw_mn)
7 |   |   SYRK(A^r_mk, A^rw_mm)
```

Initialization and Deinitialization. When initializing StarPU with `starpu_init`, StarPU automatically detects the topology of the machine and launches one thread per processing unit to execute the tasks. Calling `starpu_shutdown()` releases all the resources.

Registering and Unregistering data Since the tasks composing the tile Cholesky factorization work on tiles, the matrix to be factored is itself split into tiles. Each tile is registered separately into StarPU to be associated with a handle. As shown in Figure A.1, the `tile_handle[m][n]` StarPU abstraction is obtained from each actual memory pointer, `tile[m][n]`. Several data types are pre-defined for the handles. Here, tiles are registered as matrices since a submatrix is itself a matrix. When all tasks have been executed, we stop maintaining data coherency and put the tiles back into main memory by unregistering the different data handles.

```
1  float *tile[mt][nt];                    // Actual memory pointers
2  starpu_data_handle tile_handle[mt][nt]; // StarPU abstraction
3
4  starpu_init(NULL); // launch StarPU
5
6  for (n = 0; n < nt; n++) //loop on cols
7  for (m = 0; m < mt; m++) //loop on rows
8    starpu_matrix_data_register(&tile_handle[m][n], 0, &tile[m][n], M, M, N, sizeof(float));
9
10 (...) // task submission
11
12 for (n = 0; n < nt; n++) //loop on cols
13 for (m = 0; m < mt; m++) //loop on rows
14   starpu_matrix_data_unregister(&tile_handle[m][n]);
15
16 starpu_shutdown(); // stop StarPU
```

Figure A.1: Initializing StarPU and registering tiles as handles of matrix data type. Data handles are unregistered at the end of the computation and all resources are released when stopping StarPU.

Codelet definition As shown at lines 39-45 for the `sgemm_codelet` in Figure A.2, a codelet is a structure that describes a multi-versioned kernel (here, *sgemm*). It contains pointers to the functions that implement the kernel on the different types of units: lines 1-15 for the CPU and 17-32 for the GPU. The prototype of these functions is fixed: an array of pointers to the data interfaces that describe the local data replicates, followed by a pointer to some user-provided argument for the codelet. The `STARPU_MATRIX_GET_PTR` is a helper function that takes a data interface in the matrix format and returns the address of the local copy.

Function `starpu_unpack_cl_args` is also a helper function that retrieves the arguments stacked in the `cl_arg` pointer by the application. Those arguments are passed when the tasks are inserted. In this example, the implemented routines are wrappers on top of the respective actual *sgemm* CPU and GPU BLAS CPU kernels.

Task insertion In StarPU, a task consists of a codelet working on a list of handles. The access mode (e.g., read-write) of each handle is also required so that the runtime can compute the dependencies between tasks. A task may also take values as arguments (passed through pointers). A task is inserted with the `starpu_insert_task` function.[1] Lines 32-40 in Figure A.3 shows how the *sgemm* task is inserted. The first argument is the codelet, `sgemm_codelet`. The following arguments are either values (key-word VALUE) or handles (when an access mode is specified). For instance, a value is specified at line 33, corresponding to the content of the `notrans` variable. On the right of line 39, the handle of the tile (m,n) is passed in read-write mode (key-word INOUT). Figure A.3 is a direct translation of the Tile Cholesky decomposition given on Algorithm 3, showing the ease of programmability. Once all tasks have been submitted, the application can perform a barrier using the `starpu_task_wait_for_all()` function (line 52 in Figure A.3).

Choice or design of a scheduling strategy. Once the above steps have been completed, the application is fully defined and can be executed as it is. However, the choice of scheduling strategy may be critical for performance. StarPU provides several built-in, pre-defined strategies the user

[1] Other interfaces not discussed here are also available.

```
1  void sgemm_cpu_func(void *descr[], void *cl_arg)
2  {
3      int transA, transB, M, N, K, LDA, LDB, LDC;
4      float alpha, beta, *A, *B, *C;
5
6      A = STARPU_MATRIX_GET_PTR(descr[0]);
7      B = STARPU_MATRIX_GET_PTR(descr[1]);
8      C = STARPU_MATRIX_GET_PTR(descr[2]);
9
10     starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
11                           &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);
12
13     sgemm(CblasColMajor, transA, transB, M, N, K,
14                           alpha, A, LDA, B, LDB, beta, C, LDC);
15 }
16
17 void sgemm_cuda_func(void *descr[], void *cl_arg)
18 {
19     int transA, transB, M, N, K, LDA, LDB, LDC;
20     float alpha, beta, *A, *B, *C;
21
22     A = STARPU_MATRIX_GET_PTR(descr[0]);
23     B = STARPU_MATRIX_GET_PTR(descr[1]);
24     C = STARPU_MATRIX_GET_PTR(descr[2]);
25
26     starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
27                           &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);
28
29     cublasSgemm(magma_const[transA][0], magma_const[transB][0],
30                           M, N, K, alpha, A, LDA, B, LDB, beta, C, LDC);
31     cudaThreadSynchronize();
32 }
33
34 struct starpu_perfmodel_t cl_sgemm_model = {
35     .type   = STARPU_HISTORY_BASED,
36     .symbol = "sgemm"
37 };
38
39 starpu_codelet sgemm_codelet = {
40     .where     = STARPU_CPU|STARPU_CUDA, // who may execute?
41     .cpu_func  = sgemm_cpu_func,         // CPU implementation
42     .cuda_func = sgemm_cuda_func,        // CUDA implementation
43     .nbuffers  = 3,                      // number of handles accessed by the task
44     .model     = &cl_sgemm_model         // performance model (optional)
45 };
```

Figure A.2: A codelet implementing the *sgemm* kernel.

can select during initialization, depending on the specific requirements of the application. The strategy can either be selected by setting the `STARPU_SCHED` environment variable to the name of a predefined policy, or by specifying which strategy to use when calling `starpu_init`. When the performance of the kernels is stable enough to be predictable directly from the previous executions (as it is the case with Tile Cholesky factorization), one may associate an auto-tuned history-based performance model to a codelet as shown on lines 34-37 and 44 in Figure A.2. If all codelets are associated with a performance model, it is then possible to schedule the tasks according to their expected termination time. The fastest results are obtained with the **heft-tmdp-pr** policy which is described in details in Section 3.6.4.

```
1  void hybrid_cholesky(starpu_data_handle **Ahandles, int M, int N, int Mt, int Nt, int Mb)
2  {
3    int lower = Lower;     int upper = Upper; int right = Right;
4    int notrans = NoTrans; int conjtrans = ConjTrans;
5    int nonunit = NonUnit; float one = 1.0f; float mone = -1.0f;
6
7    int k, m, n, temp;
8    for (k = 0; k < Nt; k++)
9    {
10     temp = k == Mt-1 ? M-k*Mb : Mb ;
11     starpu_insert_task(&spotrf_codelet,
12       VALUE, &lower, sizeof(int), VALUE, &temp, sizeof(int),
13       INOUT, Ahandles[k][k],      VALUE, &Mb, sizeof(int), 0);
14
15     for (m = k+1; m < Nt; m++)
16     {
17       temp = m == Mt-1 ? M-m*Mb : Mb ;
18       starpu_insert_task(&strsm_codelet,
19         VALUE, &right, sizeof(int),    VALUE, &lower, sizeof(int),
20         VALUE, &conjtrans,sizeof(int), VALUE, &nonunit, sizeof(int),
21         VALUE, &temp, sizeof(int),     VALUE, &Mb, sizeof(int),
22         VALUE, &one, sizeof(float),    INPUT, Ahandles[k][k],
23         VALUE, &Mb, sizeof(int),       INOUT, Ahandles[m][k],
24         VALUE, &Mb, sizeof(int),       0);
25     }
26
27     for (m = k+1; m < Nt; m++)
28     {
29       temp = m == Mt-1 ? M-m*Mb : Mb;
30       for (n = k+1; n < m; n++)
31       {
32         starpu_insert_task(&sgemm_codelet,
33           VALUE, &notrans, sizeof(notrans),
34           VALUE, &conjtrans, sizeof(conjtrans),
35           VALUE, &temp, sizeof(int),   VALUE, &Mb, sizeof(int),
36           VALUE, &Mb, sizeof(int),     VALUE, &mone, sizeof(float),
37           INPUT, Ahandles[m][k],       VALUE, &Mb, sizeof(int),
38           INPUT, Ahandles[n][k],       VALUE, &Mb, sizeof(int),
39           VALUE, &one, sizeof(one),    INOUT, Ahandles[m][n],
40           VALUE, &Mb, sizeof(int),     0);
41       }
42
43       starpu_insert_task(&ssyrk_codelet,
44         VALUE, &lower, sizeof(int),  VALUE, &notrans, sizeof(int),
45         VALUE, &temp,  sizeof(int),  VALUE, &Mb, sizeof(int),
46         VALUE, &mone, sizeof(float), INPUT, Ahandles[m][k],
47         VALUE, &Mb, sizeof(int),     VALUE, &one, sizeof(float),
48         INOUT, Ahandles[m][m],       VALUE, &Mb, sizeof(int), 0);
49      }
50   }
51
52   starpu_task_wait_for_all();
53 }
```

Figure A.3: Actual implementation of the tile Cholesky hybrid algorithm with StarPU.

Appendix B

Tuning linear and non-linear regression-based models

B.1 Tuning linear models with the Least Square method

Let us suppose that we have a set of measurements $(p_i, t_i)_{i<n}$ where p_i corresponds to the i-th input parameter (data size by default) and t_i the duration of the i-th sample. We here detail how to model the performance of this kernel with an affine law of the form $ap + b$. The Least Square method permits to find out the optimal a and b parameters, that is to say the values that minimize the following error metric:

$$\min_{a,b} \sum_{i<n} (ap_i + b - t_i)^2 \tag{B.1}$$

If we note $\overline{p}$ (resp. $\overline{y}$) the mean of $(p_i)_{i<n}$ (resp. $(t_i)_{i<n}$), the error is minimized for:

$$b = \frac{\sum_i (t_i - \overline{t})(p_i - \overline{p})}{\sum_i (p_i - \overline{p})^2} \tag{B.2}$$

$$a = \overline{t} - b\overline{p} \tag{B.3}$$

A drawback of this method is that we would need to store all $(p_i, t_i)_{i<n}$ pairs in order to compute the a and b terms. Another limitation is that this formula is not really suited to iteratively refine these parameters every time a new measurement is available (*e.g.* after each task). It is however possible to rewrite equations B.2 and B.3 in a more convenient way:

$$b = \frac{n\sum_i (p_i t_i) - \sum_i p_i \sum_i t_i}{n\sum_i p_i^2 - (\sum_i p_i)^2} \tag{B.4}$$

$$a = \frac{\sum_i t_i - b\sum_i p_i}{n} \tag{B.5}$$

The advantage of this new expression of terms a and b is that we do not have to use $\overline{x}$ and $\overline{y}$ within the different sums. We just need to store the different sums separately (*e.g.* $\sum_i x_i^2$ and $\sum_i (x_i y_i)$), and to update them (in constant time) every time a new sample is available. This method is used in StarPU to dynamically build and maintain performance models based on linear regressions with a low overhead.

The Least Square method can also be applied to exponential models, that is to say which have the following form $\tilde{t} = \alpha p^{\beta}$. We can indeed reduce such models to an affine form by considering their logarithm.

$$ln\left(t\right) = ln\left(\alpha p^{\beta}\right) = ln\left(\alpha\right) + \beta ln\left(p\right) \tag{B.6}$$

We thus obtain an affine model that we approximate by considering the $(ln(p_i), ln(t_i))_{i<n}$ dataset to find out the optimal values of $ln\left(\alpha\right)$ and β.

B.2 Offline algorithm to tune non-linear models

Unfortunately, the linearization technique used in Equation B.6 to tune exponential models (of the form αp^{β}) is not applicable if we want to add a constant factor to model the constant overhead typically introduced by kernel launch latency ($\alpha p^{\beta} + \gamma$)[1]. In this case, we actually have to perform a non-linear regression which is much heavier to implement. While being much more expensive, adding such a constant parameter is crucial to make meaningful predictions, particularly for small inputs, when kernel launch overhead is non negligible. Let us suppose that we have $\tilde{t} = \alpha p^{\beta} + \gamma$. Then:

$$ln\left(\tilde{t} - \gamma\right) = ln\left(\alpha\right) + \beta ln\left(p\right) \tag{B.7}$$

The first step of our algorithm consists in finding the optimal γ value. Since there is an affine relationship between $n\left(p\right)$ and $ln\left(\tilde{t} - \gamma\right)$ in the $ln\left(\alpha\right)+\beta ln\left(p\right)$ term, the optimal γ value is obtained when the $ln\left(p\right) \rightarrow ln\left(\tilde{t} - \gamma\right)$ function is the closest possible to an affine function.

From the point of view of statistics, the function that is the most similar to an affine function is obtained when the absolute value of the correlation coefficient between $ln\left(p\right)$ and $ln\left(\tilde{t} - \gamma\right)$ is the closest to 1.

$$r\left(\left\{\left(x_i, y_i\right)_{i<n}\right\}\right) = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - \left(\sum x_i\right)^2}\sqrt{n\sum y_i^2 - \left(\sum y_i\right)^2}} \tag{B.8}$$

In order to find the optimal γ value, we therefore perform a dichotomy to maximize the correlation coefficient of the $\left(ln\left(p_i\right), ln\left(t_i - \gamma\right)\right)_{i<n}$ dataset[2].

Since γ represents the constant overhead found in every measurement, its value is comprised between 0 and the duration of the shortest measurement. We thus initialize the dichotomy procedure in this range. As shown on Algorithm 4, the dichotomy actually consists in taking the average value in the current interval, and to compute whether the slope of the absolute value of the regression coefficient is directed to the left or to the right. Once an optimal value has been found for γ, we apply the Least Square method to fit $ln\left(\alpha\right) + \beta ln\left(p\right)$ with the optimal affine function.

[1] In case β is known in advance, one should use an affine law and specify that the input parameter is p^{β}.

[2] We admit that $\gamma \rightarrow r\left(\left(ln\left(p_i\right), ln\left(t_i - \gamma\right)\right)_{i<n}\right)$ is a concave function in our case.

Algorithm 4: Auto-tuning of the α, β and γ terms for non-linear models of the form $\alpha p^{\beta} + \gamma$.

begin
  $\gamma_{min} \leftarrow 0$;
  $\gamma_{max} \leftarrow \min_i (t_i)$;
  $\eta \leftarrow 0.05$
  while $(\gamma_{max} - \gamma_{min}) > \epsilon$ **do**
    $\gamma_{left} \leftarrow \frac{1}{2}(\gamma_{max} + \gamma_{min}) - \eta(\gamma_{max} - \gamma_{min})$
    $\gamma_{right} \leftarrow \frac{1}{2}(\gamma_{max} + \gamma_{min}) + \eta(\gamma_{max} - \gamma_{min})$
    $r_{left} \leftarrow r(\{ln(p_i), ln(t_i - \gamma_{left})\}_{i<n})$
    $r_{right} \leftarrow r(\{ln(p_i), ln(t_i - \gamma_{left})\}_{i<n})$
    if $|r_{left}| > |r_{right}|$ **then**
      $\gamma_{min} \leftarrow \frac{1}{2}(\gamma_{max} + \gamma_{min})$
    else
      $\gamma_{max} \leftarrow \frac{1}{2}(\gamma_{max} + \gamma_{min})$
  $\gamma \leftarrow \frac{1}{2}(\gamma_{max} + \gamma_{min})$
  $\beta \leftarrow \frac{\sum_i (t_i - \bar{t})(p_i - \bar{p})}{\sum_i (p_i - \bar{p})^2}$
  $\alpha \leftarrow \bar{t} - \gamma - \beta\bar{p}$

Appendix C

Bibliography

[1] Top500 supercomputer sites. `http://www.top500.org`.

[2] *Running Unstructured Grid CFD Solvers on Modern Graphics Hardware*, June 2009. AIAA 2009-4001.

[3] *Tile QR Factorization with Parallel Panel Processing for Multicore Architectures*, 2010.

[4] José Nelson Amaral, editor. *Languages and Compilers for Parallel Computing*, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Franois Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Batrice Creusillet, and Serge Guelton. PIPS Is not (just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[6] Mehdi Amini, Fabien Coelho, Franois Irigoin, and Ronan Keryell. Static compilation analysis for host-accelerator communication optimization. In *24th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Fort Collins, Colorado, USA, September 2011. Also Technical Report MINES ParisTech A/476/CRI.

[7] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[8] Diego Andrade, Basilio B. Fraguela, James Brodman, and David Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 101–110, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[10] Apple, Inc. Apple Technical Brief on Grand Central Dispatch. `http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf`, March 2009.

[11] Apple, Inc. Introducing Blocks and Grand Central Dispatch. `http://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/`, August 2010.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[13] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A proposal to extend the openmp tasking model for heterogeneous architectures. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.

[15] Rosa M. Badia, Jess Labarta, Ral Sirvent, Josep M. Prez, Jos M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *Journal of Grid Computing*, 1:151–170, 2003. 10.1023/B:GRID.0000024072.93701.f3.

[16] J. Balart, A. Duran, M. Gonzlez, X. Martorell, E. Ayguad, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *European Workshop on OpenMP (EWOMP'04). Pp*, pages 103–109, 2004.

[17] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008. `http://www.bsc.es/media/1002.pdf`.

[18] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[19] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *CC'10*, pages 244–263, 2010.

[20] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: supporting irregular parallelism in sequoia. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 13–24, New York, NY, USA, 2011. ACM.

[21] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Exploiting Locality on the Cell/B.E. through Bypassing. In *SAMOS*, pages 318–328, 2009.

[22] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. Cellss: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[23] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[24] OpenMP Architecture Review Board. OpenMP. `http://www.openmp.org/`.

[25] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack Dongarra. Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA, 2010.

[26] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing, SEP 2010. UT-CS-10-659.

[27] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prelle. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. In *Proc. of First Workshop on Software Tools for Multi-Core Systems*, New York, NY, USA, March 2006. Mercury Computer Systems.

[28] Thomas Brandes. Exploiting advanced task parallelism in high performance fortran via a task library. In Patrick Amestoy, Philippe Berger, Michel Dayd, Daniel Ruiz, Iain Duff, Valrie Frayss, and Luc Giraud, editors, *Euro-Par99 Parallel Processing*, volume 1685 of *Lecture Notes in Computer Science*, pages 833–844. Springer Berlin / Heidelberg, 1999.

[29] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-André Wacrenier, and Raymond Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, April 2010. IEEE Computer Society Press.

[30] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press.

[31] Franois Broquedis, Nathalie Furmento, Brice Goglin, Pierre-Andr 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. Mller and Eduard Ayguad*, 38(5):418–439, 2010.

[32] Élisabeth Brunet, François Trahay, Alexandre Denis, and Raymond Namyst. A sampling-based approach for communication libraries auto-tuning. In *IEEE International Conference on Cluster Computing*, Austin, États-Unis, September 2011. http://hal.inria.fr/inria-00605735/en/.

[33] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[34] Javier Bueno, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Poster: programming clusters of gpus with ompss. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 378–378, New York, NY, USA, 2011. ACM.

[35] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures, 2007.

[36] Dan Campbell. Vsipl++ acceleration using commodity graphics processors. In *Proceedings of the HPCMP Users Group Conference*, pages 315–320, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Louis-Claude Canon and Emmanuel Jeannot. Evaluation and optimization of the robustness of dag schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 99(RapidPosts):532–546, 2009.

[38] Paul Carpenter. *Running Stream-like Programs on Heterogeneous Multi-core Systems*. PhD thesis, Universitat Polit'ecnica de Catalunya, 2011.

[39] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *In Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 1989.

[40] Ernie Chan. Runtime data flow scheduling of matrix computations, 2009.

[41] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0:44–54, 2009.

[42] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, April 2010.

[43] ClearSpeed Inc. Clearspeed inc. `http://www.clearspeed.com`.

[44] Sylvain Collange, David Defour, and Arnaud Tisserand. Power Consumption of GPUs from a Software Perspective. In *9th International Conference on Computational Science*, pages 914–923, Baton Rouge, Louisiana, USA, May 2009.

[45] Intel Corp. A Quick, Easy and Reliable Way to Improve Threaded Performance: Intel Cilk Plus, 2010.

[46] M. Cosnard and M. Loi. Automatic task graph generation techniques. *Hawaii International Conference on System Sciences*, 0:113, 1995.

[47] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 413–, Washington, DC, USA, 1999. IEEE Computer Society.

[48] Kevin Coulomb, Mathieu Faverge, Johnny Jazeix, Olivier Lagrasse, Jule Marcoueille, Pascal Noisette, Arthur Redondy, and Clment Vuchener. Vite's project page. `http://vite.gforge.inria.fr/`, 2009–2011.

[49] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *CF '08*, pages 3–12, 2008.

[50] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *EuroPar*, Lisbonne, Portugal, September 2005.

[51] DARPA. Dri-1.0 specification, September 2002.

[52] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceeding of the 4th international workshop on Multicore software engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[53] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-avoiding parallel and sequential qr factorizations. Technical report, EECS, UC Berkeley, 2008.

[54] 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, 2008.

[55] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.

[56] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *Micro, IEEE*, 18(2):66 –76, mar/apr 1998.

[57] Alejandro Duran, Roger Ferrer, Michael Klemm, Bronis de Supinski, and Eduard Ayguad. A proposal for user-defined reductions in openmp. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias Mller, Barbara Chapman, and Bronis de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin / Heidelberg, 2010.

[58] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[60] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[61] Johan Enmyren and Christoph W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[62] Naila Farooqui, Andrew Kerr, Gregory Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 9:1–9:9, New York, NY, USA, 2011. ACM.

[63] J.-P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse. Gpucv: A framework for image processing acceleration with graphics processors. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 585 –588, july 2006.

[64] 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.

[65] Massimiliano Fatica. Accelerating linpack with cuda on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, New York, NY, USA, 2009. ACM.

[66] P. Ferraro, P. Hanna, L. Imbert, and T. Izard. Accelerating query-by-humming on GPU. In *Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR'09)*, pages 279–284, Kobe, Japan, October 2009.

[67] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. `http://software.intel.com/en-us/articles/intel-avx-newfrontiers-in-performance-improvements-and-energyefficiency/`. White Paper.

[68] High Performance Forum. High performance fortran language specification. Technical Report CRPC-TR92225, Center For Research on Parallel Computing, Rice University, Houston, Tx, May 1993.

[69] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[70] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[71] F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 88 –95, oct 1998.

[72] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. Mpi-2: Extending the message-passing interface. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel*

Processing, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135. Springer Berlin / Heidelberg, 1996.

[73] Isaac Gelado, Javier Cabezas, John E. Stone, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, Pittsburgh, PA, USA, March 2010.

[74] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGARCH Comput. Archit. News*, 38:347–358, March 2010.

[75] Luigi Genovese, Matthieu Ospici, Thierry Deutsch, Jean-Francois Méhaut, Alexey Neelov, and Stefan Goedecker. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J Chem Phys*, 131(3):034103, 2009.

[76] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 199–210, New York, NY, USA, 2005. ACM.

[77] GNU Project. Plugins - GNU Compiler Collection (GCC) Internals. `http://gcc.gnu.org/onlinedocs/gccint/Plugins.html`, 2011.

[78] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40:1135–1160, November 2010.

[79] J. Greene, C. Nowacki, and M. Prelle. Pas: A parallel applications system for signal processing applications. In *International Conference on Signal Processing Applications and Technology*, 1996.

[80] Chris Gregg, Jeff Brantley, and Kim Hazelwood. Contention-aware scheduling of parallel code for heterogeneous systems. In *2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar, Berkeley, CA, June 2010.

[81] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[82] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[83] The Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. http://khronos.org/opencl/.

[84] Liang Gu, Jakob Siegel, and Xiaoming Li. Using gpus to compute large out-of-card ffts. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264, New York, NY, USA, 2011. ACM.

[85] Serge Guelton, Franois Irigoin, and Ronan Keryell. Automatic code generation for simd hardware accelerators, 2010.

[86] J. M Hammersley and D. C. Handscomb. *Monte Carlo methods*. Methuen, 1964.

[87] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[88] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[89] Sylvain Henry. OpenCL as StarPU frontend. `http://runtime.bordeaux.inria.fr/shenry/papers/HS_SOCL.pdf`.

[90] Everton Hermann, Bruno Raffin, Franois Faure, Thierry Gautier, and Jrmie 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.

[91] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[92] Torsten Hoefler, Prabhanjan Kambadur, Richard Graham, Galen Shipman, and Andrew Lumsdaine. A Case for Standard Non-blocking Collective Operations. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 125–134. Springer Berlin / Heidelberg, 2007.

[93] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. *SIGARCH Comput. Archit. News*, 38:280–289, June 2010.

[94] Chao Huang, Orion Lawlor, and L. Kal. Adaptive MPI. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 306–322. Springer Berlin / Heidelberg, 2004.

[95] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7705, April 2010.

[96] Innovative Computing Laboratory (ICL). MAGMA. `http://icl.cs.utk.edu/magma/`.

[97] Innovative Computing Laboratory (ICL). PLASMA. `http://icl.cs.utk.edu/plasma/`.

[98] IMPACT group, University of Illinois. The parboil benchmark suite, 2007. `http://impact.crhc.illinois.edu/parboil.php`.

[99] IBM Inc. SPE Runtime Management Library. Technical report, IBM Corp., 2006.

[100] NVIDIA inc. *NVIDIA CUDA FFT library - CUFFT 4.0*.

[101] Toshiba Inc. Quad Core HD Processor: SpursEngine. `http://www.semicon.toshiba.co.jp/eng/product/assp/selection/spursengine/index.html`, 2008.

[102] Intel Corp. Intel Ct Technology. `http://software.intel.com/en-us/data-parallel/`.

[103] Intel Corp. Intel Threading Building Blocks. `http://www.threadingbuildingblocks.org/`.

[104] Intel Corp. Single-chip cloud computer, DEC 2009. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[105] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[106] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC*, pages 19–33, 2009.

[107] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[108] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[109] Laxmikant V. Kale, David M. Kunzman, and Lukasz Wesolowski. Accelerator Support in the Charm++ Parallel Programming Model. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, pages 393–412. CRC Press, Taylor & Francis Group, 2011.

[110] Jacques Chassin De Kergommeaux, Benhur De Oliveira Stein, and Montbonnot Saint Martin. Paje: An extensible environment for visualizing multi-threaded program executions. In *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, pages 133–144, 1900.

[111] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[112] Volodymyr V. Kindratenko and Robert J. Brunner. Accelerating cosmological data analysis with fpgas. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:11–18, 2009.

[113] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR*, 2009.

[114] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[115] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43:114–124, June 2008.

[116] D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the PMUP Workshop*, Seattle, WA, USA, September 2006.

[117] David Kunzman. Charm++ on the Cell Processor. Master's thesis, Dept. of Computer Science, University of Illinois, 2006.

[118] J. Kurzak, A. Buttari, and J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1175–1186, 2008.

[119] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19:1175–1186, 2008.

[120] Jakub Kurzak and Jack Dongarra. Implementation of the mixed-precision high performance. In *LINPACK Benchmark on the CELL Processor, University of Tennessee Computer Science, Tech. Rep. UT-CS-06-580, LAPACK Working Note 177*, 2006.

[121] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In Bo Kgstrm, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin / Heidelberg, 2007.

[122] Jakub Kurzak, Rajib Nath, Peng Du, and Jack Dongarra. An implementation of the tile qr factorization for a gpu and multiple cpus. Technical Report 229, LAPACK Working Note, September 2010.

[123] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[124] O.S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –8, 31 2009-sept. 4 2009.

[125] Fabian Lecron, Sidi Ahmed Mahmoudi, Mohammed Benjelloun, Saïd Mahmoudi, and Pierre Manneback. Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images. *International Journal of Biomedical Imaging: Parallel Computation in Medical Imaging Applications*, 07 2011.

[126] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. COMIC: a coherent shared memory interface for Cell BE. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 303–314, New York, NY, USA, 2008. ACM.

[127] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[128] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44:227–242, October 2009.

[129] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceeding of ICCS'09, Baton Rouge, Louisiana, U.S.A.*, 2009.

[130] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[131] Michael David Linderman. *A programming model and processor architecture for heterogeneous multicore computers*. PhD thesis, Stanford University, Stanford, CA, USA, 2009. AAI3351459.

[132] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. A scalable high performant cholesky factorization for multicore with gpu accelerators. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 93–101, Berlin, Heidelberg, 2011. Springer-Verlag.

[133] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[134] Sidi Ahmed Mahmoudi, Fabian Lecron, Pierre Manneback, Mohammed Benjelloun, and Saïd Mahmoudi. GPU-Based Segmentation of Cervical Vertebra in X-Ray Images. In *IEEE International Conference on Cluster Computing*, Crete. Greece, 09 2010.

[135] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[136] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[137] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform, 2006.

[138] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11):648–662, 2007.

[139] Stéphanie Moreaud, Brice Goglin, and Raymond Namyst. Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access. In Edgar Gabriel Rainer Keller and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface. The 17th European MPI User's Group Meeting (EuroMPI 2010)*, volume 6305 of *Lecture Notes in Computer Science*, pages 239–248, Stuttgart, Germany, September 2010. Springer-Verlag. Best paper award.

[140] Movidius Corp. Movidius - The Mobile Video Processor Company. `http://www.movidius.com`.

[141] Raymond Namyst and Jean-Franois Mhaut. *Marcel : Une bibliothque de processus lgers*. LIFL, Univ. Sciences et Techn. Lille, 1995.

[142] Rajib Kumar Nath. Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an Autotuned and Algorithmic Approach. Master's thesis, Dept. of Computer Science, University of Tennessee, Knoxville, August 2010.

[143] Maik Nijhuis, Herbert Bos, and Henri E. Bal. A component-based coordination language for efficient reconfigurable streaming applications. In *ICPP*, page 60, 2007.

[144] NOAA Earth System Research Laboratory. The f2c-acc fortran-to-cuda compiler. `http://www.esrl.noaa.gov/gsd/ab/ac/F2C-ACC.html`.

[145] NVIDIA Corp. Fermi Compute Architecture White Paper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[146] NVIDIA Corp. NVIDIA CUDA Toolkit v4.0 Release Notes for Windows, Linux, and Mac OS X. `http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_Toolkit_Release_Notes.txt`.

[147] NVIDIA Corp. Nvidia npp library. `http://www.nvidia.com/object/npp.html`.

[148] NCSA University of Illinois at Urbana Champaign. Gpu computing. `http://gpucomputing.net/`.

[149] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –10, April 2008.

[150] OpenMP Architecture Review Board. OpenMP Program Interface Version 3.0, May 2008.

[151] 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.

[152] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, 2010.

[153] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen mei W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *SASP*, pages 35–42, 2009.

[154] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer Berlin / Heidelberg, 1996.

[155] Petitet, A., R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers - Version 2. `http://netlib.org/benchmark/hpl/`, 2008.

[156] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor - a multi-core soc. In *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on*, pages 49 – 52, may 2005.

[157] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierchical task based programming with StarSs. *International Journal of High Performance Computing Application*, 23:284, 2009.

[158] Antoniu Pop and Albert Cohen. A stream-computing extension to openmp. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[159] FLAME Project, MAY 2010.

[160] Filadelfio Puglisi, Renzo Ridi, Francesca Cecchi, Aurelio Bonelli, and Robert Ferrari. Segmental vertebral motion in the assessment of neck range of motion in whiplash patients. *International Journal of Legal Medicine*, 118(4):235–9, 2004.

[161] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

[162] Bertrand Putigny. Optimisation de code sur processeur Cell. Master's thesis, Laboratoire PRISM, Université Versailles Saint-Quentin-En-Yvelines, 2009.

[163] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:427–436, 2009.

[164] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM.

[165] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26:28–38, June 1993.

[166] Dylan W. Roeh, Volodymyr V. Kindratenko, and Robert J. Brunner. Accelerating cosmological data analysis with graphics processors. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 1–8, New York, NY, USA, 2009. ACM.

[167] S. Rosario-Torres and M. Velez-Reyes. Speeding up the matlab hyperspectral image analysis toolbox using gpus and the jacket toolbox. In *Hyperspectral Image and Signal Processing: Evolution in Remote Sensing, 2009. WHISPERS '09. First Workshop on*, pages 1 –4, August 2009.

[168] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[169] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[170] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 733–740, Washington, DC, USA, 2010. IEEE Computer Society.

[171] Fethulah Smailbegovic, Georgi N. Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format.

[172] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[173] Kyle Spafford, Jeremy S. Meredith, and Jeffrey S. Vetter. Quantifying NUMA and contention effects in multi-GPU systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 11:1–11:7, New York, NY, USA, 2011. ACM.

[174] John Stratton, Sam Stone, and Wen-mei Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In Jos Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 2008.

[175] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*, ICS '96, pages 18–25, New York, NY, USA, 1996. ACM.

[176] Mellanox Technologies. NVIDIA GPUDirect Technology Accelerating GPU-based Systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`, May 2010.

[177] George Teodoro, Timothy D. R. Hartley, Umit Catalyurek, and Renato Ferreira. Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of*

the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 13–24, New York, NY, USA, 2010. ACM.

[178] The PIPS team. Pips: Automatic parallelizer and code transformation framework. `http://pips4u.org`.

[179] The Portland Group. PGI Fortran & C Accelerator Programming Model white paper. `http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.0.pdf`.

[180] The Portland Group. Pgi cuda-x86. `http://www.pgroup.com/resources/cuda-x86.htm`, 2011.

[181] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *Proceedings of the 13th International Euro-par Conference*, volume 4641 of *Lecture Notes in Computer Science*, Rennes,France, 8 2007. ACM, Springer.

[182] 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, April 2010.

[183] 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.

[184] François Trahay and Alexandre Denis. A scalable and generic task scheduling system for communication libraries. In *Proceedings of the IEEE International Conference on Cluster Computing*, New Orleans, LA, September 2009. IEEE Computer Society Press.

[185] George Tzenakis, Konstantinos Kapelonis, Michail Alvanos, Konstantinos Koukos, Dimitrios S. Nikolopoulos, and Angelos Bilas. Tagged Procedure Calls (TPC): Efficient Runtime Support for Task-Based Parallelism on the Cell Processor. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2010.

[186] Stanly Tzeng, Anjul Patney, and John D. Owens. Poster: Task management for irregular workloads on the gpu. In *Proceeding of NVIDIA GPU Technology Conference*, 2010.

[187] John D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.

[188] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[189] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, jan. 2008.

[190] Jeffrey S. Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, and Sudhakar Yalamanchili. Keeneland: Bringing heterogeneous gpu computing to the computational science community. *Computing in Science Engineering*, 13(5):90 –95, sept.-oct. 2011.

[191] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.

[192] Richard Vuduc, James Demmel, and Katherine Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC'05*, Journal of Physics: Conference Series, San Francisco, CA, June 2005. Institute of Physics Publishing.

[193] Lukasz Wesolowski. An Application Programming Interface for General Purpose Graphics Processing Units in an Asynchronous Runtime System. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. `http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml`.

[194] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[195] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *International Parallel And Distributed Processing Symposium (IPDPS)*, Miami, 2008.

[196] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31:50–59, March 2011.

[197] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: a fast flexible architecture for secure communication. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 110 –119, 2001.

[198] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels.

[199] Baida Zhang, Shuai Xu, Feng Zhang, Yuan Bi, and Linqi Huang. Accelerating matlab code using gpu: A review of tools and strategies. In *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, pages 1875 –1878, August 2011.

Appendix D

Publications

[AAD+10a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, USA, 07 2010.

[AAD+10b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, 09 2010.

[AAD+11a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Egypt, June 2011. Best Paper Award.

[AAD+11b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, 5 2011.

[ACOTN10] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, December 2010.

[AN08] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, volume 5415 of *Lecture Notes in Computer Science*, pages 174–183, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proceedings of*

the International Euro-Par Workshops 2009, HPPC'09, volume 6043 of *Lecture Notes in Computer Science*, pages 56–65, Delft, The Netherlands, August 2009. Springer.

[ATN10] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, INRIA, March 2010.

[ATNN09] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *SAMOS Workshop - International Workshop on Systems, Architectures, Modeling, and Simulation*, Lecture Notes in Computer Science, Samos, Greece, July 2009.

[ATNW09] 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 International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, August 2009. Springer.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[Aug08] Cédric Augonnet. Vers des supports d'exécution capables d'exploiter les machines multicoeurs hétérogènes. Mémoire de DEA, Université Bordeaux 1, June 2008.

[Aug09] Cédric Augonnet. StarPU: un support exécutif unifié pour les architectures multicœurs hétérogènes. In *19èmes Rencontres Francophones du Parallélisme*, Toulouse / France, September 2009. Best Paper Award.

[BPT+11] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, 31(5):28–41, 2011.

[NBBA09] Maik Nijhuis, Herbert Bos, Henri E. Bal, and Cédric Augonnet. Mapping and synchronizing streaming applications on cell processors. In *HiPEAC*, pages 216–230, 2009.

[Web] INRIA RUNTIME Team Website. Starpu: A unified runtime system for heterogeneous multicore architectures. `http://runtime.bordeaux.inria.fr/StarPU/`.