Apprehending heterogeneity at (very) large scale

Raphaël Bleuse

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Raphaël BLEUSE

Thèse dirigée par **Denis TRYSTRAM**
et codirigée par **Grégory MOUNIÉ**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'École Doctorale **MSTII**

Appréhender l'hétérogénéité à (très) grande échelle

Apprehending heterogeneity at (very) large scale

Thèse soutenue publiquement le **11 octobre 2017**,
devant le jury composé de :

Lionel EYRAUD-DUBOIS
Chargé de Recherche, LaBRI, Inria, France, Examinateur

Nectarios KOZIRIS
Professeur d'Université, School of Electrical and Computer Engineering, National Technical University of Athens, Grèce, Rapporteur

Vitus J. LEUNG
Principal Member of Technical Staff, Sandia National Laboratories, États-Unis, Rapporteur

Grégory MOUNIÉ
Maître de Conférence, LIG, Univ. Grenoble Alpes, France, Co-Directeur de thèse

Alix MUNIER
Professeure d'Université, LIP6, UPMC, France, Examinatrice

Yves ROBERT
Professeur d'Université, LIP, ENS Lyon, France, Président

Denis TRYSTRAM
Professeur d'Université, LIG, Univ. Grenoble Alpes, France, Directeur de thèse

À cette grande mère, Mimi.

L'expression est œuvre difficile, lente et tortueuse, — et l'erreur est de croire que n'est pas ce qui ne peut d'abord s'énoncer.

— **Antoine de Saint-Exupéry**

Remerciements

(Acknowledgments)

I would like to first thank the jury members, and especially the two reviewers Nectarios KOZIRIS and Vitus J. LEUNG who provided insightful comments on my dissertation. I would like to thank the DGA-MRIS for the scholarship they granted me. I would like to thank the JLESC for funding my travels to the USA. I am grateful to the NCSA members who welcomed me; notably Greg BAUER, Jeremy ENOS, Andriy KOT, and Sharif ISLAM.

Mes travaux de thèse ne seraient pas ce qu'ils sont sans Denis et Grégory. Je souhaite chaudement remercier Denis pour ses conseils avisés et sa patience. Je remercie Grégory pour son enthousiame, son optimisme à toute épreuve et son foisonnement d'idées. Je souhaite aussi vivement remercier les collègues des deux équipes MESCAL/MOAIS (ou plutôt DATAMOVE/POLARIS) pour l'environnement exceptionnel de recherche qu'ils arrivent à faire vivre. Merci en particulier à Jean-Marc de m'avoir integré dans ton effort de médiation. Merci aussi à Olivier et Florence pour ces discussions de cafét' pendant cet été de rédaction. Annie tient aussi une place spéciale au travers du soutien quotidien qu'elle apporte aux deux équipes. Merci à Alexis, David et David, thésards du bureau 443 pour ces moments de travail/rédaction/codage/discussion/etc. partagés. Je ne remercie pas Millian afin de mieux le troller[1]. Une attention spéciale va à Fernando, en qui j'ai trouvé un frère brésilien habile du fer souder mais rétif au français.

Un grand merci aussi à mes parents ainsi qu'à mes deux sœurs pour leur compagnie, soutien, dessins, etc.

Je tiens aussi à remercier mes compagnes et compagnons de liberté—que sont notamment Hélène, Manu, Pauline et Sylvain-Lio'—pour ces aventures verticales et spirituelles qui ont marqué ces dernières années.

Enfin, merci à Mathou d'Être qui Elle Est.

[1] Vous aurez, bien entendu, saisi le ton ironique.

Abstract / Résumé

Abstract

The demand for computation power is steadily increasing, driven by the need to simulate more and more complex phenomena with an increasing amount of consumed/produced data. To meet this demand, the High Performance Computing platforms grow in both size and heterogeneity. Indeed, heterogeneity allows splitting problems for a more efficient resolution of sub-problems with *ad hoc* hardware or algorithms. This heterogeneity arises in the platforms' architecture and in the variety of processed applications. Consequently, the performances become more sensitive to the execution context.

We study in this dissertation how to qualitatively bring—at a reasonable cost—context-awareness/obliviousness into allocation and scheduling policies. This study is conducted from two standpoints: within single applications, and at the whole platform scale from an inter-applications perspective.

We first study the minimization of the *makespan* of sequential tasks on platforms with a mixed architecture composed of multiple CPUs and GPUs. We integrate context-awareness into schedulers with an affinity mechanism that improves local behavior. This mechanism has been implemented in a parallel run-time, and experiments show that it is able to reduce the memory transfers while maintaining a low makespan. We then extend the model to implicitly consider parallelism on the CPUs with the moldable-task model. We propose an efficient algorithm formulated as an integer linear program with a constant performance guarantee of $\frac{3}{2} + \epsilon$.

Second, we devise a new modeling framework where constraints are a first-class tool. Rather than extending existing models to consider all possible interactions, we reduce the set of feasible schedules by further constraining existing models. We propose a set of reasonable constraints to model application spreading and I/O traffic. We then instantiate this framework for unidimensional topologies, and propose a comprehensive case study of the makespan minimization under convex and local constraints.

Résumé

Le besoin de simuler des phénomènes toujours plus complexes accroît les besoins en puissance de calcul, tout en consommant et produisant de plus en plus de données. Pour répondre à cette demande, la taille et l'hétérogénéité des plateformes de calcul haute performance augmentent. L'hétérogénéité permet en effet de découper les problèmes en sous-problèmes, pour lesquels du matériel ou des algorithmes *ad hoc* sont plus efficients. Cette hétérogénéité se manifeste dans l'architecture des plateformes et dans la variété des applications exécutées. Aussi, les performances sont de plus en plus sensibles au contexte d'exécution.

L'objet de cette thèse est de considérer, qualitativement et à faible coût, l'impact du contexte d'exécution dans les politiques d'allocation et d'ordonnancement. Cette étude est menée à deux niveaux : au sein d'applications uniques, et à l'échelle des plateformes au niveau inter-applications.

Nous étudions en premier lieu la minimisation du temps de complétion pour des tâches séquentielles sur des plateformes hybrides intégrant des CPU et des GPU. Nous proposons de tenir compte du contexte d'exécution grâce à un mécanisme d'affinité améliorant le comportement local des politiques d'ordonnancement. Ce mécanisme a été implémenté dans un *run-time* parallèle. Une campagne d'expérience montre qu'il permet de diminuer les transferts de données tout en conservant un faible temps de complétion. Puis, afin de prendre implicitement en compte le parallélisme sur les CPU, nous enrichissons le modèle en considérant les tâches comme *moldables* sur CPU. Nous proposons un algorithme basé sur la programmation linéaire en nombres entiers. Cet algorithme efficace a un rapport de compétitivité de $\frac{3}{2} + \epsilon$.

Dans un second temps, nous proposons un nouveau cadre de modélisation dans lequel les contraintes sont des outils de premier ordre. Plutôt que d'étendre les modèles existants en considérant toutes les interactions possibles, nous réduisons l'espace des ordonnancements réalisables via l'ajout de contraintes. Nous proposons des contraintes raisonnables pour modéliser l'étalement des applications ainsi que les flux d'E/S. Nous proposons ensuite une étude de cas exhaustive dans le cadre de la minimisation du temps de complétion pour des topologies unidimensionnelles, sous les contraintes de convexité et de localité.

Contents

Acknowledgments **v**

Abstract / Résumé **vii**

Contents **ix**

1 Introduction **1**
1.1 Background . . . 1
1.2 Contextualization . . . 2
1.2.1 Intra-Application Level . . . 5
1.2.2 Inter-Applications Level . . . 5
1.3 Explicit *vs.* Implicit Modeling of Communications . . . 6
1.3.1 Existing Explicit Models . . . 7
1.3.2 Limits of Explicit Models . . . 8
1.3.3 Implicit Modeling . . . 8
1.4 Contributions . . . 9
1.5 Content . . . 12

2 Scheduling Independent Sequential Tasks on Multi-Cores with GPUs **15**
2.1 Problem Definition . . . 17
2.2 Related Work . . . 17
2.2.1 Algorithmic Results . . . 17
2.2.2 Parallel Run-times . . . 19
2.3 XKaapi Scheduling Framework . . . 20
2.3.1 Execution Flow . . . 21
2.3.2 Performance Model . . . 21
2.4 Scheduling Policies . . . 22
2.4.1 HEFT: Heterogeneous Earliest-Finish-Time . . . 22
2.4.2 Dual Approximation Based Algorithms . . . 23
Pure Dual Approach . . . 24
Bringing Context-Awareness in: Affinity . . . 24
2.5 Usability of Scheduling Policies for Linear Algebra . . . 27

2.6 Performance Evaluation 29
2.6.1 Experimental Setup: Platform and Benchmarks 29
Platform 29
Benchmarks 29
Methodology 30
2.6.2 Impact of the Affinity Control Parameter α 30
2.6.3 Comparison of Scheduling Policies 31
Experimental Evaluation 32
Discussion 32
2.7 Summary 34

3 Scheduling Independent Moldable Tasks on Multi-Cores with GPUs 37
3.1 Problem Definition 38
3.2 Related Work 40
3.3 Algorithm APPROX-3/2 41
3.3.1 Partitioning Tasks 41
3.3.2 Mathematical Formulation 43
Objective Function and Constraints 43
Filtering 44
Integer Linear Program 46
3.4 Analysis of the Algorithm APPROX-3/2 46
3.4.1 Structure of a Schedule of Makespan λ 47
3.4.2 Structure of the Partitioning 48
3.4.3 Correctness of the Dual Approximation 51
3.4.4 Building the Schedule 54
3.5 Algorithm APPROX-2 54
3.5.1 Sketch 55
3.5.2 Analysis 55
3.6 Experimental Evaluation 56
3.6.1 Problem Instances 56
3.6.2 HEFT-like Heuristics 59
3.6.3 Implementation Details 60
3.6.4 Experimental Results 61
3.7 Summary 65

4 Geometric Constraints as a First-Class Modeling Tool 67
4.1 General Problem Setting 68
4.1.1 Intrinsic Constraints 70
4.1.2 Extrinsic Metrics 74

4.1.3 Extension of Graham Notation 75
4.2 Related Work 76

5 Unidimensional Problem's Instantiations 79
5.1 Formal Instantiation 79
5.1.1 Structural Properties 80
5.2 Study of *unpinned* I/O 81
5.2.1 Complexity 81
5.2.2 Meta Approximation Algorithm 82
Sketch 83
Analysis 83
5.3 Study of *pinned* I/O 84
5.3.1 Complexity 84
5.3.2 Approximation Algorithm 86
Algorithm with a single I/O node per job 86
Analysis with a single I/O node per job 90
Extending to any number of I/O nodes per job 92
5.4 Summary 95

6 Conclusion and Future Steps 97
6.1 Future Steps 98

Bibliography A1

List of Figures A11

List of Tables A13

1 Introduction

1.1 Background

In **H**igh **P**erformance **C**omputing (HPC), the demand for computation power is steadily increasing, driven by the need to simulate more and more complex phenomena (structures at the atomic level, protein folding, fluid dynamics, etc.) with an increasing amount of consumed/produced data. In the near future, the great challenge for the HPC community is to build the ecosystem for both *exaflop* ($10^{18}\,\mathrm{Flop/s}$) and *sustained petaflop* ($10^{15}\,\mathrm{Flop/s}$) performance levels. To measure up to such a challenge while keeping both construction and operating costs at a minimum, the HPC platforms (or supercomputers) require disruptive evolutions [Don+11]. Among these evolutions, two aspects require particular attention: *scale* and *heterogeneity*.

First, the extreme scale of the platforms is a challenge on its own. To absorb the demand in computing power, the platforms now integrate about ten million computing cores [@top500]. Still, increasing the number of cores is not sufficient as it only tackles the question of the available raw theoretical processing power. At such an extreme scale, the issues of feeding input data to the cores and retrieving the produced resulting data become the predominant difficulty. Due to the cost and scaling constraints, most of the current supercomputers have a unique and multi-purpose interconnection network. These networks, such as the DragonFly or SlimFly interconnects [Kat+15], are usually organized as very dense local subnetworks that are interconnected by a global sparse network. Such topologies further emphasize the effects of locality.

Second, the platforms and their usages are growing in heterogeneity. This heterogeneity arises from various factors:

- The tremendous amount of data transiting in the platforms requires dedicated nodes within the platform to handle the generated load. It is indeed impossible for the computing nodes to keep in their local memories all the data required for computations.
- The platforms are now hybrid systems built with computing cores of various architectures. For example, the platforms may integrate slow general purpose

cores, or CPUs; efficient SIMD cores, such as GPUs or MICs; programmable cores dedicated to complex specific computations, or FPGAs; etc.

- The various network traffics induced by the applications share a unique network resource.
- The applications running on the platforms vary in many ways: their size, the time to process them, their communication patterns, etc.

As of today, the most efficient, state-of-the-art HPC platforms are able to achieve power efficiency in the order of $10^{10}\,\mathrm{Flop/s\,W^{-1}}$ [@green500]. A quick calculation leads to an estimated power consumption of $100\,\mathrm{MW}$ for an exascale platform. Due to the ever-increasing power generation costs, such a tremendous power consumption for a single platform is not reasonable, and justifies the need for disruptions. However, we do not directly target the power consumption. We rather look at it through the prism of communications: communications, and more precisely data movements, are indeed the most power hungry operations. As a consequence, optimizing data movements will lower power consumption, but is particularly challenging for two reasons. First, as aforementioned, the volume of data is tremendous: the biggest applications routinely manage volumes of $10\,\mathrm{TB}$ to $100\,\mathrm{TB}$. Such volume of data considerably stresses the interconnection network, and this volume of data is expected to follow the growth of applications and platforms. Second, the gap between memory, network and processor speeds is widening. This gap is such that applications, if not properly scheduled, spend more time waiting for data than computing [Ash+10].

1.2 Contextualization

By essence, any *model* reduces the object under study to a simpler set of characteristics, parameters and postulates. While this reduction is mandatory to apprehend the studied object, it is done at the cost of losing accuracy. In the case of executing applications on modern parallel platforms, phenomena such as network contention, complex data dependencies, memory hierarchies or tightly coupled tasks often are not modeled, though they have a huge impact on performance. We define *context* as such external elements that are not included into the model, but still have a significant impact. **The governing idea of this dissertation is to bring—at a reasonable cost—context awareness, or even context obliviousness, in the scheduling policies.** This idea is derived at two levels: within a single application at a fine grain scale, and with multiple concurrent applications at a coarser level.

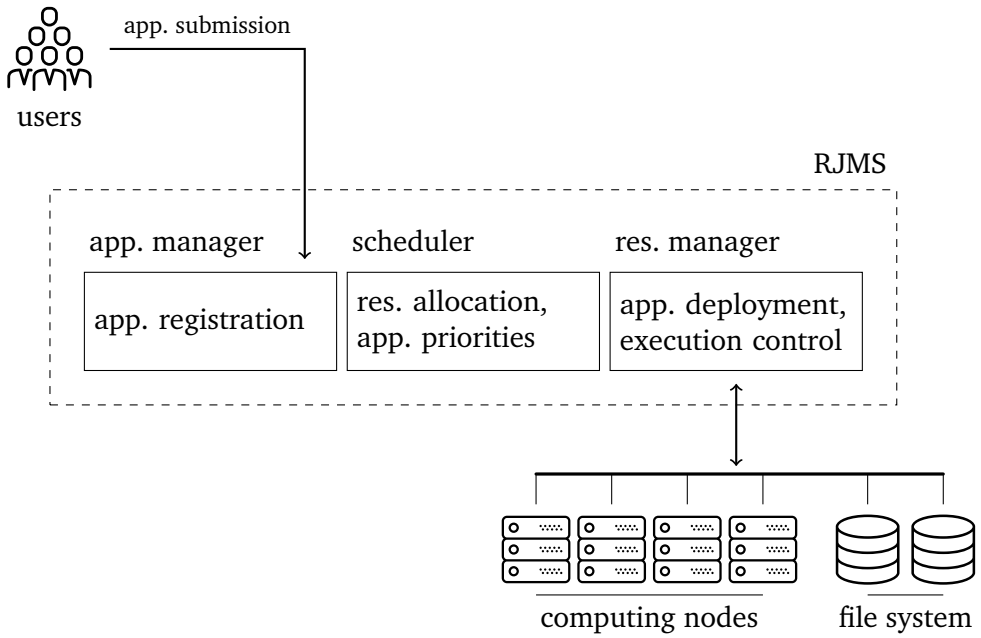


Figure 1.1 Overview of an application submission process on a typical parallel platform [Geo10].
The application manager is the interface between the users and the scheduler: the users send the description of their applications to the application manager.
The resource manager is the interface between the scheduler and the platform: it monitors the resources, deploy applications, and control the execution of the applications.
With respect to the resources status reported by the resource manager, the scheduler decides where (which resources to allocate) and when an application will execute.
[Credits: icons by Madebyoliver and Zlatko Najdenovski; `https://www.flaticon.com`]

Before detailing both levels, let us sketch an application execution. When users of a parallel platform want to run an application, they submit the description of the application to the RJMS (**R**esource and **J**ob **M**anagement **S**ystem). An application description usually consists in an executable, resources requirements, and an estimation of the time needed to run the application. The RJMS is responsible for managing the applications (collecting users' requests, queuing applications), scheduling them (allocating resources to the applications), and managing the resources (deploying and starting applications). The critical role of the scheduler component is to satisfy the users' demand for computation by allocating some resources of the platform for some amount of time to their applications. An overview of an application submission process on a parallel platform is depicted on Figure 1.1. The decisions of the scheduler are often visualized as Gantt charts, as shown on Figure 1.2.

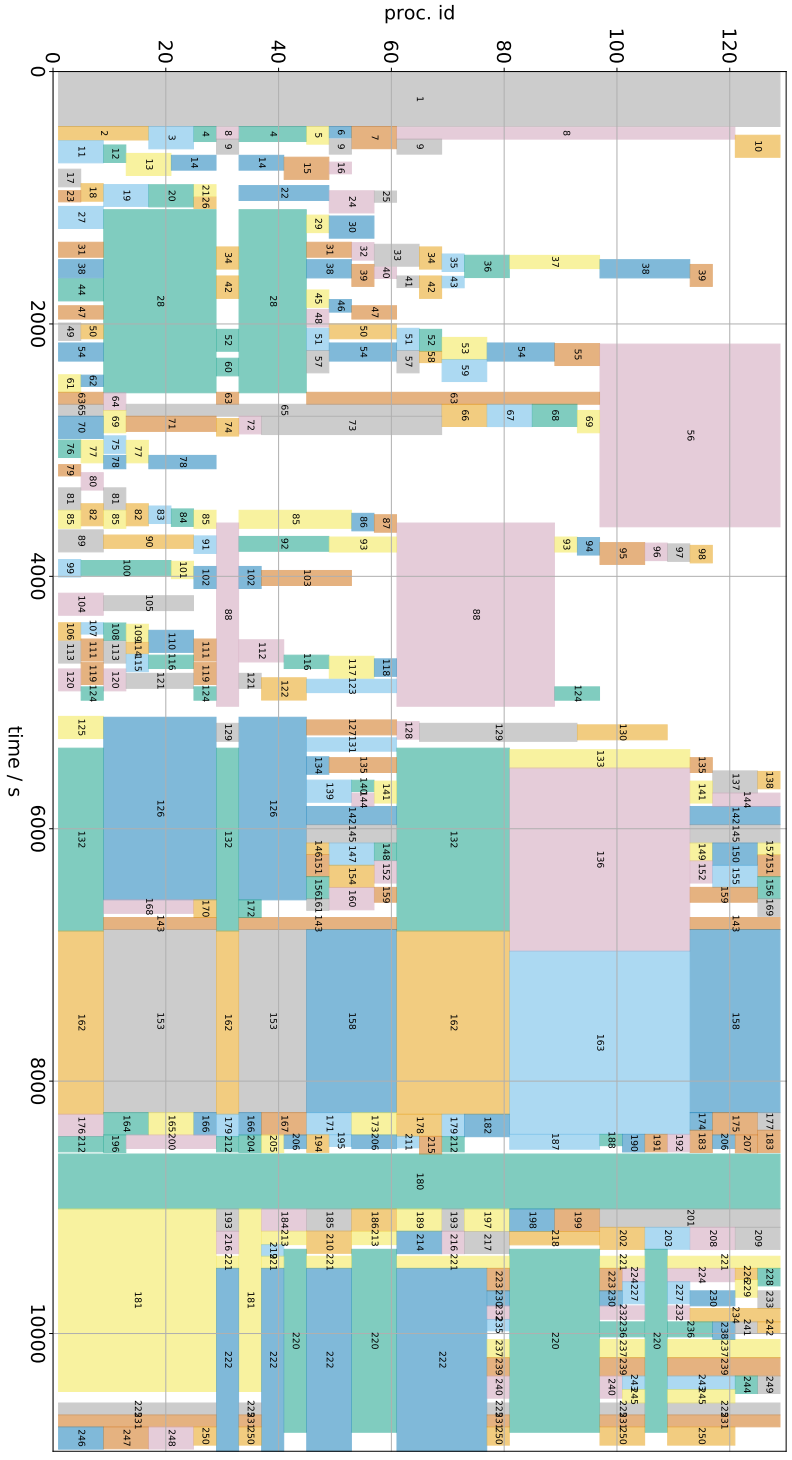


Figure 1.2 Gantt chart of a typical HPC workload. Jobs executed by processors (y-axis) are depicted by rectangles along the time line (x-axis). This chart represents the execution of 250 jobs on a platform with 128 processors. Two distinct phases are visible: the system starts to populate during a transient phase until the processors are saturated (around 5000 s), the system then enter a steady state and processors are almost always used.

1.2.1 Intra-Application Level

From the RJMS point of view, an application may be viewed as a black box (as depicted by rectangles on the Gantt chart on Figure 1.2), and we do not want to open this Pandora's box. Once some resources have been allocated to an application, the application itself is responsible for their correct usage.

An application is usually described as a set of tasks to process. These tasks are the core concept at this level: they represent the basic unit of computations. To efficiently leverage the allocated resources, the application needs to consider various factors. Some factors to consider are trustworthy and stable, such as the application structure or the topology of the allocated resources of the platform. Other factors can only be unreliably measured, such as the effective processing time or the interconnect (internal buses or platform network). Moreover, the execution of an application is perturbed by factors alien to the application [Bha+13]. For example, an application may share some bandwidth with other applications, the temperature of the platform may raise and degrade performance due to other overdemanding applications, etc.

Facing such uncertainties, there is a need for simple, robust and context-aware scheduling policies. These policies also need to be able to handle the heterogeneity in the computing resources. There exist scheduling policies based on strip packing problems that provide efficient solutions with provable approximation ratios [Bou+11]. However, these policies are not robust as they require very precise estimations of the time to process tasks, but such estimations do not exist [HC16].

Locality is a well-known mechanism that drastically improves the execution performance. We propose to bring context-awareness through *contextual affinity*, which we define as a score derived from rough measures of the context (e.g., memory usage at a given moment). Each task prioritizes the computing resources it may be executed on with this qualitative score, and this serves as a guide for the scheduler.

Still, as of today, the application knowledge of the platform is limited to its allocated resources. However, as the platform is shared by several applications, it is not sufficient to consider the intra-application level.

1.2.2 Inter-Applications Level

As stated earlier, HPC platforms now include millions of computing nodes. Very few applications, if any, are able to leverage that many resources. It is however crucial to maintain a high utilization of the platform. Hence, many applications are executed

simultaneously. Similarly to the tasks at the intra-application level, applications are competing for the computing and the communication resources. Yet, when working at this level, the core concept is the single application, and we do not care about too detailed a description of the applications.

As the platforms and applications grow in size, the amount of data transiting increases. Still, even if the memory density increases, the computing nodes cannot keep the whole datasets in their local memories. These datasets are hence stored on parallel file systems (e.g., Lustre) that reside on dedicated hardware. To ensure that the computing nodes can access data is the responsibility of the interconnection network. There exist various cost-effective and efficient interconnection topologies: multidimensional torus, fat-tree, SlimFly, DragonFly [Kat+15]. Such interconnection networks are meant to be unique and all-purpose within the platforms. Consequently, intra-application communications and I/O traffic have to share a global network: this induces complex interactions such as network contention between applications.

These nocuous interactions can either be tackled once applications have been scheduled (reactive strategy) or while scheduling them (proactive strategy). A typical reactive strategy would be—given some application allocations—to schedule the data movements [Gai+15]. On the other hand, a proactive strategy would tackle the interactions by enforcing smart allocations. With these smart allocations, the applications would induce less traffic. It is worth noticing that reactive strategies need a fine knowledge of the context, while proactive strategies do not.

We consider context in an oblivious manner at this inter-application level. Rather than optimizing communications once the applications have been scheduled, we propose to enforce constraints on the allocations. These constraints allow us to completely abstract the context of execution while guaranteeing the applications will behave well by isolating them.

1.3 Explicit *vs.* Implicit Modeling of Communications

The 2010 Turing award laureate, Leslie Valiant, described a good model as a bridge between the chosen programming model and the hardware [Val90]. More precisely, a good model has to be “easy to implement in hardware” and “efficiently targeted by programmers”. As an example, the von Neumann model (a.k.a. RAM model) of the computer is a very good abstraction for sequential programming. On the other hand, the search for a good parallel programming model is still going on.

1.3.1 Existing Explicit Models

PRAM The natural extension of the von Neumann model for parallel platform is the PRAM model (**P**arallel **R**andom-**A**ccess **M**achine) [FW78]. A parallel platform is modeled as an unbounded number of processors with access to an unbounded global memory. At each step in this model, the processors simultaneously execute three successive operations: an optional read, followed by a computation, finished by an optional write. Despite being a powerful classification tool, the PRAM model ignores the communications cost and the synchronization overhead, and it is not representative of modern parallel platforms.

To overcome this weakness of the PRAM model, many modeling approaches have been proposed. We quickly review three classical model families aiming at integrating communication: communication delays, BSP, and LogP.

Communication Delays The communication delays modeling techniques rely on the representation of a computation as a directed acyclic graph of dependencies: vertices represent computations, and edges represent data dependencies between computations. A parallel platform is modeled as a set of processors with local memory connected by an interconnection network. Following the edges of the dependency graph, a data exchange is delayed if the computations are not scheduled on the same processor. A review of various models with communication delays can be found in [BGT97].

BSP The BSP model (**B**ulk **S**ynchronous **P**arallel) has been proposed in the 1980s by Valiant [Val90]. The BSP model integrates the communication from a macroscopic point of view. A parallel platform is modeled as a set of processors, a network to exchange messages, and some hardware support for synchronization. The computations under BSP are modeled as a series of supersteps: each superstep consists in a phase of concurrent computations, a phase of global communications, and a synchronization phase.

LogP The LogP model tackles the issue of communications from an architectural point of view [Cul+93]. A parallel platform under LogP is modeled as a set of processors communicating thanks to an interconnection network. However, contrarily to the BSP model, the processors communicate asynchronously with messages of small size (e.g., 16 bytes in [Dus+96]). The time to exchange messages is then

modeled with three parameters: the latency of the network, the overhead (time spent to process a message) and the gap (reciprocal of the network bandwidth).

1.3.2 Limits of Explicit Models

Parallel platforms are complex systems, where integrating more and more components emphasizes some unforeseen patterns. Existing models exhibit various limitations that prohibit their use at the extreme scales we target. Explicit modeling of the communications faces the issue of choosing between models that are too precise or too coarse.

Too precise models are unable to scale due to the number of parameters needed to apprehend the exascale. Moreover, gathering valid data to instantiate models is an arduous task: the volume of data is tremendous, the granularity of measures is often too coarse to get useful data, instrumentation of code is tedious, etc. Furthermore, the patterns of application submission times are ever-changing: they depend, for example, on the day of the week, whether the location of a conference is attractive or not, how tight the next deadline is. Such behaviors are barely quantifiable. In light of these points, what is the purpose of having ultraprecise model when we are unable to instantiate them?

On the other side of the spectrum, coarse models are unable to capture phenomena such as contention. For example, the BSP model makes the assumption that the network is in a steady state by considering the "basic throughput [...] when in continuous use" [Val90]. This assumption presupposes the latency in the interconnection network can be bound: this is at best a very imprecise assumption, at worst a false hypothesis. Thus, the contention on networks is either imprecisely modeled, or falsely taken into account. Although the LogP model cannot be considered a coarse grain model, it is flawed as it makes the assumption that the bandwidth is sufficient with respect to the message sizes.

1.3.3 Implicit Modeling

As stated in the previous section, explicit models are unable by design to capture contention at scale. Precise models are unable to scale as the number of parameters to account for explodes, while coarse models are too simple to model contention. The goal is instead to find a good balance between the two worlds, by keeping the simplicity of coarse models while capturing phenomena such as contention.

The moldable task model arose from this idea [DMT04; Fei+97]. In this model, the communication times are implicitly taken into account by a penalty function that represents the overhead caused by the parallelization. This model is however limited as the only factor of parallelization is the number of resources allotted to a task. It does not capture, for example, the impact of the allocation shape. Nonetheless, it showed that implicitly modeling communications is an effective way to model modern parallel platform. Rather than trying to capture more and more of the complexity of the modern parallel platform, we keep a simpler model than we enrich with constraints. These extrinsic constraints are cheap safeguards that derive either from the structure of the platform or from the structure of the application. These constraints do not aim to model the context. They however ensure the whole system will behave correctly with regard to the context. As an example, a constraint could forbid the scheduler to use certain allocation shapes if they degrade performance. With such an approach, we are able to mitigate the curse of dimensionality while capturing the context.

1.4 Contributions

The focus of this dissertation is to bring—at a reasonable cost—awareness of the execution context into scheduling policies. The contributions of this dissertation are twofold: we develop this idea within a single application (i.e., intra-application level), and with multiple concurrent applications (i.e., inter-applications level). We address as a first step the intra-application level. More precisely, we study in Chapters 2 and 3 the problem of scheduling independent tasks on a single computing node composed of multiple CPUs and GPUs. Secondly, we propose a generic model of modern large-scale parallel platforms. We expose in Chapter 4 the proposed model framework, and we study in Chapter 5 some instantiations of the model for unidimensional topologies.

Efficient implementations of parallel applications on heterogeneous hybrid architectures require a careful balance between computations and communications with accelerator devices. Even if most of the communication time can be overlapped by computations, it is essential to reduce the total volume of exchanged data. The literature therefore abounds with *ad hoc* methods to reach that balance, but these methods depend on both the targeted architecture and application. We propose a generic mechanism to automatically optimize the scheduling of sequential independent tasks between CPUs and GPUs. This mechanism consists of grouping the tasks by *affinity* before running a fast dual approximation [HS87]. This affinity-based algorithm

has been tested against the reference algorithm HEFT [THW02] by simulation and experimentation. We assessed the performance with standard dense linear algebra kernels. We ran the experiments on a hybrid parallel machine composed of twelve CPUs and eight GPUs. The results show that HEFT and the new algorithm perform well for various experimental conditions. The new algorithm scales better for larger systems and number of GPUs. Moreover, in most cases, the new algorithm generates much lower data transfers than HEFT to achieve the same performance.

However, as the approach detailed above targets sequential tasks, it still relies on an explicit view of the parallelism on the CPUs. We further extend the model to assume tasks are parallelizable on CPUs by using the moldable model: the final number of cores allotted to a task is decided and set by the scheduler. We design an algorithm aimed at minimizing the makespan—the maximum completion time of all tasks—for this scheduling problem. The proposed algorithm combines a dual approximation scheme with a fast integer linear program (ILP). A worst-case analysis shows that the algorithm has an approximation ratio of $\frac{3}{2} + \epsilon$. Since the time complexity of the ILP-based algorithm could be non-polynomial, we also present a polynomial-time algorithm with an approximation ratio of $2 + \epsilon$. We complement the theoretical analysis of our two novel algorithms with a simulation study. In these simulations, we compare our algorithms to a modified version of the classical HEFT algorithm, which we adapted to handle moldable tasks. The simulation results show that our algorithm with the $\left(\frac{3}{2} + \epsilon\right)$-approximation ratio produces significantly shorter schedules than the modified HEFT for most of the instances. In addition, our results provide evidence that our ILP-based algorithm can solve large problem instances in a reasonable amount of time.

Moving up to the inter-application level, schedulers for large parallel platform face challenges to efficiently share the platform between users. These challenges mainly arise from the scale of the machines, the heterogeneity of the resources, and the structure of the interconnection network. As the interconnection network is unique and shared, the interweaving of network flows begets complex interactions detrimental to performance. Among the various types of network traffic induced by applications, we distinguish the flows induced by data exchanges for computations (i.e., intra-application flows) and the flows targeting the external storage. We propose here a new direction to consider and limit such complex interactions. We do not try to directly optimize the communications. The core idea is to use geometric constraints as first class modeling tools. We propose to derive constraints from the topology of the parallel platform. These constraints restrict the allocations the scheduler is allowed to use. We classify the constraints, and identify the constraints of interest for the integration of the execution context. We address the issue of

intra-application flows with structurally intrinsic constraints such as convexity or contiguity. For example, the convexity constraint ensures intra-application flows do not interact with any other concurrent application. Furthermore, we address the issue of I/O flows with extrinsic constraints, such as compacity or proximity. These extrinsic constraints ensure applications are allocated resources close to the location of their datasets. The generic scheduling problem is then defined as an optimization problem with the application description and the platform (nodes and topology) as input. The objective for example is to minimize makespan or maximize throughput, while respecting the constraints on the allocations.

We then study with this methodology several instances of the generic problem with unidimensional topologies. Unidimensional topologies are indeed a first step towards the study of instances with characteristics deriving from machines such as Curie [@curie] (fat-tree interconnect) or Blue Waters [@bw] (3D-torus interconnect). More precisely, we study the line and the ring topologies. We start by stating structural dominance of some properties. These dominance results are independent of the optimization objective. Second, we study thoroughly the problem of minimizing the makespan under convexity and locality constraints. We prove complexity results for various setups. We then propose approximation algorithms with constant approximation ratios.

Work dissemination

The following communications result from the preparation of this dissertation.

Journal Articles

- Raphaël Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. “Scheduling independent tasks on multi-cores with GPU accelerators”. In: *Concurrency and Computation: Practice and Experience* 27.6 (2015), pp. 1625–1638. DOI: 10.1002/cpe.3359.
- Raphaël Bleuse, Sascha Hunold, Safia Kedad-Sidhoum, et al. “Scheduling Independent Moldable Tasks on Multi-Cores with GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (Sept. 2017), pp. 2689–2702. DOI: 10.1109/TPDS.2017.2675891.

Peer reviewed international conferences

- Raphaël Bleuse, Thierry Gautier, João Vicente Ferreira Lima, Grégory Mounié, and Denis Trystram. "Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures". In: *Euro-Par*. Vol. 8632. Lecture Notes in Computer Science. Springer, Aug. 2014, pp. 560–571. DOI: 10.1007/978-3-319-09873-9_47.

International Workshops

- Raphaël Bleuse, Giorgio Lucarelli, Grégory Mounié, and Denis Trystram. "Interference-Aware Scheduling with 2D-Torus as a Case Study". Presented at *ECCO XXX*. Koper, Slovenia, May 2017.
- Raphaël Bleuse. "Affinity Scheduling: from Quantitative to Qualitative". Presented at the *2nd JLESC Workshop*. Chicago, USA, Nov. 2014.
- Raphaël Bleuse, Giorgio Lucarelli, and Denis Trystram. "Convex Allocations under IO Constraints". Presented at *New Challenges in Scheduling Theory*. Aussois, France, Mar. 2016.
- Raphaël Bleuse, Giorgio Lucarelli, and Denis Trystram. "Convex Allocations under IO Constraints". Presented at the *5th JLESC Workshop*. Lyon, France, June 2016.

Popularization

- "CS Unplugged: Alice déménage". Workshop animation at *Fête de la Science*. Grenoble, France, Oct. 2014, 2015, 2016.

1.5 Content

The remainder of the dissertation is organized as follows. Each chapter includes a section addressing the relevant related work. At first, we study the intra-application level. We propose in Chapter 2 a new algorithm based on affinity to schedule independent sequential tasks on a platform composed of multiple CPUs and GPUs. We then report the performances of its implementation in a parallel run-time. We extend the model in Chapter 3, by allowing the tasks to be moldable on the CPUs. We propose two algorithms with performance guarantees, and study their performances

by simulation. The next two chapters address the inter-applications level. We propose in Chapter 4 a generic framework to model current and upcoming parallel platforms. More precisely, we identify constraints and metrics of interest for the study of such platforms. We then instantiate the proposed model for unidimensional topologies in Chapter 5, and study all variants of makespan minimization under the constraints of convexity and locality. Finally, a conclusion and future work perspectives are presented in Chapter 6.

2 Scheduling Independent Sequential Tasks on Multi-Cores with GPUs

This chapter presents new methods to schedule independent sequential tasks on hybrid CPU-GPU architectures designed for HPC. We propose and study the integration of scheduling algorithms based on the dual approximation technique [HS87]. We propose an extension of such algorithms that is able to find a compromise between performance and transfers. This is done by bringing context awareness through an affinity based abstraction. The contributions of this chapter are first the design and implementation of new algorithms, and second their evaluation on three dense linear algebra algorithms in double precision floating-point operations from PLASMA [But+09]: namely Cholesky, LU, and QR.

With the recent evolution of processor design, the future generations of processors will contain hundreds of cores. To increase the performance per watt ratio, the cores will be non-symmetric with few highly powerful cores (CPU) and numerous simpler cores (GPU) [Lee+10]. The success of such machines will rely on the ability to schedule the disparate workload at run-time, even for small problem instances. One of the main challenges is to define a scheduling policy that may be able to exploit all potential parallelisms on a heterogeneous architecture composed of multiple CPUs and multiple GPUs. These new hybrid architectures have given rise to new scheduling problems. In the field of High Performance Computing (HPC), one of the most studied scheduling problem is the minimization of the maximum completion time (makespan) of the schedule. Some Polynomial-Time Approximation Schemes (PTAS) exist for problems of minimizing the makespan on heterogeneous processors [BW12; HS88], but their running times make them impractical for solving scheduling problems on actual computing platforms.

In the field of parallel processing, there exists a huge number of papers dealing with *ad hoc* implementations of algorithms using GPUs or hybrid architectures. These works expand over several aspects of parallelism from operating system, run-time, application implementation or languages. However, few of them focus on the intermediate problem of scheduling on hybrid platforms [PDB13]. An online algorithm

with a performance guarantee [CYZ13] has recently been developed for CPU-GPU platforms, but there is no performance guarantee for any offline problem on these systems. Many works in the literature consist of studying the gains and performances of parallel implementations of some specific numerical kernels [Agu+11b; STD12], specific applications like multiple alignments of biological sequences [Bou+10b], or molecular dynamics [PSS08]. As a result, most of the existing scheduling algorithms and tools are not well-suited for general purpose applications. Scheduling is done on a case by case basis, and often offers good performances. However, we lack high-level mechanisms that provide transparent and efficient schedules for any application.

Some actual run-time systems include the basic mechanisms for developing scheduling algorithms like OmpSs [Bue+12], StarPU [Aug+11] or XKaapi [Gau+13]. Several scheduling algorithms have been implemented on top of these systems. Previous works demonstrate the efficiency of policies such as static distribution [Son+10; TDB10], centralized list scheduling with data locality [Bue+12], cost models [Agu+11a; Agu+11b; ATN09; Aug+11] based on Heterogeneous Earliest Finish Time scheduling (HEFT) [THW02], and dynamic scheduling for some specific application domains [Bos+12; Her+10]. Locality-aware work stealing [Gau+13], with a careful implementation to overlap communication by computation [Fer+12], improves significantly the performance of compute-bound linear algebra problems such as matrix product and Cholesky factorization. Nevertheless, none of the works cited above considers scheduling strategies from the viewpoint of a compromise between performance and locality.

The chapter is organized as follows. A formal description of the scheduling problem with k GPUs is provided in Section 2.1. In Section 2.2 we report existing works from an algorithmic point of view, and we briefly survey related works on run-time systems, scheduling policies and performance prediction. Section 2.3 provides an overview of XKaapi run-time system, describes the XKaapi scheduling framework and the cost model applied for performance prediction. Section 2.4 details the scheduling policies that have been studied. We study in Section 2.5 the usability of the proposed scheduling policies. Section 2.6 presents the experimental results obtained with the retained scheduling policies on a heterogeneous machine composed of twelve CPUs and eight GPUs. Finally, a synthesis is provided in Section 2.7.

The work presented in this chapter has led to two publications [Ble+14; Ble+15], and has been done in collaboration with Thierry GAUTIER, Safia KEDAD-SIDHOUM, João V. FERREIRA LIMA, Florence MONNA, Grégory MOUNIÉ, and Denis TRYSTRAM.

2.1 Problem Definition

We consider a multi-core parallel platform composed of m identical CPUs and k identical GPUs. The m CPUs are considered independent from the GPUs. Each GPU is driven by a dedicated CPU. The driving CPUs are not considered in the scheduling problem because they do not execute any task. An instance of the problem is described as a set $T_1, \ldots, T_n$ of n independent sequential tasks. The processing time of any task T_j is denoted by $\overline{p_j}$ if T_j is processed on a CPU, and by $\underline{p_j}$ if the task is processed on a GPU. We assume that the processing times are known in advance. This is a common assumption when scheduling dense linear algebra kernels (as we do in the experiments).

The *scheduling* problem consists in computing a function σ that associates every task to a starting time and either a CPU or a GPU. The makespan is defined as the maximum completion time of the last finishing task. The objective is to minimize the makespan $C_{\max}$ of the whole schedule, which is the maximum of the makespan on CPUs and the makespan on the GPUs. The problem is denoted by $(Pm, Pk) \mid\mid C_{\max}$.

2.2 Related Work

This section is organized in two parts: first we discuss complexity and algorithmic results, and second we briefly describe existing work in parallel run-times.

2.2.1 Algorithmic Results

With the constraint that processing times match on both CPUs and GPUs (i.e., $\forall j, \overline{p_j} = \underline{p_j}$), the problem $(P1, P1) \mid\mid C_{\max}$ is equivalent to $P2 \mid\mid C_{\max}$. As $P2 \mid\mid C_{\max}$ is **NP**-hard [GJ79], the problem of scheduling with GPUs is **NP**-hard. Hence, we are interested in proposing efficient approximation algorithms with performance guarantees. Recall that for a given minimization problem, the approximation ratio ρ_A of an algorithm A solving this problem is defined as the maximum over all the instances I of the ratio $\frac{f(I)}{f^*(I)}$ where f is the minimization objective and f^* is its optimal value [HS87].

The problem targeted here is a new problem. It is harder than the classical problem of scheduling on uniform machines $P \mid\mid C_{\max}$. Although $(Pm, Pk) \mid\mid C_{\max}$ is a specific

case of the problem of scheduling on unrelated machines $R \mid\mid C_{\max}$ [Bła+07], it is possible to get better results by considering the problem on its own. Lenstra et al. proposed a Polynomial-Time Approximation Scheme (PTAS) for the problem $R \mid\mid C_{\max}$ with a running time bounded by the product of $(n+1)^{\frac{m}{\epsilon}}$ and a polynomial of the input size [LST90]. Let us notice that if parameter m is not fixed, then the algorithm is not fully polynomial. The authors also proved that unless **P** = **NP**, $R \mid\mid C_{\max}$ admits no PTAS with an approximation factor better than $\frac{3}{2}$. They proposed a 2-approximation algorithm for $R \mid\mid C_{\max}$. This algorithm works by optimally solving with a linear program the preemptive version of the problem, and by rounding this optimal solution. Shmoys and Tardos generalized this technique to obtain the same approximation factor for the generalized assignment problem [ST93]. Furthermore, they generalized the rounding technique to hold for any fractional solution. Recently, Shchepin and Vakhania introduced a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$ [SV05]. This technique has a similar time complexity as the one developed by Lenstra et al. [LST90]. To the best of our knowledge, this is so far the best approximation result for this problem. However, the prohibitive computational cost of these algorithms prevents their usage on actual computing platforms.

It is worth noticing that if all the tasks of the problem have the same acceleration on the GPUs, the problem reduces to the classical problem of scheduling on related machines $Q \mid\mid C_{\max}$, with two machine speeds. For $Qm \mid\mid C_{\max}$, Friesen proved that the approximation ratio of the well-known Longest Processing Time (LPT) scheduling policy satisfies $1.52 \leqslant \frac{C_{\max}^{\text{LPT}}}{C_{\max}^{*}} \leqslant \frac{5}{3}$ [Fri87]. The first PTAS for $Q \mid\mid C_{\max}$ was given by Hochbaum and Shmoys [HS88]. The overall running time of their algorithm is $\mathcal{O}\left(\left(\log(m) + \log(\frac{3}{\epsilon})\right)\frac{m}{\epsilon}\frac{n}{\epsilon}\frac{1}{\epsilon}\right)$. However, these solving methods would only work for specific instances of the problem of scheduling on hybrid platforms, where the acceleration factors $\frac{\overline{p_j}}{\underline{p_j}}$ would be equal to a constant. This is not the case in practice.

Another direction is to consider the problem of scheduling on unrelated machines of few different types. Indeed, the $R \mid\mid C_{\max}$ reference problem can be refined to better fit the constraints of the hybrid platforms. Bonifaci and Wiese [BW12] presented a PTAS to solve a scheduling problem with unrelated machines of few different types. The tools used in their solving method are somewhat similar to the ones used for solving $R \mid\mid C_{\max}$. The rounding phases of the algorithm require a significant amount of time, raising the time complexity of the algorithm to an impractical level. Despite considering only two machine speeds with $(Pm, Pk) \mid\mid C_{\max}$, their method is too costly to be implemented in a parallel run-time in our case. There is a need to consider other algorithms than these PTAS to design algorithms that could

be implemented on actual platforms. A PTAS with a reasonable time complexity has been developed for the online version of the problem of the assignment of sporadic tasks on hybrid platforms [RN12]. However, an offline version of the problem with non-periodic tasks has not been studied, and the algorithm cannot be trivially extended to the problem $(Pm, Pk) \mid\mid C_{\max}$. On another hand, Imreh presented different greedy algorithms for the problem of scheduling on two sets of identical machines, with varying approximation ratios including $2 + \frac{m-1}{k}$ and $4 - \frac{2}{m}$ (for m CPUs and k GPUs) [Imr03]. These algorithms are fast enough for being implemented in modern platforms, nevertheless the approximation ratios of these algorithms are quite high since usually the number of GPUs is much lower than the number of CPUs.

The objective is to build a bridge between purely theoretical algorithms with performance guarantees and practical low cost heuristics. Thus, we propose a trade-off solution with a provable performance guarantee and a reasonable time complexity.

2.2.2 Parallel Run-times

From a practical perspective, the scheduling policy is a key point for the performance of an application. Tuning scheduling algorithms for a specific case (problem and computer architecture) is common. Fast heuristics without performance guarantee are often used on computing platforms, time efficiency being the crucial factor. However, simple policies are not sufficient to guarantee the performance for more general cases potentially far from the specific one, and *ad hoc* methods cannot be reused. The performance portability is difficult to achieve when the number of CPUs and GPUs varies or when the speedup of the various parts of the application is evolving during the execution.

When writing a parallel run-time for hybrid machines, one of the main challenges is to define a scheduling policy that may be able to exploit all potential parallelisms on a heterogeneous architecture composed of multiple CPUs and multiple GPUs.

StarPU [Aug+11], OmpSs [Bue+12] and QUARK [YKD11] are programming environments or libraries that enable the automatic scheduling of tasks with data-flow dependencies. OmpSs is based on OpenMP-like pragmas while StarPU and QUARK are C libraries. QUARK does not schedule tasks on multi-GPU architecture and implements a centralized greedy list scheduling algorithm. OmpSs locality-aware scheduling, similar to our data-aware heuristic from [Gau+13], computes an affinity score based on both data location and size. Then, the task is placed on the

highest affinity resource or in a global list, otherwise. The StarPU scheduler uses the HEFT [THW02] algorithm to schedule all ready tasks in accordance with the cost models for data transfer and task execution time [ATN09]. Our data transfer model is based on the StarPU model with minor extension. In the context of dense linear algebra algorithms, PLASMA [But+09] provides fine-grained parallel linear algebra routines with dynamic scheduling through QUARK, which was conceived specially for numerical algorithms on multi-CPU architecture. MAGMA [TDB10] implements static scheduling for linear algebra algorithms on heterogeneous systems composed of GPUs. Recently it has included some methods with dynamic scheduling in multi-CPU and multi-GPU systems on top of StarPU, in addition to the static multi-GPU version. In [Son+10], the authors based their Cholesky factorization on 2D block cyclic distribution with an owner compute rule to map tasks to resources. DAGuE [Bos+12] is a parallel framework focused on multi-core clusters and supports single-GPU nodes. Other papers reported performance results of task-based algorithms with HEFT cost model scheduling on heterogeneous architectures for the Cholesky [Aug+11], LU [Agu+11a], and QR [Agu+11b] factorizations. All the results report evaluation of single floating-point arithmetics with up to three GPUs. Due to the small number of GPUs, such studies cannot observe contention and scalability.

2.3 XKaapi Scheduling Framework

The XKaapi data-flow model [GBP07]—as in Cilk, Intel TBB, OpenMP-3.0, or OmpSs [Bue+12]—enables non-blocking task creation: the caller creates the task, and proceeds with the program execution. Parallelism is explicit while the detection of synchronizations is implicit [GBP07]: the dependencies between tasks and memory transfers are automatically managed by the run-time system.

The XKaapi run-time system is structured around the notion of a *worker*: it is the internal representation of a kernel thread. A worker executes the code of the tasks, and takes local scheduling decisions. Each worker owns a local queue of ready tasks. Our interface is mainly inspired by the one of a work stealing scheduler, and is composed of three operations that act on workers' queues of tasks: *pop*, *push* and *steal*. Previous work demonstrated the efficiency of XKaapi's locality-aware work stealing, as well as the corresponding multi-GPU run-time support using specialized implementation of these operations [Gau+13]. A new operation, called *activate*, has been defined to push ready tasks to a worker's queue.

2.3.1 Execution Flow

A framework interface for scheduling policies is not a new concept in heterogeneous systems. Bueno et al. [Bue+12] and Augonnet et al. [Aug+11] described a minimal interface to design scheduling policies with selection at run-time. However, there is little information available on the comparison of different policies. Most of them reported performance on centralized list scheduling and performance models. The XKaapi framework is composed of three classical operations in the work stealing context, plus an action to activate tasks when their predecessors have completed:

- The *push* operation inserts a task into a queue. A worker can push a task into any other workers' queue.
- A *pop* removes a task from the local queue owned by the caller worker.
- A *steal* removes a task from the queue of a remote worker. The operation is called by an idle thread—the *thief*—in order to pick tasks from a randomly selected worker—the *victim*.
- The *activate* operation is called after the completion of a task. The role of this operation is to allocate the tasks that are ready to be executed. The scheduling decision are computed during this operation.

The sketch of the execution mechanism is as follows. At each step, either the own local queue of worker is non-empty and the worker uses it; or the worker emits a steal request to a randomly selected worker to get a task to execute. Once a worker finishes processing a task, the *activate* operation is called. With respect to the dependencies, this operation marks as ready the successors of the completed task that become ready for execution (i.e., the completed task was the last blocking dependency). The scheduling policy then schedules the ready tasks. The execution of the application terminates when all the tasks have been executed.

2.3.2 Performance Model

Cost models depend on a certain knowledge of both application algorithm and the underlying architecture to predict performance at run-time. In order to predict performance, we designed a StarPU-like performance model for both task execution time and communication [ATN09]. The prediction of task processing time relies on a history-based model, and the estimation of transfer time is based on asymptotic bandwidth. These predictions are given to the scheduling policies that need task processing time such as HEFT and dual approximation based policies.

The XKaapi run-time system gets information from every internal events (such as start/end of task execution, or start/end of communication with a GPU) to calibrate the performance model, and corrects erroneous predictions due to unpredictable or unknown behavior (e.g., operating system state or I/O disturbance). StarPU [Aug+11] uses similar run-time measurements in order to correct the performance predictions in its HEFT implementation.

To efficiently balance the load XKaapi maintains for each processor—as a shared time-stamp—the predicted time at which the processor will become idle. The completion date of the last executed task is also kept for each processor. The update and incrementation of the time-stamps are efficiently implemented with atomic operators.

2.4 Scheduling Policies

This section introduces the scheduling policies designed on top of the XKaapi scheduling framework. We consider a multi-core parallel architecture with m homogeneous CPUs and k homogeneous GPUs. First, we describe our implementation of HEFT [THW02]. Then, we recall the principle of the dual approximation scheme [HS87]. We propose a new algorithm—**D**istributed **A**ffinity **D**ual **A**pproximation (DADA)—based on this paradigm which takes into account the affinity between tasks.

In the following, the number of tasks is denoted by n. We denote by $\overline{p_j}$ the processing time of task T_j on a CPU and $\underline{p_j}$ on a GPU. We define the speedup S_j of task T_j as the ratio $S_j = \frac{\overline{p_j}}{\underline{p_j}}$.

2.4.1 HEFT: Heterogeneous Earliest-Finish-Time

The **H**eterogeneous **E**arliest-**F**inish-**T**ime algorithm (HEFT), proposed by [THW02], is a scheduling algorithm for a bounded number of heterogeneous processors. HEFT is the natural extension of list algorithms for unrelated processors. This is however not a list scheduling algorithm as some resources may stay idle. The algorithm time complexity is in $\mathcal{O}\left(n^2 \cdot (m+k)\right)$. It has two major phases: *task prioritizing* and *worker selection*. Our XKaapi version of HEFT implements both phases during the *activate* operation. The *task prioritizing* phase computes for all ready tasks T_j its speedup S_j relative to an execution on GPU. Next, it sorts the list of ready tasks by

decreasing speedups. Whereas the original HEFT rule sorts the tasks by decreasing upward rank (average path length to the end), our rule gives priority on minimizing the sum of the execution times. In the *worker selection* phase, the algorithm selects tasks in the order of their speedup S_j and schedules each task on the worker which minimizes the completion time. Algorithm 1 describes the basic steps of HEFT over XKaapi.

Algorithm 1: HEFT—*activate* operation

Input : a list LR of ready tasks T_j
Output : tasks T_j pushed to the worker's queues

```
foreach T_j ∈ LR do
    S_j ← p̄_j / p_j
end
sort all ready tasks T_j by decreasing speedup S_j
foreach T_j ∈ LR do
    schedule T_j on the worker i achieving the earliest finish time
    push T_j into queue of worker i
    update processor load time-stamps on worker i
end
```

As of today, HEFT is the *de facto* reference algorithm to schedule tasks on machines with diverse computation resources. From a theoretical point of view, it is not a satisfying algorithm as its worst case performance ratio is arbitrarily bad. More precisely, Monna proved the following result:

Lemma 2.1 ([Mon14]). *For problem* $(Pm, P1) \mid\mid C_{\max}$*, the worst case performance ratio of* HEFT *is at least* $\frac{m}{2}$.

2.4.2 Dual Approximation Based Algorithms

The algorithms exposed in this section all rely on the dual approximation technique [HS87]. Let us recall that dual approximation is a powerful technique to design approximation algorithms for minimization problems. A g-dual approximation algorithm takes an estimation (guess) λ of the value of the objective to minimize. If a feasible solution of objective value at most λ exists, the algorithm is guaranteed to find a solution of objective value at most $g\lambda$. On the contrary, if there exists no feasible solution of objective value at most λ, the algorithm correctly reject the guess λ. Given a lower bound and an upper bound of the optimal objective value, a binary search on the guess until a given precision ϵ is reached.

In the context of scheduling, the dual approximation algorithm uses a guess λ of the makespan. The g-dual approximation algorithm then finds a schedule of makespan at most $g\lambda$ if there exists a feasible schedule of makespan at most λ, or the algorithm rejects λ.

Pure Dual Approach

The dual approximation technique is a powerful tool to reduce the complexity of the algorithms. Having an estimation of the makespan allows searching feasible solutions with a specific structure. We detail here the algorithmic aspects of the studied algorithms.

DA-2 DA-2 is a 2-approximation algorithm [Ble+15]. The computing resources are split into two shelves: one shelf contains all the CPUs, the other shelf contains all the GPUs. The key idea to allocate the tasks is to minimize the work on the CPUs by offloading as much work as possible to the GPUs.

Given a guess λ of the makespan, the algorithm works as follows. There is no choice for the tasks that fits in λ only on one type of architecture: allocate them to the corresponding architecture. Sort the remaining unallocated tasks by decreasing speedup. Following this order, schedule each task on the least loaded GPUs until each GPU is loaded more than λ. Schedule the remaining tasks on the CPUs with a list algorithm.

DP-4/3 DP-4/3 [Ble+15] is based on the same principles as DA-2: it partitions the computing resources into shelves, allocates the tasks to shelves, and schedules the tasks within their shelves. The algorithms provide a guarantee of $\frac{4}{3} + \frac{1}{3k} + \epsilon$ by using a finer partitioning of the tasks into eight shelves. The allocation of the tasks to the shelves is done with a dynamic program of computational cost of $\mathcal{O}\left(n^2m^2k^3\right)$ per step of the binary search.

Bringing Context-Awareness in: Affinity

The pure dual based algorithms presented neglect a key point as they use a flat view of the resources within each partition. Although scheduling ready tasks only allows us to consider tasks as independent, the choice of where to allocate a task needs to

take into account the context. DADA builds a compromise taking into account both raw performance (makespan) and transfers (context).

The idea is to split the scheduling in two successive phases: a first *local* phase targeting the reduction of the communications, and a second *global* phase which counter-balances the induced serialization. The combination of the two phases follows the scheme introduced by Stein and Wein [SW97]. The local phase uses the affinity abstraction described below. The second phase aims at a global balance. Any algorithm optimizing the makespan could be used for the second phase. We use here DA-2. In order to gain a finer control, the length of the first phase is controlled by a parameter (denoted by α, $0 \leqslant \alpha \leqslant 1$). A value of 0 for α means that the affinity is not taken into account: DADA is then equivalent to DA-2. While at the opposite end, a value close to 1 allows a length up to λ for the first phase, thus giving a greater weight to affinity.

Each pair (task, computation resource) is given an affinity score. Maximizing the score over the whole schedule makes possible to consider local impacts. The affinity scores are computed using extra information automatically gathered by the run-time system. In our implementation, they were computed using the amount of data updated by each task. For instance, a task that *writes* or *modifies* a data stored on a resource i has a high score and is prone to be scheduled on i.

The dual approximation part of Algorithm 2 consists in the following steps:

- Choice of the initial guess λ (lines 2 and 4);
- Extract the tasks which fit only into GPUs ($\overline{p_j} > \lambda$), and symmetrically those which are dedicated to CPUs (line 9);
- Keep this schedule if the tasks fit into λ (line 12). Otherwise, reject it if there is a task larger than λ on both CPUs and GPUs (line 15);
- Add to the tasks allocated to the GPU those which have the largest speedup S_j up to overreaching the threshold λ (line 19) which guarantees the ratio $\rho = 2$;
- Put all the remaining tasks in the m CPUs and execute them using an earliest-finish-time scheduling policy (line 19).

Algorithm 2: DADA—*activate* operation

Input : a list LR of ready tasks T_j
Output : tasks T_j pushed to the worker's queues

$lower \leftarrow 0$
$upper \leftarrow \sum_j \max(\overline{p_j}, \underline{p_j})$
while $(upper - lower) > \epsilon$ **do**
    $\lambda \leftarrow (upper + lower)/2$
    begin *local affinity phase*
        schedule tasks of LR per affinity score on its affinity processor, loading each processor up to overreaching $\alpha\lambda$
    end
    begin *global balance phase*
        schedule LR to minimize finish time using λ as hint
        if *tasks do fit into* $(\rho + \alpha)\lambda$ **then**
            $upper \leftarrow \lambda$
            keep current schedule
        else
            $lower \leftarrow \lambda$
            reject current schedule
        end
    end
end
push each task T_j of LR on queue of worker i based on the last fitting schedule and update processor load time-stamps

2.5 Usability of Scheduling Policies for Linear Algebra

The performances of the algorithms have been first assessed by simulation [Ble+15]. The dual approximation policies computed schedule with smaller makespans. We study in this section the practical usability of the scheduling policies. The algorithms have been implemented as scheduling policies in the XKaapi run-time.

We measured the performances of the HEFT, DA-2 and DP-4/3 policies with linear algebra kernels because these kernels are extensively used, well-maintained, and the processing times of the generated tasks are well-known and stable.

Some preliminary experiments have been conducted on a quad-core Intel i7-3840QM running at $3.8\,\mathrm{GHz}$, with hyperthreading enabled, and with $32\,\mathrm{GB}$ of memory. The machine is augmented with a single NVIDIA Quadro K1000M GPU (Kepler architecture) of 192 GPU cores (scalar processors) running at $850\,\mathrm{MHz}$ with $2\,\mathrm{GB}$ DDR3 memory.

We study the variation of the computation time as a function of the block size for the same matrix size of a Cholesky factorization (`DPOTRF`) extracted from the MAGMA library. Varying the block size allows us study the behavior of the run-time under various conditions as the block size has a direct impact on memory transfers between the CPUs and the GPU. The block sizes also impacts the number of scheduled tasks: the smaller the block size, the greater the number of tasks.

We observe on Figure 2.1 that the execution time using DP-4/3 increases drastically with the number of tasks (small block size). This behavior is caused by the time to compute the schedule dominating the execution time. Regarding the processing time of each task (i.e., linear algebra kernel), DP-4/3 is too slow to compute a schedule. Despite providing a better performance guarantee, DP-4/3 does it at too great a cost (quadratic in the number of tasks) for linear algebra kernels.

The algorithm DP-4/3, and its better performance guarantee, is therefore usable for cases where the processing time of tasks is greater than the time to compute a schedule. In the context of linear algebra kernels, DP-4/3 is not the best suited algorithm. On the other hand, HEFT and DA-2 are fast enough to compute schedules in such a setup.

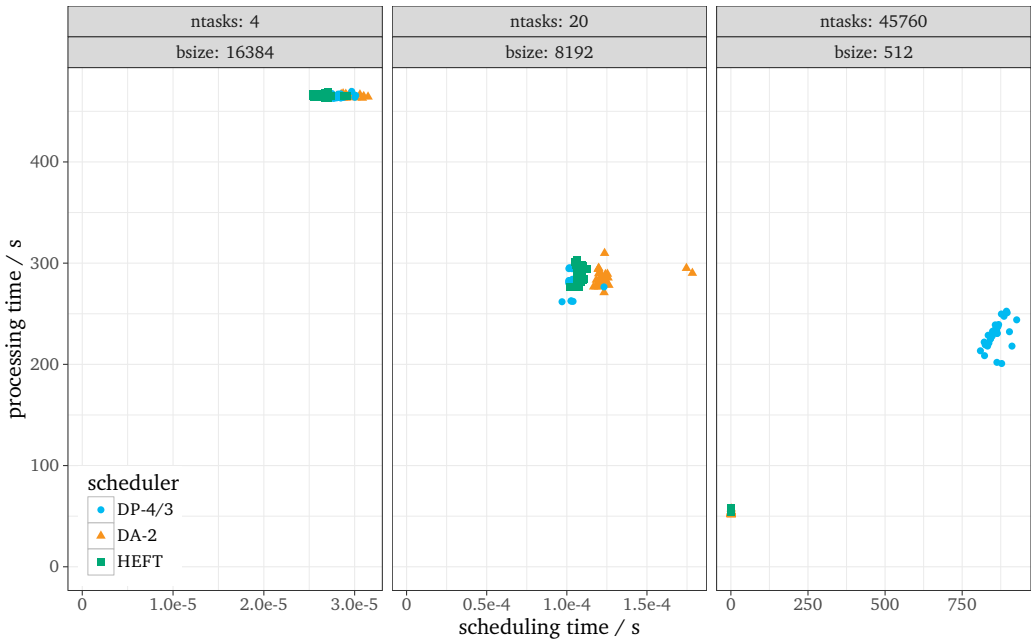


Figure 2.1 Distribution of the execution time between scheduling time and effective processing time for various block sizes.
The sum of both times gives the real execution time of the factorization. The computed kernel is a Cholesky factorization (`DPOTRF`) of a matrix of size 32768. Block sizes between 1024 and 4096 exhibit the same behavior as with a block size of 512: they have been removed for the sake of clarity.
The factorization is scheduled by DA-2, DP-4/3, and HEFT on three hyperthreaded cores and a single GPU.

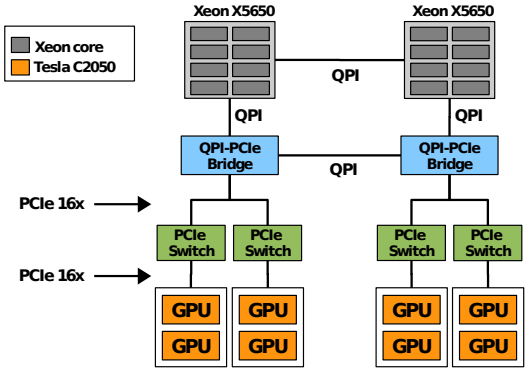


Figure 2.2 Architecture of a multi-cores with GPUs system [Gau+13]. CPUs and GPUs do not share memory, and communicate via the PCIe buses.

2.6 Performance Evaluation

As stated in Section 2.5, DP-4/3 is too costly to compute schedules for linear algebra. We hence compare the performances of HEFT, DA-2, and DADA. Recall that DADA is a variation of DA-2 with an improved context-aware mapping (affinity).

2.6.1 Experimental Setup: Platform and Benchmarks

Platform

All experiments have been conducted on a heterogeneous, multi-CPU, multi-GPU system. The machine is composed of two hexa-core Intel Xeon X5650 CPUs running at $2.66\,\mathrm{GHz}$ with $72\,\mathrm{GB}$ of memory. The machine is enhanced with eight NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at $1.15\,\mathrm{GHz}$ each (2688 GPU cores total) with $3\,\mathrm{GB}$ GDDR5 memory per GPU ($18\,\mathrm{GB}$ total). The machine has four PCIe switches to support up to eight GPUs. Figure 2.2 depicts the architecture of the machine. When two GPUs share a switch, their aggregated PCIe bandwidth is bounded by the one of a single PCIe 16x. Experiments using up to four GPUs avoid this bandwidth constraint by using at most one GPU per PCIe switch.

Benchmarks

All benchmarks ran on top of a GNU/Linux Debian 6.0.2 *squeeze* with kernel 2.6.32-5-amd64. We compiled with GCC 4.4 and linked against CUDA 5.0 and the li-

brary ATLAS 3.9.39 (BLAS and LAPACK). All experiments use the tile algorithms of PLASMA [But+09] for Cholesky (`DPOTRF`), LU (`DGETRF`), and QR (`DGEQRF`). The QUARK API [YKD11] has been implemented and extended in XKaapi to support task multi-specialization: the XKaapi run-time system maintains the CPU and GPU versions for each PLASMA task. At the task execution, our QUARK version runs the appropriate task implementation in accordance with the worker architecture. The GPU kernels of QR and LU are based on previous works from [Agu+11a; Agu+11b] and adapted from PLASMA CPU algorithm and MAGMA from [TDB10]. Each running GPU monopolizes a CPU core to manage its worker. The remaining CPU cores are involved in the application computations.

Methodology

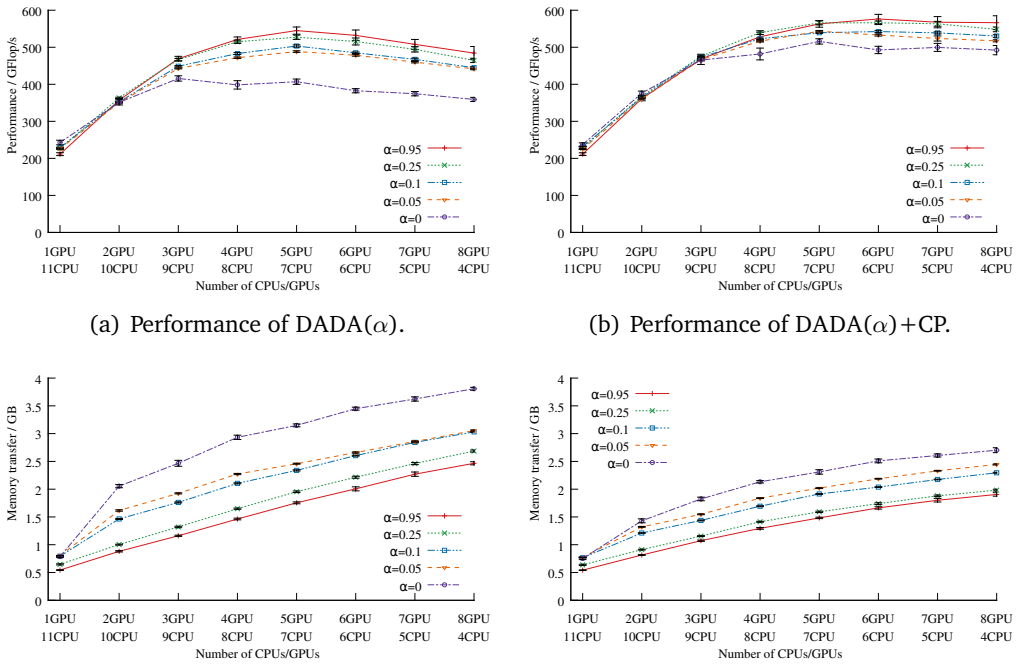
Each experiment has been executed at least 30 times for each set of parameters. We report on all the figures (Figures 2.3, 2.4, 2.5 and 2.6) the mean and the 95% confidence interval. The factorizations have been done in double precision floating-point operations with a PLASMA internal block (*IB*) of size 128 and tiles of size 512. For each of them, we plot the highest performance obtained on various matrix sizes with the discussed scheduling policies.

In the following, DADA(α) represents DADA parametrized by α. We denote by DADA(α)+CP the algorithm using Communication Prediction as supplementary information. Note that DA-2 and DADA(0) denote the same algorithm: we use the latter form to denote this algorithm. HEFT policy always computes the earliest finish time of a task taking into account the time to transfer data before executing the task.

2.6.2 Impact of the Affinity Control Parameter α

This section highlights the impact of the affinity control parameter α on the compromise between performance and data transfers. The measures have been done with the Cholesky decomposition on matrices of size 8192×8192 and 16384×16384. However, we present only results for the smallest size as we observe similar behaviors for both matrix sizes.

Figure 2.3 shows both performance (Figures 2.3(a) and 2.3(b)) and total memory transfers (Figures 2.3(c) and 2.3(d)) for several values of α with respect to the number of GPUs. Both metrics are shown without (Figures 2.3(a) and 2.3(c)) and



(a) Performance of DADA(α).

(b) Performance of DADA(α)+CP.

(c) Memory transfer of DADA(α).

(d) Memory transfer of DADA(α)+CP.

Figure 2.3 Impact of the affinity parameter α on Cholesky factorization (`DPOTRF`) with matrix of size 8192×8192.

with (Figures 2.3(b) and 2.3(d)) communication prediction taken into account. Once affinity is considered (i.e., $\alpha \neq 0$), the higher the value of α, the better the policy scales. Using as little information as possible (i.e., DADA(0) and no communication prediction), the policy performance does not scale with more than two GPUs because of too many transfers.

2.6.3 Comparison of Scheduling Policies

We present in this section the results for the three kernels (`DPOTRF`, `DGETRF`, and `DGEQRF`) with matrix size 8192×8192. Other tested sizes have the same behavior. The idea is to evaluate the behavior of each policy with different workloads. We study both performance and data transfers of the policies introduced above: HEFT, DADA(0), DADA(α) and DADA(α)+CP.

Experimental Evaluation

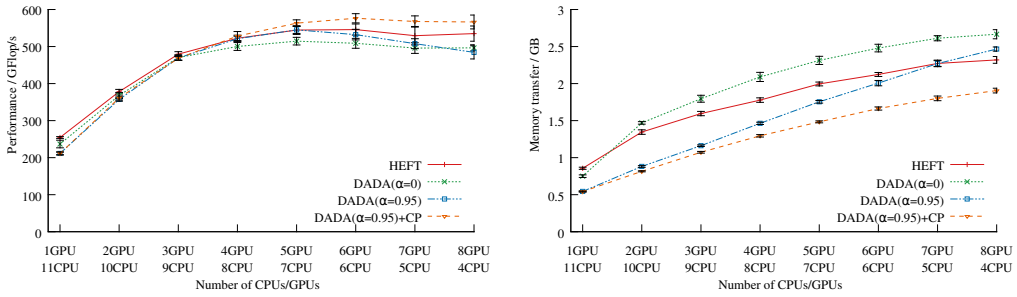
Figure 2.4 reports the behavior of the Cholesky decomposition (`DPOTRF`) with respect to the number of GPUs used. It studies both performance results (Figure 2.4(a)) and total memory transfers (Figure 2.4(b)). All scheduling algorithms have similar performances. DADA(α)+CP scales slightly better with the number of GPU. As expected, DADA(α)+CP and DADA(α) are the policies with the lowest bandwidth footprint up to six GPUs. Yet, as the number of GPU grows, the use of communication prediction allows to reduce the communication volume with sustained high performances.

Figure 2.5 reports the behavior of the LU factorization (`DGETRF`). It studies both performance results (Figure 2.5(a)) and total memory transfers (Figure 2.5(b)). Apart from the performance of DADA(α)+CP for six CPUs and six GPUs (with a large confidence interval), all scheduling policies sustain the same performance. Data transfers seem to have a little impact on performance. However, DADA(α)+CP generates less memory movements than other policies. DADA(0) is the costliest policy while DADA(α) and HEFT have similar impacts. The total memory transfers of the LU and the Cholesky factorizations behave similarly. Still, the gap between the curves is widening: DADA(α)+CP is 3.5 times less demanding in bandwidth than HEFT for only a slowdown of about 1.13 in performance for eight GPUs.

Finally, Figure 2.6 reports the behavior of the QR factorization (`DGEQRF`) with respect to the number of GPUs used. Both performance results (Figure 2.6(a)) and total memory transfers (Figure 2.6(b)) are studied. All dual approximations (DADA(0), DADA(α), DADA(α)+CP) behave the same, and are outperformed by HEFT. Even the low transfer footprint of both DADA(α) is not able to sustain performance. It seems that the dependencies between tasks for QR factorization have a strong impact on the schedule computed by all dual approximation algorithms. We are still investigating this particular point.

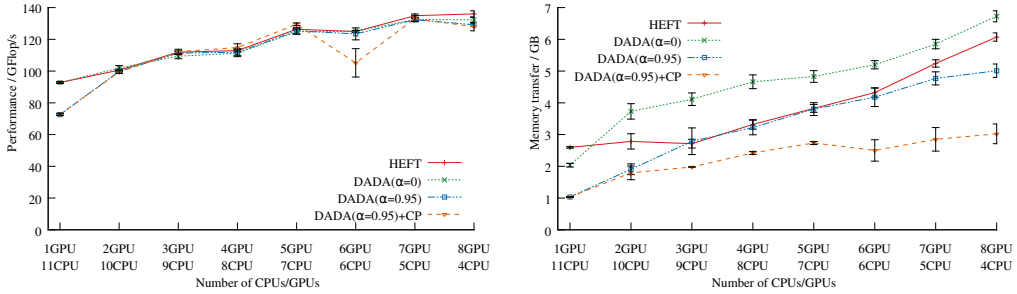
Discussion

Communication Prediction Affinity is a viable alternative to communication modeling. Indeed, DADA without communication prediction is comparable to HEFT in terms of performance. Moreover, affinity based policy combined with communication prediction allows to further reduce the amount of memory transfers (up to a factor 3.5 when compared to HEFT).



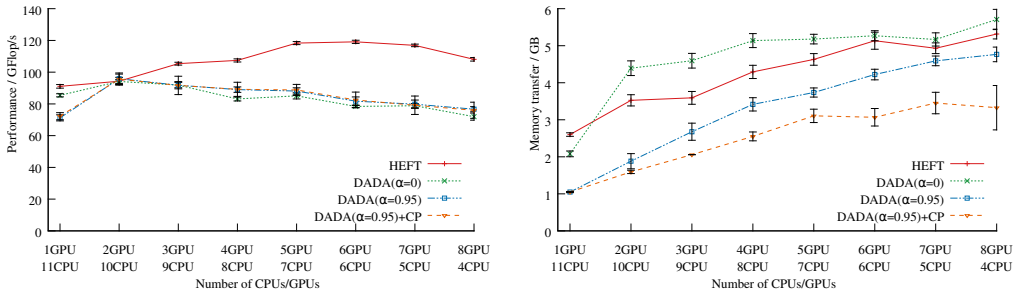
(a) Performance (8192×8192). (b) Memory Transfer (8192×8192).

Figure 2.4 Benchmarks of Cholesky factorization (`DPOTRF`).



(a) Performance (8192×8192). (b) Memory Transfer (8192×8192).

Figure 2.5 Benchmarks of LU factorization (`DGETRF`).



(a) Performance (8192×8192). (b) Memory Transfer (8192×8192).

Figure 2.6 Benchmarks of QR factorization (`DGEQRF`).

Comparison with Work Stealing Scheduling Algorithm For the sake of completeness, we also tested the work stealing algorithm. However we did not plot the results in previous figures for the sake of readability. We briefly discuss them now. The naive work stealing algorithm is cache unfriendly, especially with small matrices as its random choices are heavily penalizing [Gau+13]. On the contrary, the affinity policies proposed here are suitable for this case. When scheduling for medium and large matrix sizes, the impact of modeling inaccuracies grows. Model oblivious algorithms such as work-stealing behave well by efficiently overlapping communications and computations while HEFT is misled by the imprecise communication predictions. Hence, our approach is much more robust than work stealing and HEFT since it does not need too precise a communication model, and adapts well to various matrix sizes.

2.7 Summary

In this chapter we designed new algorithms, and we proposed the evaluation of various generic scheduling algorithms for hybrid architectures consisting in multi-core machines augmented with GPUs. This evaluation has been conducted by implementing the algorithms on top of the XKaapi parallel run-time. The behaviors of the algorithms have been studied with three dense linear algebra kernels. The studied algorithms (DP-4/3, and DADA variations) are based on the dual approximation, and provide performance guarantees. Their performances have been compared to the *de facto* standard HEFT algorithm. Among the studied algorithms, DADA is a new approach. DADA relies on *affinity* to introduce context-awareness at a tractable cost.

We first validate that the trade-off between the computational complexity and the performance guarantee matters. Despite having a better guarantee than DADA, DP-4/3 is not a viable option in such a setup as the algorithm is too slow to compute a schedule.

Although HEFT achieves the best absolute performance on the QR factorization, DADA has similar or better performances on the Cholesky and LU factorizations for large numbers of GPU. Nevertheless, DADA allows to significantly reduce the amount of data transfers with a negligible impact on raw performances.

More interestingly, thanks to its affinity criterion, DADA proposes a qualitative way to take communications into account. This is a valuable alternative to the precise

communication cost models which are required by HEFT to predict the completion time of tasks.

3 Scheduling Independent Moldable Tasks on Multi-Cores with GPUs

In the previous chapter, we showed first steps to devise generic scheduling algorithms for heterogeneous compute nodes. In particular, we have developed an approximation algorithm with a constant worst-case performance guarantee that provides solutions for the scheduling problem of independent, sequential tasks onto CPUs or GPUs for the makespan objective. However, the DP-4/3 algorithm had two main drawbacks. First, even though the algorithm has a polynomial-time complexity, it relies on dynamic programming, in which a vast state space has to be explored. For that reason, the practical applicability of the algorithm is limited due to its large run-time. Second, tasks could potentially benefit from internal (data-)parallelism on CPUs, and the previous algorithm worked for sequential tasks only. Thus, in the current chapter we assume that tasks are *moldable*, i.e., they are computational units that may be executed by several (more than one) processors. Then, the run-time of such a moldable task depends on the number of processors allotted to it. Such a model allows us to exploit the two types of parallelism offered by hybrid parallel computing platforms: the inherent parallelism induced by GPU's architecture, and the parallelization of tasks on several CPUs. The objective of this chapter is to propose a generic method to leverage these two different kind of parallelism.

This chapter reflects a publication [Ble+17] done in collaboration with Sascha HUNOLD, Safia KEDAD-SIDHOUM, Florence MONNA, Grégory MOUNIÉ, and Denis TRYSTRAM.

Compared to the state of the art, the contributions of this chapter are:

- To present a novel algorithm—combining dual approximation and integer linear programming—that can solve the scheduling problem of independent, moldable tasks on hybrid parallel compute platforms consisting of m CPUs and k GPUs.

- To prove that this algorithm has an approximation ratio of $\frac{3}{2} + \epsilon$.

- To show through a sequence of experiments that even though our algorithm is based on integer linear programming, which may be theoretically non-polynomial, it is still practically relevant, as it provides competitive schedules, and has a relatively short run-time.

- To present a fully polynomial-time algorithm for the same scheduling problem, for which we prove an approximation ratio of $2 + \epsilon$.

The chapter is organized as follows: in Section 3.1, we define the scheduling problem targeted in this work. We examine existing related works on scheduling with GPUs and moldable tasks in Section 3.2. We present our novel scheduling algorithm, which is based on integer linear programming (ILP), in Section 3.3 and provide the theoretical analysis of the algorithm in Section 3.4. Section 3.5 presents a fully polynomial approximation algorithm, which we introduce to compare with our ILP-based algorithm. In Section 3.6, we present an experimental study that compares the solution quality (makespan) of various scheduling algorithms for a variety of test instances. We conclude the chapter in Section 3.7.

3.1 Problem Definition

We target the same architecture as in Chapter 2. We consider a multi-core parallel platform composed of m identical CPUs and k identical GPUs. An instance of the problem is described as a set $\{T_1, \ldots, T_n\}$ of n independent tasks considered as *moldable* when assigned to the CPUs and sequential when assigned to a GPU. The processing time of any task T_j is represented by a function $\overline{p_j} : l \mapsto \overline{p_{j,l}}$ that represent the processing time when executed on l CPUs and by $\underline{p_j}$ that is the processing time when executed on a GPU. We assume that these processing times are known in advance.

Recall that the *scheduling* problem consists in finding a function σ that associates for each task T_j its starting time and the computing resources assigned for its execution. A task is either assigned to a single GPU or to a subset of the available CPUs, under the constraints that the task starts its execution simultaneously on all the allocated resources and occupies them without interruption until its completion time.

We define the CPU work function w_j of a task T_j as $w_j : l \mapsto w_{j,l} = l \times \overline{p_{j,l}}$ for $l \leqslant m$. The value $w_{j,l}$ corresponds to the computational area—in the Gantt chart representation of a schedule—of the task T_j when executed on l CPUs. According to the usual executions of parallel programs, we assume that the tasks assigned to the

CPUs are monotonic: assigning more CPUs to a task usually decreases its execution time at a price of an increased work. This is due to some internal communications and synchronizations. There are two types of monotonicities: namely the *time monotonicity* which is achieved when $\overline{p_j}$ is a non-increasing function for any task, and the *work monotonicity* that is achieved when w_j is a non-decreasing function for any task. A set of tasks is said to be *monotonic* when it achieves both monotonicities. This assumption may be interpreted by the well-known Brent's lemma [Bre74], which states that the parallel execution of a task achieves some speedup if it is large enough, but does not lead to super-linear speedups. Notice that an instance of the problem can always be transformed to fulfill the time monotonicity property by replacing function $\overline{p_j}$ by $\overline{p_j}' : l \mapsto \min\{\overline{p_{j,q}} \mid q \in 1, \ldots, l\}$. Such a transformation does not affect the optimal solution of the scheduling. In the sequel, we always assume that the set of tasks of the considered instance is monotonic. There is no need for such a hypothesis on the GPUs as the tasks are considered sequential on this architecture.

The makespan is defined as the maximum completion time of the last finishing task. For the problem considered here, the objective is to minimize the makespan of the whole schedule, which is the maximum of the makespan on the CPUs and the makespan on the GPUs. The problem is denoted by $(Pm, Pk) \mid mold \mid C_{\max}$.

Observe that if all the tasks are sequential and the processing times are the same on both devices ($\underline{p_j} = \overline{p_{j,1}}$) for $j \in 1, \ldots, n$, the problem $(Pm, Pk) \mid mold \mid C_{\max}$ is equivalent to the classical $P \mid\mid C_{\max}$ problem, which is **NP**-hard. Thus, the problem of scheduling moldable tasks with GPUs is also **NP**-hard, and we are looking for efficient algorithms with guaranteed approximation ratio. Recall that for a given problem the approximation ratio ρ_A of an algorithm A is defined as the maximum over all the instances I of the ratio $\frac{f(I)}{f^*(I)}$ where f is any minimization objective and f^* is its optimal value.

This study considers algorithms that provide non-preemptive schedules with contiguous processor assignment. It is clear that the optimal assignment could use CPUs that are not consecutive ones. However, this restriction does not hinder the achieved results [MRT07].

3.2 Related Work

We present in this section existing results dealing with moldable tasks. Please refer to Section 2.2 for a discussion about heterogeneous scheduling and the problem $(Pm, Pk) \mid\mid C_{\max}$.

The problem of scheduling independent moldable tasks on homogeneous parallel systems has been extensively studied in the last decade. Among other reasons, the interest in studying this problem was motivated by scheduling jobs in batch processing in HPC clusters. For instance, the documentation of TORQUE mentions a basic moldable submission mechanism [@torque]. A noteworthy work is the implementation and evaluation of a moldable scheduler for OAR by Lionel Eyraud [Eyr06].

Jansen and Porkolab [JP99] proposed a Polynomial-Time Approximation Scheme based on a linear programming formulation for scheduling independent moldable tasks. The complexity of their scheme, although linear in the number of tasks, is highly dependent on the accuracy of the approximation due to an exponential factor in the number of processors. Thus, although the result is of significant theoretical interest, this algorithm cannot be considered for a practical use.

Most existing previous works are based on a two-phase approach, initially proposed by Turek, Wolf and Yu [TWY92]. The basic idea here is to select first an assignment (the number of processors assigned to each task), and in a second step to solve the resulting rigid (non-moldable) scheduling problem. The rigid task scheduling problem is a classical scheduling problem with multiprocessor tasks. As far as the makespan objective is concerned, this problem is related to a 2-dimensional strip-packing problem for independent tasks [Bou+10a; Cof+80].

It is clear that applying an approximation of guarantee ρ for the rigid problem on the assignment of an optimal solution provides the same guarantee ρ for the moldable problem if ever an optimal assignment can be found. Two complementary ways have been proposed for solving the problem, either focusing on the first phase of assignment or on the scheduling (second phase). Ludwig improved the complexity of the assignment selection in the special case of monotonic tasks leading to a 2-approximation [LT94]. The other way corresponds to choosing an assignment such that the resulting non-moldable problem is not a general instance of strip-packing, and hence better specific approximation algorithms can be applied. Using the knapsack problem as an auxiliary problem for the selection of the assignment, this technique leads to a $(\frac{3}{2} + \epsilon)$-approximation algorithm for any $\epsilon > 0$ [MRT07].

Furthermore, an extensive, both theoretical and experimental comparison of low-cost scheduling algorithms for moldable tasks has been carried out by Fan et al. [Fan+12].

3.3 Algorithm APPROX-3/2

The principle of the algorithm is based on a dual approximation [HS87]. Recall that a ρ-dual approximation algorithm for our scheduling problem uses a guess λ of the makespan. The ρ-dual approximation algorithm then finds a schedule of makespan at most $\rho\lambda$ if there exists a feasible schedule of makespan at most λ, or the algorithm rejects λ. We target $\rho = \frac{3}{2}$. Let λ be the current real number input for the dual approximation. In the whole section, we suppose there exists a schedule of length lower than λ, and we show how to build a schedule of length at most $\frac{3\lambda}{2}$.

Given a positive number h, we define—as in [MRT07]—for each task T_j its *canonical number* of CPUs $\gamma(j,h)$. The number $\gamma(j,h)$ is the minimal number of CPUs needed to execute task T_j in time at most h. If T_j cannot be executed in time less than h on m CPUs, we set by convention $\gamma(j,h) = +\infty$. Observe that $w_{j,\gamma(j,h)}$ is the minimal work area needed to execute T_j on CPUs in time at most h. Also note that if the set of tasks is monotonic, the canonical number of CPUs can be found in time $\mathcal{O}(\log(m))$ by binary search.

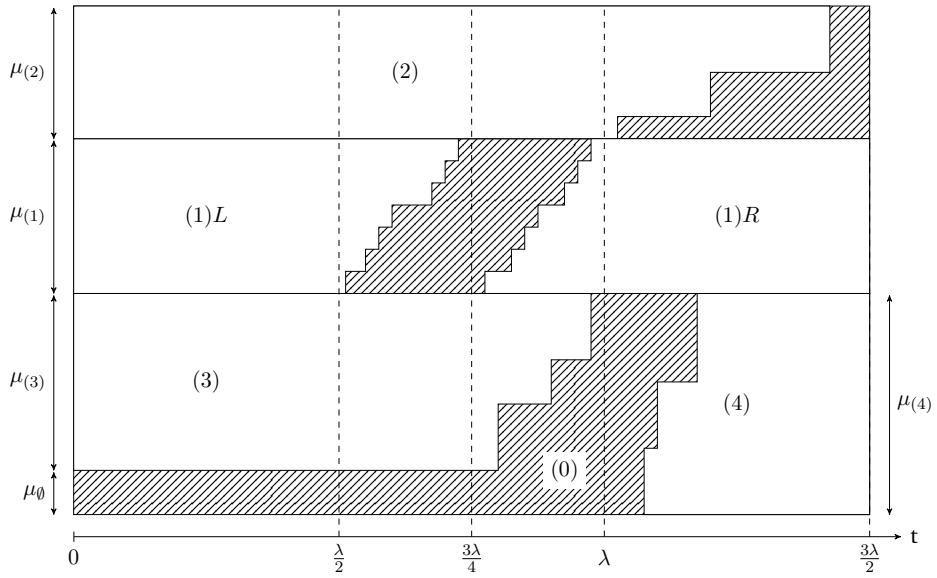
3.3.1 Partitioning Tasks

The idea of the algorithm is to partition the set of tasks on the CPUs into five sets, and on the GPUs into two sets, as depicted in Figure 3.1. This choice of the task assignement to CPUs is detailed below:

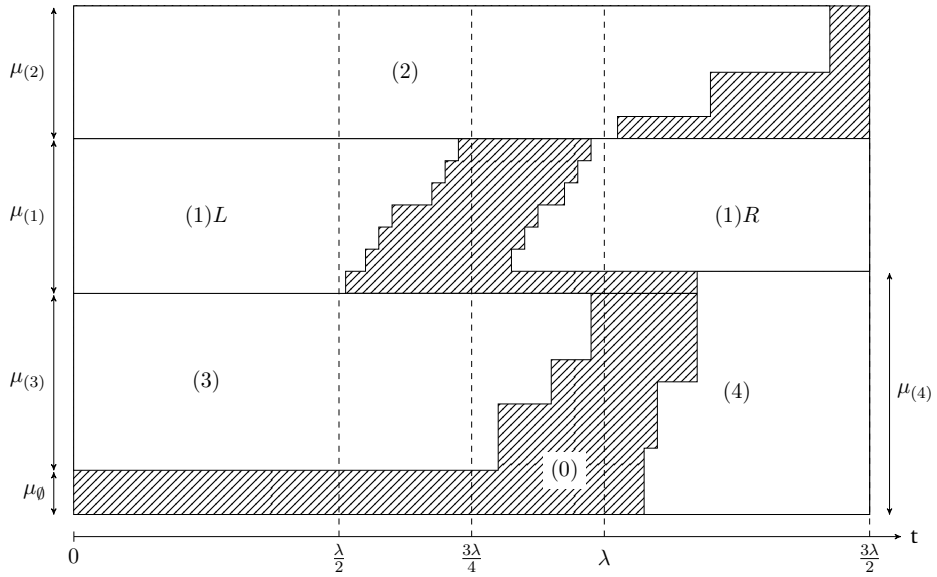
(0): the set containing the sequential tasks assigned to the CPUs with a processing time at most $\frac{\lambda}{2}$.

(1): the set containing the sequential tasks assigned to the CPUs with a processing time greater than $\frac{\lambda}{2}$ and at most $\frac{3\lambda}{4}$. This set is partitioned in two shelves as depicted in Figure 3.1: namely, the left set $(1)L$ and the right set $(1)R$.

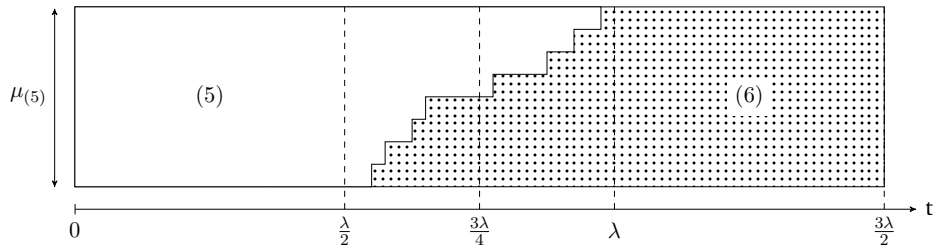
(2): the set containing the tasks—either parallel or sequential—assigned to the CPUs with different canonical numbers of CPUs for the times λ and $\frac{3\lambda}{2}$. Task T_j is then assigned to $\gamma\left(j, \frac{3\lambda}{2}\right)$ CPUs.



(a) Structure of the schedule on the CPUs with an even number of tasks in set (1).



(b) Structure of the schedule on the CPUs with an odd number of tasks in set (1).



(c) Structure of the schedule on the GPUs.

Figure 3.1 Structure in seven sets of the schedule. The number of processors used by set (i) is denoted by $\mu_{(i)}$. The number of CPUs below set (3) is denoted by $\mu_{\emptyset}$.

(3): the set containing the tasks assigned to their canonical number of CPUs for time λ. If this number is 1, then the processing time of the corresponding task is strictly greater than $\frac{3\lambda}{4}$.

(4): the set containing the parallel tasks assigned to their canonical number of CPUs for time $\frac{\lambda}{2}$. Note that $\gamma\left(j, \frac{\lambda}{2}\right)$ is greater than 1.

Similarly, the tasks assigned to GPUs are partitioned in two sets:

(5): the set containing the tasks assigned to a GPU with a processing time greater than $\frac{\lambda}{2}$ and at most λ.

(6): the set containing the tasks assigned to a GPU with a processing time at most $\frac{\lambda}{2}$.

Such a partition ensures that both the makespans on the CPUs and on the GPUs are lower than $\frac{3\lambda}{2}$.

Note that if there is an even number of tasks assigned to set (1), both sets $(1)L$ and $(1)R$ occupy the same number of CPUs. On the contrary, if set (1) contains an odd number of tasks, the right set occupies one less processor (as shown in Figure 3.1(b)).

3.3.2 Mathematical Formulation

Partitioning tasks into the seven above-mentioned sets using a list algorithm does not achieve the desired performance guarantee. Therefore, we propose an Integer Linear Program (ILP) for solving the assignment problem.

Objective Function and Constraints

We define $W_{\mathcal{C}}$ as the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the works of the tasks assigned to some CPUs:

$$W_{\mathcal{C}} = \sum_{T_j \in (0)\cup(1)} w_{j,1} + \sum_{T_j \in (2)} w_{j,\gamma\left(j,\frac{3\lambda}{2}\right)} + \sum_{T_j \in (3)} w_{j,\gamma(j,\lambda)} + \sum_{T_j \in (4)} w_{j,\gamma\left(j,\frac{\lambda}{2}\right)}$$

We want to obtain a specific five-set schedule on the CPUs and a two-set schedule on the GPUs. The assignment *minimizes* the total computational area $W_{\mathcal{C}}$ on the CPUs, and the assignment must satisfy the following constraints:

(C_1) The total computational area $W_{\mathcal{C}}$ on the CPUs is at most $m\lambda$.

(C_2) Sets $(1)L$, (2) and (3) use a total of at most m processors.

(C_3) Sets $(1)R$, (2) and (4) use a total of at most m processors.

(C_4) The total computational area on the GPUs is at most $k\lambda$.

(C_5) Set (5) uses a total of at most k processors.

(C_6) Each task is assigned to exactly one set.

(C_7) The number of tasks assigned to set (1) is the total number of tasks processed in its two shelves.

(C_8) The tasks of set (1) are evenly shared between its two sets $(1)L$ and $(1)R$, i.e. there is at most one task less in $(1)R$. The idle time induced by the difference is used to process a fraction of a task assigned to set (4).

Such an assignment clearly defines a schedule of length at most $\frac{3\lambda}{2}$ which allows us to build a solution.

Filtering

The structure of the schedule allows tasks to belong only to a limited number of shelves. Hence we define for each task j the filtering function $F(j)$ computing the set of possible containing shelves. For each set (i) we also define the set $\mathcal{T}^{(i)}$ of tasks eligible for an allocation in (i). The eligible allocation sets are explicitly defined in Equation (3.1). We furthermore define for each task T_j several binary variables $x_j^{(i)}$, where $i \in F(j)$. If T_j is assigned to set (i), we define $x_j^{(i)}$ to be 1. Otherwise we set $x_j^{(i)}$ to be 0. We also define for set (1) the variables $left^{(1)}$ and $right^{(1)}$, which

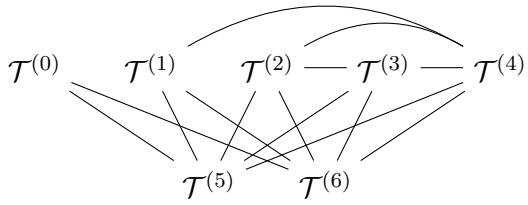


Figure 3.2 Intersection graph of the eligible allocation sets, in its most generic shape. Actual instances may have fewer edges.

correspond to the number of tasks assigned to the left ($(1)L$) and to the right ($(1)R$) of set (1), respectively (see Figure 3.1).

$$
\begin{aligned}
\mathcal{T}^{(0)} &= \left\{ j \mid \overline{p_{j,1}} \leqslant \frac{\lambda}{2} \right\} \\
\mathcal{T}^{(1)} &= \left\{ j \mid \frac{\lambda}{2} < \overline{p_{j,1}} \leqslant \frac{3\lambda}{4} \right\} \\
\mathcal{T}^{(2)} &= \left\{ j \mid \lambda < \overline{p_{j,\gamma\left(j,\frac{3\lambda}{2}\right)}} \leqslant \frac{3\lambda}{2} \right\} \\
\mathcal{T}^{(3)} &= \left\{ j \mid \frac{\lambda}{2} < \overline{p_{j,\gamma(j,\lambda)}} \leqslant \lambda \right\} \setminus \mathcal{T}^{(1)} \\
\mathcal{T}^{(4)} &= \left\{ j \mid \overline{p_{j,\gamma\left(j,\frac{\lambda}{2}\right)}} \leqslant \frac{\lambda}{2} \ \wedge \ \gamma\left(j, \frac{\lambda}{2}\right) > 1 \right\} \\
\mathcal{T}^{(5)} &= \left\{ j \mid \frac{\lambda}{2} < \underline{p_j} \leqslant \lambda \right\} \\
\mathcal{T}^{(6)} &= \left\{ j \mid \underline{p_j} \leqslant \frac{\lambda}{2} \right\}
\end{aligned}
\tag{3.1}
$$

This filtering step helps a lot reducing the search space. The intersection graph of the eligible allocation sets shown in Figure 3.2 explains this behavior. Each task can simultaneously belong to only a limited number of sets, since most sets are mutually exclusive. In most cases, a task belong to 2 or 3 sets.

Integer Linear Program

Determining if such an assignment exists reduces to solving an ILP that can be formulated as follows:

$$\begin{aligned}
\min W_{\mathcal{C}}^{(ILP)} = & \sum_{j\in\mathcal{T}^{(0)}} w_{j,1}x_j^{(0)} + \sum_{j\in\mathcal{T}^{(1)}} w_{j,1}x_j^{(1)} + \sum_{j\in\mathcal{T}^{(2)}} w_{j,\gamma\left(j,\frac{3\lambda}{2}\right)}x_j^{(2)} \\
& + \sum_{j\in\mathcal{T}^{(3)}} w_{j,\gamma(j,\lambda)}x_j^{(3)} + \sum_{j\in\mathcal{T}^{(4)}} w_{j,\gamma\left(j,\frac{\lambda}{2}\right)}x_j^{(4)} \\
\text{s.t. } & W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda & (C_1) \\
& \sum_{j\in\mathcal{T}^{(2)}} \gamma\left(j,\frac{3\lambda}{2}\right)x_j^{(2)} + \sum_{j\in\mathcal{T}^{(3)}} \gamma\left(j,\lambda\right)x_j^{(3)} + left^{(1)} \leqslant m & (C_2) \\
& \sum_{j\in\mathcal{T}^{(4)}} \gamma\left(j,\frac{\lambda}{2}\right)x_j^{(4)} + \sum_{j\in\mathcal{T}^{(2)}} \gamma\left(j,\frac{3\lambda}{2}\right)x_j^{(2)} + right^{(1)} \leqslant m & (C_3) \\
& \sum_{j\in\mathcal{T}^{(5)}} \underline{p_j}x_j^{(5)} + \sum_{j\in\mathcal{T}^{(6)}} \underline{p_j}x_j^{(6)} \leqslant k\lambda & (C_4) \\
& \sum_{j\in\mathcal{T}^{(5)}} x_j^{(5)} \leqslant k & (C_5) \\
& \sum_{i\in F(j)} x_j^{(i)} = 1 \quad \forall j \in \{1,\ldots,n\} & (C_6) \\
& \sum_{j\in\mathcal{T}^{(1)}} x_j^{(1)} = left^{(1)} + right^{(1)} & (C_7) \\
& 0 \leqslant left^{(1)} - right^{(1)} \leqslant 1 & (C_8) \\
& x_j^{(i)} \in \{0,1\} \quad \forall j \in \{1,\ldots,n\}, \forall i \in F(j) & (C_9) \\
& left^{(1)}, right^{(1)} \in \{0,\ldots,m\} & (C_{10})
\end{aligned}$$

The first eight equations of this integer linear program correspond to the constraints listed above in order to obtain a five-set schedule on the CPUs and a two-set schedule on the GPUs. The last two equations (C_9), and (C_{10}) are integrity constraints for the variables of the integer linear program.

3.4 Analysis of the Algorithm APPROX-3/2

The integer linear program presented above derives from the structural properties of the schedule we aim to construct. The analysis—rather technical—is structured in three steps. First we explain how the estimation of the instance's makespan λ

helps us to sort and allocate tasks. We then give some insight on the structure of the proposed partitioning. We finally prove the correctness of the dual approximation, i.e. we prove the reject condition is actually matched by the algorithm.

3.4.1 Structure of a Schedule of Makespan λ

To take advantage of the dual approximation paradigm, we have to make explicit the consequences of the assumption that there exists a schedule of length at most λ. We state below some straightforward properties of such a schedule. They should give the insight for the construction of the solution.

Proposition 3.1. *In a solution of makespan at most λ, the execution time of each task is at most λ. The computational areas on the CPUs and GPUs are at most $m\lambda$ and $k\lambda$, respectively.*

Remark that for the problem of scheduling moldable tasks on identical processors [MRT07], it is enough to look at the $2m$ tasks with the longest processing times. If they have a computational area larger than $m\lambda$, then a schedule of length λ cannot exist. In the case of heterogeneous processors some of these tasks can be assigned to a GPU, therefore the n tasks have to be considered in our case.

Proposition 3.2. *In a solution of makespan at most λ, if there exist two consecutive tasks on the same processor such that one of the task has an execution time greater than $\frac{\lambda}{2}$, then the other task has an execution time lower than $\frac{\lambda}{2}$.*

Proposition 3.3. *Two tasks with sequential processing times on CPU greater than $\frac{\lambda}{2}$ and lower than $\frac{3\lambda}{4}$ can be executed successively on the same CPU within a time at most $\frac{3\lambda}{2}$.*

We now look at exploiting the properties of a schedule of makespan at most λ, in order to construct the seven sets. The constraints of the integer linear program derive from these properties.

From Proposition 3.3, as we aim at a makespan of $\frac{3\lambda}{2}$, two tasks from set (1) can be executed successively on the same CPU. The whole set occupies $\mu_{(1)}$ CPUs. The number of tasks in set $(1)R$ is given by $\mu_{(1)} - \mathbf{1}_{(1)\text{odd}}$ where $\mathbf{1}_{(1)\text{odd}}$ is an indicator function equals to 1 when the number of tasks in set (1) is odd.

From Proposition 3.2, the tasks whose execution times on CPUs are greater than $\frac{\lambda}{2}$ do not use more than $m - \mu_{(1)}$ CPUs, and hence can be executed concurrently on the CPUs in set (3). They occupy $\mu_{(3)}$ CPUs.

Set (2) does not exist in a solution of makespan λ, since the processing times of all the tasks in (2) are greater than λ with the number of CPUs they are assigned to. However, with this assignment and the work monotonicity of the tasks on CPUs, the work of the tasks in (2) is lower than their corresponding work in the optimal schedule. Therefore, every task assigned to (2) in the constructed schedule is a gain on the total work on the CPUs. The tasks of (2) occupy $\mu_{(2)}$ CPUs and the inequality $\mu_{(1)} + \mu_{(2)} + \mu_{(3)} \leqslant m$ must be satisfied.

The remaining tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU, and those that are not sequential can be executed within a time at most $\frac{\lambda}{2}$ in set (4). These tasks cannot be executed on the CPUs occupied by tasks from set (2) but can be processed after the tasks from set (3). They cannot be on the CPUs that already process two tasks of (1), but if the number of tasks in (1) is odd, there is a CPU that only processes one task from $(1)L$ and a task from (4) can be executed on this CPU. Therefore, if we denote by $\mu_{(4)}$ the number of CPUs occupied by tasks of (4), the inequality $\mu_{(1)} - \mathbf{1}_{(1)\text{odd}} + \mu_{(2)} + \mu_{(4)} \leqslant m$ must be satisfied.

The remaining sequential tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU and are assigned to set (0).

With the same reasoning, the tasks on GPUs whose execution times are greater than $\frac{\lambda}{2}$ do not use more than k GPUs, and hence can be executed concurrently in set (5).

The remaining tasks on GPUs have execution times lower than $\frac{\lambda}{2}$ on GPU and can be executed within a time at most $\frac{\lambda}{2}$ in set (6) on the GPUs. It can be after a task from (5) or on the remaining free GPUs.

3.4.2 Structure of the Partitioning

We now have to prove that, under the assumption that the dual approximation does not reject the current guess λ (i.e., $W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda$) the ILP solution leads to a feasible seven-shelf schedule.

The structure of the partitioning verifies some properties exposed hereinafter.

Lemma 3.1. *With the assumption that $W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda$, the tasks assigned to sets (1), (2), (3) and (4) occupy at most m CPUs, in a time at most $\frac{3\lambda}{2}$.*

Proof. From Constraints (C_2) and (C_3), the assignment of the tasks of the four sets is such that they occupy at most m CPUs. The tasks are scheduled two by two in (1). According to Constraint (C_8), set (1) may have an even number of tasks (see Figure 3.1(a)) or an odd number of tasks (see Figure 3.1(b)). Whenever set (1) is assigned an odd number of tasks, an extra processor is available to compute tasks from set (4). The tasks of (4) are scheduled after tasks of (3) or on remaining free CPUs. With this schedule, at most m CPUs are occupied and the makespan is at most $\frac{3\lambda}{2}$. □

Lemma 3.2. *If $W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda$, the tasks assigned to set (0) fit in the remaining free computational space, while keeping the makespan under $\frac{3\lambda}{2}$.*

Proof. The tasks of set (0) all have a sequential processing time on CPU lower than $\frac{\lambda}{2}$ by construction, and they necessarily fit into the remaining computational space in the allowed area of $\frac{3\lambda}{2}\,m$ (represented by the dashed area in Figure 3.1). The schedule would otherwise contradict Proposition 3.1.

The following algorithm can be used to schedule these tasks:

1. Consider the remaining tasks $T_1, \ldots, T_f$ ordered by non-increasing sequential processing time on CPU, where f is the number of remaining tasks.

2. At each step s ($s \in 1, \ldots, f$) assign task T_s to the least loaded CPU, at the latest possible date, or between set (3) and set (4) if relevant. Update the CPU's load.

At each step, the least loaded CPU has a load at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Proposition 3.1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{2}$, and can contain the task T_s. This proves the correctness of the algorithm above. □

Lemma 3.3. *If $W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda$, the tasks assigned to sets (5) and (6) occupy at most k GPUs, in a time at most $\frac{3\lambda}{2}$.*

Proof. When the tasks of set (5) are assigned to the GPUs, they take up to k GPUs from Constraint (C_5), and their processing time is lower than λ: the dual approximation would otherwise reject the solution. The tasks of set (5) are scheduled first, one per GPU.

The tasks of set (6) all have a processing time on GPU lower than $\frac{\lambda}{2}$ by construction and they necessarily fit into the remaining computational space in the allowed area of $\frac{3\lambda}{2}\,k$. The schedule would otherwise contradict Proposition 3.1 and Constraint (C_4).

The following algorithm can be used to schedule these tasks:

1. Consider the remaining tasks $T_1, \ldots, T_f$ ordered by non-increasing processing time on GPU, where f is the number of remaining tasks.
2. At each step s ($s \in 1, \ldots, f$) assign task T_s to the least loaded GPU, at the latest possible date. Update the GPU's load.

At each step, the least loaded GPU has a load at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $k\lambda$ (according to Proposition 3.1 and Constraint (C_4)). Hence, the idle time interval on the least loaded GPU has a length at least equal to $\frac{\lambda}{2}$ and can contain the task T_s. The correctness of the algorithm above is proved. □

These three lemmas allow us to derive the following theorem:

Theorem 3.4. *If $W_{\mathcal{C}}^{(ILP)} \leqslant m\lambda$, then there exists a schedule of length at most $\frac{3\lambda}{2}$ built upon the assignment of the tasks given by the solution of ILP.*

Proof. The solution of ILP returns an assignment such that the computational area on the CPUs is minimized, therefore its value $W_{\mathcal{C}}^{(ILP)}$ is lower than the computational area on the CPUs in the optimal schedule, $W_{\mathcal{C}}^{*}$, which is lower than $m\lambda$ since we assumed that there exists a schedule of makespan at most λ. The three lemmas show that the schedule constructed with the assignment of the tasks given by the solution of ILP has a makespan lower than $\frac{3\lambda}{2}$. □

If the computational area on the CPUs, i.e. the objective of the integer linear program $W_{\mathcal{C}}^{(ILP)}$, is greater than $m\lambda$, the dual approximation algorithm rejects λ. Indeed, this computational area is minimized in the resolution of (ILP): if we had $\lambda \leqslant C_{\max}^{*}$, we would get $W_{\mathcal{C}}^{(ILP)} \leqslant W_{\mathcal{C}}^{*}$, which is impossible since we have $W_{\mathcal{C}}^{*} \leqslant m\lambda$. Therefore in that case there exists no solution with a makespan at most λ, and the algorithm rejects the current guess λ.

We have so far proved that—for a given guess λ of the dual approximation algorithm—if the solution of ILP has a computational area on the CPUs greater than $m\lambda$, then

there is no solution of makespan λ, and the guess has to be rejected. If the solution of ILP has a computational area on the CPUs lower than $m\lambda$, then we can construct a solution with a makespan at most $\frac{3\lambda}{2}$, with the corresponding sets on the CPUs and GPUs.

3.4.3 Correctness of the Dual Approximation

It remains to be proved that the existence of a solution of makespan at most λ implies the existence of a solution with the seven-shelf structure. To do so, we first expose and prove two technical lemmas before stating the existence theorem (Theorem 3.7).

Lemma 3.5. *Suppose there exists a solution σ_{ref} of makespan at most λ. The assignment of tasks on the GPUs in σ_{ref} is compatible with the seven-shelf structure.*

Proof. All tasks assigned to the GPUs by σ_{ref} are sequential. Hence we can assign these tasks in two distinct sets: tasks with a processing time strictly greater than $\frac{\lambda}{2}$ and tasks with a processing time lower than $\frac{\lambda}{2}$. These two sets exactly match the sets (5) and (6) of the structure we seek. □

Lemma 3.5 allows us to only consider tasks assigned to the CPUs in the proof of the existence of the sought schedule.

Lemma 3.6. *If there exists a solution σ_{ref} of makespan at most λ, then there exists a solution σ_{struct} with the seven-shelf structure whose sub-solution $\overline{\sigma}_{struct}$ (considering only tasks assigned to CPUs) uses at most m CPUs with a lower CPU load than the CPU load of σ_{ref}.*

Proof. First, let us prove that the big tasks of σ_{ref}, namely tasks with a processing time greater than $\frac{\lambda}{2}$, fit in sets (1), (2) and (3) without using more than m CPUs and without increasing the CPU load:

- The tasks assigned to set (1) are sequential tasks of length greater than $\frac{\lambda}{2}$: their work is minimal. Since their processing time is at most $\frac{3\lambda}{4}$, only one of the tasks assigned to set (1) can fit on one CPU in σ_{ref}, whereas in σ_{struct}, these tasks are stacked by pair, one in shelf $(1)L$, the other in shelf $(1)R$. As a result, the tasks in set (1) in σ_{struct} use fewer processors than they would in σ_{ref}.

- The tasks assigned to sets (2) and (3) are using their canonical number of CPUs for a time limit at least λ, hence they generate a lower or equal work than they would in σ_{ref}. As these tasks use their canonical number of processors for a time limit greater than λ, they use fewer processors than they would in σ_{ref}. Observe that the tasks assigned to set (2) use fewer processors than they do in σ_{ref} thanks to the relaxed time limit.

We now have to consider the tasks of σ_{ref} assigned to CPUs with a processing time lower than $\frac{\lambda}{2}$. All the tasks with a sequential time at most $\frac{\lambda}{2}$ are assigned to set (0). The remaining tasks are the tasks that have been assigned to more than one CPU in σ_{ref}, with a processing time lower than $\frac{\lambda}{2}$. The monotonicity assumption ensures that they can fit in any set among sets (1), (2), (3) and (4) without increasing the computational load. In order to prove that there exists such an assignment of these remaining tasks, we consider the integer linear program introduced in Section 3.3.2 that we relax by removing Constraint (C_3). This allows set (4) to occupy as many CPUs as needed. The tasks already assigned to the GPUs as in σ_{ref} thanks to Lemma 3.5 have their corresponding variables in the integer linear program set according to their assignment. The same is done for the variables of the integer linear program corresponding to the tasks already assigned to sets (1), (2), (3) and (0) above in the proof. We let the integer linear program choose the remaining assignments. By doing so, since Constraint (C_3) was removed, set (4) could use too many CPUs. It remains to prove that the assignment returned by the revised integer linear program does not need to use more than m CPUs. Two cases are to be distinguished: either every CPU is busy or some CPUs remain idle after assigning tasks to sets (1), (2) and (3). The first case's proof is straightforward while the second case's proof is done in three steps.

Let us first consider the case where every CPU is busy. By construction, at most one processor—in set (1)—is loaded less than λ but more than $\frac{\lambda}{2}$. As all the tasks assigned to set (4) have a processing time larger than $\frac{\lambda}{4}$, we cannot use more than m CPUs without contradicting the facts that the integer linear program is minimizing the CPU load and that σ_{ref} exists.

Let us consider now the case where some CPUs remain idle. We denote by $\mu_\emptyset$ their number.

(i) We begin by proving that at most one task in set (4) does not fit. As $\mu_\emptyset > 0$, every task of set (4) has a work greater than $\mu_\emptyset\lambda$, otherwise it would have been assigned to set (3) by the integer linear program. The maximum amount of work by which a task of set (4) could be overreaching is bounded by the gap left between $m\lambda$ and the work of the tasks filling sets (1), (2) and (3). Because of the five-set structure on

the CPUs, such a gap is at most $\frac{3\lambda}{4}\mu_\emptyset + \frac{\lambda}{4}\mathbf{1}_{(1)\text{odd}}$, which is strictly smaller than the work of any task assigned to set (4). The existence of a task in set (4) executed only on processors not meant to do so by the five set structure would contradict the fact that sets (1), (2) and (3) were filled by the integer linear program while minimizing the CPU load. Therefore there is at most a fraction of a task assigned to (4) whose execution requires processors that do not exist.

In the next two steps, we consider an arbitrary assignment for the tasks assigned to set (4) and suppose exactly one task does not fit. We focus on this particular task, denoted by T_Δ. Proving its existence contradicts the fact that the work is minimized by the integer linear program. We denote by $(3)\Delta$ the subset of (3) that shares processors with the task T_Δ.

(ii) We show now that the inequality $\mu_{(3)\Delta} > 2\,\mu_\emptyset$ holds under the assumption that T_Δ exists. The integer linear program chose to assign task T_Δ to set (4). As set (4) is the one creating the most work amongst sets (1), (2), (3) and (4), this choice had to be made because otherwise constraints would have been violated. We know for sure that $\mu_{(3)\Delta} > 0$, otherwise this would contradict Step ***(i)***. Moreover, as T_Δ was not assigned on $\mu_\emptyset$ processors in set (2), its work is greater than $\frac{3\lambda}{2}\mu_\emptyset$. Such a case is only possible if we have enough space next to set (3), which is equivalent to the following inequality:

$$\frac{3\lambda}{4}\mu_{(3)\Delta} + \frac{3\lambda}{2}\mu_\emptyset < (\mu_{(3)\Delta} + \mu_\emptyset)\lambda \tag{3.3}$$

This inequality reduces to the one we are interested in, i.e. $\mu_{(3)\Delta} > 2\,\mu_\emptyset$.

(iii) To finish the proof, let us shows that the previous point leads to a contradiction, hence $\overline{\sigma}_{\text{struct}}$ fits into m CPUs. Inequality (3.3) can be rewritten in the following form:

$$\frac{3\lambda}{4}\mu_{(3)\Delta} + \frac{\lambda}{2}\left(\mu_{(3)\Delta} + \mu_\emptyset\right) > \lambda\left(\mu_{(3)\Delta} + \mu_\emptyset\right)$$

The left part of the sum is a lower-bound of the work of set $(3)\Delta$. The monotonicity ensures that the work of T_Δ is greater than $\frac{\lambda}{2}\left[\gamma\left(T_\Delta, \frac{\lambda}{2}\right) - 1\right]$, and we know that the number of processors needed by task T_Δ is at least $\mu_{(3)\Delta} + \mu_\emptyset + 1$. Hence the work of T_Δ is greater than $\frac{\lambda}{2}\left(\mu_{(3)\Delta} + \mu_\emptyset\right)$. This brings the contradiction as this would mean that the total work is greater than $m\lambda$. □

Theorem 3.7. *If there exists a solution of makespan at most λ, then there exists a solution with the seven-shelf structure we are looking for with a makespan at most $\frac{3\lambda}{2}$ and a lower CPU load. The theorem is a direct consequence of Lemmas 3.5 and 3.6.*

The correctness of the reject condition of the dual approximation is given by the contrapositive of Theorem 3.7.

3.4.4 Building the Schedule

We have described the core step of the dual-approximation algorithm, with a fixed guess. A binary search is used with successive guesses to approach the optimal makespan. Using an initial lower bound $B_{\min}$ and an initial upper bound $B_{\max}$ of the optimal makespan, the number of iterations of this binary search is bounded by $\log(B_{\max} - B_{\min})$.

Each iteration of the dual approximation algorithm consists in solving an ILP. The complexity of this step is not bounded by a polynomial function. However, solving the ILP with a standard linear solver (e.g., CPLEX or Gurobi) shows a very good efficiency as described in Section 3.6.4. Indeed, the filtering functions allow to reduce the search space size of the ILP, since a task can be assigned to at most four sets instead of seven. Moreover, as the number of tasks increases, every task's relative execution time shrinks. Thus, for large instances, most of the tasks will be assigned to sets (0) and (6) only.

This has to be compared to an algorithm using dynamic programming to solve the allocation problem. Even if this paradigm would lead to a proved polynomial complexity, the size of the search space makes it intractable to explore. Adapting the technique proposed in [Ble+15] would result in an algorithm whose complexity is $\mathcal{O}(n^2m^4k^2)$ in our case.

3.5 Algorithm APPROX-2

As stated in Section 3.4.4, APPROX-3/2 is not proved to be polynomial. To get more insight on dual approximation algorithms, we devise APPROX-2 that is a simpler polynomial-time algorithm providing an approximation ratio of $2 + \epsilon$. APPROX-2 uses the same principle as APPROX-3/2: it partitions the computing resources, allocates the tasks to a partition, and then schedules them within their partition.

3.5.1 Sketch

We consider a guess λ of the optimal makespan value. The scheduling problem on the CPUs is simplified by forcing the number of CPUs a task can use to its canonical number of CPUs, with respect to λ. The algorithm then works as follows:

1. Allocate the tasks that fits in λ only on one type of architecture.
2. Sort the tasks by decreasing work ratio $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. The approximation ratio derives from this sort as explained in Lemma 3.9.
3. Allocate the tasks on the GPUs until each GPU has a load more than λ.
4. Schedule the remaining rigid tasks on the CPUs with a 2-approximation algorithm. List algorithms or strip-packing algorithms are viable options.

If all the tasks do not fit with a makespan at most 2λ, then the algorithm rejects this guess. Otherwise, we have founded a valid schedule.

3.5.2 Analysis

We now analyze some properties of APPROX-2. First we study the approximation ratio, then the complexity.

Lemma 3.8. *The makespan of the tasks allocated to the GPUs is smaller than* 2λ.

Proof. By construction, all the tasks considered for an allocation on a GPU are smaller than λ. As the algorithm stops loading a GPU when its load exceeds λ, the makespan bound is straightforward. □

Lemma 3.9. *If there exists a solution of makespan at most* λ*, then the makespan of the tasks allocated to the CPUs is smaller than* 2λ.

Proof. Using the canonical number of CPUs with respect to λ ensures that every task allocated to some CPUs generates a minimal amount of work (as stated in Section 3.3). In particular, this amount of work is at most the amount of work generated in the optimal schedule. The GPUs have been—by construction—allocated a greater share of work than the optimal solution. Moreover, the tasks are sorted by decreasing work ratio $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. This specific order implies that the work remaining on the CPUs is smaller than $m\lambda$ if there exists a solution of makespan at most λ. The makespan bound follows from the fact we schedule the remaining tasks with a

list algorithm [GG75] or a strip-packing algorithm [Ste97]. Using the strip-packing algorithm would provide a contiguous solution. □

The two previous lemmas prove that APPROX-2 provides a solution whose makespan is at most 2λ.

APPROX-2 is an algorithm of low polynomial complexity. It indeed only relies on sorting the tasks, and on keeping track of the computing resources using priority queues. Moreover, each task is considered at most once when scheduled. Hence, and more precisely, the complexity of the algorithm belongs to $\mathcal{O}\left(n\left[\log(n) + \log(k) + m\log(m)\right]\right)$.

3.6 Experimental Evaluation

After providing the theoretical foundation for solving the given scheduling problem, we will now examine the applicability of our approach. For that reason, we will compare the makespans produced by APPROX-3/2 and APPROX-2 as well as the run-time required to compute the solutions. In our evaluation, we will also consider the scheduling solutions from heuristics, which are modifications of the classical Heterogeneous Earliest Finish Time algorithm (HEFT) [THW02].

We start by explaining the problem instances used in our analysis. After that, we give a description of the heuristics used to evaluate our proposed algorithms. We present implementation details for all algorithms, and last, we show and discuss the experimental results.

3.6.1 Problem Instances

Finding the right problem instances for evaluating scheduling algorithms is generally a hard problem. Real world instances are often considered to be essential when choosing test instances. However, testing an algorithm on a small set of real world instances will often not support the claim that an algorithm is generally well applicable. Another problem is that influencing factors, such as the number of tasks or the size of tasks, are fixed in real world instance. Therefore, we generated scheduling instances that allows us to study the general applicability of our algorithms and to investigate the influence of experimental factors.

description	variable	values
number of tasks	n	$\{10, 50, 100, 1000\}$
number of CPUs	m	$\{4, 16, 64, 128, 256, 384, 512\}$
number of GPUs	k	$\{1, 2, 4, 8, 16, 32\}$
minimum sequential run-time of tasks	$\overline{p}^{min}$	10
maximum sequential run-time of tasks	$\overline{p}^{max}$	100
minimum sequential fraction of a task	β^{min}	0
maximum sequential fraction of a task	β^{max}	0.5
mean speedup factor for tasks on GPUs	$mean_g$	0.2
standard deviation of speedup factor for tasks on GPUs	sd_g	0.5
minimum speedup factor for tasks on GPUs	min_g	0.1 (10× speedup on the GPU)
maximum speedup factor for tasks on GPUs	max_g	1.5 (50% slowdown on the GPU)

Table 3.1 Parameter settings used to generate scheduling instances.

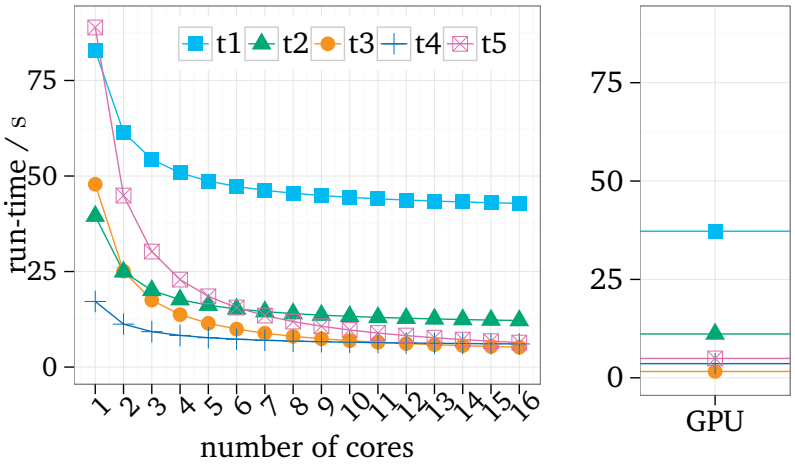


Figure 3.3 Example of a problem instance: Each of the five tasks exhibits a different parallel scalability on the multi-core machine (left) and has a different run-time on the GPU (right).

To generate the instances, we first select the number of tasks (n), the number of CPUs (m), and the number of GPUs (k). Then, the instance generator decides on the run-time of all tasks as follows:

1. It randomly picks the sequential time on the CPU of one task.
2. It defines the speedup of one task on the CPU, by picking the sequential fraction of this task. The time for the sequential fraction defines the lower bound of a task's run-time, as only the run-time of the parallel fraction of a task can be reduced by adding more CPUs (Amdahl's law).
3. It picks a speedup factor that defines how much faster a particular task can run on a GPU compared to being executed on all m CPUs.
4. The generation process is repeated for all the n tasks.

We now provide a more detailed description of each step of the instance generation process.

Step 1: The sequential run-time $\overline{p_{j,1}}$ of task T_j is picked from a uniform distribution in the interval $[\overline{p}^{min}, \overline{p}^{max}]$.

Step 2: Next, the speedup model of each task is determined. To this end, we apply Amdahl's law to model the speedup of a moldable task. The law states that the parallel execution time is bounded by the sequential fraction of a program. Therefore, we select the sequential fraction β_j of each task, where β_j follows uniform distribution in $[\beta^{min}, \beta^{max}]$. The knowledge of the sequential run-time $\overline{p_{j,1}}$ and the sequential fraction β_j allows us to model and compute the parallel execution time on l CPUs of task T_j as: $\overline{p_{j,l}} = \beta_j \overline{p_{j,1}} + (1 - \beta_j)\frac{\overline{p_{j,1}}}{l}$ (for all l in $2, \ldots, m$).

Step 3: We assume that GPUs can accelerate the execution of a task, i.e., a task will—most likely—be faster on a GPU than on all CPUs of the multi-core system. Thus, we model the time for task T_j on the GPU relative to the parallel time and all m CPUs $\overline{p_{j,m}}$. To obtain the time on the GPU ($\underline{p_j}$), we pick a speedup factor g, and set $\underline{p_j} = g\,\overline{p_{j,m}}$. The value of g follows a normal distribution with mean $mean_g$ and standard deviation sd_g. In this way, we also allow tasks that are slower on the GPU than on the CPUs. We also bound the maximum speedup and maximum slowdown for each task on the GPU. For this purpose, we introduce the variables min_g and max_g to denote the minimum and maximum values of g, respectively.

We generated 10 samples for each parameter combination of n, m, and k with the values shown in Table 3.1. Figure 3.3 shows the characteristics of one of these instances, which contains only five tasks for the sake of readability. Each task has

a different scalability behavior (caused by a different sequential fraction), and all tasks in this example have a shorter run-time when executed on a GPU.

3.6.2 HEFT-like Heuristics

We have proposed here two algorithms that provide approximate solutions to the scheduling problem stated in Section 3.1. In order to compare our approaches with practically relevant algorithms, we also include HEFT-like algorithms in our evaluation. We call them HEFT-like algorithms as they work similar to the original HEFT algorithm [THW02], but target a slightly different scheduling problem. HEFT-like algorithms are used in practice. For instance, the run-time system StarPU uses a very similar algorithm (called MCT for minimum completion time) to schedule tasks on CPUs and GPUs [Aug+11].

Now, we describe our variants and implementation of the HEFT-like algorithms for scheduling moldable tasks on a multi-core system with multiple GPUs. Our implementation resembles the general idea of the original algorithm proposed by Topcuoglu et al. [THW02], except that—since we have no precedence constraints—we change the priority function used to sort the tasks. Similar to HEFT, our algorithm places the highest priority task on the CPUs or one of the GPUs that minimize the *earliest finish time* (**EFT**). We expect that HEFT-like algorithms are sensitive to the type of prioritization function. To avoid a possible bias towards one prioritization function, we consider three different strategies, which are:

1. **LPT**: This strategy sorts the tasks in decreasing order of their execution times (**L**ongest **P**rocessing **T**ime).

2. **SPT**: This strategy sorts the tasks in increasing order of their execution times (**S**hortest **P**rocessing **T**ime).

3. **RATIO**: This strategy sorts the tasks in decreasing order of the following ratio: execution time on the CPUs over the run-time on a GPU, i.e., $\frac{\overline{p_{j,l}}}{\underline{p_j}}$, where l is either 1 for sequential tasks or m for parallel tasks.

For the strategies **LPT** and **SPT**, the execution time of task T_j is computed as $\max\left(\overline{p_{j,l}}, \underline{p_j}\right)$, for $l \in \{1, m\}$.

The question is now: how many CPUs should be assigned to each task when computing the schedule? We use two simple schemes: the strategy **PAR** allots all CPUs to a task ($l = m$), whereas the strategy **SEQ** allots only one CPU to a task ($l = 1$). Considering the monotonicity assumption for the run-time of moldable

tasks, the strategy **PAR** is a greedy way of minimizing the execution time of a task, as the run-time function is non-increasing in the number of CPUs. The second strategy (**SEQ**) favors task parallelism and minimizes every task's work. It is certainly possible that these HEFT-like implementations can be improved, but such considerations are outside the scope of this study. In total, we have created six different HEFT-like heuristics, called Heuristic 1–6, which are listed in Table 3.2.

3.6.3 Implementation Details

We implemented APPROX-3/2 using the programming languages Julia and Python. Logically, the algorithm APPROX-3/2 consists of two steps: (i) find the best λ by applying the bisection method to partition the tasks into sets, and (ii) build the schedule from the computed partitioning. The first step has been implemented in Julia, as it features the domain-specific modeling language JuMP, which provides an abstraction layer above different ILP solvers, such as Gurobi, CPLEX, or GLPK. Hence, we only need to write our ILP problem using the JuMP API, and we can use different solvers to find a solution. The second step, the building of the schedule, has been implemented in Python.

As stated above, the lower and upper bound of the scheduling problem are adjusted during the iterative search for the best λ using the bisection method. The bisection method stops when the ratio between upper and lower bound is below a certain threshold (the *cutoff* value). For both algorithms, APPROX-3/2 and APPROX-2, we have used a *cutoff* value of 1.01 ($\sim 1\%$) in all experiments.

The algorithm APPROX-2 has been entirely implemented in Julia. Here, the algorithm also intends to find the best λ, but since it maps tasks to devices (CPUs or GPUs) greedily, the actual schedule is built on the fly.

The HEFT-like heuristics has been written in Python. Similarly to the implementation of APPROX-2, the actual schedule can be built directly, as there is no previous partitioning step.

We have used the following software versions: Julia $0.3.11$, Python $2.7.10$, JuMP $0.10.1$, Gurobi binding for JuMP $0.1.29$, CPLEX binding for JuMP $0.0.13$, Gurobi Optimizer for OS X $6.0.0$, CPLEX Optimization Studio for OS X $12.6.1.0$, and Mac OS X $10.10.5$. For the experiments shown in the present study, we have used the Gurobi Optimizer to solve the integer linear programs.

name	mapping	sorting	parallel tasks on CPUs
Heuristic 1	**EFT**	**LPT**	no (**SEQ**)
Heuristic 2	**EFT**	**SPT**	no (**SEQ**)
Heuristic 3	**EFT**	**RATIO**	no (**SEQ**)
Heuristic 4	**EFT**	**LPT**	yes (**PAR**)
Heuristic 5	**EFT**	**SPT**	yes (**PAR**)
Heuristic 6	**EFT**	**RATIO**	yes (**PAR**)

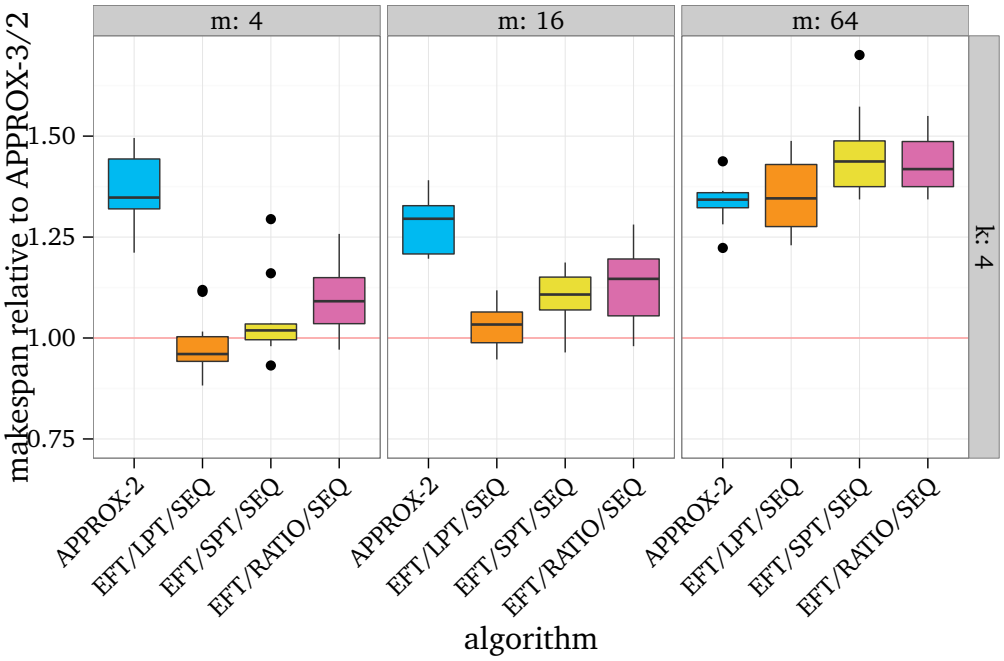
Table 3.2 HEFT-like heuristics used for comparison.

3.6.4 Experimental Results

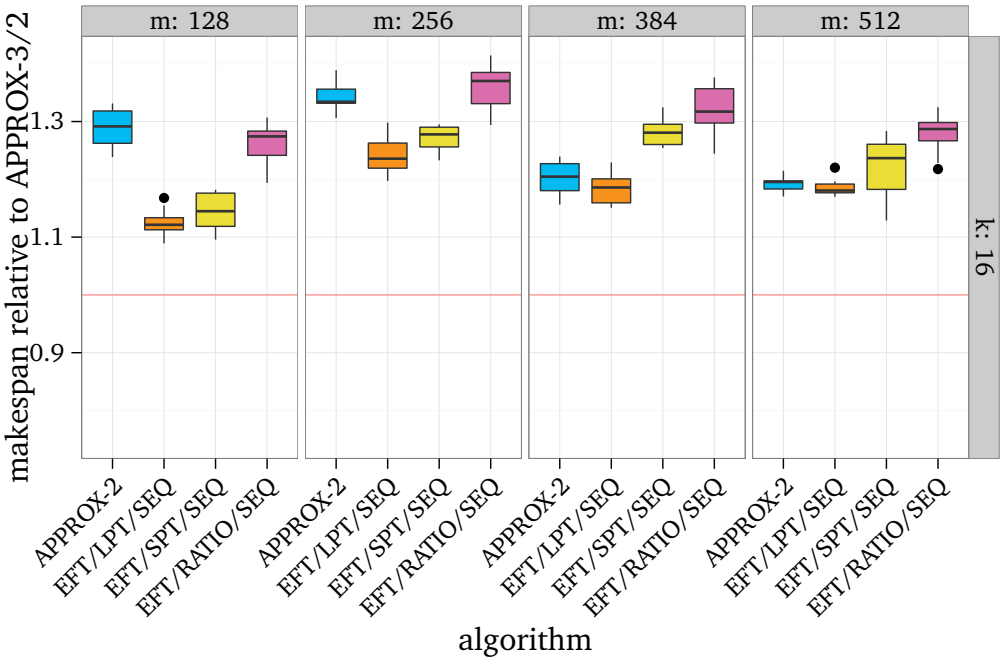
First, we evaluate the produced makespan of each scheduling instance, which is the most important property of the scheduling algorithms described in this chapter.

Figure 3.4 compares the makespans of the schedules generated by APPROX-3/2, APPROX-2, and the six different HEFT-like heuristics. For a better comparability, we normalize the makespan for each scheduling instance by the makespan obtained from APPROX-3/2. Thus, the algorithm APPROX-3/2 will always have a relative makespan of 1.0 (red horizontal line). The relative makespan of the other algorithms, APPROX-2 and the six heuristics, will most likely differ from 1.0. If the computed relative makespan is smaller than 1.0, the produced schedule of one of the competing scheduling algorithms is shorter than the one of APPROX-3/2. Similarly, if the relative makespan is larger than 1.0 then APPROX-3/2 was able to find a shorter schedule. We can observe that the HEFT-like heuristics produce competitive results when the number CPUs and GPUs is small (see Figure 3.4(a), case $m = 4$ and $k = 4$). If the number of tasks, CPUs, and GPUs is increased, the results in Figure 3.4(b) provide evidence that APPROX-3/2 produces significantly shorter schedules than its competitors. The results of the heuristics 4–6 using the **PAR** strategy (see Table 3.2) have been omitted, as they have been found to be largely inferior compared to the **SEQ** versions. Among the HEFT-like algorithms, the heuristics that use an **LPT** strategy produced the shortest schedules. Interestingly, the solutions obtained from the approximation algorithm APPROX-2 are most often not better than the much simpler HEFT-like heuristics, indicating that an approximation factor of 2 is simply too large for a practical applicability.

The solution quality (the makespan) is only one metric to assess scheduling algorithms. The algorithm APPROX-3/2 requires to solve an ILP for each value of λ. Therefore, an analysis of the run-time of the algorithms is of equal importance. The run-times measured do not include the time to read and parse the input files and the time to write the final schedules to disk. In addition, the results are only meant



(a) $n = 100$



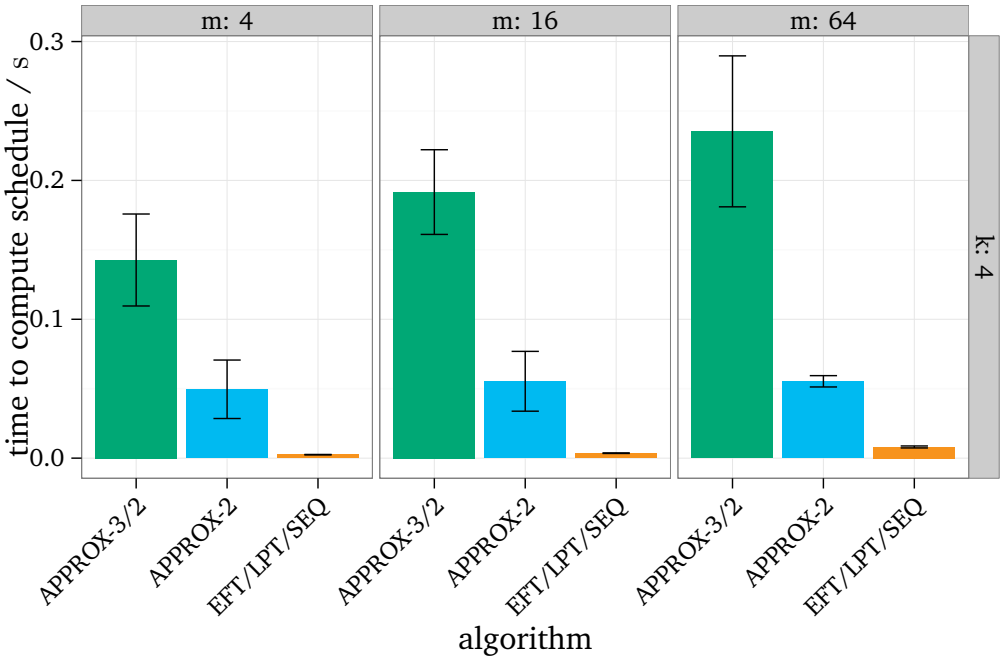
(b) $n = 1000$

Figure 3.4 Comparison of the relative makespan obtained with APPROX-2 and the HEFT-like algorithms with respect to the makespan produced by APPROX-3/2 (n tasks, m CPUs, k GPUs).

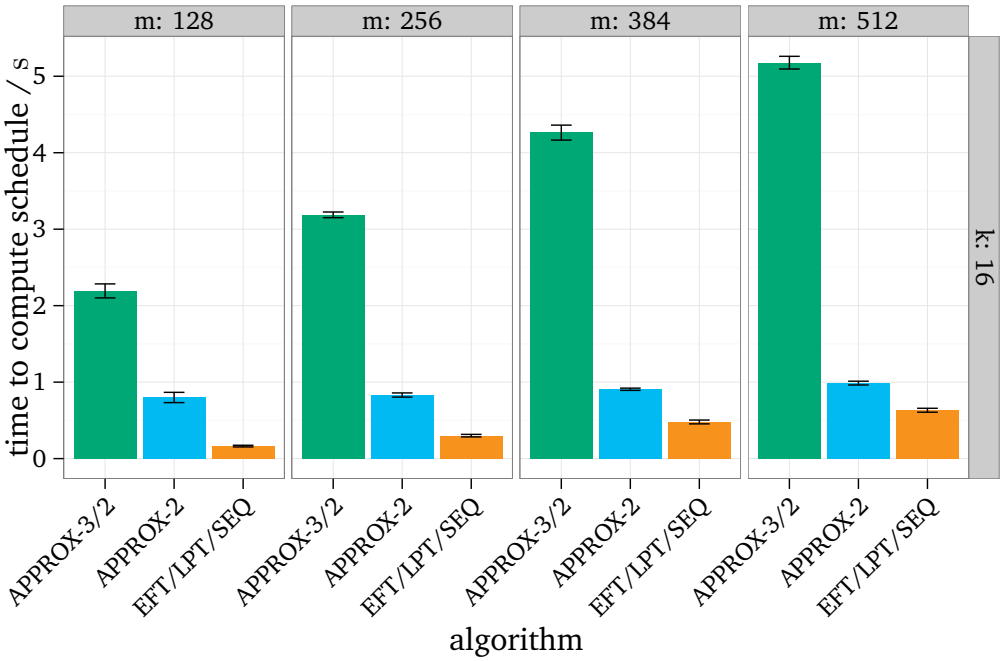
to show general trends of the run-time requirements of the different algorithms, as the algorithms have been implemented using different programming languages.

Figure 3.5 compares mean run-time of the different scheduling algorithms for various values of n, m, and k. In particular, the run-time of the algorithms APPROX-3/2 and APPROX-2 includes all iterations that were required to obtain the final value of λ. The experiments were conducted on a quad-core Intel i7-3615QM with a clock speed of $2.3\,\mathrm{GHz}$. Since the run-times of the various HEFT-like heuristics were very similar, as only the prioritization function is changed, we only show the time for Heuristic 1 (**EFT**, **LPT**, **SEQ**). As expected, the run-time of Heuristic 1 grows linearly with the number of tasks, CPUs, or GPUs, and has been found to be the shortest among all scheduling algorithms tested. The run-time of the APPROX-2 algorithm is significantly longer than the run-time of the heuristics due to the iterative nature of the algorithm. It is also not surprising that the APPROX-3/2 algorithm has the longest mean run-time for all considered cases. We can also see that the run-time of APPROX-3/2 grows proportionally faster than the run-times of the other algorithms, which is a consequence of solving an ILP in each iteration. Nevertheless, APPROX-3/2 computes the solutions relatively quickly, as it takes about five seconds to compute the schedule for the largest instance in our test set ($n = 1000$, $m = 512$, $k = 16$, see Figure 3.5(b)). It goes without saying that this run-time is too large to schedule many small-grained tasks onto CPUs or GPUs, but it is a very promising alternative algorithm for scheduling longer running tasks (or even different parallel application).

We have also studied the effectiveness of the filtering step that we introduced in Section 3.3.2, and Figure 3.6 shows these results. For different numbers of tasks (n), the graphs show the distributions of the mean number of possible partitions per task after the filtering has been applied. We recall that the internal ILP finds a partitioning of all tasks into seven disjoint sets. That means, each of the n tasks can only be in one of the seven partitions. Thus, the ILP initially allocates a table of $n \times 7$ binary variables. In the filtering step, some variables are set to 0, i.e., the number of partitions that a task can be assigned will be reduced. Ideally, the number of available partitions per task reduces from seven to one when the filtering is applied, and the solution can be obtained immediately. Figure 3.6 shows the number of available partitions for increasing values of n. The "mean number of possible partitions" is computed over all the tasks of one iteration. For example, for one problem instance, the mean number of possible partitions (over all the tasks) is 2 in iteration 1 and 2.5 in iteration 2. In this case, the distributions shown in Figure 3.6 will contain the values 2 and 2.5. We observe that for the majority of the tasks, except in the case of $n = 10$, only two partitions are available (on average)



(a) $n = 100$



(b) $n = 1000$

Figure 3.5 Comparison of the mean run-time (incl. 95% confidence interval) of each scheduling algorithms to compute the solutions (n tasks, m CPUs, k GPUs).

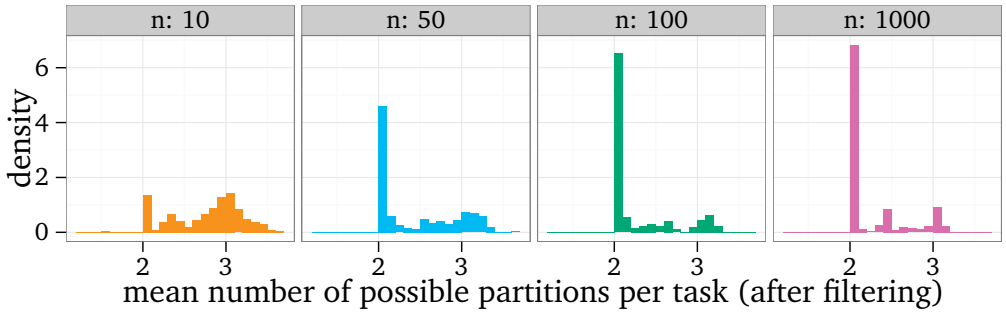


Figure 3.6 Distribution of the (mean) number of possible partitions per task after the filtering has been applied for APPROX-3/2. The graphs show distributions for all values of m and k presented in Table 3.1.

after applying the filtering step, which supports our claim that the filtering step is effective.

Figure 3.7 complements the previous results by an analysis of the number of iterations required, such that the bisection method converges. In the experiments conducted as part of the present study, the required number of iterations was ranging from 10 to 17.

In summary, we can state that APPROX-3/2 is able to find significantly shorter schedules than the APPROX-2 algorithm or the HEFT-like heuristics. On the contrary, APPROX-3/2 needs more time to compute the solutions. However, even for larger instances ($n = 1000$, $m = 512$, $k = 16$) APPROX-3/2 can be used to obtain the solution in a few seconds. If the average task duration lies in the range of seconds, applying APPROX-3/2 will definitely provide an advantage compared to the other scheduling algorithms.

3.7 Summary

In this chapter, we presented a new scheduling algorithm using a generic methodology (as opposed to the design of specific *ad hoc* algorithms) for hybrid architectures (multi-core machine with GPUs) with the moldable task model on CPUs. We proposed an algorithm with a constant approximation ratio of $\frac{3}{2}+\epsilon$. The main idea of the approach is to determine an adequate partition of the set of tasks on the CPUs and the GPUs using a dual approximation scheme and integer linear programming. Still, we do not provide any proof of the complexity of this ILP-based approach. Instead, we compared this approach with another algorithm with a proved polynomial-time

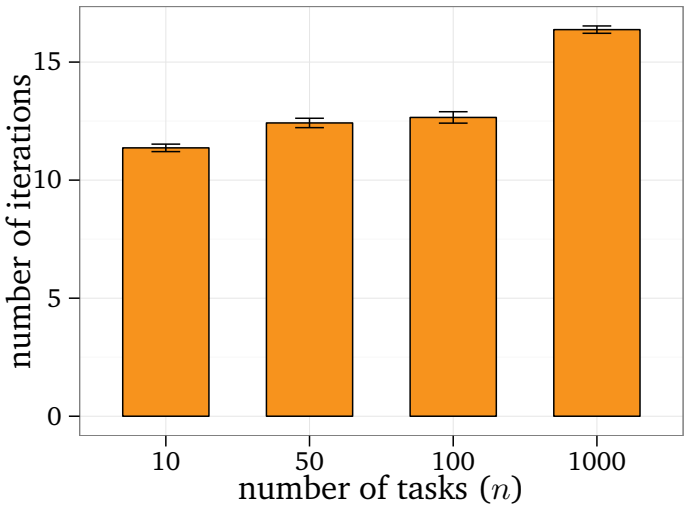


Figure 3.7 Distribution of the number of iterations (of the bisection method) performed by APPROX-3/2 to converge to a solution.

complexity, at the cost of degrading the approximation ratio to $2 + \epsilon$. An experimental analysis on realistic instances has been provided to assess the computational efficiency and the schedule quality of the proposed method when compared to classical HEFT algorithms. The main conclusion is that the ILP-based algorithm is stable because of its approximation guarantee, with a reasonable running time. Moreover, this proposed algorithm outperforms all HEFT algorithms when dealing with instances of large size, which is the case dealt with HPC platforms.

4 Geometric Constraints as a First-Class Modeling Tool

In the two previous chapters we focused our attention on the fine grain scale, that is within a single computing node. To meet the challenge of greater performance, while being constrained by ever growing energy costs, the architecture of supercomputers also grows in complexity at the whole machine scale. This complexity arises from various factors: **i)** the size of the machines (supercomputers now integrates millions of cores); **ii)** the heterogeneity of the resources (various architectures of computing nodes, mixed workloads of computing and analytics, nodes dedicated to I/O, etc.); **iii)** the interconnection topology. The architectural evolutions of the interconnection networks at the whole machine scale pose two main challenges: first, the community is proposing new topologies [Kat+15]; and second, the interconnection network is now unique within the machine (the network is shared for various mixed data flows). Sharing such a single multi-purpose interconnection network begets complex interactions (e.g., network contention) between running applications. These interactions have a strong impact on the performances of the applications [Bha+13; Eno+14], and hamper the understanding of the system by the users [Che+16]. As the volume of processed data increases, so does the impact of the network.

We propose in this chapter a *generic framework for interference-aware scheduling*. More precisely, we identify two main types of interleaved flows: the flows induced by data exchanges for computations, and the flows related to I/O. Rather than explicitly taking into account these network flows, we address the issue of harmful interactions by constraining the shape of the allocations. Such an approach aims at taking into account the complexity of the new platforms in a qualitative way that is more likely to scale properly. The scheduling problem is then defined as an optimization problem with the platform (nodes and topology) and the jobs' description as input. The objective is, for example, to minimize the maximum completion time or the throughput while enforcing constraints on the allocations.

The development of the modeling framework proposed here is an ongoing work done in collaboration with Giorgio Lucarelli, Grégory Mounié, and Denis Trystram. Some of this work has been done during three stays in the team administering Blue Waters at NCSA.

4.1 General Problem Setting

We model in this chapter a platform as of a set $\mathcal{V}$ of m nodes divided in two sets: m^{C} nodes dedicated to computations $\mathcal{V}^{\mathrm{C}}$, and $m^{I/O}$ nodes that are entry points to a high performance file system $\mathcal{V}^{I/O}$. The nodes are indexed by $i \in 0, \ldots, m-1$. This numbering provides an *arbitrary ordering* of the nodes. We distinguish two interesting distributions of the nodes: **i)** *coupled I/O*, where some compute nodes also are entry points for the I/O (i.e., $\mathcal{V}^{I/O} \subseteq \mathcal{V}^{\mathrm{C}} = \mathcal{V}$); **ii)** *separate I/O*, when there is no overlap (i.e., $\mathcal{V}^{I/O} \cap \mathcal{V}^{\mathrm{C}} = \emptyset$). We also distinguish two ways of interacting with the I/O nodes: *shared I/O* when any number of jobs can access an I/O node at any time, and *exclusive I/O* when an I/O node is exclusively allocated to a job for the job's lifespan. We further annotate node symbols with $\star^{I/O}$ ($\star^{\mathrm{C}}$, resp.) if there is a need to distinguish I/O nodes (compute nodes, resp.).

The nodes can communicate thanks to an interconnection network with a given *topology* (i.e., the connected graph of the interconnection), and the localization of every node within the topology is known. We define the distance that intrinsically derives from this topology as follows:

Definition 4.1 (Distance)**.** *The distance* $\mathrm{dist}\,(i, i')$ *between two nodes* i *and* i' *(either compute or I/O) is defined as the minimum number of hops to go from* i *to* i'*.*

Batch schedulers are a critical part of the software stack managing supercomputers: their goal is to efficiently allocate resources (nodes from $\mathcal{V}$ in our case) to the jobs submitted by the users of the platform. The jobs are queued in a set $\mathcal{J}$ of n jobs. Each job j requires a number of compute nodes q_j^{C} and some I/O nodes $q_j^{I/O}$. The I/O nodes requirements can either be a number of nodes (*unpinned I/O*), or a dedicated subset of $\mathcal{V}^{I/O}$ (*pinned I/O*). The number of allocated nodes is fixed (i.e., the job is *rigid* [Fei+97]). We denote by $\mathcal{V}(j)$ the nodes allocated to the job j. Each job j requires a certain time p_j to be processed, and it is *independent* of every other jobs. Once a job starts executing, it runs until completion (i.e., it is *not preemptible*). Finally, any compute node is able to process at most one job at any time.

Figure 4.1 summarizes the organization of a typical HPC platform. Along with the compute and I/O nodes, the platform embeds helper nodes (login nodes, monitoring nodes, etc.) that we do not consider in the model.

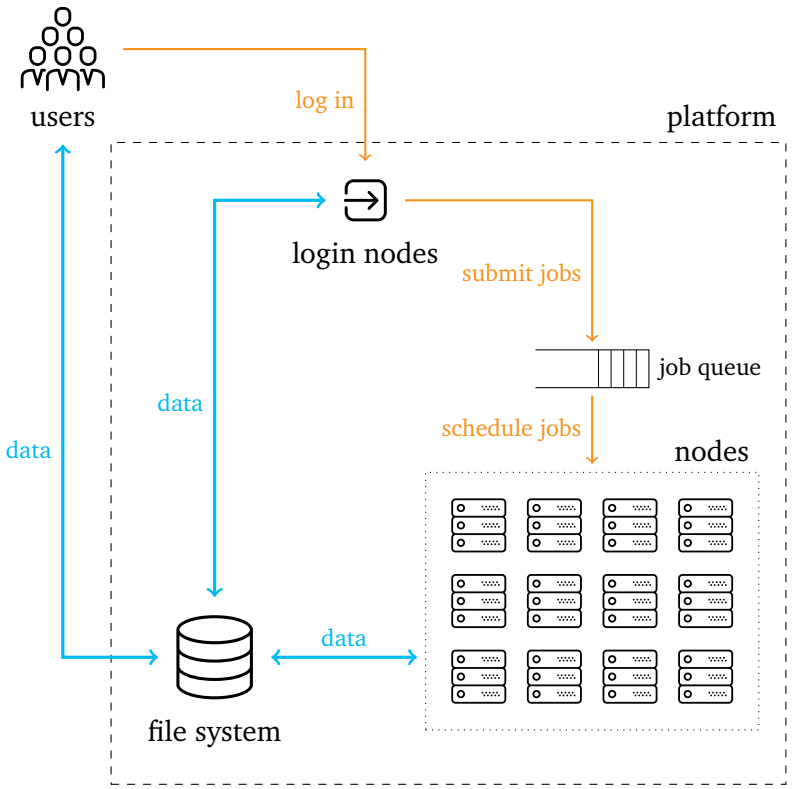


Figure 4.1 Illustration of a HPC platform.
Blue bidirectional arrows depict data flows. Orange unidirectional arrows depict jobs flows.
[Credits: icons by Gregor Cresnar, Madebyoliver and Zlatko Najdenovski; `https://www.flaticon.com`]

4.1.1 Intrinsic Constraints

As stated in the introduction of this dissertation, we do not aim at finely modeling the context of execution. We propose here to model the platform in such a way that network interactions are *implicitly* taken into account. We augment the scheduling problem with alien geometric constraints on the allocations deriving from the platform topology or the application structure.

Naive implementations of schedulers allocate resources greedily. This is known to impact performances [Eno+14], and is the core difference between parallel machine scheduling and packing problems. Constraining the allocations to enhance performance is however no new idea. For example, Lucarelli et al. studied—for the fat tree topology—the impact of enforcing contiguity or locality constraints in backfilling scheduling [Luc+15]. They showed that enforcing these constraints can be done at a small computational cost, and has minimum negative impact on usual metrics such as makespan (i.e., maximum completion time), flow-time (i.e., absolute time spent in the system), or stretch (i.e., time spent in the system relative to each job size). One may refer to [Bru07; Dro09] for a detailed definition of classic optimization objectives in scheduling. More recently, Jain et al. studied the feasibility of isolating jobs through partitioning on low diameter networks (e.g., DragonFly or SlimFly [Kat+15]) [Jai+17].

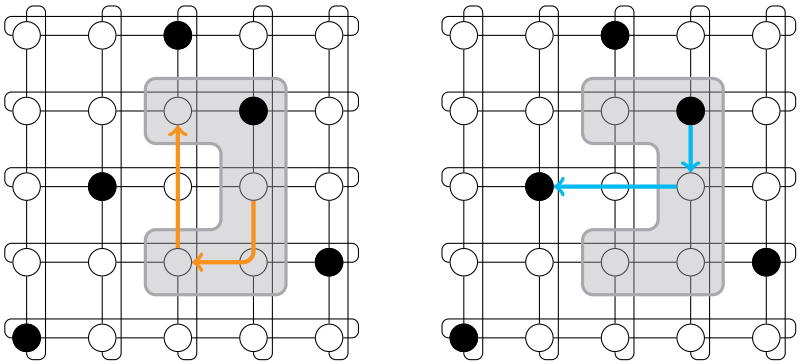
We go further with this model as we target heterogeneous machines, and distinguish network flows. We seek the following properties for constraints:

- It *captures part of the execution context*: enforcing the constraint should help minimize nocuous effects arising from the execution context.

- It *derives from minimal reliable data*. Constraints on the allocations are enforced ahead of the scheduling decisions. As a result, the proposed constraints only use the topology of the interconnection network and the size of the allocation as input data.

- It is *cheap to compute*: enumerating the list of allocations respecting some constraints cannot be a performance bottleneck for the scheduler.

Before presenting the constraints we consider in this work, we need to precisely define the network flows we target. We distinguish two types of flows, directly deriving from the fact that we are dealing with two kinds of nodes.



(a) *compute* communications (b) *I/O* communications

Figure 4.2 Figuration of the two distinguished types of communications. Note that some communications stay within the allocation, while others do not. White nodes represent compute nodes, and black nodes represent I/O nodes.

Definition 4.2 (Communication types)**.** *We distinguish two types of communications (see Figure 4.2):*

compute *communications* *are the communications induced by data exchanges for computations. Such communications occur between two compute nodes allocated to the same application.*

I/O *communications* *are the communications induced by data exchanges between compute nodes and I/O nodes. Such communications occur when compute nodes read input data, checkpoint the state of the application, or save output results.*

Avoiding compute-communication interactions Considering this classification of network flows, we first expose three constraints targeting compute communications.

Definition 4.3 (Connectivity)**.** *An allocation π is said to be* connected *iff there exists a subset $\mathcal{V}_\pi$ of $\mathcal{V}^{I/O}$ such that $\left(\pi \cap \mathcal{V}^C\right) \cup \mathcal{V}_\pi$ is connected in the graph-theory sense. $\mathcal{V}_\pi$ may be empty.*

The *connectivity* constraint ensures, for a given allocation, that there exists a path without interference between any pair of compute nodes of the allocation. This however, with regard to the interconnection topology, can either require support for dynamic routing or demand to the application to implement its own routing policy. Moreover, it may lead to islets of isolated compute nodes. Hence, although satisfactory from the graph theoretical point of view, the connectivity constraint is not sufficient to ensure that compute communication do not interfere. We propose

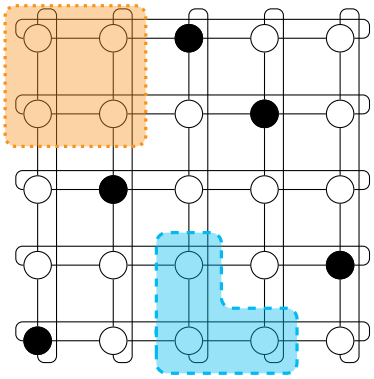


Figure 4.3 Example of a convex allocation (dotted orange contour), and a non-convex, but connected allocation (dashed blue contour). The underlying topology is a 2D-torus, with dimension-order routing. White nodes represent compute nodes, and black nodes represent I/O nodes.

the *convexity* constraint with the goal of overcoming these limits. An example of connected and convex allocations is depicted on Figure 4.3.

Definition 4.4 (Convexity)**.** *An allocation is said to be* convex *iff it is impossible for compute communications from any other potential allocation to share an interconnect link with respect to the underlying routing algorithm.*

By taking into account the effective routing policy, and by forbidding any potential sharing, the *convexity* constraint does forbid interactions.

Note that the convexity constraint dominates the connectivity constraint, as stated in the following Proposition.

Proposition 4.1. *Given any topology, any convex allocation is connected.*

Definition 4.5 (Contiguity [Błą+15; Luc+15])**.** *An allocation is said to be* contiguous *if and only if the nodes of the allocation form a contiguous range with respect to the nodes' ordering.*

One has to note that the contiguity constraint is intrinsically unidimensional as it relies on the nodes' ordering. For topologies such as trees, lines or rings the ordering is natural. On higher dimension topologies, no natural ordering exists, and an arbitrary mapping is needed. An usual strategy to order nodes is to use space-filling curves (e.g., Z-order curve [Mor66], Hilbert curve [Hil91], etc.) as they enforce a strong spatial locality. Albing proposes various orderings that may be more suited for HPC use cases, and a method to evaluate them [Alb15]. Contiguity is an interesting relaxation of convexity as it offers good spatial locality properties for a reasonable computing cost. It is however unable to ensure that no jobs could interact.

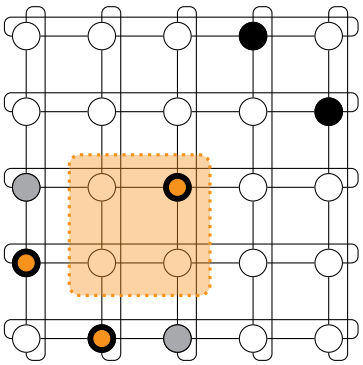


Figure 4.4 Given an allocation (dotted orange contour) for a job j, the allocation is local iff j uses a subset of the I/O nodes marked with the orange dot. Foreign compute nodes potentially impacted by I/O communications of j are depicted in gray: these nodes can only be in the neighborhood of the allocation thanks to the locality constraint. The underlying topology is a 2D-torus, with dimension-order routing. White nodes represent compute nodes, and black nodes represent I/O nodes.

Avoiding I/O-communication interactions The constraints exposed so far are well suited to take into account the compute communications, but not the I/O communications. Indeed, the compute communications may occur between any pair of compute nodes within an allocation: we usually describe this pattern as all-to-all communication. I/O communications, on the other hand, generate traffic towards few identified nodes in an all-to-one or one-to-all pattern. Hence, we propose the *locality* constraint, whose goal is to limit the impact of the I/O flows to the periphery of the job allocations (see Figure 4.4). We must emphasize that the locality constraint proposed here is not related to the locality constraint described by Lucarelli et al. [Luc+15].

Definition 4.6 (Locality)**.** *A given allocation for a job j is said to be* local *iff it is connected, and every I/O nodes from $\mathcal{V}^{I/O}(j)$ are adjacent to compute nodes from $\mathcal{V}^{C}(j)$, with respect to the underlying topology. In other words, $\mathcal{V}^{I/O}(j)$ is a subset of the closed neighborhood of $\mathcal{V}^{C}(j)$.*

Interestingly, the locality constraint enforces a bound on the number of concurrent jobs that can target a given I/O node.

Proposition 4.2. *Given any topology, any I/O node i, at any time, the number of local jobs targeting i cannot exceed the number of adjacent compute nodes of i.*

As a consequence, if the I/O nodes can be shared, the number of concurrent jobs targeting a given I/O node is bounded by the degree of this I/O node. This identity obviously also holds for exclusive I/O, but has limited interest in this case.

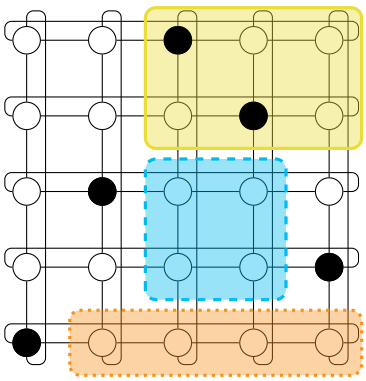


Figure 4.5 Example of four-compute-node allocations with different compacity. The underlying topology is a 2D-torus, with dimension-order routing. White nodes represent compute nodes, and black nodes represent I/O nodes. The bottom allocation (dotted orange contour) has a compacity of $\frac{3}{2} = 1.5$. Note that this allocation is not convex due to the routing policy on the torus. The middle allocation (dashed blue contour) has a compacity of $\frac{4}{3} \approx 1.33$. The top allocation (solid yellow contour) has a compacity of $\frac{11}{6} \approx 1.83$.

4.1.2 Extrinsic Metrics

The constraints presented in the previous section only target inter-application interactions. Moreover, the constraints are binary and omit that many allocation shapes are compatible. Considering all shapes compatible with a given constraint, not all are equal in terms of application performance, and there is a need for more finesse in the choice of allocations. The number of hops traversed to communicate between two nodes is known to be a key factor for the performance of the applications [Eno+14; PML14; Leu+02]. These intra-application behaviors can be captured with dispersal metrics.

Reflecting the fact that compute communications occur between any two compute nodes of an allocation, we define the *compacity* metric as follows:

Definition 4.7 (γ-Compacity [PML14])**.** *The* compacity *characterizes how spread an allocation is. It is defined as the average distance between any pair of compute nodes within the allocation. More formally, the compacity of an allocation of q_j^C compute nodes for job j is defined as*

$$\gamma = \frac{1}{q_j^C(q_j^C - 1)} \sum_{i \in \mathcal{V}^C(j)} \sum_{i' \in \mathcal{V}^C(j)} \text{dist}\,(i, i')$$

The allocation is then said to be γ-compact.

A comprehensive review of the various existing dispersal metrics could be found in [PML14]. The experiments in this work motivated our choice of the average

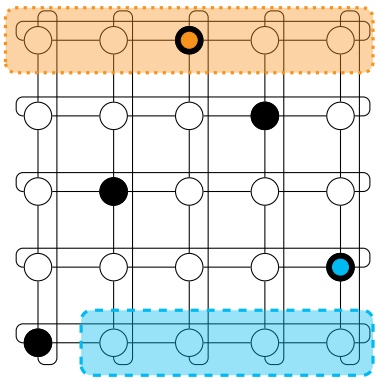


Figure 4.6 Example of four-compute-node allocations with different proximities. The underlying topology is a 2D-torus, with dimension-order routing. White nodes represent compute nodes, and black nodes represent I/O nodes. The bottom allocation (dashed blue contour, and blue I/O node) has a proximity of 1. The top allocation (dotted orange contour, and orange I/O node) has a proximity of $\frac{3}{4} = 0.75$. Note that both allocations are local, but only the top one is convex.

distance as compacity metric, as they show that the average distance is highly correlated with application performance: the smaller the compacity is, the better the performances are. Figure 4.5 depicts an example of two convex allocations of the same size with different compacity values.

As stated in the previous section, the I/O communications occur between compute nodes and identified I/O nodes. This observation leads to the following definition of the *proximity* metric, which is an adaptation of the compacity metric for the I/O nodes. Similarly to the compacity, the lower the proximity, the better. Figure 4.6 depicts two allocations of the same size with different proximity values.

Definition 4.8 (ρ-Proximity)**.** *The* proximity *characterizes an allocation by measuring how far the compute nodes are from the I/O nodes. More formally, the proximity of an allocation for job j is defined as*

$$\rho = \max_{i \in \mathcal{V}^{I/O}(j)} \frac{1}{q_j^C} \left(\min_{i' \in \mathcal{V}^C(j)} \operatorname{dist}(i, i') + \max_{i' \in \mathcal{V}^C(j)} \operatorname{dist}(i, i') \right)$$

The allocation is then said to be ρ-proximate.

4.1.3 Extension of Graham Notation

We use throughout this work the $\alpha \mid \beta \mid \gamma$ notation proposed by Graham et al. [Gra+79]. Drozdowski did a thorough survey of parallel jobs' scheduling in his book [Dro09]: we make the notation more specific, and extend it to suit our modeling framework.

symbol	meaning
$size_j^C$	jobs require a number of compute nodes (i.e., rigid)
$size_j^{I/O}$	jobs require a number of I/O nodes (i.e., unpinned I/O)
$fix_j^{I/O}$	jobs require identified I/O nodes (i.e., pinned I/O)
$excl^{I/O}$	exclusive use of I/O nodes
$connect$	connected allocations
$convex$	convex allocations
$contig$	contiguous allocations
$local$	local allocations

Table 4.1 Extension of Graham notation for the β field.

We summarize in Table 4.1 the symbols added to the β field. Note that the symbols $size_j^{I/O}$ and $fix_j^{I/O}$ are mutually exclusive. We assume by default that I/O nodes can be accessed by multiple jobs simultaneously (shared I/O). If this is not the case, we indicate that I/O nodes cannot be shared with $excl^{I/O}$ (exclusive I/O).

Second, we extend the γ field with the following optimality criteria (all of them are minimization objectives): $\max_j compact_j$, $\max_j prox_j$, $\sum_j compact_j$, and $\sum_j prox_j$.

4.2 Related Work

Tackling the nocuous interactions arising from the context of execution, or, more specifically, network contention, can be done either by preventing these interactions from happening or by mitigating them. Still, the approaches discussed above require knowledge of the application communication patterns (either compute or I/O communications). We start reviewing related work in the prevention/mitigation of interactions before discussing monitoring techniques.

Interactions Prevention Some steps have been taken towards integrating more knowledge about the communication patterns of applications into the batch scheduler. For example, Georgiou et al. studied the integration of TreeMatch into SLURM [Geo+17]. Given the communication matrix of an application, the scheduler minimizes the load of the network links by smartly mapping the application's processes on the resources. This approach however is limited to tree-like topologies, and does not consider the temporality of communications. Targeting the mesh/torus topologies, the works of Tuncer et al. [TLC15] and Pascual et al. [PML14] are noteworthy. Another way to prevent interactions is to force the scheduler to use only

certain allocation shapes with good properties: this strategy has been implemented in the Blue Waters scheduler [Eno+14]. The administrators of Blue Waters let the scheduler pick a shape among 460 precomputed cuboids.

Yet, the works proposed above only target compute communications. HPC applications usually rely on highly tuned libraries such as MPI-IO, parallel netCDF or HDF5 to perform their I/O. Tessier et al. propose to integrate topology awareness into these libraries [Tes+16]. They show that performing data aggregation while considering the topology allow to diminish the bandwidth required to perform I/O. The CLARISSE approach proposes to coordinate the data staging steps while considering the full I/O stack [ICR16].

Interactions Mitigation Given a set of applications, Gainaru et al. propose to schedule I/O flows of concurrent applications [Gai+15]. Their work aim at mitigating I/O congestion once applications have been allocated computation resources. To achieve such a goal, their algorithm relies on past I/O patterns of the applications to either maximize the global system utilization, or minimize the maximum slowdown induced by sharing bandwidth.

Application/Platform Instrumentation A lot of effort has been put into developing tools to better understand the behavior of HPC applications. Characterizing I/O patterns is key as it allows the developers to identify performance bottlenecks, and allows the system administrator to better configure the platforms. Some tools, such as Darshan [Car+11], instrument the most used I/O libraries, and record every I/O-related function call. The gathered logs provide valuable data for postmortem analysis. Taking a complementary path, Omnisc'IO aims at predicting I/O performances during execution [Dor+16]. The predictions rely on a formal grammar to model the I/O behavior of the instrumented application.

These instrumentation efforts allow for a better use of the scarce communication resources. However, as they are application-centric, they fail to capture inter-application interactions. Monitoring of the platform is a way of getting insight on the inter-application interactions [Age+14; EBB16]. For example, the OVIS/LDMS system deployed on Blue Waters collect 194 metrics on every 27648 nodes every minute [Age+14]. Among the metrics of interest are the network counters: the number of stalls is a good indicator of congestion [Dev+14].

5 Unidimensional Problem's Instantiations

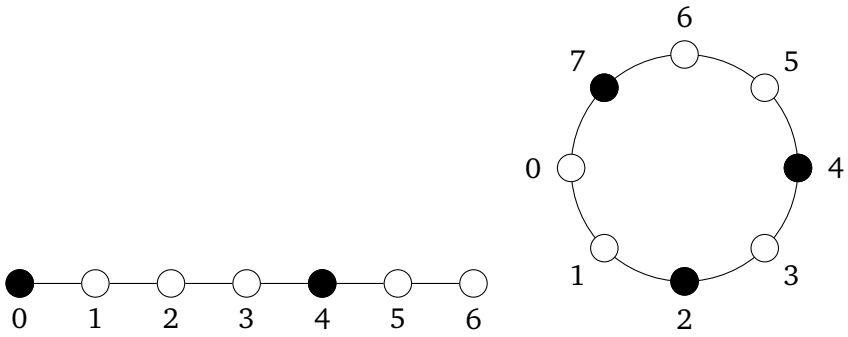
We proposed in the previous chapter a framework to model modern HPC platform where constraints are a first-class modeling tool. With this framework, we aim at developing low cost algorithms with performance guarantees while being able to apprehend the scale of modern platforms. We study, as a first step, the instantiation of this framework with unidimensional topologies. Namely, these are the line (Figure 5.1(a)) and the ring (Figure 5.1(b)). Studying topologies of one dimension is a first step towards the study of state-of-the-art platforms. The line may indeed be seen as a degenerate tree. Fat-tree topologies are a common interconnect, and are for example used in the Curie [@curie] and Oakforest-PACS [@ofp] platforms. The toric topologies, such as the one used by Blue Waters [@bw] and Titan [@titan] (3D torus) or the K computer [@kcomp] (6D torus), may be studied from the ring with embedding techniques. Moreover, these simple topologies provide lower bounds for the other topologies.

We first recall the notations, and give a formal definition of the studied instantiations. We then expose some properties specific to unidimensional topologies. As a second step, we study the complexity, and give constant-ratio approximation algorithms for the studied instantiations.

The work presented in this chapter is an ongoing collaboration with Konstantinos Dogeas, Giorgio Lucarelli, Grégory Mounié, and Denis Trystram

5.1 Formal Instantiation

Let us recall that a platform is a set $\mathcal{V}$ of m nodes. We consider in this chapter *separate I/O* where the platform is partitioned in two sets: a subset $\mathcal{V}^{\mathrm{C}}$ of m^{C} compute nodes and a subset $\mathcal{V}^{I/O}$ of $m^{I/O}$ I/O nodes. As a consequence, we have $m = m^{\mathrm{C}} + m^{I/O}$. The nodes are indexed by $i \in 0, \ldots, m-1$. On the line and the torus, the natural order is such that $i \in 1, \ldots, m-2$ is adjacent to $i-1$ and $i+1$. On the torus, we set the index such that nodes 0 and $m-1$ also are adjacent.



(a) Line topology. The nodes are ordered with the natural order.

(b) Ring topology.

Figure 5.1 Example of platforms with unidimensional topologies. White nodes represent compute nodes, and black nodes represent I/O nodes.

Let us also recall that jobs are queued in a set $\mathcal{J}$ of n jobs. Each job j requires p_j units of time to be processed, some compute nodes q_j^C, and some I/O nodes $q_j^{I/O}$. We consider jobs to be rigid and non-preemptible. Moreover, any compute node is able to process at most one job at any time.

We are interested in this chapter in minimizing the maximum completion time, while enforcing the convexity (Definition 4.4) and locality (Definition 4.6) constraints. We study first the case of *unpinned I/O*, followed by *pinned I/O*. The differences between *exclusive I/O* and *shared I/O* are highlighted along the chapter when relevant.

5.1.1 Structural Properties

As stated in Section 4.1.1, the convexity constraint dominates the connectivity constraint (see Proposition 4.1). However, the unidimensional topologies are more constrained, and the following properties also hold in this chapter:

Proposition 5.1. *Let us consider the line topology, any connected allocation is convex. Hence, connectivity and convexity constraints are equivalent on a line topology.*

Proposition 5.2. *Let us consider the line topology, ordered with its naturally associated order. The constraints of contiguity and connectivity are equivalent.*

5.2 Study of *unpinned* I/O: $\boldsymbol{P^{\mathsf{C}}, P^{I/O} \mid size_j^{\mathsf{C}}, size_j^{I/O}, convex, local \mid C_{\max}}$

We study in this section the problem with *unpinned* I/O. In this configuration, each job requires a given number of I/O nodes. Such a setup occur when jobs have minimum requirements in bandwidth towards the file system, or when the availability of the I/O nodes is crucial (i.e., minimizing the latency).

We start by a complexity analysis of the problem, and continue by proposing a meta approximation algorithm based on packing techniques.

5.2.1 Complexity

Theorem 5.1. *$P^{C}, P^{I/O} \mid size_j^{C}, size_j^{I/O} = 1, convex, local \mid C_{\max}$ is **NP**-complete. Furthermore, if there are at least three I/O node in the platform, the problem is **NP**-complete in the strong sense.*

Proof. The problem clearly belongs to **NP**. The proof of the strong **NP**-completeness is done by reducing the problem to **3-PARTITION**[1] [GJ79].

Let us consider a set A of $3M$ positive integers, a bound $B \in \mathbb{Z}^+$ such that for each $a \in A$, $\frac{B}{4} < a < \frac{B}{2}$, and such that $\sum_{a \in A} a = MB$. The **3-PARTITION** decision problem is to decide whether A can be partitioned in M disjoint sets $A_1, A_2, \ldots, A_M$, such that for $1 \leqslant i \leqslant M, \sum_{a \in A_i} a = B$.

The corresponding input for our problem is the following:

- $m^{\mathsf{C}} = B$, $m^{I/O} = 3$;
- the topology is a line starting with an I/O node, followed by $\frac{B}{2}$ compute nodes, an I/O node, $\frac{B}{2}$ compute nodes, and finishing with a third I/O node;
- for each $a \in A$, we create a job j with $q_j^{\mathsf{C}} = a$, $q_j^{I/O} = 1$, and $p_j = 1$.

Let us suppose there exists a partition. We build a schedule of makespan at most M by scheduling jobs from A_i at time i. The size constraint $\forall a \in A$, $\frac{B}{4} < a < \frac{B}{2}$ ensures the I/O nodes cannot be shared while the locality constraint is met by the schedule. This constraint indeed ensures at most three jobs can be executed in parallel.

[1] **3-PARTITION** is indexed as SP15 in [GJ79].

Let us suppose now there exists a schedule of makespan M. As the whole work is MB, there is no idle time. The partition is directly derived by assigning jobs that start at time i to A_i.

The proof of the **NP**-completeness when the platform contains exactly two I/O nodes is done by reducing the problem to **PARTITION**[2] [GJ79]. Consider the same topology as above, and remove the middle I/O node. The partition derives from the choice of assigning a job either to the left or to the right I/O node. The full proof is left to the reader.

When the platform contains a single I/O node, the complexities of the exclusive I/O model and the shared I/O model differ. On one hand, the problem with exclusive I/O constraint becomes solvable in linear time: it is an instance of $1 \mid\mid C_{\max}$. On the other hand, with the shared I/O model, the problem can be reduced to **PARTITION** [GJ79]. We however consider a topology with a single I/O node between two compute nodes. Each element a from the set to partition is associated to a job j with $q_j^{\mathsf{C}} = q_j^{I/O} = 1$, and $p_j = a$. Similarly to the proof with two I/O nodes, the partition derives from the choice of assigning a job either to the left or to the right while adding the processing times. The full proof is left to the reader. □

Note that if the distance between the I/O nodes is greater than the maximum size q_j^{C} of the jobs, and under the constraint that each job requires exactly one I/O node, the problem reduces to $P \mid\mid C_{\max}$.

5.2.2 Meta Approximation Algorithm

We propose in this section a meta approximation algorithm for the line topology. Additionally, we consider in this section that the I/O nodes are uniformly distributed on the line. In other words, the I/O nodes are equidistant from each other. Without loss of generality, let us assume that m is divisible by $m^{I/O}$. We then denote by $\delta = \frac{m}{m^{I/O}}$ the distance separating two consecutive I/O nodes. Furthermore, we constraint each job to require exactly one I/O node.

[2] **PARTITION** is indexed as SP12 in [GJ79].

Sketch

The key idea is to notice that all the small jobs (i.e., $q_j^{\mathsf{C}} < \delta$) need special care to be local. The remaining jobs, with size at least δ, are structurally guaranteed to be adjacent to at least an I/O node.

We propose a meta approximation algorithm that works by scheduling small and big jobs apart. We partition the set $\mathcal{J}$ of jobs in two subsets: $\mathcal{J}_{<\delta} = \left\{j \in \mathcal{J} \mid q_j^{\mathsf{C}} < \delta\right\}$ and $\mathcal{J}_{\geqslant\delta} = \left\{j \in \mathcal{J} \mid q_j^{\mathsf{C}} \geqslant \delta\right\}$. The meta algorithm schedules the jobs from the subsets in independent shelves: jobs from $\mathcal{J}_{<\delta}$ are scheduled as a $P \mid\mid C_{\max}$ problem, and jobs from $\mathcal{J}_{\geqslant\delta}$ are scheduled with a strip-packing algorithm.

Analysis

Lemma 5.2. *If there exists a ρ-approximation algorithm for the $P \mid\mid C_{\max}$ scheduling problem, then there exists a ρ-approximation algorithm to schedule jobs of size less than δ, where δ is the distance between two consecutive I/O nodes (i.e., $\delta = \frac{m}{m^{I/O}}$).*

Proof. Due to their small size, jobs in $\mathcal{J}_{<\delta}$ need special care to be local. However, their small size also ensures they do not interact with each other. Hence, choosing which I/O node to allocate to each small job is sufficient to schedule them. With this consideration, the problem of scheduling small jobs is the same as $P \mid\mid C_{\max}$ with $m^{I/O}$ machines. □

Lemma 5.3. *If there exists a ρ-approximation algorithm for the strip-packing problem, then there exists a ρ-approximation algorithm to schedule jobs with sizes at least δ.*

Proof. We can consider the problem of scheduling large jobs as a strip-packing problem. The width of the strip is m as a node may process at most one job at any time. Moreover, jobs of size at least δ are structurally guaranteed to be local as their needs in compute node exceed the distance between two consecutive I/O nodes. Hence, any solution given by solving the strip-packing problem is a valid feasible solution for our problem. The makespan of the solution is the height produced by the packing. □

Theorem 5.4. *The meta algorithm computes a feasible schedule with an approximation ratio of $\rho + \rho'$, where ρ and ρ' are the approximation ratios for the $P \mid\mid C_{\max}$ and strip-packing problems, respectively. The proof directly derives from Lemmas 5.2 and 5.3.*

For example, one may schedule small jobs with the **L**ongest **P**rocessing **T**ime (LPT) list algorithm [Gra69], and big jobs with Steinberg's algorithm [Ste97]. Such an instantiation would have a performance guarantee of $\frac{4}{3}+2$ with a computational cost in $\mathcal{O}\left(n \cdot \log n + \frac{n \cdot \log^2 n}{\log \log n}\right)$, where n is the number of jobs.

5.3 Study of *pinned* I/O: $\boldsymbol{P^{\mathsf{C}},\ P^{\mathit{I/O}} \mid size_j^{\mathsf{C}},\ fix_j^{\mathit{I/O}},\ convex,\ local \mid C_{\max}}$

We study in this section the problem with *pinned* I/O. Let us recall that in this case, each job requests a specific set of I/O nodes. Such a model is representative of HPC platforms where the parallel file system is organized in stripes. For example, this is the case with the configuration of the Lustre file system in Blue Waters, where each I/O node is responsible for an address range (i.e., a stripe). The input data of jobs resides on multiple stripes. Hence, the jobs will request the I/O nodes corresponding to their data.

We first study the computational complexity of this model. We then propose a constant-approximation algorithm.

5.3.1 Complexity

Definition 5.1 (**Numerical 3-Dimensional Matching**[3] [GJ79])**.** *An instance consists in three disjoint sets W, X, and Y, each containing M positive integers, and a bound $B \in \mathbb{Z}^+$.*

The decision problem is to decide whether $W \cup X \cup Y$ can be partitioned into M disjoint sets $A_1, A_2, \ldots, A_M$ such that each A_i contains exactly one element from each of W, X, and Y; such that for $1 \leqslant i \leqslant M$, $\sum_{a \in A_i} a = B$.

*This decision problem is **NP**-complete in the strong sense.*

Lemma 5.5. *Let us consider the decision problem* **Numerical 3-Dimensional Matching** *(denoted **N3DM** in short), and let us enforce that all elements that belong to one of the disjoint sets are greater than $\frac{B}{2}$. This constrained version of **N3DM** remains **NP**-complete in the strong sense.*

[3] **Numerical 3-Dimensional Matching** is indexed as SP16 in [GJ79].

Proof. The proof is done by reducing the constrained version of **N3DM** to the original version of **N3DM**.

Let us consider three disjoint sets W, X, and Y, each containing M positive integers, and a bound $B \in \mathbb{Z}^+$. We build the following instance that is compatible with the size constraint: $W' = W$, $Y' = Y$, $X' = \{x + B, x \in X\}$, and $B' = 2B$.

Let us suppose there exists a partition for the original instance. The exact same partition is a solution for the constrained instance. Similarly, if there exists a partition for the constrained instance, it is also a solution for the original instance. □

Theorem 5.6. *P^C, $P^{I/O}$ | $size_j^C$, $fix_j^{I/O}$, $convex$, $local$ | $C_{\max}$ is **NP**-complete in the strong sense.*

Proof. It is clear the problem belongs to **NP**. Thanks to Lemma 5.5, we prove the problem is **NP**-complete by reducing it to the constrained version of **N3DM**.

Let us consider three disjoint sets W, X, and Y, each containing M positive integers, and a bound $B \in \mathbb{Z}^+$. Furthermore, we impose that any element belonging to X is greater than $\frac{B}{2}$. The decision problem is explained in Definition 5.1.

We craft the following corresponding input for our problem:

- $m^C = B$, $m^{I/O} = 3$;
- the topology is a line starting with an I/O node, followed by $\frac{B}{2}$ compute nodes, an I/O node, $\frac{B}{2}$ compute nodes, and finishing with a third I/O node;
- for each $a \in A$, we create a job j with $q_j^C = a$, and $p_j = 1$. All jobs derived from sets W, X, and Y target the first, second, and third I/O node, respectively. The jobs mapped to the middle I/O nodes derive from the set with a size constraint.

Let us suppose there exists a matching. The schedule built by scheduling jobs from A_i at time i has a makespan of M, and fulfills the convexity and locality constraints. The size constraint $\forall x \in X, x > \frac{B}{2}$ ensures the I/O nodes cannot be shared, and at most three jobs are processed concurrently.

Let us suppose now there exists a schedule of makespan M. There is no idle time as the whole work is MB. The partition is derived by assigning to A_i the jobs starting at time i. □

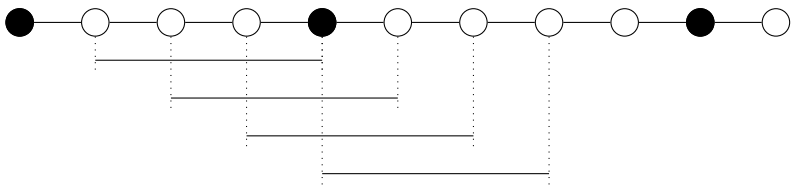


Figure 5.2 Potential allocations for a job requesting the middle I/O node and three compute nodes. White nodes represent compute nodes, and black nodes represent I/O nodes.

It should be pointed out that if the distance between the I/O nodes is greater than the maximum size of the jobs, and under the constraint that each job requires exactly one I/O node, the problem reduces to several independent instances $1 \mid\mid C_{\max}$. In this specific case, the optimal solution can be computed in linear time.

5.3.2 Approximation Algorithm

We propose in this section a constant-approximation algorithm. First we propose an 6-approximation algorithm for the sub-case where all jobs require a single I/O node. We then show that any instance can be reduced to a single I/O instance.

Algorithm with a single I/O node per job

Sketch The algorithm is split in two phases. We first reduce the problem to an instance of the dedicated processors scheduling problem ($P \mid fix_j \mid C_{\max}$ in Graham notation) by determining the compute nodes allocated to each job. The dedicated processors scheduling problem can be seen as the DYNAMIC STORAGE ALLOCATION[4] problem [GJ79] where time and space have been swapped. This first phase (*dedication phase*) is done by solving a linear program, and rounding the given solution. In a second phase (*scheduling phase*), we set the starting time of every job.

Dedication Phase We are interested in allocations that are simultaneously convex and local. As a consequence, there exist at most $q_j^{\mathrm{C}} + 1$ valid allocations for each job j. An example is depicted on Figure 5.2. In order to transform our problem into an instance of the dedicated processors scheduling problem, we need to choose a single allocation for every job.

[4] DYNAMIC STORAGE ALLOCATION is indexed as SR2 in [GJ79].

Let us formally introduce the notations we use in this section. We define the local load L_i of a node i as the sum of the processing time of the jobs allocated to node i. The (global) load of an allocation is defined as $\Lambda = \max_i L_i$. We define by $\alpha^{\mathsf{C}}(s,q)$ the smallest convex allocation containing q compute nodes of index at least s. If such an allocation does not exist, we set $\alpha^{\mathsf{C}}(s,q) = \emptyset$. Such a definition is ambiguous on a ring: this technical point is addressed later (see page 92). We extend the definition of the distance of a specific node to an allocation as the minimum distance between this specific node and any node belonging to the allocation. The distance to the empty allocation is set to infinity. For the purpose of notation, when a job j requires a single I/O node, we will designate this I/O node by $\mathcal{V}^{I/O}(j)$. We furthermore define for each job j the variables $x_{j,s}$, that corresponds to the fraction of j computed on $\alpha^{\mathsf{C}}\left(s, q_j^{\mathsf{C}}\right)$.

We are seeking an assignment of compute nodes for each job such that the allocation minimizes the maximum load Λ. The choice of the allocation for each job is done with the linear program outlined below.

$$
\begin{aligned}
&\min \Lambda, \\
&\text{s.t.} \quad \Lambda \geqslant L_i && \forall i && (C_1)\\
&\qquad L_i \geqslant \sum_j \sum_s x_{j,s} p_j \mathbf{1}_{i \in \alpha^{\mathsf{C}}\left(s, q_j^{\mathsf{C}}\right)} && \forall i && (C_2)\\
&\qquad \sum_s x_{j,s} = 1 && \forall j && (C_3)\\
&\qquad \operatorname{dist}\left(\alpha^{\mathsf{C}}\left(s, q_j^{\mathsf{C}}\right), \mathcal{V}^{I/O}(j)\right) \leqslant 1 && \forall j\, \forall s && (C_4)\\
&\qquad 0 \leqslant x_{j,s} \leqslant 1 && \forall j\, \forall s && (C_5)
\end{aligned}
$$

Constraints (C_1) and (C_2) report the load induced by the choices of the $x_{j,s}$ variables. Constraints (C_3) and (C_5) ensure that each job is entirely processed. Constraint (C_4) represents the locality constraint.

The solution of the previous linear program is then rounded to an integral solution. A convex allocation can be identified by its leftmost node (i.e., the node of the smallest index, or s) and its size. The allocation chosen for a job j is the allocation with the smallest index σ such that $\sum_{s \leqslant \sigma} x_{j,s}$ is at least a half. The rounding is done independently for each job. The rounding procedure is detailed on Figure 5.3.

In the setup where I/O nodes can be shared, the linear program is modified to ignore the load on the I/O nodes. This is done by altering the Constraint (C_2).

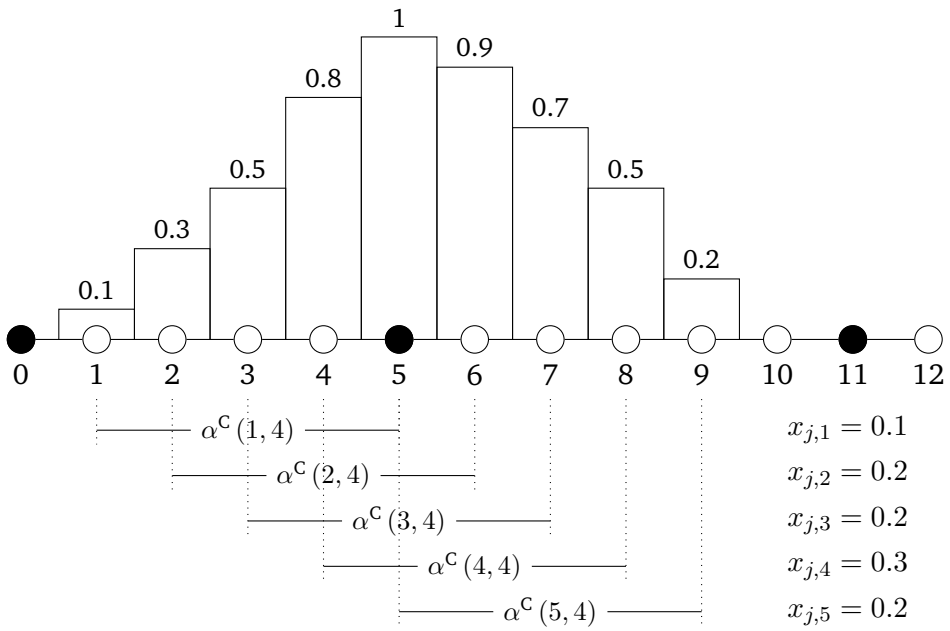


Figure 5.3 Rounding procedure of the linear program for the dedication phase.
White nodes represent compute nodes, and black nodes represent I/O nodes.
Below the nodes are represented the valid allocations for a job j requesting the I/O node of index 5 and four compute nodes. Among the valid allocations for j, the linear program distributed the load as indicated by the $x_{j,s}$ variables. This distribution is influenced by the other jobs that are not represented for the sake of clarity.
Above the nodes is depicted the induced cumulated load on each node. Without loss of generality, we assume here $p_j = 1$.
The rounding procedure would dedicate the nodes $\alpha^C(3,4)$ to job j, as the node of index 3 is the node of the smallest index with a load of at least 0.5.

Scheduling Phase Once an allocation has been chosen for every job, our problem is an instance of the dedicated processors scheduling problem. We compute a feasible schedule with the algorithm proposed by Gergov [Ger99] (see Algorithm 3). The algorithm is split in two phases: first it constructs an infeasible schedule such that two jobs can overlap but the intersection of any three jobs is empty, then it transforms this infeasible schedule into a feasible one.

Algorithm 3: Gergov's algorithm

Input: a set $\mathcal{J}$ of jobs; the number m of nodes
Output: a feasible schedule σ

```
1  begin Incremental 2-Allocation
2      P ← {(0, 0, m − 1)}   /* priority queue with lexicographic order */
3      σ′ ← empty schedule
4      while J ≠ ∅ do
5          p = (t, x_l, x_r) ← P.popmin()
6          if ∃j ∈ J s.t. j only intersects with p then
7              remove j from J
8              σ′(j) ← t
9              L ← min q_j                          /* leftmost node of j */
10             R ← max q_j                          /* rightmost node of j */
11             P.insert((t + p_j, max(x_l, L), min(x_r, R)))        /* ∩ pillar */
12             if x_l < L then
13                 P.insert((t, x_l, L))                          /* left pillar */
14             end
15             if R < x_r then
16                 P.insert((t, R, x_r))                         /* right pillar */
17             end
18         end
19     end
20 end
21 begin Feasible Scheduling
22     color jobs using first fit
23     build a feasible schedule σ using σ′ and one shelf per color
24 end
```

The first phase relies on an auxiliary data structure called the *pillar structure*. A pillar is a convex subset of nodes with an associated height where the height corresponds to the load of the nodes in the pillar. It is represented as a triple (h, L, R), where h, L and R are the height, leftmost node and rightmost node, respectively. The pillars are stored in a priority queue with ascending lexicographic order. The pillar structure is initialized with a single pillar of height 0 containing all the nodes (line 2). Then, jobs are successively stacked on the pillars. At each iteration of the algorithm, the

smallest pillar is removed from the priority queue (line 5). If there does not exist a job that only intersects with this pillar, the pillar is simply removed. Otherwise, the job is stacked on the pillar, and the pillar structure is updated accordingly (lines 6 to 18). Three new pillars at most are created. Figure 5.4(b) on page 93 depicts an infeasible schedule produced by this first phase.

For the second phase, the algorithm assigns a color to each job such that two overlapping jobs have different colors (line 22). This coloring is done with a first-fit strategy. Then, a shelf is created for each color, and all jobs of a given color are scheduled in the corresponding shelf (line 23). The final produced schedule is shown on Figure 5.4(c), page 93.

When dealing with shared I/O, the I/O nodes are removed from the topology seen by Gergov's algorithm.

Analysis with a single I/O node per job

We show in this section that the proposed algorithm is a 6-approximation for the instances where each job requires a single I/O node. To ensure this performance guarantee, we link both phases by bounding the maximum induced load.

Lemma 5.7. *The dedication phase finds an allocation for each job such that the maximum load is at most two times the maximum load of the optimal allocation.*

Proof. Without loss of generality, we consider a single job j of unitary processing time. To prove the correctness of the rounding procedure, we need to show there exists a large enough convex set of compute nodes processing at least half of the job.

The locality constraint ensure a unitary load on the targeted I/O node. Hence, there exists a compute node that is loaded at least half of the job processing time. We denote by σ the leftmost—according to the order exposed above—compute node loaded at least half the processing time of the job. By construction, we have $x_{j,\sigma} + \sum_{s<\sigma} x_{j,s} + \sum_{s>\sigma} x_{j,s} = 1$ and $\sum_{s<\sigma} x_{j,s} < 0.5$. We can derive from both these relations that $x_{j,\sigma} + \sum_{s>\sigma} x_{j,s} = 1 - \sum_{s<\sigma} x_{j,s} > 0.5$. Hence, all the compute nodes belonging to the allocation with the leftmost node σ are loaded at least half the processing time of the job.

Allocating the whole job to the above-mentioned allocation doubles at most the (optimal) load computed by the linear program. □

Lemma 5.8. *Gergov's algorithm computes a feasible schedule for the dedicated processor scheduling problem such that the makespan of the schedule is at most three times the maximum load. The proof of the lemma is available in the original publication [Ger99].*

Theorem 5.9. *The two-phases algorithm is a* 6*-approximation algorithm. This is a direct consequence of Lemmas 5.7 and 5.8.*

Theorem 5.10. *The* 3*-approximation ratio of Gergov's algorithm is asymptotically tight.*

Proof. We prove the approximation ratio of the algorithm is tight with an adversary argument.

Given a target makespan of M for the optimal solution, we craft an instance with $3M + 1$ nodes and $4M$ jobs of unitary processing time (i.e., $p_j = 1$). The nodes are indexed by $i \in 0, \ldots, 3M$, and the jobs are indexed by $j \in 0, \ldots, 4M - 1$. The jobs are divided in four families: jobs aligned on the left edge when $j \equiv 0 \pmod 4$, jobs aligned on the right edge when $j \equiv 1 \pmod 4$, jobs centered on the node of index M when $j \equiv 2 \pmod 4$, and jobs centered on the node of index $2M$ when $j \equiv 3 \pmod 4$. More specifically, the jobs require the following nodes:

$$
\begin{array}{llll}
q_0 = \{0\}, & q_{4k} = \{0, \ldots, M-k\} & \forall k \in 1, \ldots, M-1 \\
q_1 = \{3M\}, & q_{4k+1} = \{3M-k, \ldots, 3M\} & \forall k \in 1, \ldots, M-1 \\
q_2 = \{M\}, & q_{4k+2} = \{M-k, \ldots, M+k\} & \forall k \in 1, \ldots, M-1 \\
q_3 = \{2M\}, & q_{4k+3} = \{M+k, \ldots, 3M-k\} & \forall k \in 1, \ldots, M-1
\end{array}
$$

The key element in this structure is that the node of index 0 is only requested by jobs that belongs to the first family. Similarly, the nodes of index $3M$, M and $2M$ are only requested by the second, third and fourth family, respectively.

On one hand, an optimal schedule of this instance has a makespan of M, as shown on Figure 5.4(a). An area argument shows the optimality of the depicted schedule as it reaches the lower bound of the total work divided by the number of available nodes.

On the other hand, let us suppose jobs are sorted by increasing index. In such a case, thanks to the structure in four families, the four first jobs define four pillars that span the entire first phase of the algorithm. As jobs are sorted by increasing index, they are scheduled by layers of jobs of index with identical quotient when divided by four. The jobs in the middle intersect other jobs at both their extrema. These intersections impose to use three colors in the coloring step, as depicted by Figure 5.4(b). Finally,

after the shelving step, the solution computed by the algorithm has a makespan of $3M - 2$ (see Figure 5.4(c)).

Hence, for any positive ϵ, there exists an M such that the approximation ratio is at least $3 - \epsilon$. This concludes the proof. □

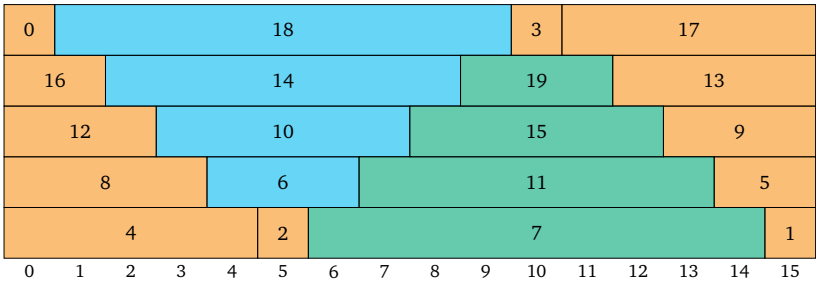
Ring reduction Let us point out that we supposed in both the formulation of the linear program and the proofs that the nodes can be totally ordered. This is trivial with the line topology. However, finding such an order on the ring topology is not straightforward as it is a cyclic graph. We can nonetheless order the allocations with the intuitive leftmost order introduced above. Note that the allocation ordering and the rounding are job dependent, and are independent of other jobs. The ring ($\mathbb{Z}_{/\star\mathbb{Z}}$) can be seen as the quotient space of an infinite line ($\mathbb{Z}$). The projection of an allocation is the set of all the nodes whose representative node on the ring belongs to the initial allocation. This set is a collection of intervals. We choose to consider a single interval to account for the allocation. The choice is done by selecting the interval that contains an origin node. This origin node is arbitrarily chosen, on a per job basis, as a node on the infinite line equivalent to the requested I/O node on the ring. This transformation clearly brings us to the same setup as the line. Such a transformation is depicted on Figure 5.5.

Extending to any number of I/O nodes per job

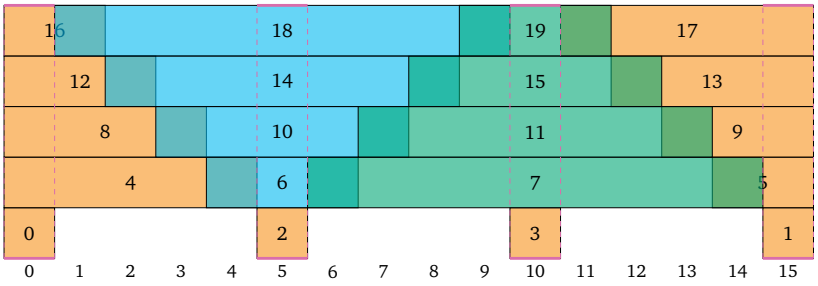
Under a line topology, the more I/O nodes are required by a convex local job, the easier the problem becomes. As the number of required I/O nodes grows, the problem gets more constrained, and it leaves fewer choices for the compute nodes intervals. The algorithm proposed in Section 5.3.2 can easily be adapted for any instance.

Let us consider a job that requires exactly two I/O nodes. Note than requiring more than two I/O nodes reduces to requiring exactly two I/O nodes: the same transformation considering only the extreme I/O nodes works. The two I/O nodes can be considered as one by using the following transformation:

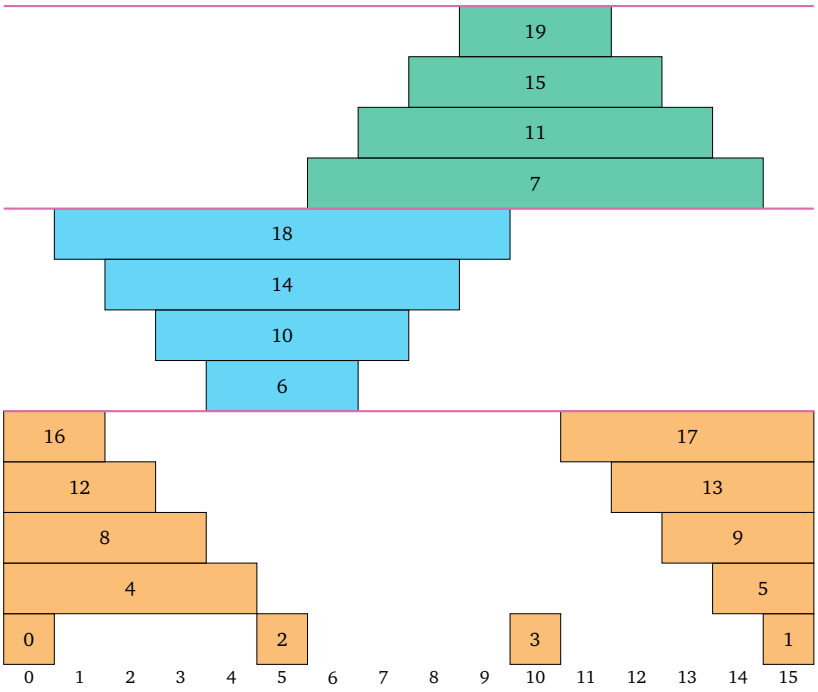
- if there are at least q_j^{C} compute nodes between the two I/O nodes, there exists only a single valid allocation for the job: the interval whose extrema are the I/O nodes;



(a) Optimal schedule.



(b) Infeasible schedule produced by the *Incremental 2-Allocation* phase.



(c) Gergov's final schedule.

Figure 5.4 Comparison of the Gergov's schedule and an optimal schedule for the dedicated processors scheduling problem ($P \mid fix_j \mid C_{\max}$ in Graham's notation).
The instance depicted is the instance described in the proof of the asymptotic tightness with $M = 5$ (see Theorem 5.10). The jobs are colored with the colors from the coloring step of Gergov's algorithm. The nodes' indexes are shown on the horizontal axis, and time is on the vertical axis.

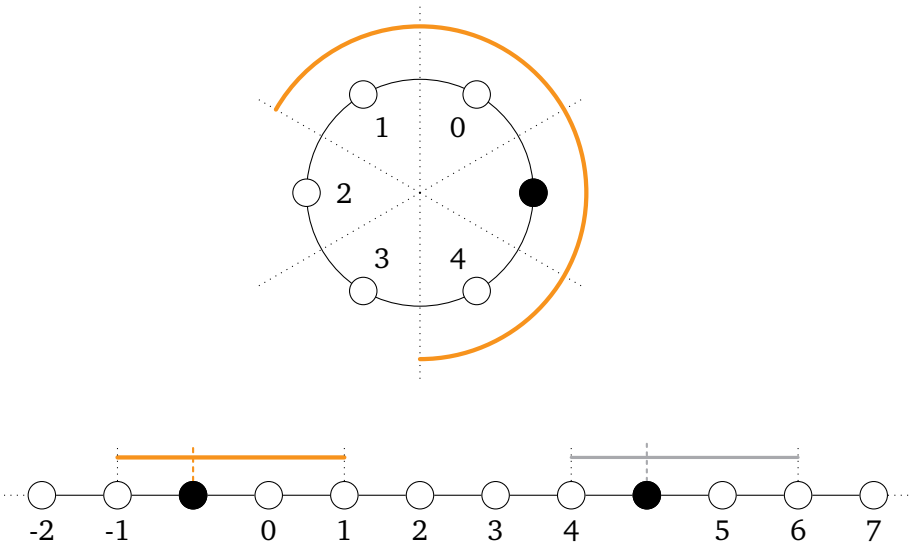


Figure 5.5 Projection of the ring on an infinite line topology. White nodes represent compute nodes, and the black node represents an I/O node. We choose (arbitrarily) the interval $[-1, \ldots, 1]$ to account for the allocation of the job.

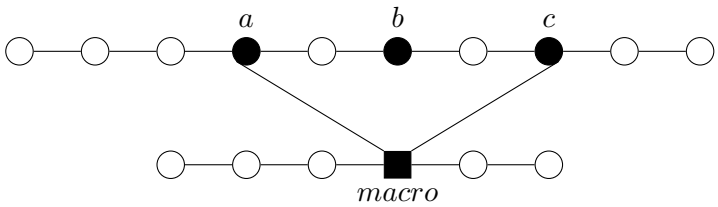


Figure 5.6 Aggregation of many I/O nodes into a macro I/O node through edge contraction. A job j requesting four compute nodes and the I/O nodes a and c can be viewed as another job j' requesting a single $macro$ I/O node and two compute nodes. White nodes represent compute nodes, and black nodes represent I/O nodes.

- if not, consider the I/O nodes and the compute nodes within their range as a single macro I/O. The transformed job is now requiring a single I/O node (the macro I/O) and q_j^{C} compute nodes diminished by the number of compute nodes in the macro I/O.

Such a transformation is depicted on Figure 5.6, and uses edge contraction as a base operation.

This transformation also shows that even if the combination of both the convexity and locality constraints is of great theoretical interest, it might be too great a constraint for systems in production. Ensuring both constraints at the same time obliges the scheduler to give more resources than needed to each job, leading to an underutilization of the machine.

The gap between the line topology and the ring topology widens when the number of requested I/O nodes increases. The transformation from the ring to the line requires to choose the origin (i.e., the base I/O node) to order the allocations (see page 92). Requesting more than a single I/O node brings a more complex combinatorics as the number of potential origins grows: there are indeed as many potential origins as requested I/O nodes. It is however possible to tackle the increased complexity at the cost of a doubled approximation ratio (leading to a ratio of 12).

Theorem 5.11. *Transforming a ring topology into a line topology by cutting between any two nodes doubles at most the maximum load.*

Proof. Consider an optimal allocation of minimal load on a ring. We obtain a derived line topology by cutting between any two nodes. All jobs whose allocations include the cut have to be re-allocated above the remaining uncut jobs. This represents an extra load of at most the maximum load. Hence the load on this derived line topology is bounded by twice the maximum load of the ring topology. □

5.4 Summary

We studied in this chapter some simple instantiations of the framework proposed in Chapter 4. Namely, we studied the minimization of the makespan on the line and ring topologies, while the allocations were constrained to be both convex and local. We gave constant-ratio approximation algorithms for all the variants, considering an uniform distribution of I/O nodes, and jobs requiring a single I/O node and an arbitrary number of compute nodes. Table 5.1 recapitulates the technical results proposed in this chapter for the line topology.

	pinned I/O ($set_j^{I/O}$)		unpinned I/O ($fix_j^{I/O}$)	
	shared	exclusive	shared	exclusive
$p_j = 1$	**NP**h	sNPh	sNPh	
	2-approx.		ρ_{BP}-approx.	
p_j	NPh (cf. $p_j = 1$)	sNPh (cf. $p_j = 1$)	sNPh (cf. $p_j = 1$)	
	6-approx.		$(\rho + \rho_{SP})$-approx.	

Table 5.1 Summary of complexity classes and approximation ratios for the line. The approximation ratios ρ, ρ_{BP} and ρ_{SP} are the ratios for the $P \mid\mid C_{\max}$, Bin Packing and Strip Packing problems, respectively.

As future steps, one could implement the proposed algorithms, and study their performances through simulation. From a theoretical point of view, the tightness results show the limits of a two-phase approach. The approximation ratios might be improved by scheduling the problem in a single phase. This could be done, for example, by only using a linear program to compute the schedule.

6 Conclusion and Future Steps

Increasing the number of computing resources embedded in the HPC platforms is not enough to address the ever-increasing demand for computing power. As we are reaching the limits of a sensible energetic envelop for these platforms, technological disruptions are needed. As a consequence, the heterogeneity in HPC platforms becomes more and more pervasive. This heterogeneity comes in various forms, and manifests itself in the architecture of the new platforms as well as in the variety of processed applications. Consequently, this impacts the software stacks managing the resources (and notably the RJMS). In this work, we studied from two standpoints how scheduling policies can leverage heterogeneity, and manage the execution context: within a single application (intra-application level), and at the whole platform scale from the inter-applications perspective.

Intra-Application Level We studied the minimization of the makespan for platforms composed of multiple CPUs and GPUs. For such platforms, the approach of partitioning tasks in two subsets mapped to each architecture has proved powerful. This is however not sufficient to take into account the impact of communications, as such an approach ignores the individuality of resources. We introduced context-awareness with an affinity mechanism: this affinity guides the global view of the scheduler with qualitative hints improving local behaviors. We showed this is an efficient and cheap mechanism to reduce memory transfers while maintaining a low makespan.

We extended the model to implicitly consider parallelism on the CPUs with the moldable-task model. This leaves more flexibility to leverage the parallelism, as we shift from a *sequential* vs. *vectorial* to a *parallel* vs. *vectorial* paradigm. Starting with the structure of the scheduling problem with CPUs only, we proposed an algorithm based on an integer linear program that directly derives a feasible schedule for the problem with GPUs. This formulation is able to break the complexity barrier that penalizes a dynamic programming approach. Furthermore the formulation of the algorithm as an integer linear program makes it easier to reuse as a future building block.

Inter-Applications Level Moving up to the whole platform scale, it is not realistic to schedule each application down to the single computing resource. Rather, we have considered applications as black boxes with coarse resources requirements. However, quantifying every possible combination of concurrent applications remains unrealistic as the combinatorics explode with the number of applications. Instead, we have proposed to reduce the set of all feasible schedules by further constraining the scheduling problem. These constraints are considered a first-class modeling tool. We have proposed a set of reasonable constraints to model application spreading and I/O traffic.

More precisely, we then instantiated this modeling framework with unidimensional topologies. This case study focused on minimizing the makespan under the convex and local constraints. This is a first step towards the study of production grade topologies. Although the optimization problems remain hard, we were able to derive low-complexity constant-ratio approximation algorithms. This theoretical study shows that it is viable to use adequate constraints as a modeling tool to obliviously apprehend the execution context.

6.1 Future Steps

We proposed in this work generic low-cost algorithms that are able to take into account the execution context. However, some points were not fully addressed, and deserve to be further studied.

First, from a technical point of view, the algorithms proposed in this work may be implemented in other run-time than XKaapi. The performances of the algorithms could also be assessed with different workloads.

The two-phase approach (partitioning then scheduling) proved to be an efficient technique to design scheduling policies for the platforms with CPUs and GPUs. However, as the local interactions have a greater impact in the new proposed modeling framework, it is unclear if such an approach is the good one in this case. Trying to design algorithms with a unique scheduling phase is hence a path to consider.

A natural extension of the unidimensional case study is the study of topologies of greater dimensions such as 2D or 3D tori. Following this path, the study of the tree-based topologies also is a step towards the study of topologies such as DragonFly or SlimFly.

Lastly, all our work was focused on the heterogeneity of the resources in the targeted platforms. Another direction would be to consider the heterogeneity of applications instead. Indeed, by distinguishing applications sensitive to the surrounding I/O traffic from the insensitive applications, one could efficiently co-schedule these classes of applications. A first step in this direction would be to identify reasonable classes of applications.

Bibliography

[@bw] *Blue Waters User Portal*. URL: https://bluewaters.ncsa.illinois.edu/ (visited on Aug. 9, 2017). | cit. on pp. 11, 79

[@curie] *TGCC Curie Supercomputer*. URL: http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm (visited on Aug. 9, 2017). | cit. on pp. 11, 79

[@green500] Wu Feng and Tom Scogland. *Green500 list*. URL: https://www.green500.org/lists/ (visited on June 29, 2017). | cit. on p. 2

[@kcomp] *K Computer*. URL: http://www.aics.riken.jp/en/k-computer/about/ (visited on Aug. 9, 2017). | cit. on p. 79

[@ofp] *Oakforest-PACS System*. URL: http://jcahpc.jp/eng/ofp_intro.html (visited on Aug. 9, 2017). | cit. on p. 79

[@titan] *Titan Cray XK7*. URL: https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/ (visited on Aug. 9, 2017). | cit. on p. 79

[@top500] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *TOP500 list*. URL: https://www.top500.org/lists/ (visited on June 16, 2017). | cit. on p. 1

[@torque] *Torque 6.1.1 Administrator Guide*. URL: http://docs.adaptivecomputing.com/torque/6-1-1/adminGuide/help.htm (visited on June 16, 2017). | cit. on p. 40

[Age+14] Anthony Agelastos, Benjamin A. Allan, Jim M. Brandt, et al. "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications". In: *SC*. IEEE, Nov. 2014, pp. 154–165. DOI: 10.1109/SC.2014.18. | cit. on p. 77

[Agu+11a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, et al. "LU Factorization for Accelerator-based Systems". In: *AICCSA*. IEEE, Dec. 2011, pp. 217–224. DOI: 10.1109/AICCSA.2011.6126599. | cit. on pp. 16, 20, 30

[Agu+11b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, et al. "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In: *IPDPS*. IEEE, May 2011, pp. 932–943. DOI: 10.1109/IPDPS.2011.90. | cit. on pp. 16, 20, 30

[Alb15] Carl Albing. "Characterizing Node Orderings for Improved Performance". In: *PMBS@SC*. ACM, 2015, 6:1–6:11. DOI: 10.1145/2832087.2832094. | cit. on p. 72

[Ash+10] Steve Ashby, Pete Beckman, Jackie Chen, et al. *Opportunities and Challenges of Exascale Computing*. Tech. rep. U.S. Department of Energy, 2010. URL: https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf. | cit. on p. 2

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures". In: *Euro-Par Workshops*. Vol. 6043. Lecture Notes in Computer Science. Springer, Aug. 2009, pp. 56–65. DOI: 10.1007/978-3-642-14122-5_9. | cit. on pp. 16, 20, 21

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (Nov. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. | cit. on pp. 16, 19, 20, 21, 22, 59

[BGT97] Evripidis Bampis, Frédéric Guinand, and Denis Trystram. "Some models for scheduling parallel programs with communication delays". In: *Discrete Applied Mathematics* 72.1 (Jan. 1997), pp. 5–24. DOI: 10.1016/S0166-218X(96)00034-0. | cit. on p. 7

[Bha+13] Abhinav Bhatele, Kathryn Mohror, Steve H. Langer, and Katherine E. Isaacs. "There Goes the Neighborhood: Performance Degradation due to Nearby Jobs". In: *SC*. ACM, Nov. 2013, 41:1–41:12. DOI: 10.1145/2503210.2503247. | cit. on pp. 5, 67

[Bła+07] Jacek Błażewicz, Klaus H. Ecker, Erwin Pesh, Günter Schmidt, and Jan Weglarz. *Handbook on Scheduling: From Theory to Applications*. International Handbooks on Information Systems. Springer, 2007. DOI: 10.1007/978-3-540-32220-7. | cit. on p. 18

[Błą+15] Iwo Błądek, Maciej Drozdowski, Frédéric Guinand, and Xavier Schepler. "On contiguous and non-contiguous parallel task scheduling". In: *Journal of Scheduling* 18.5 (Oct. 2015), pp. 487–495. DOI: 10.1007/s10951-015-0427-z. | cit. on p. 72

[Ble+14] Raphaël Bleuse, Thierry Gautier, João Vicente Ferreira Lima, Grégory Mounié, and Denis Trystram. "Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures". In: *Euro-Par*. Vol. 8632. Lecture Notes in Computer Science. Springer, Aug. 2014, pp. 560–571. DOI: 10.1007/978-3-319-09873-9_47. | cit. on p. 16

[Ble+15] Raphaël Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. "Scheduling independent tasks on multi-cores with GPU accelerators". In: *Concurrency and Computation: Practice and Experience* 27.6 (2015), pp. 1625–1638. DOI: 10.1002/cpe.3359. | cit. on pp. 16, 24, 27, 54

[Ble+17] Raphaël Bleuse, Sascha Hunold, Safia Kedad-Sidhoum, et al. “Scheduling Independent Moldable Tasks on Multi-Cores with GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (Sept. 2017), pp. 2689–2702. DOI: 10.1109/TPDS.2017.2675891. | cit. on p. 37

[Bos+12] George Bosilca, Aurélien Bouteiller, Anthony Danalis, et al. “DAGuE: A generic distributed DAG engine for High Performance Computing”. In: *Parallel Computing* 38.1 (Jan. 2012), pp. 37–51. DOI: 10.1016/j.parco.2011.10.003. | cit. on pp. 16, 20

[Bou+10a] Marin Bougeret, Pierre-François Dutot, Klaus Jansen, Christina Otte, and Denis Trystram. “A Fast 5/2-Approximation Algorithm for Hierarchical Scheduling”. In: *Euro-Par (1)*. Vol. 6271. Lecture Notes in Computer Science. Springer, Aug. 2010, pp. 157–167. DOI: 10.1007/978-3-642-15277-1_16. | cit. on p. 40

[Bou+10b] Azzedine Boukerche, Jan Mendonça Correa, Alba Cristina Magalhaes Alves de Melo, and Ricardo P. Jacobi. “A Hardware Accelerator for the Fast Retrieval of DIALIGN Biological Sequence Alignments in Linear Space”. In: *IEEE Transactions on Computers* 59.6 (June 2010), pp. 808–821. DOI: 10.1109/TC.2010.42. | cit. on p. 16

[Bou+11] Marin Bougeret, Pierre-François Dutot, Klaus Jansen, Christina Robenek, and Denis Trystram. “Approximation Algorithms for Multiple Strip Packing and Scheduling Parallel Jobs in Platforms”. In: *Discrete Mathematics, Algorithms and Applications* 3.4 (Dec. 2011), pp. 553–586. DOI: 10.1142/S1793830911001413. | cit. on p. 5

[Bre74] Richard Peirce Brent. “The Parallel Evaluation of General Arithmetic Expressions”. In: *Journal of the ACM* 21.2 (Apr. 1974), pp. 201–206. DOI: 10.1145/321812.321815. | cit. on p. 39

[Bru07] Peter Brucker. *Scheduling Algorithms*. Fifth Edition. Springer, 2007. DOI: 10.1007/978-3-540-69516-5. | cit. on p. 70

[Bue+12] Javier Bueno, Judit Planas, Alejandro Duran, et al. “Productive Programming of GPU Clusters with OmpSs”. In: *IPDPS*. IEEE, May 2012, pp. 557–568. DOI: 10.1109/IPDPS.2012.58. | cit. on pp. 16, 19, 20, 21

[But+09] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In: *Parallel Computing* 35.1 (Jan. 2009), pp. 38–53. DOI: 10.1016/j.parco.2008.10.002. | cit. on pp. 15, 20, 30

[BW12] Vincenzo Bonifaci and Andreas Wiese. “Scheduling Unrelated Machines of Few Different Types”. In: *CoRR* abs/1205.0974 (May 2012). URL: https://arxiv.org/abs/1205.0974. | cit. on pp. 15, 18

[Car+11] Philip H. Carns, Kevin Harms, William E. Allcock, et al. “Understanding and Improving Computational Science Storage Access through Continuous Characterization”. In: *ACM Transactions on Storage* 7.3 (Oct. 2011), 8:1–8:26. DOI: 10.1145/2027066.2027068. | cit. on p. 77

[Che+16] Nan-Chen Chen, Sarah S. Poon, Lavanya Ramakrishnan, and Cecilia R. Aragon. “Considering Time in Designing Large-Scale Systems for Scientific Computing”. In: *CSCW*. ACM, Feb. 2016, pp. 1533–1545. DOI: 10.1145/2818048.2819988. | cit. on p. 67

[Cof+80] Edward Grady Coffman Jr., Michael Randolph Garey, David Stifler Johnson, and Robert Endre Tarjan. “Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms”. In: *SIAM Journal on Computing* 9.4 (Nov. 1980), pp. 808–826. DOI: 10.1137/0209062. | cit. on p. 40

[Cul+93] David E. Culler, Richard M. Karp, David A. Patterson, et al. “LogP: Towards a Realistic Model of Parallel Computation”. In: *PPOPP*. ACM, May 1993, pp. 1–12. DOI: 10.1145/155332.155333. | cit. on p. 7

[CYZ13] Lin Chen, Deshi Ye, and Guochuan Zhang. “Online Scheduling on a CPU-GPU Cluster”. In: *TAMC*. Vol. 7876. Lecture Notes in Computer Science. Springer, 2013, pp. 1–9. DOI: 10.1007/978-3-642-38236-9_1. | cit. on p. 16

[Dev+14] Mehmet Deveci, Sivasankaran Rajamanickam, Vitus J. Leung, et al. “Exploiting Geometric Partitioning in Task Mapping for Parallel Computers”. In: *IPDPS*. IEEE, May 2014, pp. 27–36. DOI: 10.1109/IPDPS.2014.15. | cit. on p. 77

[DMT04] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. “Scheduling Parallel Tasks Approximation Algorithms”. In: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Computer & Information Science Series. Chapman and Hall/CRC, Apr. 2004. DOI: 10.1201/9780203489802.ch26. | cit. on p. 9

[Don+11] Jack Dongarra, Peter H. Beckman, Terry Moore, et al. “The International Exascale Software Project roadmap”. In: *International Journal of High Performance Computing Applications* 25.1 (Jan. 2011), pp. 3–60. DOI: 10.1177/1094342010391989. | cit. on p. 1

[Dor+16] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Robert B. Ross. “Using Formal Grammars to Predict I/O Behaviors in HPC: The Omnisc’IO Approach”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.8 (Aug. 2016), pp. 2435–2449. DOI: 10.1109/TPDS.2015.2485980. | cit. on p. 77

[Dro09] Maciej Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009. DOI: 10.1007/978-1-84882-310-5. | cit. on pp. 70, 75

[Dus+96] Andrea C. Dusseau, David E. Culler, Klaus E. Schauser, and Richard P. Martin. “Fast Parallel Sorting Under LogP: Experience with the CM-5”. In: *IEEE Transactions on Parallel and Distributed Systems* 7.8 (Aug. 1996), pp. 791–805. DOI: 10.1109/71.532111. | cit. on p. 7

[EBB16] R. Todd Evans, James C. Browne, and William L. Barth. “Understanding Application and System Performance Through System-Wide Monitoring”. In: *IPDPS Workshops*. IEEE, May 2016, pp. 1702–1710. DOI: 10.1109/IPDPSW.2016.145. | cit. on p. 77

[Eno+14] Jeremy Enos, Gregory H. Bauer, Robert Brunner, et al. “Topology-Aware Job Scheduling Strategies for Torus Networks”. In: *Cray User Group*. May 2014. URL: https://cug.org/proceedings/cug2014_proceedings/includes/files/pap182.pdf. | cit. on pp. 67, 70, 74, 77

[Eyr06] Lionel Eyraud. “Théorie et pratique de l’ordonnancement d’applications sur les systèmes distribués”. PhD thesis. ID-IMAG, Institut National Polytechnique de Grenoble, Grenoble, France, Oct. 2006. URL: http://graal.ens-lyon.fr/~leyraudd/These/manuscrit.pdf. | cit. on p. 40

[Fan+12] Liya Fan, Fa Zhang, Gongming Wang, and Zhiyong Liu. “An effective approximation algorithm for the Malleable Parallel Task Scheduling problem”. In: *Journal of Parallel and Distributed Computing* 72.5 (May 2012), pp. 693–704. DOI: 10.1016/j.jpdc.2012.01.011. | cit. on p. 41

[Fei+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. “Theory and Practice in Parallel Job Scheduling”. In: *JSSPP*. Vol. 1291. Lecture Notes in Computer Science. Springer, 1997, pp. 1–34. DOI: 10.1007/3-540-63574-2_14. | cit. on pp. 9, 68

[Fer+12] João Vicente Ferreira Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. “Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs”. In: *SBAC-PAD*. IEEE, Oct. 2012, pp. 75–82. DOI: 10.1109/SBAC-PAD.2012.28. | cit. on p. 16

[Fri87] Donald K. Friesen. “Tighter Bounds for LPT Scheduling on Uniform Processors”. In: *SIAM Journal on Computing* 16.3 (June 1987), pp. 554–560. DOI: 10.1137/0216037. | cit. on p. 18

[FW78] Steven Fortune and James Wyllie. “Parallelism in Random Access Machines”. In: *STOC*. ACM, May 1978, pp. 114–118. DOI: 10.1145/800133.804339. | cit. on p. 7

[Gai+15] Ana Gainaru, Guillaume Aupy, Anne Benoit, et al. “Scheduling the I/O of HPC Applications Under Congestion”. In: *IPDPS*. IEEE, May 2015, pp. 1013–1022. DOI: 10.1109/IPDPS.2015.116. | cit. on pp. 6, 77

[Gau+13] Thierry Gautier, João Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. “XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures”. In: *IPDPS*. IEEE, May 2013, pp. 1299–1308. DOI: 10.1109/IPDPS.2013.66. | cit. on pp. 16, 19, 20, 29, 34

[GBP07] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. “KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors”. In: *PASCO*. ACM, 2007, pp. 15–23. DOI: 10.1145/1278177.1278182. | cit. on p. 20

[Geo+17] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. “Topology-aware Resource Management for HPC Applications”. In: *ICDCN*. ACM, 2017, 17:1–17:10. DOI: 10.1145/3007748.3007768. | cit. on p. 76

[Geo10] Yiannis Georgiou. “Contributions for Resource and Job Management in High Performance Computing”. PhD thesis. LIG, Univ. Grenoble Alpes, France, Nov. 2010. URL: https://tel.archives-ouvertes.fr/tel-01499598. | cit. on p. 3

[Ger99] Jordan Gergov. “Algorithms for Compile-Time Memory Optimization”. In: *SODA*. ACM/SIAM, Jan. 1999, pp. 907–908. URL: https://dl.acm.org/citation.cfm?id=314500.315082. | cit. on pp. 89, 91

[GG75] Michael Randolph Garey and Ronald Lewis Graham. “Bounds for Multiprocessor Scheduling with Resource Constraints”. In: *SIAM Journal on Computing* 4.2 (June 1975), pp. 187–200. DOI: 10.1137/0204015. | cit. on p. 56

[GJ79] Michael Randolph Garey and David Stifler Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. | cit. on pp. 17, 81, 82, 84, 86

[Gra+79] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. “Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey”. In: *Annals of Discrete Mathematics* 5.2 (1979), pp. 287–326. DOI: 10.1016/S0167-5060(08)70356-X. | cit. on p. 75

[Gra69] Ronald Lewis Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal on Applied Mathematics* 17.2 (Mar. 1969), pp. 416–429. DOI: 10.1137/0117039. | cit. on p. 84

[HC16] Sascha Hunold and Alexandra Carpen-Amarie. “Reproducible MPI Benchmarking is Still Not as Easy as You Think”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.12 (Dec. 2016), pp. 3617–3630. DOI: 10.1109/TPDS.2016.2539167. | cit. on p. 5

[Her+10] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. “Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations”. In: *Euro-Par (2)*. Vol. 6272. Lecture Notes in Computer Science. Springer, Aug. 2010, pp. 235–246. DOI: 10.1007/978-3-642-15291-7_23. | cit. on p. 16

[Hil91] David Hilbert. “Ueber die stetige Abbildung einer Line auf ein Flächenstück”. In: *Mathematische Annalen* 38.3 (Sept. 1891), pp. 459–460. DOI: 10.1007/BF01199431. | cit. on p. 72

[HS87] Dorit S. Hochbaum and David B. Shmoys. “Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results”. In: *Journal of the ACM* 34.1 (Jan. 1987), pp. 144–162. DOI: 10.1145/7531.7535. | cit. on pp. 9, 15, 17, 22, 23, 41

[HS88] Dorit S. Hochbaum and David B. Shmoys. “A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach”. In: *SIAM Journal on Computing* 17.3 (1988), pp. 539–551. DOI: 10.1137/0217033. | cit. on pp. 15, 18

[ICR16] Florin Isaila, Jesús Carretero, and Robert B. Ross. “CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms”. In: *CCGrid*. IEEE, May 2016, pp. 346–355. DOI: 10.1109/CCGrid.2016.24. | cit. on p. 77

[Imr03] Csanád Imreh. “Scheduling Problems on Two Sets of Identical Machines”. In: *Computing* 70.4 (Aug. 2003), pp. 277–294. DOI: 10.1007/s00607-003-0011-9. | cit. on p. 19

[Jai+17] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Todd Gamblin, and Laxmikant V. Kalé. “Partitioning Low-diameter Networks to Eliminate Inter-job Interference”. In: *IPDPS*. IEEE, May 2017, pp. 439–448. DOI: 10.1109/IPDPS.2017.91. | cit. on p. 70

[JP99] Klaus Jansen and Lorant Porkolab. “Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks”. In: *SODA*. ACM/SIAM, Jan. 1999, pp. 490–498. URL: https://dl.acm.org/citation.cfm?id=314500.314870. | cit. on p. 40

[Kat+15] Georgios Kathareios, Cyriel Minkenberg, Bogdan Prisacari, Germán Rodríguez, and Torsten Hoefler. “Cost-Effective Diameter-Two Topologies: Analysis and Evaluation”. In: *SC*. ACM, Nov. 2015, 36:1–36:11. DOI: 10.1145/2807591.2807652. | cit. on pp. 1, 6, 67, 70

[Lee+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, et al. “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU”. In: *ISCA*. ACM, June 2010, pp. 451–460. DOI: 10.1145/1815961.1816021. | cit. on p. 15

[Leu+02] Vitus J. Leung, Esther M. Arkin, Michael A. Bender, et al. “Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies”. In: *CLUSTER*. IEEE, Sept. 2002, pp. 296–304. DOI: 10.1109/CLUSTR.2002.1137758. | cit. on p. 74

[LST90] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. “Approximation Algorithms for Scheduling Unrelated Parallel Machines”. In: *Mathematical Programming* 46.1 (Jan. 1990), pp. 259–271. DOI: 10.1007/BF01585745. | cit. on p. 18

[LT94] Walter Ludwig and Prasoon Tiwari. “Scheduling Malleable and Nonmalleable Parallel Tasks”. In: *SODA*. ACM/SIAM, Jan. 1994, pp. 167–176. URL: https://dl.acm.org/citation.cfm?id=314464.314491. | cit. on p. 40

[Luc+15] Giorgio Lucarelli, Fernando Machado Mendonça, Denis Trystram, and Frédéric Wagner. “Contiguity and Locality in Backfilling Scheduling”. In: *CCGRID*. IEEE, May 2015, pp. 586–595. DOI: 10.1109/CCGrid.2015.143. | cit. on pp. 70, 72, 73

[Mon14] Florence Monna. “Scheduling for new computing platforms with GPUs”. PhD thesis. LIP 6, Pierre and Marie Curie University, Paris, France, Nov. 2014. URL: https://tel.archives-ouvertes.fr/tel-01127919. | cit. on p. 23

[Mor66] G. M. Morton. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep. IBM Ltd., Mar. 1966. URL: https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39. | cit. on p. 72

[MRT07] Grégory Mounié, Christophe Rapine, and Denis Trystram. "A 3/2-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks". In: *SIAM Journal on Computing* 37.2 (2007), pp. 401–412. DOI: 10.1137/S0097539701385995. | cit. on pp. 39, 40, 41, 47

[PDB13] Frédéric Pinel, Bernabé Dorronsoro, and Pascal Bouvry. "Solving very large instances of the scheduling of independent tasks problem on the GPU". In: *Journal of Parallel and Distributed Computing* 73.1 (Jan. 2013), pp. 101–110. DOI: 10.1016/j.jpdc.2012.02.018. | cit. on p. 15

[PML14] Jose Antonio Pascual, José Miguel-Alonso, and José Antonio Lozano. "Application-aware metrics for partition selection in cube-shaped topologies". In: *Parallel Computing* 40.5 (May 2014), pp. 129–139. DOI: 10.1016/j.parco.2014.04.006. | cit. on pp. 74, 76

[PSS08] James C. Phillips, John E. Stone, and Klaus Schulten. "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters". In: *SC*. IEEE, Nov. 2008, 8:1–8:9. DOI: 10.1145/1413370.1413379. | cit. on p. 16

[RN12] Gurulingesh Raravi and Vincent Nélis. "A PTAS for Assigning Sporadic Tasks on Two-type Heterogeneous Multiprocessors". In: *RTSS*. IEEE, Dec. 2012, pp. 117–126. DOI: 10.1109/RTSS.2012.64. | cit. on p. 19

[Son+10] Fengguang Song, Hatem Ltaief, Bilel Hadri, and Jack Dongarra. "Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems". In: *SC*. IEEE, Nov. 2010, pp. 1–11. DOI: 10.1109/SC.2010.48. | cit. on pp. 16, 20

[ST93] David B. Shmoys and Éva Tardos. "An approximation algorithm for the generalized assignment problem". In: *Mathematical Programming* 62.1 (Feb. 1993), pp. 461–474. DOI: 10.1007/BF01585178. | cit. on p. 18

[STD12] Fengguang Song, Stanimire Tomov, and Jack Dongarra. "Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems". In: *ICS*. ACM, 2012, pp. 365–376. DOI: 10.1145/2304576.2304625. | cit. on p. 16

[Ste97] A. Steinberg. "A Strip-Packing Algorithm with Absolute Performance Bound 2". In: *SIAM Journal on Computing* 26.2 (Apr. 1997), pp. 401–409. DOI: 10.1137/S0097539793255801. | cit. on pp. 56, 84

[SV05] Evgeny V. Shchepin and Nodari Vakhania. "An optimal rounding gives a better approximation for scheduling unrelated machines". In: *Operations Research Letters* 33.2 (Mar. 2005), pp. 127–133. DOI: 10.1016/j.orl.2004.05.004. | cit. on p. 18

[SW97] Clifford Stein and Joel Wein. “On the existence of schedules that are near-optimal for both makespan and total weighted completion time”. In: *Operations Research Letters* 21.3 (Oct. 1997), pp. 115–122. DOI: 10.1016/S0167-6377(97)00025-4. | cit. on p. 25

[TDB10] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. “Towards dense linear algebra for hybrid GPU accelerated manycore systems”. In: *Parallel Computing* 36.5 (June 2010), pp. 232–240. DOI: 10.1016/j.parco.2009.12.005. | cit. on pp. 16, 20, 30

[Tes+16] François Tessier, Preeti Malakar, Venkatram Vishwanath, Emmanuel Jeannot, and Florin Isaila. “Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers”. In: *COMHPC@SC*. IEEE, Nov. 2016, pp. 73–81. DOI: 10.1109/COMHPC.2016.013. | cit. on p. 77

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (Mar. 2002), pp. 260–274. DOI: 10.1109/71.993206. | cit. on pp. 10, 16, 20, 22, 56, 59

[TLC15] Ozan Tuncer, Vitus J. Leung, and Ayse Kivilcim Coskun. “PaCMap: Topology Mapping of Unstructured Communication Patterns onto Non-contiguous Allocations”. In: *ICS*. ACM, June 2015, pp. 37–46. DOI: 10.1145/2751205.2751225. | cit. on p. 76

[TWY92] John Turek, Joel L. Wolf, and Philip S. Yu. “Approximate Algorithms for Scheduling Parallelizable Tasks”. In: *SPAA*. June 1992, pp. 323–332. DOI: 10.1145/140901.141909. | cit. on p. 40

[Val90] Leslie Gabriel Valiant. “A Bridging Model for Parallel Computation”. In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181. | cit. on pp. 6, 7, 8

[YKD11] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. *QUARK Users’ Guide: QUeueing And Runtime for Kernels*. Tech. rep. ICL-UT-11-02. University of Tennessee, Apr. 2011. URL: http://www.icl.utk.edu/sites/icl/files/publications/2011/icl-utk-454-2011.pdf. | cit. on pp. 19, 30

List of Figures

1.1 Overview of an application submission process 3
1.2 Gantt chart of a typical HPC workload 4

2.1 Comparison of DA-2, DP-4/3, and HEFT performances 28
2.2 Example of a multi-CPU, multi-GPU system 29
2.3 Performance impact of the affinity parameter 31
2.4 Benchmarks of Cholesky factorization (`DPOTRF`) 33
2.5 Benchmarks of LU factorization (`DGETRF`) 33
2.6 Benchmarks of QR factorization (`DGEQRF`) 33

3.1 Structure in seven sets of the schedule 42
3.2 Intersection graph of the eligible allocation sets 45
3.3 Example of a problem instance 57
3.4 Comparison of the quality of the computed solutions 62
3.5 Comparison of the mean run-time to compute the solutions 64
3.6 Impact of the filtering on the number of possible partitions per task . 65
3.7 Distribution of the number of iterations to converge to a solution . . 66

4.1 Illustration of a HPC platform 69
4.2 Figuration of the two distinguished types of communications 71
4.3 Figuration of the convexity and connectivity constraints 72
4.4 Figuration of the locality constraint 73
4.5 Figuration of the compacity metric 74
4.6 Figuration of the proximity metric 75

5.1 Example of unidimensional topologies 80
5.2 Potential allocations for a job with a single pinned I/O node 86
5.3 Rounding procedure of the LP for the dedication phase 88
5.4 Comparison of the Gergov's schedule and an optimal schedule . . . 93
5.5 Projection of the ring on an infinite line topology 94
5.6 Aggregation of many I/O nodes into a macro I/O node 94

List of Tables

3.1 Parameter settings used to generate scheduling instances 57
3.2 HEFT-like heuristics used for comparison 61

4.1 Extension of Graham notation for the β field 76

5.1 Summary of complexity and approximation results for the line . . . 95

Abstract

The demand for computation power is steadily increasing, driven by the need to simulate more and more complex phenomena with an increasing amount of consumed/produced data. To meet this demand, the High Performance Computing platforms grow in both size and heterogeneity. Indeed, heterogeneity allows splitting problems for a more efficient resolution of sub-problems with *ad hoc* hardware or algorithms. This heterogeneity arises in the platforms' architecture and in the variety of processed applications. Consequently, the performances become more sensitive to the execution context.

We study in this dissertation how to qualitatively bring—at a reasonable cost—context-awareness/obliviousness into allocation and scheduling policies. This study is conducted from two standpoints: within single applications, and at the whole platform scale from an inter-applications perspective.

We first study the minimization of the *makespan* of sequential tasks on platforms with a mixed architecture composed of multiple CPUs and GPUs. We integrate context-awareness into schedulers with an affinity mechanism that improves local behavior. This mechanism has been implemented in a parallel run-time, and experiments show that it is able to reduce the memory transfers while maintaining a low makespan. We then extend the model to implicitly consider parallelism on the CPUs with the moldable-task model. We propose an efficient algorithm formulated as an integer linear program with a constant performance guarantee of $\frac{3}{2} + \epsilon$.

Second, we devise a new modeling framework where constraints are a first-class tool. Rather than extending existing models to consider all possible interactions, we reduce the set of feasible schedules by further constraining existing models. We propose a set of reasonable constraints to model application spreading and I/O traffic. We then instantiate this framework for unidimensional topologies, and propose a comprehensive case study of the makespan minimization under convex and local constraints.

Résumé

Le besoin de simuler des phénomènes toujours plus complexes accroît les besoins en puissance de calcul, tout en consommant et produisant de plus en plus de données. Pour répondre à cette demande, la taille et l'hétérogénéité des plateformes de calcul haute performance augmentent. L'hétérogénéité permet en effet de découper les problèmes en sous-problèmes, pour lesquels du matériel ou des algorithmes *ad hoc* sont plus efficients. Cette hétérogénéité se manifeste dans l'architecture des plateformes et dans la variété des applications exécutées. Aussi, les performances sont de plus en plus sensibles au contexte d'exécution.

L'objet de cette thèse est de considérer, qualitativement et à faible coût, l'impact du contexte d'exécution dans les politiques d'allocation et d'ordonnancement. Cette étude est menée à deux niveaux : au sein d'applications uniques, et à l'échelle des plateformes au niveau inter-applications.

Nous étudions en premier lieu la minimisation du temps de complétion pour des tâches séquentielles sur des plateformes hybrides intégrant des CPU et des GPU. Nous proposons de tenir compte du contexte d'exécution grâce à un mécanisme d'affinité améliorant le comportement local des politiques d'ordonnancement. Ce mécanisme a été implémenté dans un *run-time* parallèle. Une campagne d'expérience montre qu'il permet de diminuer les transferts de données tout en conservant un faible temps de complétion. Puis, afin de prendre implicitement en compte le parallélisme sur les CPU, nous enrichissons le modèle en considérant les tâches comme *moldables* sur CPU. Nous proposons un algorithme basé sur la programmation linéaire en nombres entiers. Cet algorithme efficace a un rapport de compétitivité de $\frac{3}{2} + \epsilon$.

Dans un second temps, nous proposons un nouveau cadre de modélisation dans lequel les contraintes sont des outils de premier ordre. Plutôt que d'étendre les modèles existants en considérant toutes les interactions possibles, nous réduisons l'espace des ordonnancements réalisables via l'ajout de contraintes. Nous proposons des contraintes raisonnables pour modéliser l'étalement des applications ainsi que les flux d'E/S. Nous proposons ensuite une étude de cas exhaustive dans le cadre de la minimisation du temps de complétion pour des topologies unidimensionnelles, sous les contraintes de convexité et de localité.