Scheduling for new computing platforms with GPUs

Florence Monna

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Florence MONNA

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Ordonnancement pour les nouvelles plateformes de calcul avec GPUs

devant le jury composé de :

M. Jacek Blazewicz	Examinateur	Université de Technologie de Poznan
M. Christophe Cérin	Rapporteur	LIPN, Université Paris XIII
Mme Safia Kedad-Sidhoum	Directrice de thèse	LIP6, Université Pierre et Marie Curie
M. Grégory Mounié	Examinateur	LIG, Université de Grenoble
Mme Alix Munier	Examinateur	LIP6, Université Pierre et Marie Curie
M. Rizos . Sakellariou	Rapporteur	Université de Manchester
M. Samuel Thibault	Examinateur	INRIA, Université de Bordeaux
M. Denis Trystram	Directeur de thèse	ENSIMAG

Résumé

Depuis de nombreuses années, les problèmes d'ordonnancement ont traité des systèmes avec des processeurs en parallèle ou bien avec des processeurs dédiés. Avec le développement de nouvelles architectures de calcul, cette classification n'est plus si évidente. De plus en plus d'ordinateurs utilisent des architectures hybrides combinant des processeurs multi-coeurs (CPUs) et des accĺeérateurs matériels comme les GPUs (Graphics Processing Units). Ces plates-formes parallèles hybrides exigent de nouvelles stratégies d'ordonnancement adaptées. Cette thèse est consacrée à une caractérisation de ce nouveau type de problèmes d'ordonnancement. L'objectif le plus étudié dans ce travail est la minimisation du makespan, qui est un problème crucial pour atteindre le potentiel des nouvelles plates-formes en Calcul Haute Performance.
Après une introduction approfondie de ce nouveau type de systèmes de calcul, une extension de la notation classique des problèmes d'ordonnancement est proposée. Le problème central étudié dans ce travail est le probeème d'ordonnancement efficace de n tâches séquentielles indépendantes sur une plateforme de m CPUs et k GPUs, où chaque tâche peut être exécutée soit sur un CPU ou sur un GPU, avec un makespan minimal. Après un aperçu des méthodes de résolution qui ont été utilisées dans ce travail pour s'attaquer à ce nouveau problème, et les problèmes classiques associés, nous présentons les méthodes que nous avons développées pour résoudre le problème d'ordonnancement en premier lieu sur un seul CPU et un GPU, puis ensuite sur m CPUs et k GPUs. Ces problèmes d'ordonnancement sont NP-difficiles, nous proposons donc des algorithmes d'approximation avec des garanties de performance allant de 2 à $\frac{2q+1}{2q} + \frac{1}{2qk}$, $q > 0$, et des complexités polynomiales correspondantes de $\mathcal{O}(n \log n)$ à $\mathcal{O}(n^2 k^{q+1} m^q)$, augmentant lorsque les ratios diminuent, en gardant à l'esprit qu'une véritable plate-forme de calcul a besoin d'efficacité autant que de précision dans l'ordonnancement de ses calculs. La méthode de résolution est basée sur un schéma d'approximation duale qui utilise la programmation dynamique de façon à répartir de manière équitable la charge entre les ressources hétérogènes. La méthode de résolution proposée dans ce travail est le premier algorithme générique pour la planification sur des machines hybrides avec une garantie de performance théorique qui peut être utilisé à des fins pratiques.
Des variates du problème d'ordonnancement avec m CPUs et k GPUs sont étudiées. Un cas particulier où toutes les tâches sont accĺeérées quand elles sont affectées à un GPU, avec un algorithme d'approximation rapide avec un ratio de $\frac{3}{2}$ pour n'importe quel nombre de GPUs est analysé. Une attention est également accordé aux

préemptions, qui peuvent être autorisées sur les CPUs, mais pas sur les GPUs en raison de leur architectures différentes. Nous considérons ensuite le problème de l'intégration du modèle de tâches malléables dans la problématique de l'ordonnancement sur plate-forme hétérogène, et proposons un algorithme avec un ratio d'approximation de $\frac{3}{2}$. Certains de ces algorithmes ont été implémentés. Des expériences basées sur des critères réalistes ont été réalisées. Ces algorithmes ont été intégré dans l'ordonnanceur du système d'exécution xKaapi pour les noyaux d'algèbre linéaire, et comparés au classique algorithme HEFT.
Enfin, nous étudions le problème de planification de tâches dépendantes sur des CPUs et GPUs. Nous proposons un algorithme d'approximation avec une garantie de performance de 6 pour ce problème. L'algorithme a une méthode de résolution en deux phases: une première phase basée sur l'arrondi d'une solution fournie par la résolution d'une formulation en programmation linéaire pour l'affectation des tâches aux ressources. Une deuxième phase utilise un algorithme classique de liste pour planifier les tâches en fonction de l'affectation déterminée dans la première phase. C'est le premier algorithme avec une garantie de performance pour la planification des tâches avec contraintes de précédence sur les plates-formes hybrides avec des ressources CPUs et GPUs.

Abstract

For many years, scheduling problems have been concerned either with parallel processor systems or with dedicated processors. With the development of new computing architectures this partition is no longer so obvious. More and more computers use hybrid architectures combining multi-core processors (CPUs) and hardware accelerators like GPUs (Graphics Processing Units). These hybrid parallel platforms require new scheduling strategies. This work is devoted to a characterization of this new type of scheduling problems. The most studied objective in this work is the minimization of the makespan, which is a crucial problem for reaching the potential of new platforms in High Performance Computing.

After a thorough introduction of this new type of computing systems, an extension of the classical notation of scheduling problems is proposed. The core problem studied in this work is scheduling efficiently n independent sequential tasks with m CPUs and k GPUs, where each task of the application can be processed either on a CPU or on a GPU, with minimum makespan.

After an overview of the solving methods that were used in this work to tackle this new problem, and the classical problems associated with them, we present the methods we developed to solve the problem of scheduling on first only one CPU and one GPU, then m CPUs and k GPUs. These scheduling problems are NP-hard, therefore we propose approximation algorithms with performance ratios ranging from 2 to $\frac{2q+1}{2q} + \frac{1}{2qk}$, $q > 0$, and corresponding polynomial time complexities from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n^2 k^{q+1} m^q)$, increasing when the ratios drop, keeping in mind that a real computing platform need efficiency as much as accuracy in the scheduling of its calculations. The solving method is based on a dual approximation scheme which uses dynamic programming to balance evenly the load between the heterogeneous resources. The proposed solving method is the first general purpose algorithm for scheduling on hybrid machines with a theoretical performance guarantee that can be used for practical purposes.

Some variants of the scheduling problem with m CPUs and k GPUs are studied. A special case where all the tasks are accelerated when assigned to a GPU, with a faster $\frac{3}{2}$-approximation algorithm for any number of GPUs is analyzed. An attention is also paid to preemptions, that can be allowed on CPUs but not on GPUs due to their different architectures. We also consider the problem of integrating the model of malleable tasks into the problem of scheduling on heterogeneous platform, and proposed an algorithm with a performance ratio of $\frac{3}{2}$.

Some of these algorithms were implemented. Experiments based on realistic benchmarks have been conducted. These algorithms have been integrated into the scheduler of the xKaapi runtime system for linear algebra kernels, and compared to the state-of-the-art algorithm HEFT.
Finally, we study the problem of scheduling dependent tasks on CPUs and GPUs. We provide an approximation algorithm with a performance guarantee of 6 to solve this problem. The algorithm is a two-phase solving method: a first phase based on rounding the solution provided by solving a linear programming formulation for the assignment of the tasks to the resources. A second phase uses a classical list algorithm to schedule the tasks according to the assignment determined in the first phase. This is the first algorithm with a performance guarantee for scheduling tasks with precedence constraints on hybrid platforms with CPUs and GPUs resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

In several domains, complex and powerful computations are necessary. Their applications are very diverse, such as real-time finance, weather predictions, molecular modeling, and countless areas of physics.

This need for more computational resources and the considerable technological advances of these last few years have led to the construction of large-scale hierarchical computing platforms for High Performance Computing (HPC). These new platforms are constituted of parallel multi-core processors with a great number of computing units (again called processors), where these units can be heterogeneous: at the finest level, classical processors (CPUs) share a large memory with additional hardware accelerators like General Purpose Graphical Processing Units (GPGPUs, or, in short GPUs) [56].

Indeed, in some domains requiring HPC, the parallelism of processors of the same type is not the best solution. An example where different types of parallel processors are used is DNA assembling problem, where hundreds of millions of DNA chains have to be aligned and the resulting chromosome is to be constructed. In short, this approach requires at the first stage (alignment of DNA chains) a multi-GPU machine, while the second stage (construction of a corresponding DNA graph and finding the resulting path) should be done on a parallel CPU system [7, 8, 53], meaning several CPUs working in parallel to execute complex calculations.

There is an increasing complexity within the internal nodes of such hybrid parallel systems, mainly due to the heterogeneity of the computational resources. To take advantage of the benefits offered by these new features in terms of performance, there is an important need for an effective, automatic management of these hybrid resources at the finest level. Indeed, no just a computing platform does not execute one calculation at a time. There are only a few of these machines for a much greater number of customers with calculations to perform.

These new characteristics have given rise to new scheduling problems, consisting in allocating and sequencing the computations on the different resources such that a given

objective is optimized. The objective in High Performance Computing (HPC) is to execute as fast as possible all the tasks of an application. This means that the aim is to determine the ending time of the execution of the application defined by the largest completion time (makespan) of the tasks on CPUs and GPUs.

The existing scheduling algorithms and tools, abundantly studied and used on previous generation execution systems, are often not well-suited for these new platforms. Then the main challenge is to create adequate generic scheduling methods and software tools that fulfill the requirements for optimizing the performances.

In the field of parallel processing, a huge amount of work has been devoted to implementations of *ad hoc* algorithms using GPU or hybrid CPU-GPU architectures. They expand over several aspects of parallelism from operating system, runtime, application implementation or languages. However, only few of them focus on the intermediate problem of scheduling on hybrid platforms [71]. Most of the works in the literature consist in studying the gains and performances of parallel implementation of some specific numerical kernels [1, 80], or specific applications like multiple alignments of biological sequences [13], or molecular dynamics [70]. The existing scheduling algorithms and tools are usually not well-suited for general purpose applications since the internal hardware organization of a GPU highly differs from a CPU and thus, the GPU should be considered as a new type of resources in order to determine efficient approaches. Scheduling is usually done on a case by case basis and often offers good performances, however, it lacks high-level mechanisms that provide transparent and efficient schedules for any application. Some actual runtime systems include the basic mechanisms for developing scheduling algorithms like OMPSS [16], StarPU [3] or XKaapi [34]. Several scheduling algorithms have been implemented on top of these systems and most scheduling policies are restricted to fast greedy algorithms or work stealing [10, 59]. An online algorithm with a performance guarantee [18] has recently been developed for CPU-GPU platforms, but , to the best of our knowledge, there is no performance guarantee for any offline problem on these systems.

This means that if a customer of a computing platform wishes his/her calculation done in a reasonable amount of time by the platform, considering other users' calculations, there is a chance that the platform scheduling algorithm will assign the calculation to a not-so-well suited processor that will upset the whole schedule of the platform and delay the obtaining of results for all the users of the computing platform.

Let us consider for example the case of one user: a nuclear physicist needs to calculate the independent trajectories of 10 billion neutrons, photons, electrons and positrons inside a nuclear reactor to determine the energy deposition that results from these particle movements inside the reactor [87]. In order to simulate these trajectories, he requests for 512 processors on the CEA computing platform Curie (see Chapter 2) for approximately 24 hours. The batch scheduler of Curie receives the request and assigns the 10 billion calculations a priority depending on the physicist computational quota on Curie. When there is no more calculations with a higher priority in the queue of Curie or if the occupation of 512 processors for 24 hours has not impact on the completion

time of any task with a higher priority waiting to be scheduled, the physicist calculations are assigned to the first group of 512 processors that become free on the platform. The scheduler then considers those 512 processors occupied for the next 24 hours whereas the calculations could be finished earlier with a finer scheduling on these 512 processors. That is the problem we focused on during this PhD. We worked on providing scheduling algorithms for a given set of calculations on a given set of processors, composed on CPUs and GPUs, all gathered on a computing platforms.

1.2 Objectives and Contributions

Since no generic method existed for scheduling calculations on a CPU-GPU platform prior to this work, our objective is to propose a characterization of this type of platform in the scheduling area as well as new scheduling algorithms for a general purpose execution on hybrid CPU-GPU architectures designed for HPC, algorithms that may remain suitable for the successive generations of the evolving computing platforms. The methods that we developed determine the assignment and schedule of the tasks of an application to the computing units, CPUs and GPUs. To the best of our knowledge, there was no automatic approach to solve this strategic problem prior to this work. Various sides were possible to address this problem. A first possibility was to adapt existing models such as unrelated processors or dedicated processors. Another way was to see this problem as the placement of malleable tasks with varying processing times. We could have also considered scheduling problems where it is assumed that the duration of tasks can be reduced with a compression cost [76]. Approaches such as work stealing [10, 59] from GPU to CPU might also have been considered.
The approach we followed was to first determine an appropriate model, capable of taking into account the new characteristics of these systems, and devise appropriate notations for the corresponding scheduling problems. We then developed several algorithms for the case with independent tasks, using several methods such as dynamic programming and the dual approximation technique [43]. Those algorithms are a major contribution to the field of heterogeneous scheduling, being the first in this field to have both practical efficiency and performance guarantee. From this basis we moved on to more specific or complex instances, such as the specific case where all the tasks to be scheduled are accelerated when assigned to a GPU, but not necessarily with the same acceleration factor for all the tasks, which is a case frequently encountered in practice, for instance in DNA sequence comparisons. Another case studied was the case where preemptions are allowed for the tasks assigned to the CPUs, but not for the tasks on the GPUs, since preemptions are possible on classical processors but not on GPUs. The case where the tasks are considered malleable when they are assigned to a CPU and sequential when assigned to a GPU was also considered, since the malleable task model is often used when communications occur within a platform. Finally, not every calculation is independent from the others on a computing platform, therefore the case where the tasks are linked by precedence relations. The last two problems mentioned

represent again a significant contribution to the heterogeneous scheduling field. We validated these algorithms conventionally in the combinatorial optimization community through complexity and approximation analysis, but also by real-sized tests on cards that were available, notably in Grenoble[1], and applied them to DNA sequence comparisons on real genomic database.

1.3 Outline

The outline of the manuscript is as follows. We present in Chapter 2 an introduction to the multiprocessor architectures and the uses of GPUs. In Chapter 3 we introduce new notations for this type of scheduling problems, and present some related works in the field of scheduling and highlight the gaps that need filling in the area of CPU-GPU scheduling. We present in Chapter 4 a formal description of the problem of minimizing the makespan with independent tasks on m CPUs and k GPUs, which is followed by the detail of different approaches, and the corresponding experiments. In Chapter 5, we generalize the approach developed in the previous chapter into a whole family of approximation algorithms for the same scheduling problem. Chapter 6 deals with other scheduling problems with independent tasks we investigated, such as the special case where all the tasks are accelerated when assigned to a GPU, or the problem where the tasks are considered malleable, or when preemptions are allowed on the CPUs. Experiments realized for the problems studied in these chapters are presented in Chapter In Chapter 8, we present the problem of scheduling tasks linked by precedence constraints on CPUs and GPUs and the approximation algorithm we developed to solve this problem. Finally, the conclusion and perspectives of this work is presented in Chapter9.

[1] the tests were performed by the MOAIS team from the LIG, notably Grégory Mounié, Raphaël Bleuse and Fernando Mendonca.

Chapter 2

An Introduction to High Performance Computing and GPUs

In Chapter 1, we have seen the need for large computing platforms with a great number of processors, an introduction to these platforms should be given more thoroughly, and the same should be done concerning the Graphical Processing Units (GPUs) that are the focus of this work. This chapter is more technical than the rest of the thesis, in order to highlight the major differences and therefore specificities of the heterogeneous platforms with GPUs that we dealt with in this work in terms of scheduling.

2.1 High Performance Computing and Supercomputers

The first large-scale computing platforms were designed in the 1960s by Seymour Cray [19] for the biggest company in the field of supercomputers until the 1970s, Control Data Corporation (CDC). Seymour Cray left CDC in the 1970s, and founded Cray Research, a company that surpassed CDC and its other opponents until 1990 [72]. During the 1980s, a lot of small companies went in the business of supercomputers, but most of them sank during the crash of this market in the middle of the 1990s. In the 21st century, large-scale computing platforms are mostly conceived as unique objects by traditional computer firms such as IBM, HP or Bull, whether they have a long lasting tradition in the domain (IBM) or that they bought in the 1990s some specialized companies to acquire their expertise.
The term computing platform varied with time, since the most powerful computers in the world at one moment in time tend to be equaled and then surpassed by ordinary desktop computers later on. The first supercomputers CDC were simple computers with a single processor (but having sometimes up to ten peripheral processors for the inputs and outputs) around ten times faster that their opponents [89]. During the 1970s, most supercomputers adopted vectorial processors, that decoded an instruction only once to apply it to a whole series of operations. It is only at the end of the 1980s that the technique of massively parallel systems was adopted, with the use in one computing

platform of thousands of processors [27]. Nowadays, some computing parallel platforms use Reduced Instruction Set Computer (RISC) microprocessors designed for serial PCs, such as PowerPC (IBM) or PA-RISC (HP) processors [4]. Others use cheaper processors with a Complex Instruction Set Computer (CISC) [54] outer appearance that are microprogrammed in RISC in the chip (AMD, Intel), such as x86 processors: the performances are a little hindered, but the memory access, usually a key parameter, is far less solicited.
Computing platforms are used for all the tasks that need large computing power, such as weather predictions, climate studies, DNA sequencing, molecular modeling, physics simulations (aerodynamics, material resistance, nuclear explosions, nuclear fusion...), cryptography, finance and insurance simulations, etc... Research institutions, both civil and military, are some of the biggest users of computing platforms.
The scale and capabilities of these platforms have grown considerably since the first computing platforms were designed. The Top500 website [83] lists the 500 most powerful computing platforms in terms of the number of operations per second they can achieve. In the June 2014 list, the Chinese computing platform Tianhe-2 was ranked number one with a computing power of 33.86 PFlops (10^{15} FLoating point Operations Per Second). It is composed of 16,000 computer nodes, each comprising two Intel Ivy Bridge Xeon CPUs and three Xeon Phi accelerator chips, counting a total of 3,120,000 cores.
The second place on the June 2014 list is occupied by a heterogeneous platform with GPUs: Titan, built by Cray Inc. for Oak Ridge national laboratory in Tennessee (see Figure 2.1). It uses a hybrid architecture composed of 18 688 CPUs, processors with 16 cores at 2.2 GHz, AMD Opteron 6274, and 18688 Nvidia GPU accelerators, Tesla K20X.

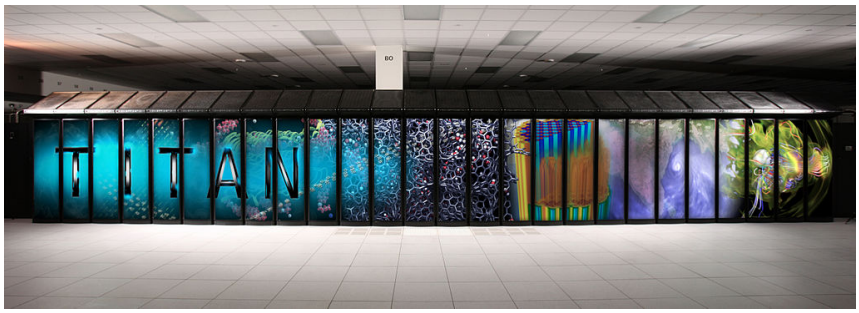


Figure 2.1: An image of Titan, computing platform with GPUs.

Titan's computing power reaches 17.59 PFlops, and could reach theoretically up to 27 PFlops at peak performance. It was also ranked 3rd on the Green500 list of November 2012, thanks to its hybrid architecture with GPUs: its performance per watt is about 2.1 GFlops/W.
In France, we find these machines in the computing centers of universities such as IDRIS, CINES, but also in the CEA and also in some large companies (Total, EDF or Meteo-France). One of these platforms is Curie, a computing platform for the CEA, designed by Bull, with a computing power of 2 PetaFlops (PFlops). It possesses three

computing architectures, the "fat" nodes, the "thin" nodes and the "hybrid" nodes, the last category being composed of heterogeneous processors with GPU accelerators: the "hybrid" nodes are composed of a combination of Intel Westmere CPUs and Nvidia M2090 T20A GPUs, for a total of 288 Intel and 288 Nvidia processors. In October 2012, Curie was the 9th most powerful computing platform in the world, and the most powerful computing platform in France until the Ada and Turing systems were installed at IDRIS in January 2013, and in march 2013, the computing platform Pangea, owned by Total, was launched, becoming the most powerful computing platform in France, with a computing power of 2.3 PFlops. Pangea and Curie were respectively ranked 16^{th} and 26^{th} on the June 2014 Top500 list.

We can see that computing platforms have reached high levels of computational power over the years and their overall complexity has grown with them. These platforms are able to process and transfer massive amounts of data in a very short amount of time. However, information cannot travel faster than the speed of light between two parts of a given platform. Therefore, when the size of a computing platform goes over several meters, the latency between some components can be counted in dozens of nanoseconds. The components of the platform have to be organized to limit the length of the cables linking the components, and the design of a computing platform must ensure that all data can be read, transferred and stored quickly, otherwise the computing power of the processors would be under-exploited. A possible solution to that problem is to use accelerating processors such as Intel Xeon Phi processors or GPUs, that are able to perform simple parallel computation at a very high speed, saving space and power. However, GPUs were not designed for such a general purpose use. Let us focus on the specificities of these processors.

2.2 Graphical Processing Units

Graphic calculations can be very costly, especially if the rendering must be of good quality. The first computers did not have graphical processors. Central processors (CPUs) did all the calculations necessary. In order to focus the CPUs resources on more demanding calculations, graphical processors were added to computers. A GPU (Graphical Processing Unit) was dedicated to the calculations regarding graphics. This specialization made it very fast, in opposition to the common CPU, with a more generic purpose and therefore slower. Over the years GPUs became more complex and versatile. At the end of the 90s, GPUs were capable of computing the calculations necessary for three dimensional graphics. During the 2000s, GPUs slowly became programmable for applications other than graphical imagery and video games. Two important companies design GPUs: NVIDIA and ATI. They increased over the years the raw computing power of their GPUs and at the same time rethought the processors' architecture to enable a more comfortable use. In 2007, NVIDIA released CUDA 1.0, a programming language only for its GPUs. ATI did not release its own software, but support a more generic language that works on the GPUs of both companies, OpenCL. From this point

forward, the new generations of GPUs are called General Purpose Graphical Processing Units (GPGPU). This PhD thesis focuses on the newest generations of GPUs that are used in High Performance Computing, therefore the GPUs considered are all GPGPUs, but for simplicity, they will be called GPUs.

2.2.1 GPU Architecture

Since it was originally designed to perform only graphical calculation, a GPU's architecture differs greatly from a CPU's architecture.
In Figure 2.2, different elements of GPUs and CPUs are represented. The size of the blocks in the figure is proportional to the real size of the components, considering the number of transistors in each component [29]. A CPU (resp. GPU) is composed of the upper block in the figure, the Dynamic Random Access Memory (DRAM) being physically separated from it.

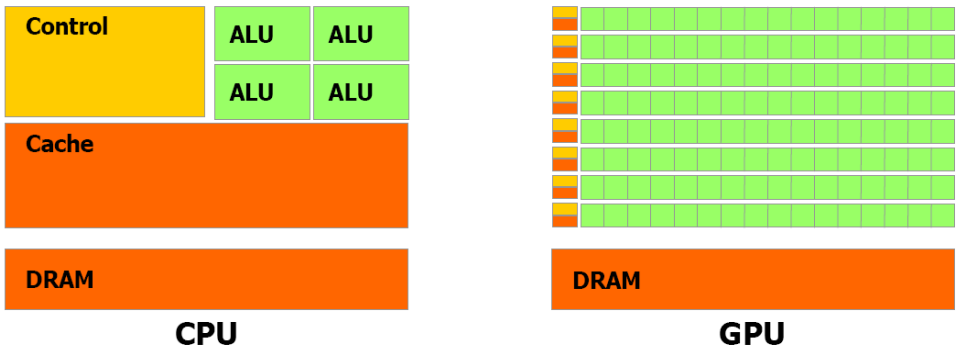


Figure 2.2: Sketch of a CPU architecture (left), and a GPU architecture (right).

A CPU first has a cache, a memory space of a small size but extremely fast. It is used as a work memory for the CPU calculations. Half of the processor's transistors compose the cache. The Arithmetic Logic Unit (ALU, see Figure 2.2) are the calculation units. Their number varies from one architecture to the other. They represent one quarter of the CPU's transistors. Finally, the control structure (Control in Figure 2.2) occupies the last quarter of the CPU's transistors. It contains the connection prediction and has two functions. A CPU is able to perform a great number of operations, so using its different calculation units at the same time can be difficult. The operations are computed in a random order (when it is possible) to optimize the occupation of the ALUs of the CPU, and the control structure determines the order of these operations. The second function of this structure deals with memory latency [55]. When the process occupying the processor needs a variable stored in the memory to go on with the calculations, there are two possibilities: the variable is in the cache (fast access) and the calculation is not slowed, or the variable is in the central memory, with a slow access. The second case is called a "cache miss", and the time lost to retrieve the variable is a time lost for the

whole processor. In order to use this time better, the connection prediction makes an assumption on the value of the variable and goes on with the calculation. When the memory access is finished, the assumed value is compared to the real value: in some cases it is identical, and the processor has not lost any time. Otherwise the time is lost anyway. This method is particularly efficient on conditional loops such as:

```
if (x>0) then
     a=b*x;
else
     a=-b*x;
end
```

The processor starts one of the two calculations and has (on average) a 50% chance to keep its calculations at the end of the memory access.
On a GPU, roughly 90% of the transistors are dedicated to the calculation units, giving it a raw calculation power extremely high. These calculation units are individually simpler (and thus less efficient) than the ones of a CPU, but their huge number greatly compensate this weakness. The GPU's calculation units are called Streaming Processor (SP), and are grouped at different scale. Eight Streaming Processor form a Streaming Multiprocessor (SM). Three Streaming Multiprocessor form a Thread Processing Cluster (TPC). All the SP of one Streaming Multiprocessor execute the same task called thread on different data. Each SM has a shared memory accessible by the eight SPs.
Cumulated, the cache memory of the GPU is smaller than the cache memory of the CPU. This does not create too many problems, the cache requirements of a GPU being lower than the ones of a CPU. Finally there is one control structure per SM. These structures are very different from the ones observed on CPU because of specific constraints. There is no prediction mechanism on the GPU because there exists a more efficient solution. In the right configuration, the GPU has more threads to compute than it can run simultaneously. Therefore, when a thread being computed needs a value from the DRAM for a variable, it is put aside and a thread in a waiting queue gets access to the GPU and starts (or resumes) its calculation. This process exchange is called a content change. When the first thread receives the value for its variable in the GPU cache, it resumes its calculation. Context changes are extremely fast on GPU, and very slow on CPU, which explains why this solution is not used with CPUs. Therefore, it is essential to occupy the GPU with a great number of simultaneous tasks to allow it context changes as often as it needs.
Since a CPU sequentially processes complex tasks, it needs complex control structures with an important number of transistors. The cache must be large enough to ensure that the majority of the variables necessary for the calculations can be included in it. Since a GPU processes groups of simple identical tasks, its control structures have a small size and small caches are sufficient.

From a scheduling point of view, this indicates that the type of a task influences the values of its processing times on CPU and on GPU: if it requires a lot of data, its

processing time on GPU will not be much better than its processing time on CPU, since any computational time gained will be hindered by the time needed to fetch the data required for the computations, too big to fit on the small GPU cache. If the calculation time is much smaller than the time for data transfer, the full execution can actually take longer on GPU than on CPU. The calculations that are good candidates for an execution on GPU are complex calculations on a small data volume.

Example 2.2.1. Calculating the sum of two diagonal square matrix of size n.
The time complexity of the calculation is in $\mathcal{O}(n)$, when the sizes of the entry data and exit data to copy vary in $\mathcal{O}(n)$.

Example 2.2.2. Inverting a square matrix of size n.
The size of the data to copy varies in $\mathcal{O}(n^2)$, and the time complexity of the calculations varies in $\mathcal{O}(n^3)$.

Example 2.2.2 seems to be a better candidate for GPU execution. Indeed, for a value of n large enough, the time of data transfer becomes negligible compared to the calculation time.
The different architectures of CPUs and GPUs leads to different memory management mechanisms that influence the processing times of a task on CPU and GPU, depending on the type of the task to compute.

2.2.2 GPU Programming

The different memory management mechanisms on CPU and GPU discussed in the previous section have an impact on the programming of GPUs. Indeed, the first phase of a calculation on a GPU must be the copy of the entry variables from the CPU memory to the GPU memory, and the last phase is always the copy of the exit variables from the GPU memory to the CPU memory. Let us take an example to visualize the different steps in a GPU calculation.

Example 2.2.3. Vector addition element by element
Compute $Y = \alpha + X$, Y and X being two vectors of 1024 float.

The program allocate memory on the CPU (`input`, `output`) and on the GPU (`input_ gpu`,`output_gpu`)

```
input = OpenCL::VArray::new(FLOAT, 1024)
output = OpenCL::VArray::new(FLOAT, 1024)
input_gpu = create_buffer(1024*4)
output_gpu = create_buffer(1024*4)
```

The command to copy the input buffer from the CPU memory to the GPU memory is the following one in the programming language OpenCL

```
enqueue_write_buffer(1024*4, input, input_gpu)
```

and the following command is for copying the output buffer from the GPU memory to the CPU memory

```
enqueue_read_buffer(1024*4, output_gpu, output)
```

As we have seen in the previous section, these memory transfers have a non negligible impact on the total execution time on GPU and must be done carefully, in order to keep these transfer times minimal compared to the computation time of a task.

The other phase in a GPU calculation is the calculation itself. In order to get a good acceleration on the processing time of a task when compared to its CPU processing time, the calculations executed on GPUs must also be programmed with a lot of parallelization in their code, and therefore they have to be parallelizable. Matrix calculations are good candidates with respect to this criterion: there can be as many threads on the GPU as there are matrix coefficients. Each SP takes care of one coefficient of the calculation. It is up to the programmer to specify the number of threads he wants to execute, as well as their organization. On GPU, the parallel routines are called kernels: the threads of one kernel execute the same code on different data. The code for the calculation on the GPU corresponding to Example 2.2.3 is the following in OpenCL:

```
prog = create_program([<<EOF
__kernel void addition( float alpha,
                        __global const float *x,
                        __global float *y) {
  size_t ig = get_global_id(0);
  y[ig] = alpha + x[ig];
}
EOF
])
create_kernel("addition",prog)
```

In a kernel, threads are organized in blocks: a block can have one, two or three dimensions, depending on the programmer's choice and the material constraints. The blocks themselves are organized into a grid of blocks. Similarly, the grid can have one, two or three dimensions. Each thread has access to variables specifying its position in the grid and in the corresponding block. Therefore, a thread in a typical kernel working on matrix starts by using these variables to define a couple of indexes (i, j) that are specific to this thread. As a result, the thread works on index (i, j) of the matrix. This corresponds to the following command in OpenCL for Example 2.2.3, that computes the kernel with the arguments and vector of size $1024 = 16 \times 64$ float split into a grid of 16 blocs, each block containing 64 threads:

```
args= set_args([OpenCL::Float::new(5.0),
                input_gpu, output_gpu])
enqueue_NDrange_kernel(prog, args, [1024], [64])
```

Since threads share the same global memory, it is necessary to prevent different threads from writing in the same memory space at the same time. Loading threads regularly is also very important. GPU executes threads in groups (or *warps*), and the processing of one group is finished when all the threads of the group are finished. It is therefore essential to avoid conditional loops that disturb load balance.
These programming difficulties have to be considered by the programmer and are in no way handled by the scheduler of a computing platform, but it affects the processing times the tasks of the programmer will have on GPU, and therefore are another reason why the processing time of a task on GPU can be very variable and may not be determined with an acceleration rule corresponding to its type or its degree of parallelization. Depending of the skills of the programmer and the hardware specifications of the platform GPUs, this degree of parallelization may not be exploited to its full potential. However, it is commonly admitted that an accurate estimation of the processing times of tasks can be obtained at compile time for regular numerical applications in HPC. Therefore, from a scheduling point of view, this aspect only add to the arbitrariness of the ratio of the processing times of tasks on CPU and on GPU, with no impact on the knowledge of these processing times.

One last characteristic of the GPU to observe: the architecture of GPUs prevents them from preempting a task while it is executed on a GPU. A GPU computation is unstoppable, and has to run its course until the end of the execution. It cannot even be canceled during the processing. This means that the scheduling problems we study have no preemption of the tasks allowed on the GPUs, not even the cancellation of the tasks being allowed on GPU.

2.3 Summary

Computing platforms have reached high levels of computational power over the years, opening new fields of interest for High Performance Computing, ranging from economy with finance computations to scientific research with nuclear physics, fluids mechanics or DNA sequencing... The overall complexity has grown with their ability to process and transfer massive amounts of data in a very short amount of time, and new techniques and processors have been developed to create these new platforms, resulting in an often heterogeneous distribution of processors within these platforms.
One type of accelerating processors used on these platforms is the GPUs, that are able to perform simple parallel computation at a very high speed, since it is what they were designed to do in their primary use, graphical processing. However, this primary purpose of the GPUs means they have an architecture that greatly differs from a common CPU architecture, creating specific characteristics that alter the processing time of a task on a GPU. The two main differences are memory management, and the parallelization of the operations of a task.
Since these difference are based on to the GPU hardware and the user's programming, when given an arbitrary set of tasks to schedule, we have to assume that the processing

times of a task on CPU and GPU cannot be linked by any rule, and therefore have to be completely arbitrary in the generic case. If the tasks to be scheduled however share the same memory characteristics and have the same parallelization potential, we can assume that all the tasks will either be accelerated when assigned to a GPU, or slowed down. Another scheduling constraint to add to our model is that the tasks assigned to a GPU cannot be preempted or canceled.
With these scheduling parameters in mind, we can define the problems we studied during the course of this PhD thesis and present new notations for this new type of scheduling problems, as well as the methods from related scheduling problems that we used during this PhD thesis to tackle these problems.

Chapter 3

New Notations and Related Works on GPU Scheduling Algorithms

New computing platforms are composed of various processors, including standard processors, CPUs, but also accelerators like GPUs. Scheduling the calculations submitted by the platform users is a crucial problem in term of efficiency for a field where performance is key. These heterogeneous processors make the scheduling problem on these platforms at least atypical and very hard in terms of known problems, especially since there was no theoretical method prior to this work to deal with this particular scheduling problem. The closest problem in the classical literature would be the problem of scheduling tasks on unrelated processors, but it is far too generic for our problem, with only two types of unrelated processors.
The classical nomenclature for scheduling problems does not have a notation adapted to the problem of scheduling tasks on a heterogeneous platform composed of CPUs and GPUs. We extend here the traditional notation $\alpha \mid \beta \mid \gamma$ introduced by Graham et al. [39] to fit our new scheduling problems, and then cover the related scheduling problems we have used during this work to establish and study a new adequate class of scheduling model.

3.1 Notations

In this work, only deterministic scheduling problems are considered, meaning that the number of tasks, the number of parallel processors, and all task characteristics (like processing times) of the problems are known in advance.
Each field of the classical *three field* notation $\alpha \mid \beta \mid \gamma$ [39] represents a particular characteristic of a scheduling problem, where

- α represents the resources of the problem, i.e. the available machines, or in our case, the number of CPUs available and the number of GPUs available. In the classical notation, when the machines are identical, $\alpha = P$, when the machines are

uniformly related i.e. when the machines have different speeds, $\alpha = Q$, and when the machines are unrelated, $\alpha = R$.

- β represents the hypothesis on the tasks and the constraints imposed on the tasks. In our case, we assume that all processing times are positive integers.

- γ represents the objective to minimize or maximize. In HPC, the favored objective is the minimization of the *makespan*, C_{max}, i.e. the maximum completion time over all tasks. Indeed, when dealing with parallel processors, the makespan becomes an objective of significant interest. In practice, one often has to deal with the problem of balancing the load on processors in parallel and by minimizing the makespan the scheduler ensures a good balance of the load.

Now we present the extensions we introduced in this notation in order to characterize our scheduling problem, starting with the α field.

3.1.1 Machines (α)

3.1.1.1 Sets of Identical CPUs and Identical GPUs

We denote by (Pm, Pk) the problem of scheduling a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of n tasks on a heterogeneous computing platform constituted of m identical CPUs (Pm) and k identical GPUs (Pk), where a task T_j has two distinct processing times, $\overline{p_j}$ if it is executed on a CPU and $\underline{p_j}$ if it is processed on a GPU. The m CPUs are considered independent from the GPUs that are commanded by some extra driving CPUs, not mentioned here because they do not execute any task. Since the CPUs (resp. GPUs) are all identical, there is no need for a more complex notation involving the number of the CPU (resp. GPU) where one task is processed. The default hypothesis is that the acceleration factor $\frac{\overline{p_j}}{\underline{p_j}} = q_j$ of the different tasks is arbitrary. Tasks with a great degree of parallelism can have their processing times greatly reduced when assigned to a GPU, while some other tasks may have similar processing times on CPU and on GPU, or some might even be slowed down when assigned to a GPU. We assume that both processing times of a task are known in advance as it is commonly admitted. As we previously mentioned, an accurate estimation can be obtained at compile time for regular numerical applications in HPC.
For instance the problem $(Pm, Pk) \mid\mid C_{max}$ will denote the problem of scheduling n independent sequential tasks (i.e. they are only executed on one processor) on m CPUs and k GPUs where the objective is to minimize the *makespan*,
$C_{max} = \max\left(C_{max}^{CPU}, C_{max}^{GPU}\right)$ (see Figure 3.1). Other classical objectives found in the literature can also be integrated in this notation, as for example the sum of completion times, $\sum C_j$.
The notation (P, P) is used when the numbers of CPUs and GPUs are arbitrary, but all the CPUs are still considered identical as well as the GPUs.

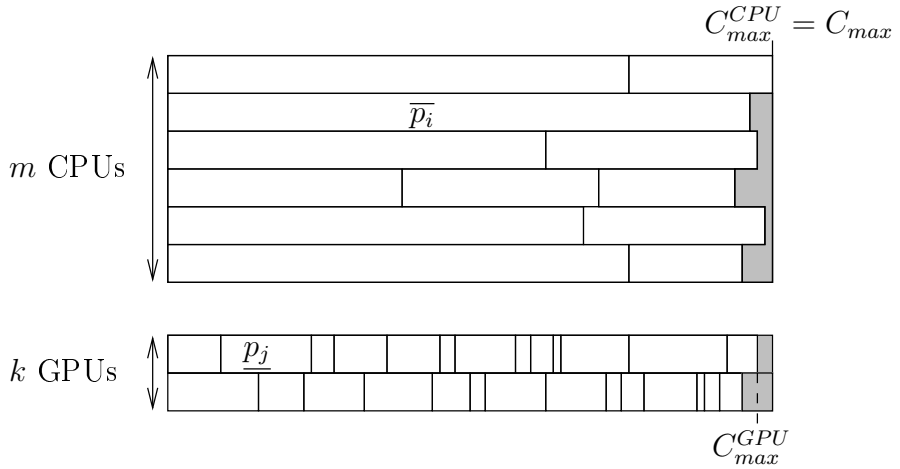


Figure 3.1: An example with $m = 6$ CPUs and $k = 2$ GPUs.

3.1.1.2 Sets of Uniform CPUs and Uniform GPUs

With the same reasoning as for identical processors, we denote by (Qm, Qk) the problem with n independent sequential tasks on a platform with m uniform CPUs (Qm) and k uniform GPUs (Qk). In this case, a task can have several distinct processing times. We denote by $\overline{p_j}$ the processing time of task j on the slowest CPU, taken as the reference CPU. From there, the processing time of task j on CPU i is defined by $\overline{p_{ij}} = \frac{\overline{p_j}}{\overline{s_i}}$, where $\overline{s_i}$ is the speedup factor of CPU i compared to the slowest CPU, whose speedup is 1, as described for classical scheduling problems with uniform machines.
We introduce the same processing times for the GPUs, where $\underline{p_j}$ denotes the processing time of a task j on the slowest GPU. The processing time of task j on GPU i is then defined by $\underline{p_{ij}} = \frac{\underline{p_j}}{\underline{s_i}}$, where $\underline{s_i}$ is the speedup factor of GPU i compared to the slowest GPU, whose speedup is 1.
Using this notation, we define similarly the acceleration ratio of a task from its parallelization on a GPU with the processing times on the reference processors: $q_j = \frac{\overline{p_j}}{\underline{p_j}}$.
Once again, the default hypothesis is that all the acceleration ratios of the different tasks can be arbitrary. The parallelization process allowing much greater acceleration that any increase in computing speed, it is assumed that even the largest speedup factor $\overline{s_i}$ among the CPUs is lower than the smallest acceleration factor q_j for a task j on the reference GPU.
Again, the notation (Q, Q) is used when the numbers of CPUs and GPUs are arbitrary, as for instance in the problem of minimizing the makespan: $(Q, Q) \mid\mid C_{max}$, but other objectives than the makespan could also be considered for this problem.

This new notation allows us to consider all the combinations for the sets of CPUs and GPUs: we could for instance study the problem $(P2, Q2)$ corresponding to a simple laptop with 2 CPU cores and its built-in GPU on which another, different, GPU has

been plugged for graphical purposes.

3.1.1.3 Unrelated CPUs and unrelated GPUs

Extending the previous notation to unrelated sets of CPUs and GPUs would bring no additional material to the notation of $\alpha = R$, the processing times being completely arbitrary from one task and one machine to another.

3.1.2 Tasks (β)

In the generic case, the tasks can be independent or linked by some precedence constraints, they can be considered either sequential (i.e. they are only executed on only one processor), or malleable (they can be executed on several processors and their processing time depends on the number of processors they are assigned to).

3.1.2.1 One type of tasks

As mentioned in the previous section, the default hypothesis in the new notation is that the acceleration factors $\frac{\overline{p_j}}{\underline{p_j}} = q_j$ for the different tasks can be arbitrary. A restricted version of this hypothesis can be made in order to consider the problems dealing with the scheduling of only one type of tasks, i.e. all the considered tasks would have the same acceleration factor: $\frac{\overline{p_j}}{\underline{p_j}} = q$ for $j = 1, \ldots, n$.
For instance, the problem $(Pm, Pk) \mid\mid C_{max}$ with only one type of tasks will be denoted by $(Pm, Pk) \mid q_j = q \mid C_{max}$ in the same way as equal processing times are denoted by $p_j = p$ in the β field of the classical notation. All other entries from the β field in the classical notation can be integrated in order to refine the problem, with the exception of the preemption which is detailed in the following section.

3.1.2.2 Partial Preemption

Due to the different architectures of the GPUs as well as the different programming languages, it is difficult and costly to start a task on a CPU, interrupt it and pick it up where it was stopped on a GPU: complete preemption cannot be allowed between a CPU and a GPU. The GPU peculiar structure requires complex management of the preemption even between the GPUs themselves [6].
We introduce the notion of "partial preemption", denoted by *ppmtn*, where preemption is only allowed for tasks remaining on the CPUs. For the rest of the manuscript, we will suppose that preemption is not allowed between GPUs, or between a CPU and a GPU. The notion may evolve in the next few years with new accelerator architectures as the Intel MIC (Many Integrated Core) architecture of the Xeon Phi, which is roughly a "standard" 60 core disk-less system. Preemption inside a MIC should be much easier. Nevertheless, efficient task migration between the CPU and the MIC remains an open problem.

With these notations, Table 3.1 summarizes the new scheduling problems we studied as well as the performance of the corresponding algorithms we developed during the course of this PhD, including several algorithms for the case with independent tasks, the specific case where all the tasks to be scheduled are accelerated when assigned to a GPU, but not necessarily with the same acceleration factor for all the tasks, the case where preemptions are allowed for the tasks assigned to the CPUs, but not for the tasks on the GPUs, the case where the tasks are considered malleable when they are assigned to a CPU and sequential when assigned to a GPU, and the case where the tasks are linked by precedence relations.

Problem	Approximation ratio achieved	Section
$(P1, P1) \mid\mid C_{max}$	$\frac{3}{2}$	4.1.3
	$1+\epsilon$	
$(Pm, Pk) \mid\mid C_{max}$	2	4.2.3
	$\frac{4}{3}+\frac{1}{3k}$	4.3, 4.4
	$\frac{2r+1}{2r}+\frac{1}{2rk},\ r>0$	5
	$\frac{2(r+1)}{2r+1}+\frac{1}{(2r+1)k},\ r\geqslant 0$	
$(Pm, Pk) \mid q_j \geqslant 1 \mid C_{max}$	$\frac{3}{2}$	6.1
$(Pm, P1) \mid ppmtn \mid C_{max}$	$1+\frac{1}{m}$	6.2.1
$(Pm, P1) \mid q_j = q,\ ppmtn \mid C_{max}$	$1+\frac{1}{q}$	
$(Pm, Pk) \mid ppmtn \mid C_{max}$	$1+\max\left(\frac{1}{m}, 1-\frac{1}{k}\right)$	6.2.2
	$1+\max\left(\frac{1}{m}, \frac{1}{2r}+\frac{1}{2rk}\right),\ r>0$	
	$1+\max\left(\frac{1}{m}, \frac{1}{2r+1}+\frac{1}{(2r+1)k}\right),\ r\geqslant 0$	
$(Pm, Pk) \mid mall \mid C_{max}$	$\frac{3}{2}$	6.3
$(Pm, Pk) \mid prec \mid C_{max}$	6	8

Table 3.1: Problems with no equivalent counterpart in the literature studied in this work.

Table 3.2 shows new scheduling problems that we linked to existing scheduling problems, that are presented in the following section.

Problem	Corresponding Problem	Section
$(Pm, Pk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$	$Q \mid p_j = 1 \mid C_{max}$	3.2.1.2
$(Qm, Qk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$		
$(Pm, Pk) \mid q_j = q \mid C_{max}$	$Q \mid\mid C_{max}$	3.2.2.1
$(Qm, Qk) \mid q_j = q \mid C_{max}$		
$(Pm, Pk) \mid\mid \sum C_j$	$R \mid\mid \sum C_j$	3.2.1.2
$(Pm, Pk) \mid ppmtn \mid \sum C_j$		

Table 3.2: Problems related to classical scheduling problems.

3.2 Related Work on Scheduling Independent Sequential Tasks

In this chapter we present the classical methods [85] used to solve scheduling problems on parallel processors with independent sequential tasks, and the best approximation results obtained with for classical problems so far. These methods were used during the course of this PhD to study the first new problems of scheduling on CPUs and GPUs. Problems with malleable tasks or with dependent tasks were also studied during the course of this PhD. The corresponding related works in the literature are presented at the beginning of the corresponding chapters (see Table 3.1).

Some specific problems with CPUs and GPUs can be directly linked to classical problems in the literature, presented in Table 3.2. In fact, if we considered an instance where all the tasks have the same ratio when comparing their processing times on CPU and on GPU, our problem would reduce to a uniform machine problem. The methods used in the literature to solve the corresponding classical problems are presented in this chapter in the specific cases of heterogeneous scheduling. We start with exact methods that solve entirely the problems they deal with.

3.2.1 Exact Methods

Exact methods cannot be used to solve directly problem $(Pm, Pk) \mid\mid C_{max}$ in polynomial time, but the techniques we present below are used in the design of the approximation algorithms developed in this work.

3.2.1.1 Linear Programming

Linear programming [88] (LP) is a method to solve optimization problems that have only *linear* constraints of equality and inequality and a *linear* objective function. Its feasible region is a convex polyhedron, which is a set defined as the intersection of finitely many half spaces, each of which is defined by a linear inequality. Its objective function is a real-valued affine function defined on this polyhedron. A linear programming algorithm finds a point in the polyhedron where this function has the smallest (resp. largest) value if such a point exists in the case where we aim at minimizing (resp. maximizing) the objective function.

Scheduling problems where preemption is allowed can typically be solved by linear programming. The problem $P \mid pmtn \mid C_{max}$ can be formulated as the following linear

program:

$$
(LP) \qquad
\begin{aligned}
&\min C_{max} && \\
&\text{s.t.} \sum_{i=1}^{m} x_{ij} = 1, && j = 1, \ldots, n \\
&\sum_{i=1}^{m} x_{ij} p_j \leqslant C_{max}, && j = 1, \ldots, n \\
&\sum_{j=1}^{n} x_{ij} p_j \leqslant C_{max}, && i = 1, \ldots, m \\
&0 \leqslant x_{ij} \leqslant 1, && i = 1, \ldots, m,\ j = 1, \ldots, n
\end{aligned}
$$

where p_j represents the processing time of a task T_j, x_{ij} is a variable in the interval $[0, 1]$ that represents the portion of task T_j that is processed on processor P_i, m is the number of processors and n the number of tasks. The objective function here corresponds to the makespan of the schedule, and the constraints represents the facts that the totality of each task must processed by the m processors, that each task must be entirely processed before the end of the schedule and that the computational load on each processor must not be larger than the makespan.
This problem can be solved very efficiently. The length of a preemptive schedule cannot be smaller than the maximum of two values: the maximum processing time of a task and the mean processing requirement of a processor i.e.:

$$C^*_{max} = \max \left\{ \max_j \{p_j\}, \frac{1}{m} \sum_{j=1}^{n} p_j \right\}.$$

An algorithm given by McNaughton [65] constructs a schedule whose length is equal to C^*_{max} with a complexity of $\mathcal{O}(n)$. This is therefore a polynomially solvable problem. However, in practice, we cannot preempt at will the tasks of an instance. Every preemption made has a cost, for example in data transfer from one processor to another, and one cannot divide a task into an infinity of very small fractions of task. This suggests the introduction of a scheduling model where task preemptions are only allowed after the tasks have been processed continuously for some given amount g of time. The value for g (preemption granularity) should be chosen large enough so that the time delay and cost overheads connected with preemption are negligible. For given granularity g, upper bounds on the preemption overhead can easily be estimated since the number of preemptions for a task of processing time p is limited by $\left\lfloor \frac{p}{g} \right\rfloor$. In [25], the problem $P \mid pmtn \mid C_{max}$ with g-restricted preemption is discussed : if $p_j \leqslant g$, then preemption is not allowed, otherwise preemption may take place after the task has been continuously processed for at least g units of time. For the remaining part of a preempted task the same rule is applied. For 2 processors, both the g-preemptive and

the exact-g-preemptive (preemptions are only allowed every g units of time, or a multiple of g) scheduling problems can be solved in time $\mathcal{O}(n)$. For more than 2 processors, both problems are NP-hard.

Problems $Q \mid pmtn \mid C_{max}$ and $R \mid pmtn \mid C_{max}$ can also be solved in polynomial time using linear programming. However these problems cannot be linked directly to any CPU-GPU problem, since the architecture of the GPUs prevents any preemption of any task, as seen in the previous chapter.

In this thesis, linear programming is used in the resolution of problem $(Pm, Pk) \mid ppmtn \mid C_{max}$ (see Chapter 6, Section 6.2) as well as in part of the resolution of $(Pm, Pk) \mid prec \mid C_{max}$ (see Chapter 8).

3.2.1.2 Transportation Networks and Network Flow Algorithms

In graph theory, a transportation network is a directed graph where each arc has a capacity and each arc receives a flow. The amount of flow on an arc cannot exceed the capacity of the arc. A flow must satisfy the restriction that the amount of incoming flow into a node equals the amount of outgoing flow, unless it is a source, which has more outgoing flow, or sink, which has more incoming flow. A transportation network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes. Such a problem can be solved polynomially.

Some specific scheduling problems may be formulated as transportation networks problems by creating sources, sinks, capacities and arcs from the original problem parameters. A transportation network formulation has been presented for problem $Q \mid p_j = 1 \mid C_{max}$ in [39], which in turn can be used to formulate problem $(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$ as a transportation network problem as follows.

There are n sources $j = 1, \ldots, n$, each corresponding to a task T_j, and $(m+k)n$ sinks (i, v) for the processors with $i = 1, \ldots, m+k$, and the positions $v = 1, \ldots, n$ (see Equation (3.1)). A task is considered to be in the v^{th} position on a processor when it is the v^{th} task executed on that processor. The first m machines correspond to the CPUs and the last k ones to the GPUs. The cost of arc $(j, (i, v))$ is

$$c_{ijv} = \begin{cases} v & \text{if machine } i \text{ is a CPU (i.e. } i = 1, \ldots, m), \\ v/q & \text{if machine } i \text{ is a GPU (i.e. } i = m+1, \ldots, m+k). \end{cases}$$

The arc flow is

$$x_{ijv} = \begin{cases} 1 & \text{if task } T_j \text{ is executed on machine } i \text{ in the } v^{\text{th}} \text{ position} \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

The problem is to minimize $C_{max} = \max\limits_{i,j,v} \{c_{ijv} x_{ijv}\}$ subject to constraints

$$\begin{aligned} \sum_{i,v} x_{ijv} &= 1 && \forall j \\ \sum_{j} x_{ijv} &\leqslant 1 && \forall i, v \\ x_{ijv} &\geqslant 0 && \forall i, j, v \end{aligned}$$

This problem can be solved by a standard transportation procedure which results in $\mathcal{O}(n^3)$ time complexity.

The problem of minimizing the sum of completion times on unrelated processors, $R \,||\, \sum C_j$, is also polynomially solvable via a transportation problem formulation [15]. The problem of scheduling on m CPUs and k GPUs with minimum sum of completion times, $(Pm, Pk) \,||\, \sum C_j$, is a specific case of the classical problem $R \,||\, \sum C_j$. We can adapt an approach to the solution of $R \,||\, \sum C_j$ to $(Pm, Pk) \,||\, \sum C_j$: the method is based on the observation that task T_j processed on machine i in the last position contributes its processing time $p_{ij} = \begin{cases} \overline{p_j} & \text{if } i \in \{1, \ldots, m\} \\ \underline{p_j} & \text{if } i \in \{m+1, \ldots, m+k\} \end{cases}$ to the sum of the completion times $\sum C_j$ for problem $(Pm, Pk) \,||\, \sum C_j$. The same task processed in the last but one position on the same processor contributes $2p_{ij}$ to $\sum C_j$ and so on. This reasoning allows us to construct an $(2n) \times n$ matrix $\mathcal{Q}$ presenting the contributions of the tasks when they are processed in different positions on different processors to the value of $\sum C_j$:

$$\mathcal{Q} = \begin{pmatrix} \overline{p_1} & \ldots & \overline{p_j} & \ldots & \overline{p_n} \\ 2\overline{p_1} & \ldots & 2\overline{p_j} & \ldots & 2\overline{p_n} \\ \vdots & & \vdots & & \vdots \\ n\overline{p_1} & \ldots & n\overline{p_j} & \ldots & n\overline{p_n} \\ \underline{p_1} & \ldots & \underline{p_j} & \ldots & \underline{p_n} \\ 2\underline{p_1} & \ldots & 2\underline{p_j} & \ldots & 2\underline{p_n} \\ \vdots & & \vdots & & \vdots \\ n\underline{p_1} & \ldots & n\underline{p_j} & \ldots & n\underline{p_n} \end{pmatrix}$$

The problem is now to carefully choose n elements from matrix $\mathcal{Q}$ in order to minimize

$$\sum_{j=1}^{n} \sum_{v=1}^{n} \left(\sum_{i=1}^{m} Q_{v,j} + \sum_{i=m+1}^{m+k} Q_{v+m,j} \right) x_{ijv}$$

under the constraints

$$\sum_{i=1}^{m+k}\sum_{v=1}^{n} x_{ijv} = 1 \qquad \forall j \in \{1,\dots,n\}$$

$$\sum_{j=1}^{n} x_{ijv} \leqslant 1 \qquad \forall i \in \{1,\dots,m+k\},\ v \in \{1,\dots,n\}$$

where

$$x_{ijv} = \begin{cases} 1 & \text{if } T_j \text{ is put on } i \text{ in the } v^{\text{th}} \text{ position, starting counting from the end,} \\ 0 & \text{otherwise.} \end{cases}$$

The problem is a transportation problem solved using classical transportation algorithms, in $\mathcal{O}(n^3)$ [15].

3.2.1.3 Dynamic programming

Dynamic programming [88] is a method for solving complex problems by breaking them down into simpler subproblems. The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often when using a more naive method, many of the subproblems are generated and solved many times. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or memorized: the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.
An example of problem solved using dynamic programming is the knapsack problem [64]. The knapsack problem is to determine, given a set of items, each with a mass and a value, the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.
This knapsack problem and its dynamic programming solving method will be used in the following chapter, Section 4.1.3, to develop an approximation algorithm for problem $(P1, P1) \mid\mid C_{max}$ and then problem $(Pm, Pk) \mid\mid C_{max}$ in the following sections.

These are some scheduling problems that are polynomially solvable with exact methods. However, when the problems become more complex, these methods do not work, and there is a need for the use of approximation algorithms.

3.2.2 Approximation Methods

Non-preemptive parallel scheduling problems with minimal makespan tend to be difficult to solve. The vast majority of them are NP-hard already for the case with a fixed number of processors. Even the scheduling problem with two identical processors $P2 \mid\mid C_{max}$ is already is NP-hard in the ordinary sense since PARTITION [33] polynomially reduces to it. Here, each processor represents a set of a partition and the tasks are the items which we want to divide evenly into these two partitions. Thus, it is unlikely (unless $P = NP$) that there exists a polynomial-time algorithm for computing a minimal makespan for a scheduling problem on hybrid platforms.
A standard way of dealing with NP-hard problems is not to search for an optimal solution, but to search for near-optimal solutions. An algorithm that returns near-optimal solutions is called an approximation algorithm [42]. If it runs in polynomial time, then it is called a polynomial time approximation algorithm.
We aim at developing approximation algorithms whose schedules are relatively close to the optimal schedule while remaining practical, i.e. with a reasonable time complexity making them good candidates for an integration on a real computing platform. To characterize the proximity of the solutions delivered by an approximation algorithm to the corresponding optimal solutions, we determine the approximation ratio of said algorithm.

Definition 3.2.1. The *approximation ratio* ρ_A, or performance guarantee of an approximation 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.

3.2.2.1 List Scheduling

The original *list scheduling* (LIST) algorithm was developed by Graham [37] in 1969 for solving problem $P \mid\mid C_{max}$. It is based on a list of tasks ready to be executed in an arbitrary order: the algorithm assigns the first task on the list when a processor becomes free.
This algorithm is not optimal but it achieves the following approximation ratio:

Proposition 3.2.2. *The worst-case performance guarantee of the LIST algorithm is:*

$$\frac{C_{max}(LIST)}{C^*_{max}} \leqslant 2 - \frac{1}{m}$$

The proof of this result is simple, but quite important since a lot of results are proven with arguments similar to the ones used in this proof (see the proof of Algorithm 4.2.3 in Chapter 4, Section 4.2.3). If we note $p_1, \ldots, p_n$ the respective processing times of the n tasks, the proof uses the notable inequality [41]:

$$\frac{C_{max}(LIST)}{C^*_{max}} \leqslant 1 + (m-1)\frac{\max\limits_j p_j}{\sum\limits_{j=1}^{n} p_j},$$

and the classical lower bounds on the optimal makespan of $P \mid\mid C_{max}$

$$C^*_{max} \geqslant \max_j p_j \quad \text{and} \quad C^*_{max} \geqslant \frac{\sum_{j=1}^{n} p_j}{m}.$$

The list principle guarantees that the idle times on the processors are regrouped at the end of the schedule, the last processor to finish its task execution determining the makespan of the schedule. This observation allowed Graham in [38] to reduce the approximation ratio of his algorithm for $P \mid\mid C_{max}$ with the assignment of the smallest tasks at the end of the schedule where they can be used to balance the loads. The new scheduling algorithm is said to be using the *longest processing time* first (LPT) rule. The algorithm assigns at time $t = 0$ the m largest tasks to the m processors. After that, whenever a processor is freed, the largest unscheduled task is put onto the processor. If the tasks are selected in the LPT order the approximation ratio of the list scheduling algorithm for problem $P \mid\mid C_{max}$ can be considerably improved:

Proposition 3.2.3. *The LPT scheduling algorithm has a performance guarantee of*

$$\frac{C_{max}(LPT)}{C^*_{max}} \leqslant \frac{4}{3} - \frac{1}{3m},$$

This new bound is tight, meaning that we can found an instance of $P \mid\mid C_{max}$ whose schedule constructed via the LPT algorithm has a makespan $\frac{4}{3}$ times greater than its optimal makespan.

Remark 3.2.4. We can note that if we order the tasks according to the LPT rule, we have a final schedule in two parts (see Figure 3.2): the first part has a number of idle processors lower than $\frac{m}{2}$, and the second part has a number of idle processors greater than $\frac{m}{2}$.

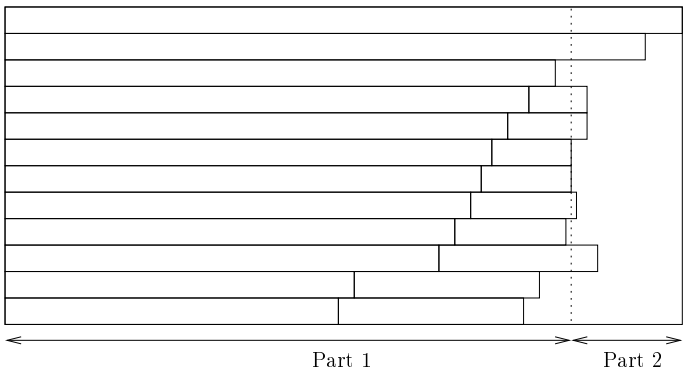


Figure 3.2: Schedule resulting from an LPT algorithm.

This remark will be used in the design of the algorithm of Chapter 4, Section 4.3.

Other problems than $P \mid\mid C_{max}$ have list algorithms to approximate them. One of these problems is the scheduling problem on uniformly related (or just related, for short) processors, $Q \mid\mid C_{max}$. Here, we are given a set of n independent tasks with sizes p_j that are to be executed on m non-identical processors. These processors run at different speeds v_i. More precisely, if task T_j is processed on processor P_i, it takes time p_j/v_i to be completed.
Graham [37, 38] generalized his LPT scheduling policy to make it applicable for the $Q \mid\mid C_{max}$ problem. This natural extension works as follows. It assigns each task, in order of non-increasing size p_j, to a processor on which it will be completed soonest, i.e., it assigns task T_j to processor i for which $\delta_i + p_j/v_i$ is minimized. Here, δ_i is the load on processor i just before the assignment of task T_j.
For the general case Gonzales et al. [35] showed

$$\frac{C_{max}(LPT)}{C^*_{max}} \leqslant 2 - \frac{2}{m+1}.$$

Additionally, they gave examples for which $C_{max}(LPT)/C^*_{max}$ approaches $\frac{3}{2}$ as m tends to infinity.

A specific version of the heterogeneous problem $(Pm, Pk) \mid\mid C_{max}$, where all the tasks have the same behavior on the GPUs, i.e. $\frac{\overline{p_j}}{\underline{p_j}} = q$ constant, denoted by $(Pm, Pk) \mid q_j = q \mid C_{max}$, could be assimilated to $Q \mid\mid C_{max}$: the m CPUs would have a speed $v_c = 1$ and the GPUs a speed $v_G = q$, for all the tasks. We can use the generalization of the LPT rule presented above. The algorithm would assign each task, in the order of longest processing time, to the processor (GPU or CPU) it will be completed soonest. The ratio is then $2 - \frac{2}{m+k+1}$.

For two uniform processors, i.e. $Q2 \mid\mid C_{max}$, Gonzales et al. showed that for any speed ratio $q \geqslant 1$, the approximation factor of the LPT algorithm is at most $\frac{1}{4}(1 + \sqrt{17}) \approx 1.28$. Here, q is the ratio between the speed of the faster processor and the speed of the slower processor. Recently, this case was investigated by Epstein and Favrholdt [26]. They gave the exact approximation factor of LPT in function of speed ratio q.
For a general setting of m uniform processors, Friesen [31] proved that the approximation factor of the LPT scheduling policy satisfies

$$1.52 \leqslant \frac{C_{max}(LPT)}{C^*_{max}} \leqslant \frac{5}{3}.$$

Another list scheduling algorithm has been presented in [60] for the specific case of $Q \mid\mid C_{max}$ with $m+1$ processors and where the first m processors have a processing speed factor equal to 1 and the remaining processor has a processing speed factor of q. The problem is denoted $Q(m+1) \mid\mid C_{max}$ and the list algorithm is as follows: the tasks

are ordered on the list in the non-increasing order of their longest processing times and processors are ordered in the non-increasing order of their processing speeds. Whenever a processor becomes free, it gets the first non-assigned task of the list. If there are two or more free processors, the fastest is chosen.

Proposition 3.2.5. *This list scheduling algorithm has a performance ratio of*

$$\frac{C_{max}(LIST)}{C_{max}*} \leqslant \begin{cases} \frac{2(m+q)}{q+2} & for\ q \leqslant 2 \\ \frac{m+q}{2} & for\ q > 2. \end{cases}$$

This problem can be also interpreted as the specific problem of scheduling tasks on m CPUs and one GPU, where all the tasks have the same acceleration when affected to the GPU, $(Pm, P1) \mid q_j = q \mid C_{max}$. The performance ratio of the ratio remains unchanged, q being the speedup of the GPU.

Remark 3.2.6. We can note that the problems $(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$ and $(Pm, Pk) \mid q_j = q \mid C_{max}$ described in the previous sections were particular cases of the classical problems $Q \mid p_j = 1 \mid C_{max}$ and $Q \mid\mid C_{max}$. We can show in a similar manner that problems $(Qm, Qk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$ and $(Qm, Qk) \mid q_j = q \mid C_{max}$ are also specific cases of $Q \mid p_j = 1 \mid C_{max}$ and $Q \mid\mid C_{max}$, respectively. The proofs would be similar and the methods used to solve these problems would remain unchanged.

Another scheduling problem with list scheduling algorithms is the problem presented by Imreh in [48], consisting in scheduling n sequential tasks on two sets of identical machines with minimum makespan. This scheduling problem corresponds exactly to $(Pm, Pk) \mid\mid C_{max}$, the first set CPU being the m CPUs, the second GPU corresponding to the k GPUs. The two sets of processors are identified as CPU and GPU in the following presentation of the associated list scheduling algorithms. We assume here that $k \leqslant m$.
The first list scheduling algorithm denoted LG for this problem is as follows:

- We first preassign the n tasks: they are divided between the sets CPU and GPU using the following rule:

  Task T_j it is assigned to GPU if $\frac{p_j}{k} \leqslant \frac{\overline{p_j}}{m}$, otherwise it is assigned to CPU.

- For each set we assume that we have an arbitrary ordered list LIST of all tasks. We then assign the tasks according to the order of LIST to the first processor available in the considered set.

Proposition 3.2.7. *Algorithm (LG) has a performance guarantee of*

$$\frac{C_{max}(LG)}{C^*_{max}} \leqslant 2 + \frac{m-1}{k}.$$

Remark 3.2.8. When, after preassigning the tasks to the sets CPU and GPU in LG, the tasks in each set could be ordered according to the LPT rule rather than choosing an arbitrary list, creating a variant of the LG algorithm, LG_{LPT}. We have then the following result for the problem $P \mid\mid C_{max}$ with f tasks: $\frac{C_{max}(LPT)}{C^*_{max}(P||C_{max})} \leqslant \frac{4}{3} - \frac{1}{3f}$.
This result can be applied to the set of tasks to be scheduled on the CPUs as well as to the set of tasks assigned to the GPUs by LG_{LPT}. If we denote by $C_{max}^{GPU^*_{LG}}(Pk \mid\mid C_{max})$ (resp. $C_{max}^{CPU^*_{LG}}(Pm \mid\mid C_{max})$) the optimal makespan for the instance of $Pk \mid\mid C_{max}$ (resp. $Pm \mid\mid C_{max}$) constituted of the tasks to be scheduled on the GPUs (resp. CPUs) by LG_{LPT}, and by C^*_{max} the optimal makespan for the corresponding instance of problem $(Pm, Pk) \mid\mid C_{max}$, we obtain:

$$\frac{C_{max}(LG_{LPT})}{C^*_{max}} \leqslant$$
$$\max\left(\left(\frac{4}{3} - \frac{1}{3k}\right)\frac{C_{max}^{GPU^*_{LG}}(Pk \mid\mid C_{max})}{C^*_{max}}, \left(\frac{4}{3} - \frac{1}{3m}\right)\frac{C_{max}^{CPU^*_{LG}}(Pm \mid\mid C_{max})}{C^*_{max}}\right).$$

A way to link the optimal makespan of the problems with identical processors to the optimal makespan of the problem with two sets could greatly improve the performance ratio of the modified LG algorithm. However, no result has been obtained on this subject.

Remark 3.2.9. If we suppose that $p_j = \alpha_j \overline{p_j} + \beta_j$, we can show that the list scheduling algorithm with a repartition rule of $\frac{\alpha_j \overline{p_j} + \beta_j}{k} \leqslant \frac{p_j}{m}$ achieves the same guarantee.

Another greedy algorithm presented by Imreh [48] has an approximation ratio of $4 - \frac{2}{m}$.
An online algorithm was designed specifically for a CPU-GPU cluster in [18], and it uses rules similar to the one from LG to schedule the tasks onto a CPU or a GPU. It approximation ratio is 4.
These algorithms are fast enough for being implemented in modern platforms, nevertheless the approximation ratios of these algorithms are quite high.

Remark 3.2.10. List scheduling algorithms are also employed to solve scheduling problems with other objectives than the makespan and can sometimes provided an exact resolution of a problem in polynomial time, for instance with the objective of the sum of the completion times of the tasks, $P \mid\mid \sum C_j$. The nature of criterion $\sum C_j$ is such that, in the case of one processor, assigning tasks in increasing order of their processing times minimizes the sum of the completion times. Conway et al. [22] showed that a generalization of this rule called *Shortest Processing Time first (SPT)* leads to an optimal list scheduling algorithm for problem $P \mid\mid \sum C_j$, with a time complexity of $\mathcal{O}(n \log n)$.
From the viewpoint of the value of the sum of the completion times, McNaughton [65] showed that preemptions are not profitable. Therefore, the SPT rule and the resulting list scheduling algorithm are also optimal for problem $P \mid pmtn \mid \sum C_j$.

If we now consider problem $(Pm, Pk) \mid ppmtn \mid \sum C_j$, where preemptions are only allowed on the CPUs that are considered identical, since the problem $P \mid pmtn \mid \sum C_j$ is polynomially solvable, the scheduling of the tasks of $(Pm, Pk) \mid ppmtn \mid \sum C_j$ can be done with the method used with the previous problem, $(Pm, Pk) \mid\mid \sum C_j$, solved in Section 3.2.1.2. Therefore, the problem remains easy to solve when partial preemptions are allowed.

Sometimes a greedy behavior such as the one of a list scheduling algorithm is not enough to approximate the solution of a problem to a satisfying degree. In those cases, one method employed in scheduling is the dual approximation technique.

3.2.2.2 Dual Approximation Technique

Definition 3.2.11. A g-dual approximation [43] algorithm for a generic problem takes a real number λ (guess) as an input, assumes that there exists a schedule of length at most λ and either delivers a schedule of makespan at most $g\lambda$, or answers correctly that there exists no schedule of length at most λ. A binary search is used to try different guesses to approach the optimal makespan as follows: we first take an initial lower bound B_{min} and an initial upper bound B_{max} of the optimal makespan. We start by solving the problem with a λ equal to the average of these two bounds, $\lambda = \frac{B_{max}+B_{min}}{2}$, and then the bounds are updated as follows:

- If the algorithm returns a schedule of makespan at most $g\lambda$, then there exists a schedule of makespan at most λ and λ becomes the new upper bound.
- If the algorithm cannot delivers a schedule of length at most $g\lambda$, then λ becomes the new lower bound and the guess is again updated accordingly.

The number of iterations of the binary search is bounded by $\log_2 (B_{max} - B_{min})$. Hence, a g-dual approximation algorithm can be converted, by bisection search, in a $g(1+\epsilon)$-approximation algorithm with a similar running time.

This dual approximation technique is first used in the following chapter, Section 4.2.3 to tackle the difficulty of having more than one CPU and one GPU. This method is the key to all the algorithms developed in this PhD. Without the guess of the dual approximation technique, it would be extremely hard to handle two sets of processors that process tasks in a completely different way.

There also exist more complex approximations algorithms with smaller approximation ratios in the literature. We give below a presentation of one of these types of algorithms, the polynomial time approximation scheme.

3.2.2.3 Polynomial Time Approximation Scheme

Definition 3.2.12. A family of $(1+\epsilon)$-approximation algorithms over all $\epsilon > 0$ with polynomial running times is called a Polynomial Time Approximation Scheme (PTAS). If the time complexity of a PTAS is also polynomially bounded in $1/\epsilon$, then it is called a Fully Polynomial Time Approximation Scheme (FPTAS).

With respect to relative performance guarantees, an FPTAS is essentially the strongest possible polynomial-time approximation result that we can derive for an NP-hard problem. The inconvenient of PTAS and FPTAS is that in order to achieve these levels of precision for the approximation ratio, the time complexity of the algorithms, although polynomial, is very high and renders them usually to time consuming to be implemented on real-time scheduling platforms, which is an objective of this PhD work.

For problem $P \mid\mid C_{max}$, Sahni [74] presented a family of approximation algorithms, where algorithm A_ϵ has a running time $\mathcal{O}(n(n^2/\epsilon)^{m-1})$, m being the number of machines, and an approximation ratio of

$$\frac{C_{max}(A_\epsilon)}{C^*_{max}} \leqslant 1+\epsilon.$$

When m is fixed, the family of algorithms A_ϵ becomes a PTAS. Later, Hochbaum and Shmoys [43] gave a better PTAS for $P \mid\mid C_{max}$ which runs in $\mathcal{O}((n/\epsilon)^{1/\epsilon})$ time, which unfortunately is still too high to be implemented in practice.

The first PTAS for $Q \mid\mid C_{max}$ was given by Hochbaum and Shmoys [44]. Since the problem is strongly NP-complete, their results are the best possible in the sense that if there were an FPTAS for this problem, then $P = NP$. Their approximation algorithm is based on a decision procedure which tests if there exists a schedule for a given problem instance where all tasks are completed by time C. Thus, the decision problem can be viewed as a bin-packing problem with variable bin sizes. The minimum value of C is computed by a simple binary search procedure. The overall running time of the algorithm is $\mathcal{O}\left(\left(\log m + \log\left(\frac{3}{\epsilon}\right)\right)\left(\frac{m}{\epsilon}\right)\left(\frac{n}{\epsilon}\right)\frac{1}{\epsilon}\right)$.
Since we saw that a specific version of our problem, $(Pm, Pk) \mid q_j = q \mid C_{max}$, where all the tasks have the same behavior on the GPUs could be assimilated to $Q \mid\mid C_{max}$, we can theoretically use the PTAS developed for $Q \mid\mid C_{max}$ for this specific case. However, the time complexity is prohibitive when it comes to practical matters.

For problem $R \mid\mid C_{max}$, Horowitz and Sahni [45] presented a non-polynomial-time dynamic programming algorithm to compute a schedule with minimum makespan. They gave also the first FPTAS to approximate an optimum schedule with minimum makespan for the case when the number of unrelated processors m is fixed. They proved that, for any $\epsilon > 0$, an $(1+\epsilon)$-approximate solution can be computed in $\mathcal{O}(nm(nm/\epsilon)^{m-1})$ time, which is polynomial in both n and $1/\epsilon$ if m is fixed. However, for the case where the number of processors is specified as a part of the problem instance, an FPTAS is unlikely to exist.

Lenstra et al. [57] also gave a PTAS for the problem with running time bounded by the product of $(n+1)^{m/\epsilon}$ and a polynomial of the input size. Although for a fixed m their algorithm is not fully polynomial, it has a much smaller space complexity than the one in [45]. In addition, the authors proved that unless $P = NP$, there is no polynomial-time approximation algorithm for the $R \mid\mid C_{max}$ problem with approximation factor less than $\frac{3}{2}$, and they also presented a polynomial-time 2-approximation algorithm. This algorithm computes first an optimal *fractional* (or preemptive) solution obtained via linear programming and then uses rounding to obtain a schedule for the discrete problem with an approximation factor of 2. Shmoys and Tardos [78] generalized this technique to obtain the same approximation factor for the generalized assignment problem. Furthermore, they generalized the rounding technique to hold for any fractional solution.
In 2004, Shchepin and Vakhania [77] introduced a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$ for a similar time complexity as [57]. To the best of our knowledge, this is so far the best low-cost approximation result for this problem. However, the prohibitive computational cost of these algorithms prevents their usage on actual computing platforms.
The fractional unrelated scheduling problem can also be formulated as a generalized maximum flow problem, where the network is defined by the scheduling problem and the capacity of some edges, that corresponds to the makespan, is minimized. This generalized maximum flow problem is a special case of linear programming (LP).

We can note that the $R \mid\mid C_{\max}$ reference problem is more generic than the problems studied in this PhD. It can be refined to better fit the constraints of the hybrid platforms.
Bonifaci and Wiese [12] 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 $R \mid\mid C_{\max}$, and the rounding phases of the algorithm require a significant amount of time, raising the time complexity of the algorithm to an impractical level, even when only two types of machines are considered, as it would be the case for a CPU-GPU platform.

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 [69]. 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 $(Pm, Pk) \mid\mid C_{\max}$.

3.2.2.4 Heuristics

Another possibility for solving difficult scheduling problems is to consider heuristic algorithms in hope of providing good results. This is the kind of scheduling algorithm that is used by most computing platforms today, most notably the HEFT

algorithm [84], that is studied in the following chapter, in Section 4.2.1. However, by using heuristics, there is usually no approximation ratio for the quality of solution, and most of the time we can only have a bound on the computation time of the schedule.

This PhD focused more on providing guarantees for the algorithms we developed while keeping the time complexity of the algorithms reasonably low to be used on a real platform.

We have seen several methods used to solve classical scheduling problems with independent sequential that could be used to tackle the scheduling problem on a hybrid platform with CPUs and GPUs. Some specific versions of this problem can be solved using some of these methods. However, for the more generic problem $(Pm, Pk) \mid\mid C_{max}$, none of the above methods are satisfactory in terms of performance guarantee and practical use. Hence, new algorithms need to be developed for these problems, with an approximation ratio and a realistic time complexity.
We started studying problem $(Pm, Pk) \mid\mid C_{max}$ and developed new algorithms for it. The first methods and subsequent algorithms are presented in the following chapter.

Chapter 4

Minimizing the Makespan with Independent Sequential Tasks

This chapter presents the first problem of scheduling on a hybrid platform with CPUs and GPUs that we studied during this PhD: minimizing the makespan with independent tasks on m CPUs and k GPUs. We analyze the problem and try different algorithms, starting with a simple version of the problem with only one CPU and one GPU, then increasing the number of processors. The organization of this chapter is progressive and the size of the problems (in terms of numbers of processors) grows as we advance in our analysis.
We consider in this chapter the problem of scheduling on a multi-core parallel platform with m identical CPUs and k identical GPUs, $(Pm, Pk) \mid\mid C_{max}$, previously described in Chapter 3, Section 3.1.1.1. We recall that the set of tasks to schedule, $\mathcal{T}$, is composed of n tasks $T_1, \ldots, T_n$, each of these tasks having two processing times depending on which type of processor it is assigned to: $\overline{p_j}$ if task T_j is processed on a CPU and $\underline{p_j}$ if it is processed on a GPU, both processing times being known in advance. The acceleration factor of task T_j is still given by the ratio $q_j = \frac{\overline{p_j}}{\underline{p_j}}$,as it was in the previous chapter. The objective is still to minimize the makespan of the whole schedule, $C_{max} = \max\left(C_{max}^{CPU}, C_{max}^{GPU}\right)$.
We observe that if both processing times are equal ($\overline{p_j} = \underline{p_j}$) for $j = 1, \ldots, n$, the problem $(P1, P1) \mid\mid C_{max}$ is equivalent to the classical $P2 \mid\mid C_{max}$ problem, which is NP-hard [32]. Thus, the problem of scheduling with GPUs is also NP-hard and we aim at finding efficient approximation algorithms with a good performance guarantee.
In order to do that, we first study the simplest version of the problem, with only one CPU and one GPU, $(P1, P1) \mid\mid C_{max}$.

4.1 Considering only one CPU and one GPU

The first method we tried to apply in order to solve problem $(P1, P1) \mid\mid C_{max}$ was the list scheduling paradigm.

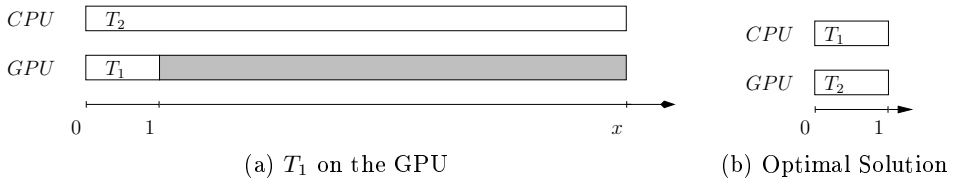
We can remark that this problem is exactly like $R2 \mid\mid C_{max}$, since we have on one side a CPU and on the other side a GPU and the processing times of tasks on these two processors cannot be linked by any law. Ibarra and Kim [47] gave an approximation algorithm for this problem with an approximation ratio of $\frac{1+\sqrt{5}}{2}$, which is quite high interesting. However, the algorithm cannot be extended to the case where the number of machines increases. We tried to approach the problem as completely new in order to developed a specific approximation algorithm that could be better adapted to the case where there are more than one CPU and one GPU.

4.1.1 An arbitrary list scheduling algorithm

In the list scheduling paradigm (see Chapter 3, Section 3.2.2.1), the set of tasks that are ready to be executed are kept in a priority list. When a computing resource becomes available, the task with the highest priority is scheduled on this resource. If no priority is specified, the tie is broken randomly. However, the use of the same strategy in a hybrid system, leads to a large value of worst case performance ratio, as demonstrated in the following lemma.

Lemma 4.1.1. *For problem $(P1, P1) \mid\mid C_{max}$, a list scheduling algorithm has a worst case performance ratio larger than the maximum acceleration of the tasks.*

Proof. Let C_{max}^{LIST} denote the value of the makespan obtained by any list scheduling algorithm and C_{max}^{*} its optimal value. Let us consider an instance of problem $(P1, P1) \mid\mid C_{max}$ composed of two tasks T_1 and T_2, with $\overline{p_1} = \underline{p_1} = 1$, $\overline{p_2} = x$ and $\underline{p_2} = 1$. If the algorithm assigns T_1 to the GPU and T_2 to the CPU, we get a makespan of $C_{max}^{LIST} = x$ (cf. Figure 4.1a). Since both processors are unrelated, we can always find an instance such as the first task selected by the list algorithm is similar to T_1.



(a) T_1 on the GPU (b) Optimal Solution

Figure 4.1: List scheduling algorithm with two different list orders.

An optimal solution can be obtained by assigning T_1 to the CPU and T_2 to the GPU leading to $C_{max}^{*} = 1$ (cf. Figure 4.1b). The approximation ratio is equal to x and thus the solution can be arbitrarily far from the optimum. □

Since this list scheduling algorithm was inconclusive in terms of performance with the objective of minimizing the makespan, we tried another approach. The main problem of the list scheduling algorithm is that it can only minimize a makespan on one type of processors, whereas the objective of minimizing the global makespan of the schedule implies to try to minimize both the makespan on the CPU and the makespan on the

GPU at the same time. Remaining with a single objective function to minimize, another objective was chosen, in order to be closer to the minimization of both makespans: we tried to minimize the sum of the makespans on the CPU and on the GPU.

4.1.2 Minimizing the sum of the makespans

We consider a combination of the two makespans on CPU and GPU to have only one makespan to minimize. We define $\delta_j = \overline{p_j} - \underline{p_j}$ for each task T_j.
We use a binary variable to characterize the assignment of a task Tj to the CPU or the GPU, for all $j \in \{1, \ldots, n\}$:

$$x_j = \begin{cases} 1 & \text{if task } T_j \text{ is assigned to the CPU} \\ 0 & \text{if task } T_j \text{ is assigned to the GPU} \end{cases}$$

The respective makespans on the CPU and the GPU can be expressed respectively as $\sum_{j=1}^{n} \overline{p_j} x_j$ and $\sum_{j=1}^{n} \underline{p_j} (1 - x_j)$. If we calculate the sum of these two makespans, we obtain $\sum_{j=1}^{n} \underline{p_j} + \sum_{j=1}^{n} \left(\overline{p_j} - \underline{p_j}\right) x_j$, which is the objective we want to minimize. In a sense, we are minimizing the global computational area of the schedule, which corresponds to the sum of the processing times of the tasks in the schedule. The term $\sum_{j=1}^{n} \underline{p_j}$ being constant, we look at minimizing

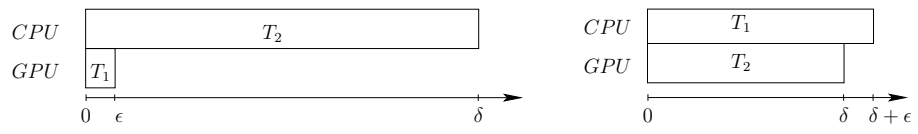
$$\sum_{j=1}^{n} \delta_j x_j.$$

Since δ_j represents the difference between the processing time of task T_j on CPU and its processing time on GPU, minimizing $\sum_{j=1}^{n} \delta_j x_j$ is equivalent to choosing to assign to the CPU the tasks whose processing time varies the least when changing processors. This means that we want to assign to the GPU the tasks that provide the largest gain in terms of computational area.
Let us consider the following greedy algorithm:

Algorithm 4.1.2.

- *Start by assigning all the tasks to the CPU.*
- *Sort the tasks by decreasing δ_j and assign them according to this order to the GPU as long as $C_{max}^{GPU} \leqslant C_{max}^{CPU}$.*

This algorithm returns a schedule of makespan $C_{max}(\delta)$. However, there exists an instance where this makespan is equal to $2C_{max}^{*}$ (see Figure 4.2), so we cannot expect to have a better performance guarantee than 2 for Algorithm 4.1.2.



(a) T_1 on the GPU (b) Optimal Solution

Figure 4.2: Scheduling with minimal makespan criteria.

Indeed, consider an instance of the problem with two tasks such as $\overline{p_1} = \delta + \epsilon + \epsilon'$, $\underline{p_1} = \epsilon$, $\overline{p_2} = 2\delta$ and $\underline{p_2} = \delta$, where δ, $\epsilon < \delta$, and $\epsilon' \ll \epsilon$ are given. According to the definition of δ_j, we have $\delta_1 = \delta + \epsilon'$ and $\delta_2 = \delta$, therefore Algorithm 4.1.2 schedules task T_1 on the GPU and task T_2 on the CPU: the resulting schedule has a makespan of 2δ. If we put task T_2 on the GPU and task T_1 on the CPU, we obtain an optimal schedule with a makespan of δ. The ratio is then 2, therefore the performance guarantee of Algorithm 4.1.2 us at least 2.
Moreover, we have two straightforward lower bounds of the optimal makespan: $C^*_{max} \geqslant \max_{1 \leqslant j \leqslant n} \underline{p_j}$ and $C^*_{max} \geqslant \frac{1}{2} \sum_{j=1}^{n} \underline{p_j}$. Assuming that the tasks are reindexed according to their assignment to the GPU first and then their assignment to the CPU, we define T_l as the last task scheduled on the GPU. The makespan on the CPU becomes $C^{CPU}_{max} = \sum_{j=l+1}^{n} \overline{p_j}$, and we have

$$\frac{\sum_{j=l+1}^{n} \overline{p_j}}{C^*_{max}} \leqslant \frac{\sum_{j=1}^{n} \overline{p_j}}{\frac{1}{2} \sum_{j=1}^{n} \underline{p_j}}$$

$$\leqslant 2 \frac{\sum_{j=1}^{n} \overline{p_j}}{\sum_{j=1}^{n} \alpha_j \overline{p_j}}$$

$$\frac{\sum_{j=l+1}^{n} \overline{p_j}}{C^*_{max}} \leqslant \frac{2}{\min_{1 \leqslant j \leqslant n} \alpha_j}$$

This guarantee is worse than the one provided by the greedy algorithm developed for the problem of scheduling on two sets of identical processors [48] mentioned in Chapter 3, Section 3.2.2.1, where we have a guarantee of 2 when $m = k = 1$. When k and m are arbitrary, we have a guarantee of $2 + \frac{m-1}{k}$ with the algorithm from [48], which is not satisfactory on large computing platforms, where the number of CPUs can be very high and the number of GPUs can remain very low.

This objective was also inconclusive in terms of performance, therefore another approach had to be explored. Since the core problem is to keep both the makespan on the CPU and the makespan on the GPU at a minimum value, we tried to minimize one of the makespans while keeping the other makespan lower than the first one.

4.1.3 A knapsack based approach

Here, we minimize one of the makespan (for example the one on the CPU) while forcing the other makespan (the one on the GPU) to remain below the first makespan, in order to obtain a knapsack formulation of our problem (see Chapter 3, Section 3.2.1.3). Defining $\sigma_j = \overline{p_j} + \underline{p_j}$, and using the same decision variables x_j as before, we can write the problem of minimizing C_{max}^{CPU} while forcing C_{max}^{GPU} to remain lower than C_{max}^{CPU} as follows:

$$\begin{aligned} \min \ & \sum_{j=1}^{n} \overline{p_j} x_j \\ \text{s.t.} \ & \sum_{j=1}^{n} \underline{p_j} \left(1 - x_j\right) \leqslant \sum_{j=1}^{n} \overline{p_j} x_j \\ & x_j \in \{0,1\} \qquad\qquad \forall j \in \{1,\ldots,n\} \end{aligned}$$

which is equivalent to

$$\begin{aligned} \max \ & \sum_{j=1}^{n} \left(-\overline{p_j}\right) x_j \\ \text{s.t.} \ & \sum_{j=1}^{n} \left(-\overline{p_j} - \underline{p_j}\right) x_j \leqslant \sum_{j=1}^{n} \left(-\underline{p_j}\right) \\ & x_j \in \{0,1\} \qquad\qquad \forall j \in \{1,\ldots,n\} \end{aligned}$$

If we define $C = \sum_{j=1}^{n} \underline{p_j}$, we obtain the following knapsack problem:

$$(K_C) \qquad \begin{aligned} \max \ & \sum_{j=1}^{n} \left(-\overline{p_j}\right) x_j \\ \text{s.t.} \ & \sum_{j=1}^{n} \left(-\sigma_j\right) x_j \leqslant -C \\ & x_j \in \{0,1\} \qquad\qquad \forall j \in \{1,\ldots,n\} \end{aligned}$$

with task T_j having a value of $\left(-\overline{p_j}\right)$, a weight of $\left(-\sigma_j\right)$, and the knapsack having a capacity of $\sum_{j=1}^{n} \left(-\underline{p_j}\right) = -C < 0$.

The other problem of minimizing the makespan of the GPU while forcing $C_{max}^{CPU} \leqslant C_{max}^{GPU}$ can also be written as a knapsack problem:

$$(K_G) \qquad \begin{aligned} \max & \sum_{j=1}^{n} \underline{p_j} x_j \\ \text{s.t.} & \sum_{j=1}^{n} \sigma_j x_j \leqslant C \\ & x_j \in \{0,1\} \qquad\qquad \forall j \in \{1,\ldots,n\} \end{aligned}$$

with task T_j having a value of $\underline{p_j}$, a weight of σ_j, and the knapsack having a capacity of $C = \sum_{j=1}^{n} \underline{p_j} > 0$.
We present now an algorithm with a performance ratio for our problem, which is based on a greedy algorithm [64] for the knapsack problem (with values p_j, weights w_j, capacity W). The greedy algorithm is as follows:

Algorithm 4.1.3. *Take the tasks by decreasing order of importance, $\frac{p_j}{w_j}$ and assign $x_j = 1$ as long as the sum of the weights of the assigned tasks stays lower than the capacity W.*

This algorithm does not have a constant guarantee, but one that has [64] can be derived from it:

Algorithm 4.1.4.

- *Compute a solution to the knapsack problem with algorithm 4.1.3, S_{imp}, and memorize the first task too big to fit in the knapsack.*
- *Create a new solution to the knapsack problem composed only of the first task discarded by algorithm 4.1.3, S_{dis}.*
- *Take the solution S of maximum value between S_{imp} and S_{dis}.*

Lemma 4.1.5. *Algorithm 4.1.4 has a performance guarantee of $\frac{3}{2}$ for a knapsack formulation of problem $(P1, P1) \mid\mid C_{max}$.*

Proof. Let T_{j_0} be the first task discarded by the decreasing order of importance assignment. We note $val(I)$ the value of a knapsack solution computed by algorithm 4.1.4 and val^* the value of the optimal solution for the associated knapsack formulation of the problem. With these notations, we have the inequality $val^* \leqslant val(S_{imp}) + \underline{p_{j_0}}$.
The value of the selected solution S is greater than the average of the values of solutions S_{imp} and S_{dis} (whose value is equal to $\underline{p_{j_0}}$), so we have

$$val(S) \geqslant \frac{val(S_{imp}) + \underline{p_{j_0}}}{2} \geqslant \frac{val^*}{2},$$

which gives us, for $(P1, P1) \mid\mid C_{max}$, represented by (K_G):

$$\sum_{j=1}^{n} \underline{p_j} x_j(S) \geqslant \sum_{j=1}^{n} \frac{\underline{p_j} x_j^*}{2},$$

$$\sum_{j=1}^{n} \underline{p_j} \left(1 - x_j(S)\right) \leqslant \frac{\sum_{j=1}^{n} \underline{p_j}}{2} + \frac{1}{2} \sum_{j=1}^{n} \underline{p_j} \left(1 - x_j^*\right),$$

$$C_{max}(S) \leqslant \frac{\sum_{j=1}^{n} \underline{p_j}}{2} + \frac{C_{max}^*}{2}.$$

We know that $\frac{\sum_{j=1}^{n} \underline{p_j}}{2} \leqslant C_{max}^*$, which gives us a performance guarantee of $\frac{3}{2}$ for the algorithm.
A similar proof can be written for the (K_C) knapsack formulation of problem $(P1, P1) \mid\mid C_{max}$, with the same performance guarantee of $\frac{3}{2}$ for the algorithm. □

Dynamic Programming We can also use dynamic programming (see Chapter 3, Section 3.2.1.3) to solve the knapsack problem, and by extension, $(P1, P1) \mid\mid C_{max}$. Ibarra and Kim designed a pseudo-polynomial algorithm with dynamic programming and an FPTAS for the knapsack problem [46]. From their algorithms we can derived a pseudo-polynomial algorithm and an FPTAS for problem $(P1, P1) \mid\mid C_{max}$. For simplicity, we will use the knapsack problem formulation (K_G):

$$(K_G) \qquad \begin{aligned} \max \ & \sum_{j=1}^{n} \underline{p_j} x_j \\ \text{s.t.} \ & \sum_{j=1}^{n} \sigma_j x_j \leqslant \sum_{j=1}^{n} \underline{p_j} \\ & x_j \in \{0, 1\} \qquad \forall j \in \{1, \ldots, n\} \end{aligned}$$

where $C = \sum_{j=1}^{n} \underline{p_j}$. We define $P = \max_{j \in \{1,\ldots,n\}} \underline{p_j}$ as the highest value of any task. Then nP is a trivial upper bound on the value that can be achieved by any solution.
We assume here that every processing time on the GPU for every task is an integer. For each $j \in \{1, \ldots, n\}$ and $p \in \{1, \ldots, nP\}$, we define a subset $S_{j,p}$ of $\{1, \ldots, j\}$ whose total

value is exactly p (i.e. $\sum_{l \in S_{j,p}} \underline{p_l} = p$) and whose total capacity, denoted by $A(j,p)$, is minimized (i.e. $A(j,p) = \sum_{l \in S_{j,p}} \sigma_l = \min_{S \subset \{1,\ldots,j\}} \sum_{l \in S} \sigma_l$. We $A(j,p) = \infty$ if no set $S_{j,p}$ defined as before can exist.
Clearly $A(1,p)$ is known for every $p \in \{1,\ldots,nP\}$. The following recurrence helps compute all values $A(j,p)$ with a time complexity in $\mathcal{O}(n^2P)$:

$$A(j+1,p) = \begin{cases} \min\left\{A(j,p), \sigma_{j+1} + A\left(j, p - \underline{p_{j+1}}\right)\right\} & \text{if } \underline{p_{j+1}} \leqslant p \\ A(j,p) & \text{otherwise} \end{cases}$$

The maximum value achievable by tasks of total weight bounded by C is

$$\max\left\{p |\ A(n,p) \leqslant C\right\}.$$

Therefore we have a pseudo-polynomial algorithm for the knapsack problem, and, by extension, for problem $(P1, P1) \mid\mid C_{max}$.

FPTAS From the dynamic programming algorithm we can build an FPTAS (see Chapter 3, Section 3.2.2.3) for our problem: if the values of our tasks in the knapsack formulation were bounded by a polynomial in n, then we would have a regular polynomial time algorithm. In our approximation scheme we ignore a certain number of least significant bits of values of tasks (depending on the error parameter ϵ), so that the modified values can be viewed as numbers bounded by a polynomial in n and $1/\epsilon$. This enables us to find a solution whose knapsack value is at least $(1-\epsilon)val^*$, where val^* is the value of an optimal solution of the corresponding knapsack formulation, in time bounded by a polynomial in n and $1/\epsilon$.

Algorithm 4.1.6 (FPTAS)**.**

1. *Given an instance I and $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.*
2. *For each task T_j, define $\underline{p'_j} = \left\lfloor \frac{p_j}{K} \right\rfloor$.*
3. *Define a new instance I' with the $\underline{p'_j}$ as values of the tasks and, using the dynamic programming algorithm, find the most valuable set S'.*
4. *Return S'.*

Lemma 4.1.7. *Algorithm 4.1.6 is an FPTAS for problem $(P1, P1) \mid\mid C_{max}$.*

Proof. For any task T_j, because of the rounding step, $K\underline{p'_j}$ can be smaller than $\underline{p_j}$ but by no more than K. Therefore,

$$val^* - val'^* \leqslant nK.$$

The dynamic programming step must return a set at least as good as the optimal one under the new values for the knapsack formulation. Therefore

$$\begin{aligned} val(S') &\geqslant Kval'^* \\ &\geqslant val^* - nK = val^* - \epsilon P \\ &\geqslant (1-\epsilon)val^*, \end{aligned} \tag{4.1}$$

where the last inequality follows from the observation that $val^* \geqslant P$.
The running time of the algorithm is in $\mathcal{O}\left(n^2 \left\lfloor \frac{P}{K} \right\rfloor\right) = \mathcal{O}\left(n^2 \left\lfloor \frac{n}{\epsilon} \right\rfloor\right)$, which is polynomial in n and $\frac{1}{\epsilon}$.
If we look at our formulation (K_G) of problem $(P1, P1) \mid\mid C_{max}$, Inequality (4.1) becomes

$$\sum_{j=1}^{n} \underline{p_j} x_j \geqslant \sum_{j=1}^{n} \underline{p_j} x_j^* - \epsilon \max_{1 \leqslant j \leqslant n} \underline{p_j},$$

where x_j^* refers to the assignment of task T_j in the optimal solution. We can reverse this inequality

$$-\sum_{j=1}^{n} \underline{p_j} x_j \leqslant -\sum_{j=1}^{n} \underline{p_j} x_j^* + \epsilon \max_{1 \leqslant j \leqslant n} \underline{p_j},$$

$$\sum_{j=1}^{n} \underline{p_j} (1 - x_j) \leqslant \sum_{j=1}^{n} \underline{p_j} \left(1 - x_j^*\right) + \epsilon \max_{1 \leqslant j \leqslant n} \underline{p_j},$$

$$C_{max}^{GPU}(S) \leqslant C_{max}^{GPU}(OPT) + \epsilon \max_{1 \leqslant j \leqslant n} \underline{p_j},$$

where $C_{max}^{GPU}(OPT)$ represents the makespan on the GPU in the optimal schedule. Moreover, $\max\limits_{1 \leqslant j \leqslant n} \underline{p_j} \leqslant C_{max}^{GPU}(OPT)$, so

$$C_{max}^{GPU}(S) \leqslant (1+\epsilon)\, C_{max}^{GPU}(OPT).$$

The same result can be obtained with the (K_C) knapsack formulation of the problem. We denote by S^G (resp. S^C) the solution obtained when solving knapsack formulation (K_G) (resp. (K_C)) with Algorithm 4.1.6, and $C_{max}^{GPU}(OPT)$ (resp. $C_{max}^{CPU}(OPT)$) the optimal solution of the corresponding problem. The makespan of the schedule obtained by Algorithm 4.1.6 becomes

$$\begin{aligned} C_{max} &= \min\left\{C_{max}^{GPU}(S^{GPU}),\ C_{max}^{CPU}(S^{CPU})\right\} \\ &\leqslant (1+\epsilon) \min\left\{C_{max}^{GPU}(OPT^{GPU}),\ C_{max}^{CPU}(OPT^{CPU})\right\} \\ &\leqslant (1+\epsilon) C_{max}^{*}. \end{aligned}$$

□

We therefore have two scheduling algorithms for $(P1, P1) \mid\mid C_{max}$, a greedy one, with a performance guarantee of $\frac{3}{2}$, and an FPTAS based on dynamic programming.
Now we move on to the problem where we have more than one CPU and more than one GPU.

4.2 Fast algorithms with m CPUs, k GPUs

In this section, we first study one of the most used scheduling algorithm on heterogeneous platforms, HEFT [84], and then propose an algorithm of our own design with a performance guarantee for our new scheduling problem: $(Pm, Pk) \mid\mid C_{max}$.

4.2.1 HEFT algorithm

The heuristic scheduler like Heterogeneous-Earliest-Finish-Time or HEFT [84] (see Chapter 3, Section 3.2.2.4) proceeds in two phases as follows:

- prioritization of the tasks that are sorted the tasks by decreasing average execution time.
- then the processor selection is obtained with the heterogeneous earliest finish time rule: tasks are scheduled in the order of prioritization and they are assigned to the processor that will allow them to finish their processing at the earliest possible time, regardless of the type of processor.

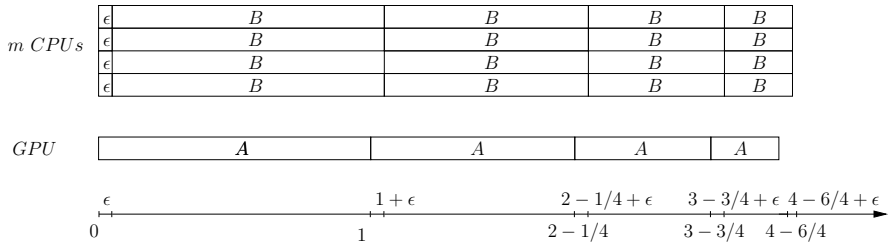
Despite appearing similar, HEFT is not a list scheduling algorithm since some computing resources may stay idle even if a task could be executed on it.

Lemma 4.2.1. *For problem $(Pm, P1) \mid\mid C_{max}$, the worst case performance ratio of HEFT is larger than $m/2$.*

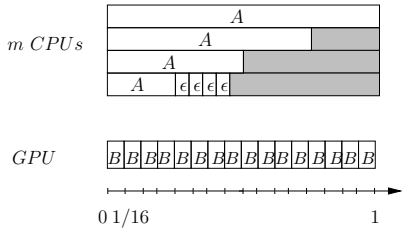
Proof. We show on the following instance (cf. Figure 4.3) that the prioritizing phase can provide a schedule whose makespan is arbitrarily far from the optimum.
Let us consider an instance with a list of the following tasks:

- m tasks of equal length such that $\overline{p_j} = \epsilon$ and $\underline{p_j} = m + k + 1$ (these tasks have a long execution time on the GPU).
- m sets of $m + 1$ tasks, with, for $i = 0, \cdots, m - 1$:
  - a single task of type A such that $\overline{p} = \underline{p} = 1 - i/m$;
  - m tasks of type B, of equal length, such that $\overline{p_j} = 1 - i/m$ and $\underline{p_j} = 1/m^2$ (these tasks are executed faster on the GPU).

On this instance, HEFT fills first the m CPUs. Then, the algorithm fills alternatively the GPU with one task of type A and the m CPUs with m tasks of type B. HEFT ends up with a makespan equal to $m/2 + 3/2 - 1/m$ (cf. Figure 4.3-a). It is easy to check that the optimal makespan is equal to $C^*_{max} = 1$ (cf. Figure 4.3-b). □



(a) HEFT schedule



(b) Optimal solution

Figure 4.3: HEFT schedule and the optimal solution with $m = 4$, $k = 1$.

HEFT is therefore not a suitable algorithm when looking for performance guarantees. We therefore turn to the methods we developed for problem $(P1, P1) \mid\mid C_{max}$, and try to adapt them to problem $(Pm, Pk) \mid\mid C_{max}$.

4.2.2 Extending the Knapsack-based Approach

We look at adapting the knapsack-based approach used for problem $(P1, P1) \mid\mid C_{max}$ for the same problem with larger values of m and k, but problem $(Pm, Pk) \mid\mid C_{max}$ cannot be decomposed in two knapsack problems such as (K_C) and (K_G) for $(P1, P1) \mid\mid C_{max}$. An idea is to consider all the CPUs as one large CPU and all the GPUs as one large GPU. The makespan of this large CPU (resp. GPU) is considered to be the computing area of the CPUs (resp. GPUs), i.e. the sum of the processing times of the tasks on all the CPUs (resp. GPUs), divided by the number of CPUs (resp. GPUs). We can then solve this problem as a $(P1, P1) \mid\mid C_{max}$ problem and have a lower bound of the makespan of problem $(Pm, Pk) \mid\mid C_{max}$. This resolution assigns each task of the original problem to a type of processor, either a CPU or a GPU. Then we can schedule with the LPT rule on the CPUs all the tasks assigned by this resolution to the large CPU and do the same on the GPUs. If we denote by $C_{max}(ALG)$ the makespan of the schedule resulting from this algorithm we call ALG, C^*_{max} the optimal makespan for problem $(Pm, Pk) \mid\mid C_{max}$, $C_{max}((P1, P1))$ and $C^*_{max}(P1, P1)$ respectively the makespan of the algorithm used to solve the corresponding problem $(P1, P1) \mid\mid C_{max}$ and the optimal makespan for this problem, we have the following lemma:

Lemma 4.2.2. $\frac{C_{max}(ALG)}{C^*_{max}} \leqslant \left(2 - \frac{1}{m}\right) \frac{C_{max}(P1,P1)}{C^*_{max}(P1,P1)}$, if $m \geqslant k$.

Proof. The assignment of the tasks according to problem $(P1, P1) \mid\mid C_{max}$ is denoted by the binary variable

$$x_j^0 = \begin{cases} 1 & \text{if task } T_j \text{ is scheduled on a CPU} \\ 0 & \text{if task } T_j \text{ is scheduled on a GPU} \end{cases}$$

We have $C_{max}(P1, P1) = \max\left(\sum_{j=1}^{n} \frac{\overline{p_j}}{m} x_j^0,\ \sum_{j=1}^{n} \frac{p_j}{\overline{k}} \left(1 - x_j^0\right)\right)$, and since solving $(P1, P1) \mid\mid C_{max}$ here is equivalent to minimizing the maximum of the computing areas of the CPUs and the GPUs, divided by their respective number of processors, we can write

$$C_{max}^*(P1, P1) = \min_{x_j} \max\left(\sum_{j=1}^{n} \frac{\overline{p_j}}{m} x_j,\ \sum_{j=1}^{n} \frac{p_j}{\overline{k}} \left(1 - x_j\right)\right) \leqslant \Sigma^*,$$

where $\Sigma^* = \max\left(\sum_{j=1}^{n} \frac{\overline{p_j}}{m} x_j^*,\ \sum_{j=1}^{n} \frac{p_j}{\overline{k}} \left(1 - x_j^*\right)\right)$, and x_j^* represents the assignment of task T_j to a CPU or a GPU in the optimal solution for problem $(Pm, Pk) \mid\mid C_{max}$.
Suppose that there exists an instance with an optimal solution such that $\frac{C_{max}(ALG)}{C_{max}^*} > \left(2 - \frac{1}{m}\right) \frac{C_{max}(P1,P1)}{C_{max}^*(P1,P1)}$. If we consider the instance with the smallest number of tasks among the instances verifying the previous inequality, the last task T_α (the one with the smallest processing time) to start its processing on the CPUs is also the one that finishes his processing last, at C_{max}^{CPU}, the makespan of the CPUs. Since all the processors are busy before T_α starts, we have

$$\begin{aligned}
C_{max}^{CPU} - \overline{p_\alpha} &\leqslant \frac{\sum_{j=1, j\neq\alpha}^{n} \overline{p_j} x_j^0}{m}, \\
C_{max}^{CPU} &\leqslant \frac{\sum_{j=1}^{n} \overline{p_j} x_j^0}{m} \\
&\leqslant \overline{p_\alpha}\left(1 - \frac{1}{m}\right) + C_{max}(P1, P1) \\
&\leqslant \overline{p_\alpha}\left(1 - \frac{1}{m}\right) + C_{max}^*(P1, P1)\ \frac{C_{max}(P1, P1)}{C_{max}^*(P1, P1)} \\
C_{max}^{CPU} &\leqslant \overline{p_\alpha}\left(1 - \frac{1}{m}\right) + \Sigma^*\ \frac{C_{max}(P1, P1)}{C_{max}^*(P1, P1)}.
\end{aligned}$$

The value of Σ^* is a lower bound of the optimal makespan C_{max}^* since it is the makespan in the case where all the tasks on the CPUs finish their processing at the same time and the same inequality holds for the GPUs. Therefore

$$C_{max}^{CPU} \leqslant \overline{p_\alpha}\left(1-\frac{1}{m}\right) + C_{max}^{*}\,\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)},$$

$$\frac{C_{max}^{CPU}}{C_{max}^{*}} \leqslant \frac{\overline{p_\alpha}\left(1-\frac{1}{m}\right)}{C_{max}^{*}} + \frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)}.$$

We assumed $\left(2-\frac{1}{m}\right)\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)} < \frac{C_{max}^{CPU}}{C_{max}^{*}}$, so we obtain $C_{max}^{*} < \overline{p_\alpha}\,\frac{C_{max}^{*}(P1,P1)}{C_{max}(P1,P1)}$, and since $C_{max}^{*}(P1,P1) \leqslant C_{max}(P1,P1)$, we have $C_{max}^{*} < \overline{p_\alpha}$.
The same reasoning can be done with the GPUs, and we obtain, with T_γ being the last task to start its processing on the GPUs, finishing at C_{max}^{GPU}:

$$\frac{C_{max}^{GPU}}{C_{max}^{*}} \leqslant \frac{\underline{p_\gamma}\left(1-\frac{1}{k}\right)}{C_{max}^{*}} + \frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)}.$$

If we suppose $k \leqslant m$, we have $\left(2-\frac{1}{k}\right)\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)} \leqslant \left(2-\frac{1}{m}\right)\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)} < \frac{C_{max}^{GPU}}{C_{max}^{*}}$, so we obtain $C_{max}^{*} < \underline{p_\gamma}\frac{C_{max}^{*}(P1,P1)}{C_{max}(P1,P1)}$ and finally $C_{max}^{*} < \underline{p_\gamma}$.
Therefore in the optimal solution we have all the assignments that are reversed in comparison to the solution derived from problem $(P1,P1)\,||\,C_{max}$.
If we look at a task T_j such as $x_j^0 = 0$, $x_j^* = 1$, we have $\overline{p_j} < \overline{p_\alpha}$ and $\overline{p_j} < \underline{p_\gamma}$, but $\underline{p_j} \geqslant \underline{p_\gamma}$, so $\underline{p_j} > \overline{p_j}$, which is impossible. This contradicts the existence of an instance such that $\left(2-\frac{1}{m}\right)\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)} < \frac{C_{max}(ALG)}{C_{max}^{*}}$. □

However, there exists an instance of problem $(P2,P2)\,||\,C_{max}$ where we have $C_{max}(ALG) = \frac{3}{2}C_{max}^{*}$. This instance consists in 4 tasks to schedule on 2 CPUs and 2 GPUs, such as $\overline{p_1} = 6$, $\underline{p_1} = 4$, $\overline{p_2} = \underline{p_2} = 1$, $\overline{p_3} = 25$, $\underline{p_3} = 3+\epsilon$, $\overline{p_4} = 4-\epsilon$ and $\underline{p_4} = 4-2\epsilon$.
The optimal solution for the corresponding $(P1,P1)\,||\,C_{max}$ problem is the assignment given in Figure 4.4. Here the ratio $\frac{C_{max}(P1,P1)}{C_{max}^{*}(P1,P1)}$ is equal to 1.

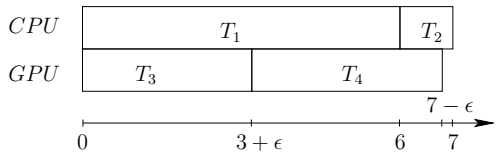
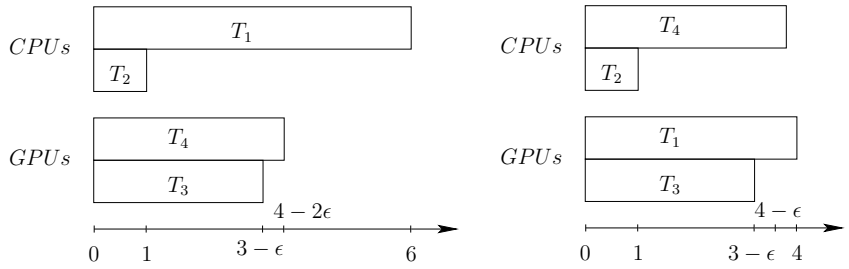


Figure 4.4: Optimal Schedule of the instance when considered as $(P1,P1)\,||\,C_{max}$, with makespan $C_{max}(P1,P1)$.

We can compare the assignments provided by algorithm ALG on our four processors with the LPT rule on the CPUs and the GPUs to the optimal solution, as we can see in Figure 4.5.



(a) Schedule of the instance with $(P1, P1)\ ||\ C_{max}$ assignments followed by LPT ($C_{max}(ALG)$)

(b) Optimal Schedule (C^*_{max})

Figure 4.5: Schedule for the $(P2, P2)\ ||\ C_{max}$ problem following the $(P1, P1)\ ||\ C_{max}$ assignments, and the optimal solution.

But with the assignments of the optimal solution, the corresponding $(P1, P1)\ ||\ C_{max}$ schedule would have been the one in Figure 4.6, which explains why this solution was not considered.

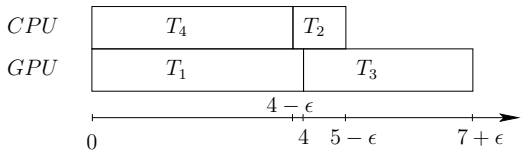


Figure 4.6: Optimal schedule for $(P2, P2)\ ||\ C_{max}$ when considered as a $(P1, P1)\ ||\ C_{max}$ problem.

The ratio is $\frac{6}{4} = \frac{3}{2}$, so this algorithm cannot have a better approximation ratio than $\frac{3}{2}C^*_{max}$.
Moreover, to this approximation ratio must be added the approximation ratio of the algorithm used to determine the solution of the corresponding $(P1, P1)\ ||\ C_{max}$ problem, which was 1 in the previous example. In the generic case, if we take the greedy algorithm presented in Section 4.1.3 for this problem, its approximation ratio is $\frac{3}{2}$. This gives us a final approximation ratio of $3 - \frac{3}{2m}$. Dynamic programming could allow us to remain at a ratio of 2 but the algorithm would not be polynomial anymore.

4.2.3 Dual approximation Scheme for solving $(Pm, Pk)\ ||\ C_{max}$

In order to get a performance ratio with a knapsack based approach derived from the resolution of $(P1, P1)\ ||\ C_{max}$ for problem $(Pm, Pk)\ ||\ C_{max}$, we use the dual approximation technique (see Chapter 3, Section 3.2.2.2): we take a guess λ, assumes that there exists a schedule of length at most λ and either delivers a schedule of makespan at most $g\lambda$ (g being the desired approximation ratio), or answers correctly that there exists no schedule of length at most λ.

The guess of the dual approximation technique allows us to consider, at each main step of the dual approximation, the $(Pm, Pk) \mid\mid C_{max}$ problem as only one large CPU and one large GPU to be filled, meaning that we can apply a knapsack algorithm similar to the one designed for the $(P1, P1) \mid\mid C_{max}$ problem (cf Figure 4.7). At one step of the dual approximation, the algorithm is as follows:

Algorithm 4.2.3.

- *Extract from the set of tasks those that are necessarily assigned to the GPUs ($\overline{p_j} > \lambda$, where λ is the current guess), put them on the GPUs and then fill the GPUs with the tasks with the largest acceleration factor (defined by $\frac{\overline{p_j}}{\underline{p_j}}$) up to the k times the guess.*
- *Put all the remaining tasks on the m CPUs, ordering them according to the LPT rule.*
- *Reorder the tasks on the GPUs according to the LPT rule.*

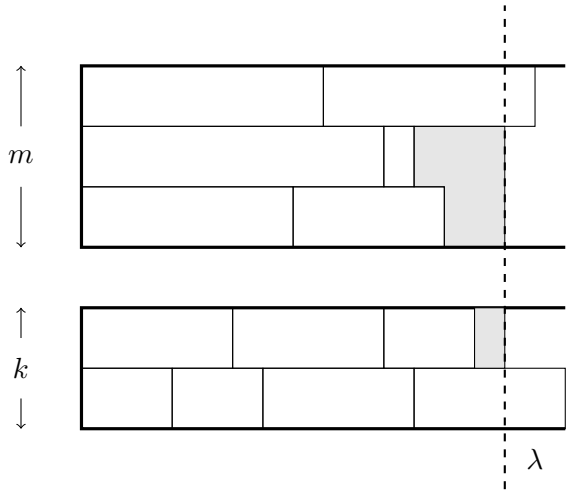


Figure 4.7: Schedule resulting from Algorithm 4.2.3 for a guess λ. The computational area on the CPUs is lower than $m\lambda$, otherwise λ is lower than C^*_{max}.

After Algorithm 4.2.3 is applied, the guess of the next step of the dual approximation has to be determined. The condition of validation of the dual approximation algorithm is here that the computational area on the CPUs must be lower than $m\lambda$. If that condition is satisfied by guess λ, then λ becomes the new upper bound in the determination of the next guess of the dual approximation, and it becomes the new lower bound if it does not satisfy the condition.

Theorem 4.2.4. *Combined with the dual approximation, Algorithm 4.2.3 has an approximation ratio of* 2*, with a time complexity in* $\mathcal{O}(n \log n)$.

Proof. We define for each task T_j a binary decision variable x_j such that $x_j = 1$ if T_j is assigned to a CPU or 0 if T_j is assigned to the GPU, as previously defined in this chapter. The makespan on the CPUs, C_{max}^{CPU}, is bounded by the following inequality:

$$C_{max}^{CPU} \leqslant \max_{1 \leqslant j \leqslant n} (\overline{p_j} x_j) + \frac{\sum_{j=1}^{n} \overline{p_j} x_j}{\sum_{j=1}^{n} x_j}$$

Let us consider one step of the dual approximation, with a guess λ satisfying the dual approximation condition, ie the computational area on the CPUs is lower than $m\lambda$. All the tasks assigned to the CPUs have a processing time lower than λ, therefore $\max_{1 \leqslant j \leqslant n} \overline{p_j} x_j \leqslant \lambda$ and $\sum_{j=1}^{n} \overline{p_j} x_j \leqslant m\lambda$ with the hypothesis that the computational area on the CPUs is lower than $m\lambda$. We obtain

$$C_{max}^{CPU} \leqslant \left(1 + \frac{m}{\sum_{j=1}^{n} x_j} \right) \lambda$$

Moreover, we can assume $\sum_{j=1}^{n} x_j > m$, otherwise the optimal solution is straightforward (one task per CPU), thus

$$C_{max}^{CPU} \leqslant 2\lambda$$

Let us examine the case of the GPUs. Let j_{last} be the index of the last task $T_{j_{last}}$ scheduled by the algorithm on the GPUs. Hence, task $T_{j_{last}}$ has no influence at all on the scheduling of all the other tasks.
Two cases hold (cf. Equation (4.2)): either task $T_{j_{last}}$ is not the last to be completed or it is. In the first case, $T_{j_{last}}$ can be removed from the schedule instance without changing the makespan. The computational area of all tasks except $T_{j_{last}}$ is smaller than $k\lambda$ thus the guarantee is the same as the one derived for the CPU schedule. In the second case, the computational area of all tasks save $T_{j_{last}}$ is also smaller than $k\lambda$ thus, when the list algorithm schedules task $T_{j_{last}}$, the least loaded of the k GPUs has a load lower than λ. Hence task $T_{j_{last}}$ ends before 2λ.

$$C_{max}^{GPU} \leqslant \begin{cases} \max_{1 \leqslant j \leqslant n | j \neq j_{last}} \left(\underline{p_j}(1 - x_j) \right) + \frac{\sum_{j=1}^{n} \underline{p_j}(1-x_j) - \underline{p_{j_{last}}}}{k} \leqslant 2\lambda \\ \left(\underline{p_j}(1 - x_j) \right) + \frac{\sum_{j=1}^{n} \underline{p_j}(1-x_j) - \underline{p_{j_{last}}}}{k} \leqslant 2\lambda \end{cases} \tag{4.2}$$

Since the makespan of the schedule is the maximum of the makespans on the CPUs and on the GPUs, we get

$$C_{max} \leqslant 2\lambda.$$

Therefore if λ satisfies the dual approximation condition, we can construct a schedule of makespan at most 2λ.
If now we suppose that λ does not satisfy the dual approximation condition, i.e. $\sum_{j=1}^{n} \overline{p_j} x_j > m\lambda$, we observe that the tasks assigned to the GPUs by Algorithm 4.2.3 have the largest acceleration factors. If we were to exchange two tasks between a CPU and a GPU to reduce the computational area on the CPUs, then the computational area on the GPUs would be increased and become greater than $k\lambda$. Therefore there is no possible assignment of the tasks that could result in a schedule of makespan λ.
With these updates on either the lower bound or the upper bound for the calculations of the guess of the dual approximation, a bisection search narrows down the value of the guess up to the optimal makespan of the schedule. Since the configuration created by Algorithm 4.2.3 is the configuration with the minimum computational area on the CPUs, we can construct a schedule of makespan at most $2C^*_{max}$, and therefore the approximation ratio of the dual approximation combined with Algorithm 4.2.3 is 2. □

Now that we have an approximation algorithm for $(Pm, Pk) \mid\mid C_{max}$, we work on improving the performance ratio for this problem. We start with problem $(Pm, P1) \mid\mid C_{max}$, then extend the results to k GPUs.

4.3 Improving the Performance Ratio for $(Pm, P1) \mid\mid C_{max}$

4.3.1 Principle of the Scheduling Algorithm

The algorithm here also uses the dual approximation technique described in Chapter 3, Section 3.2.2.2. We target here a performance ratio of $g = \frac{4}{3}$. Let λ be the current guess for the dual approximation. The key point is to show how it is possible to build a schedule of length at most $\frac{4\lambda}{3}$, starting from the assumption that there exists a schedule of length lower than λ.
The idea is to partition the set of tasks on the CPUs into two sets, each consisting of two shelves (see Figure 4.8): a first set with a shelf of length λ and the other of length $\frac{\lambda}{3}$, and a second set with two shelves of length $\frac{2\lambda}{3}$.
The partition ensures that the makespan on the CPUs is lower than $\frac{4\lambda}{3}$. If we force the makespan on the GPU to remain lower than $\frac{4\lambda}{3}$, since the tasks are independent, the scheduling strategy is straightforward when the assignment of the tasks has been determined and yields directly a solution of length at most $\frac{4\lambda}{3}$. The main problem is to assign the tasks in each shelf on the CPUs or on the GPU in order to obtain a feasible solution. This is done using dynamic programming (see Chapter 3, Section 3.2.1.3). The main steps are summarized in the following algorithmic scheme:

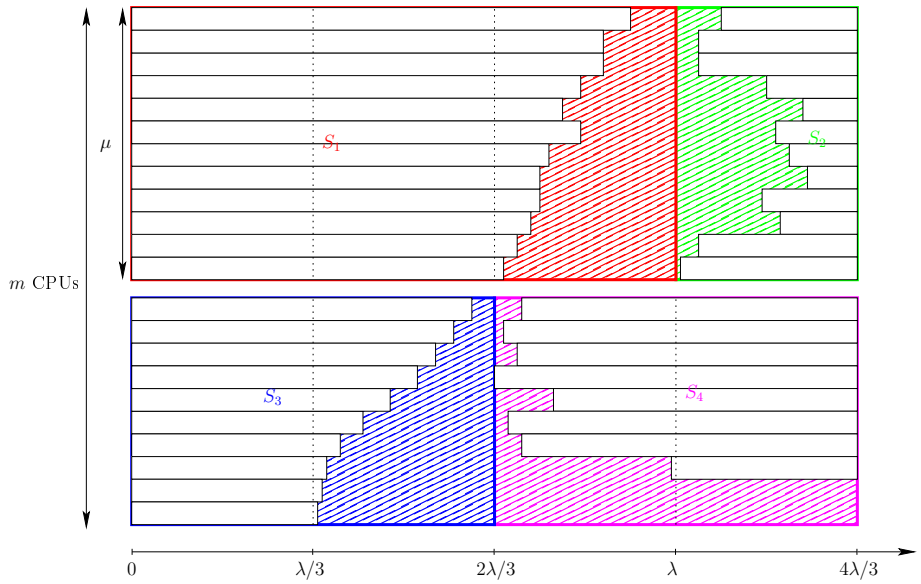


Figure 4.8: Partitioning the set of tasks on the CPUs into two sets of two shelves, the first one occupying μ CPUs, the second $m - \mu$ CPUs.

1. Compute the guess $\lambda = \frac{B_{min}+B_{max}}{2}$ where B_{min} (resp. B_{max}) is a lower (resp. upper) bound of the optimal makespan.

2. Search for an allotment of the tasks such that:

   - the total load (work) on CPUs is at most $m\lambda$,
   - the makespan on GPUs is at most λ,
   - the tasks assigned to the CPUs whose processing time is strictly greater than $\frac{2\lambda}{3}$ occupy a maximum number of CPUs denoted by μ.
   - the tasks assigned to the CPUs whose processing time is strictly greater than $\frac{\lambda}{3}$ and lower than $\frac{2\lambda}{3}$ can be assigned two by two to a maximum number of CPUs denoted by $\mu'/2$.
     The total number of CPUs must not exceed m, i.e. $\mu + \mu'/2 \leqslant m$,
   - the tasks assigned to the CPUs with processing time lower that $\frac{\lambda}{3}$ can be scheduled such that the induced makespan on the CPUs is at most equal to $\frac{4\lambda}{3}$.

3. If such an allotment does not exist, adjust the bound B_{min} to λ and restart the process (Step 1).

4. If such an allotment exists, build the corresponding schedule with sets of shelves such that the makespan is lower than $\frac{4}{3}\lambda$, adjust the bound B_{max} to λ and restart the process.

4.3.2 Structure of an Optimal Schedule

We introduce an assignment function $\pi(j)$ of a task T_j which corresponds to the processor where the task is processed. The set $\mathcal{C}$ (resp. $\mathcal{G}$) is the set of all the CPUs (resp. GPU). Therefore, if a task T_j is assigned to a CPU, we can write $\pi(j) \in \mathcal{C}$. We define W_C as the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the processing times of the tasks assigned to the CPUs: $W_C = \sum_{j \,/\, \pi(j) \in \mathcal{C}} \overline{p_j}$. This corresponds to the computational load of all the CPUs.
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 4.3.1. *In an optimal solution, the execution time of each task is at most λ, and the computational area on the CPUs is at most $m\lambda$.*

Proposition 4.3.2. *In an optimal solution, if there exist two tasks executed on the same CPU such that one of these tasks has an execution time greater than $\frac{2\lambda}{3}$, then the other one has an execution time lower than $\frac{\lambda}{3}$.*

Proposition 4.3.3. *Two tasks with processing times on CPU greater than $\frac{\lambda}{3}$ and lower than $\frac{2\lambda}{3}$ can be executed on the same CPU within a time at most $\frac{4\lambda}{3}$.*

The basic idea of the solution that we propose comes from the analysis of the shape of an optimal schedule. From Proposition 4.3.2, the tasks whose execution times on CPU are strictly greater than $\frac{2\lambda}{3}$ do not use more than m CPUs, and hence can be executed concurrently in the first set in a shelf denoted by S_1, occupying μ CPUs (see Figure 4.8). The tasks whose execution times are lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ on CPU cannot be executed on the μ CPUs occupied by S_1 from Proposition 4.3.1. Moreover, from Proposition 4.3.3, $2(m-\mu)$ of these tasks on CPU can be executed in time at most $\frac{4\lambda}{3}$ on the remaining $(m-\mu)$ CPUs in the second set and fill two shelves S_3 and S_4 of equal length $\frac{2\lambda}{3}$.
The tasks remaining to be assigned to the CPUs have a processing time lower than $\frac{\lambda}{3}$. The μ longest remaining tasks are assigned to the first set on the CPUs in another shelf denoted by S_2. The length of S_2 is $\frac{\lambda}{3}$.
W_L will denote the computational area on the CPUs remaining idle after this assignment in the schedule of length $\frac{4\lambda}{3}$. W_L corresponds to the stripped areas in Figure 4.8. Regarding the question of how the remaining tasks fit in the constructed schedule, we state the following lemma:

Lemma 4.3.4. *The tasks remaining to be assigned on the CPUs after the construction of S_1, S_2, S_3, S_4 fit in the remaining free computational space W_L between these shelves.*

Proof. The tasks remaining to be assigned after the construction of $S_1, \ldots, S_4$ all have a processing time lower than $\frac{\lambda}{3}$ by construction and they necessarily fit into the remaining

computational space W_L, otherwise the schedule would not satisfy Property 4.3.1. The following algorithm can be used to schedule these tasks:

- Consider the remaining tasks ordered by decreasing order of processing time on CPU $T_1, \ldots, T_f$, f being the total number of tasks remaining to be assigned.
- At each step i, $i = 1, \ldots, f$, assign task T_i to the least loaded processor, at the latest possible date. Update its load.

At each step, the least loaded processor has a load at most λ; otherwise it would contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Property 4.3.1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{3}$ and can contain the task T_i, which proves the correctness of the scheduling algorithm. □

4.3.3 Partitioning the Tasks into Shelves

In this section, we detail how to fill the shelves on the CPUs (see Figure 4.8) and to assign the tasks to the GPU by specifying an initial assignment of the tasks to the processors.
In order to obtain a 2-sets and 4-shelves schedule on the CPUs, we look for an assignment satisfying the following constraints:

- (C_1) The total computational area W_C on the CPUs is at most $m\lambda$.
- (C_2) The set $\mathcal{T}_1$ of tasks on the CPUs with an execution time strictly greater than $\frac{2\lambda}{3}$ in the assignment, to be scheduled in S_1, uses a total of at most m processors. We still denote by μ the number of processors they use.
- (C_3) The set $\mathcal{T}_2$ of tasks on the CPUs with an execution time lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ in the assignment, to be scheduled in S_3 or S_4, uses a total of at most $2(m - \mu)$ processors.
- (C_4) The total execution time of the tasks on the GPU is lower than $\frac{4\lambda}{3}$.

Let us notice that if Constraint (C_3) is satisfied, then Constraint (C_2) will also be satisfied. Hence, Constraint (C_2) is relaxed.
We consider for each task T_j a binary decision variable x_j such that $x_j = 1$ if T_j is assigned to a CPU or 0 if T_j is assigned to the GPU, as this was previously done in this chapter.
Determining if an allotment satisfying (C_1), (C_3) and (C_4) exists reduces to solving a two-dimensional knapsack problem that can be formulated as follows:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{4.3}$$

$$\text{s.t. } \frac{1}{2} \sum_{2\lambda/3 \geqslant \overline{p_j} > \lambda/3} x_j + \sum_{\overline{p_j} > 2\lambda/3} x_j \leqslant m \tag{4.4}$$

$$\sum_{j=1}^{n} \underline{p_j} (1 - x_j) \leqslant \frac{4\lambda}{3} \tag{4.5}$$

$$x_j \in \{0, 1\}, \quad \forall j \in \{1, \ldots, n\} \tag{4.6}$$

Equation (4.3) represents the minimal workload on all the CPUs. Constraint (4.4) imposes that no more than m tasks can be executed on the CPUs with a processing time greater than $\frac{2\lambda}{3}$, we note $\mu = \sum_{\overline{p_j} > 2\lambda/3} x_j$ their number; and that there cannot be more than $2(m - \mu)$ tasks on the CPUs with a processing time lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ (cf. Constraints (C_2)). Constraint (4.5) imposes an upper bound on the makespan of the GPU which is $\frac{4\lambda}{3} = \lambda + \frac{\lambda}{3}$ (cf. (C_4)). This problem corresponds to a two-dimensional knapsack problem.

We propose a dynamic programming algorithm in $\mathcal{O}(n^2 m^2)$ to solve the knapsack problem. For this purpose, we first discretize the processing times of the tasks on the GPU. We introduce $\nu_j = \left\lfloor \frac{\underline{p_j}}{\lambda/(3n)} \right\rfloor$ to represent the number of integer time intervals of length $\frac{\lambda}{3n}$ required for a task T_j if it is executed on the GPU, as shown in Figure 4.9. $N = \sum_{\pi(j) \in \mathcal{G}} \nu_j$ denotes the total integer number of these intervals on the GPU. We thus define the error on the processing time of each task $\epsilon_j = \underline{p_j} - \nu_j \frac{\lambda}{3n}$ induced by this time discretization.

This result allows us to consider only N states in the dynamic programming regarding the workload on the GPU. The error ϵ_j on each task is at most $\frac{\lambda}{3n}$ so if all the tasks were assigned to the GPU, we would have underestimated the processing time on the GPU by at most $n\frac{\lambda}{3n} = \frac{\lambda}{3}$. We have

$$\begin{aligned} N &= \sum_{\pi(j) \in \mathcal{G}} \nu_j \\ &= \sum_{j=1}^{n} \left\lfloor \frac{\underline{p_j}}{\lambda/(3n)} \right\rfloor (1 - x_j) \\ &= \sum_{j=1}^{n} \frac{\underline{p_j} - \epsilon_j}{\lambda/(3n)} (1 - x_j) \end{aligned}$$

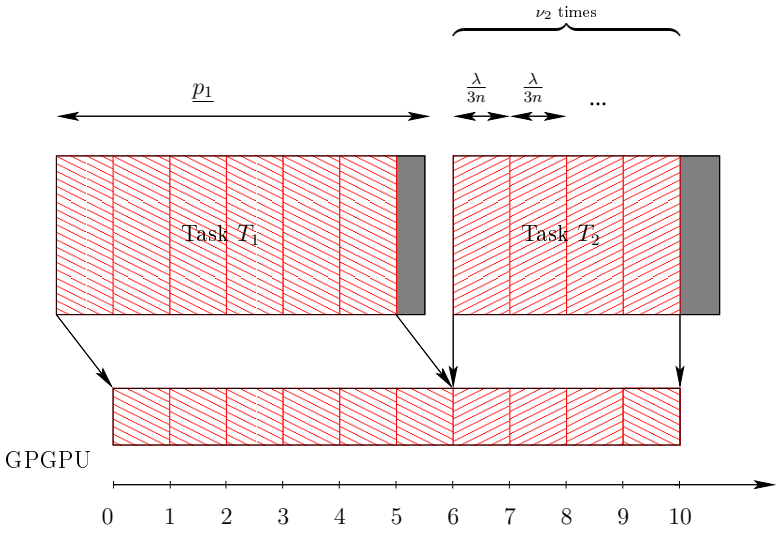


Figure 4.9: Rounded assignment of two tasks T_1 with $p_1 = 6.5$ and T_2 with $p2 = 4.7$ on a GPU.

Then we can rewrite Constraint (4.5), $\sum_{j=1}^{n} \underline{p_j}\,(1 - x_j) \leqslant \frac{4\lambda}{3}$, as

$$\sum_{j=1}^{n} \left(\underline{p_j} - \epsilon_j\right)(1 - x_j) + \sum_{j=1}^{n} \epsilon_j\,(1 - x_j) \leqslant \frac{4\lambda}{3},$$

$$\sum_{j=1}^{n} \left(\underline{p_j} - \epsilon_j\right)(1 - x_j) \leqslant \frac{4\lambda}{3} - \sum_{j=1}^{n} \epsilon_j\,(1 - x_j)\,,$$

$$\frac{\lambda}{3n} N \leqslant \frac{4\lambda}{3} - \sum_{j=1}^{n} \epsilon_j\,(1 - x_j)\,.$$

In order to always satisfy Constraint (4.5), we have to consider that we are in the worst possible case, i.e. all the tasks are assigned to the GPU and the error for each task is $\frac{\lambda}{3n}$. We obtain

$$\frac{\lambda}{3n} N \leqslant \frac{4\lambda}{3} - \frac{\lambda}{3}.$$

Therefore Constraint (4.5) becomes:

$$N = \sum_{\pi(j) \in \mathcal{G}} \nu_j \leqslant 3n \tag{4.7}$$

Remark 4.3.5. This discretization technique is used again in Chapter 5, Section 5.2.4 as well as in Chapter 6, Section 6.2.2. It is a crucial part of the algorithm, since this allows us to achieve a polynomial time complexity.

The approximated makespan of the GPU is at most λ and thus, with the underestimation detailed above, the makespan of the GPU remains lower than $\frac{4\lambda}{3}$. Some of the schedules produced by this algorithm can therefore seem non optimal because of this approximation, with a makespan on the GPU remaining lower than $\frac{4\lambda}{3}$.
We define $W_C(j, \mu, \mu', N)$ as the minimum sum of all the processing times of the tasks on the CPUs when the first j tasks are considered, with among the tasks assigned to the CPUs, μ of them having processing times greater than $\frac{2\lambda}{3}$ and μ' of them having processing times lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ and where N time intervals are occupied on the GPU.
We use a dynamic programming which allows us to compute the value of $W_C(j, \mu, \mu', N)$ using the values of W_C for $j-1$ tasks.
If task T_j is put on a CPU, the resulting sum of all the processing times of the tasks on the CPUs is then

$$F_{CPU}(j, \mu, \mu', N) = \overline{p_j} + W_C\left(j-1, \mu - I_{\left(\overline{p_j} > \frac{2\lambda}{3}\right)}, \mu' - I_{\left(\frac{2\lambda}{3} \geqslant \overline{p_j} > \frac{\lambda}{3}\right)}, N\right)$$

where $I_{\left(\overline{p_j} > \frac{2\lambda}{3}\right)}$ and $I_{\left(\frac{2\lambda}{3} \geqslant \overline{p_j} > \frac{\lambda}{3}\right)}$ are indicating functions:

$$I_{\left(\overline{p_j} > \frac{2\lambda}{3}\right)} = \begin{cases} 1 & \text{if } \overline{p_j} > \frac{2\lambda}{3} \\ 0 & \text{otherwise} \end{cases}$$

$$I_{\left(\frac{2\lambda}{3} \geqslant \overline{p_j} > \frac{\lambda}{3}\right)} = \begin{cases} 1 & \text{if } \frac{2\lambda}{3} \geqslant \overline{p_j} > \frac{\lambda}{3} \\ 0 & \text{otherwise} \end{cases}$$

If task T_j is put on the GPU, the sum of all the processing times of the tasks on the CPUs is then

$$W_C\left(j-1, \mu, \mu', N - \nu_j\right)$$

The dynamic programming is then based on the following recursive equation:

$$\underset{\substack{1 \leqslant j \leqslant n, 1 \leqslant \mu \leqslant m \\ 1 \leqslant \mu' \leqslant 2(m-\mu),\ 0 \leqslant N \leqslant 3n}}{W_C\left(j, \mu, \mu', N\right)} = \min\left(F_{CPU}(j, \mu, \mu', N), W_C(j-1, \mu, \mu', N - \nu_j)\right)$$

In order to satisfy the constraints imposing that $\mu \leqslant m$ tasks are processed on a CPU with a processing time greater than $\frac{2\lambda}{3}$ and no more than $2(m-\mu)$ tasks are processed on a CPU with a processing time lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ and that the makespan of the GPU is not greater than $\frac{4\lambda}{3}$, we have border conditions:

$$W_C\left(j, \mu, \mu', N\right) = +\infty \begin{cases} \text{if } \mu > m \\ \text{if } \mu' > 2(m-\mu) \\ \text{if } \sum\limits_{\pi(j) \in \mathcal{G}} \nu_j > 3n \end{cases}$$

The optimal value of the computational area W_C on the CPUs is then given by

$$W_C^* = \min_{0\leqslant\mu\leqslant m,\ 0\leqslant\mu'\leqslant 2(m-\mu),\ 0\leqslant N\leqslant 3n} W_c(n,\mu,\mu',N)$$

If W_C^* is greater than $m\lambda$, then there exists no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation framework. Otherwise, the guess λ is large enough, we construct a feasible solution with a makespan at most $\frac{4\lambda}{3}$, with the corresponding shelves on the CPUs and the corresponding μ, μ' and N values.
The dynamic programming algorithm represents one step of the dual-approximation algorithm, with a fixed guess λ. A binary search is then used to try different guesses to approach the optimal makespan as explained in Section 4.3.1.

Cost Analysis. Solving the dynamic program for a fixed value of λ requires to consider $\mathcal{O}(n^2m^2)$ states. Since $1 \leqslant j \leqslant n$, $1 \leqslant \mu \leqslant m$, $1 \leqslant \mu' \leqslant 2(m-\mu)$ and $0 \leqslant N \leqslant 3n$, the time complexity of each step of the binary search is $\mathcal{O}(n^2m^2)$.

We have an approximation algorithm for $(Pm, P1) \mid\mid C_{max}$ with a performance ratio of $\frac{4}{3}$. Now we extend it to the case with $k > 1$ GPUs.

4.4 Extending the $\frac{4}{3}$-appproximation Algorithm to the multi-GPUs case

The algorithm described in the previous section can be extended to the problem with $k \geqslant 2$ GPU, using the same structure for the GPUs as we did with the CPUs. The target performance ratio is $\frac{4}{3} + \frac{1}{3k}$.
An analysis of an optimal solution leads to the following properties:

Proposition 4.4.1. *In an optimal solution, the execution time of each task is at most λ and the computational area on the GPUs is at most $k\lambda$.*

Proposition 4.4.2. *In an optimal solution, if there exist two tasks executed on the same GPU such that one of these tasks has an execution time on GPU greater than $\frac{2\lambda}{3}$, then the other one has an execution time lower than $\frac{\lambda}{3}$.*

Proposition 4.4.3. *Two tasks with processing times on GPU greater than $\frac{\lambda}{3}$ and lower than $\frac{2\lambda}{3}$ can be executed on the same GPU within a time at most $\frac{4\lambda}{3}$.*

Using the same notations as before (the set $\mathcal{G}$ is now the set of all the GPUs, and k sets of $3n$ integer time intervals will be considered for the discretization phase), the problem can be formulated in the same way, with the following constraints on the GPUs:

$$\frac{1}{2}\sum_{2\lambda/3\geqslant \underline{p_j} > \lambda/3} x_j + \sum_{\underline{p_j} > 2\lambda/3} x_j \leqslant k,$$

$$N = \sum_{\pi(j)\in\mathcal{G}} \nu_j \leqslant 3kn.$$

Then, the problem becomes:

$$
\begin{aligned}
W_C^* = \min & \sum_{j=1}^{n} \overline{p_j} x_j \\
\text{s.t. } & \frac{1}{2} \sum_{2\lambda/3 \geqslant \overline{p_j} > \lambda/3} x_j + \sum_{\overline{p_j} > 2\lambda/3} x_j \leqslant m \\
& \frac{1}{2} \sum_{2\lambda/3 \geqslant \underline{p_j} > \lambda/3} x_j + \sum_{\underline{p_j} > 2\lambda/3} x_j \leqslant k \\
& N = \sum_{\pi(j) \in \mathcal{G}} \nu_j \leqslant 3kn \\
& x_j \in \{0,1\}, \quad \forall j \in \{1,\ldots,n\}
\end{aligned}
$$

If all the tasks are assigned to one GPU, the error $\epsilon = \sum_{j=1}^{n} \epsilon_j$ is still $\frac{\lambda}{3}$, so the computational area of the GPUs is lower than $\left(k + \frac{1}{3}\right)\lambda$.
The same partition of the tasks on the CPUs on two sets, each consisting of two shelves S_1 and S_2, and S_3 and S_4 can be done and leads to a schedule on the CPUs with a makespan lower than $\frac{4\lambda}{3}$.
Among the tasks assigned to the GPUs, the distribution between the different processors can be made in two sets, each consisting of two shelves S_5 and S_6, and S_7 and S_8. The first constraint we formulated on the GPUs sets that at most k tasks of processing times greater than $\frac{2\lambda}{3}$, that are put in the shelf S_5 which length is λ. We note κ the number of processors occupied on S_5. The tasks with processing times lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ are then assigned to the shelves S_7 and S_8, their number being lower than $2(k-\kappa)$. Finally, the tasks remaining to be assigned to the GPUs have a processing time lower than $\frac{\lambda}{3}$. The κ longest remaining tasks are assigned to the first set on the GPUs in another shelf denoted by S_6. The length of S_6 is $\frac{\lambda}{3} + \frac{\lambda}{3k}$. W_R denotes the computational area on the GPUs remaining idle after this assignment in the schedule of length $\frac{4\lambda}{3} + \frac{\lambda}{3k}$ (see Figure 4.10).
After the discretization of the processing times of the tasks assigned to the GPUs, the approximated computational area of the GPUs is at most $k\lambda$ and thus, the full computational area on GPU remains lower than $k\lambda + \frac{\lambda}{3}$. This allows us to answer the question of how the remaining tasks fit in the constructed schedule with the following lemma:

Lemma 4.4.4. *The tasks remaining to be assigned on the GPUs after the construction of S_5, S_6, S_7, S_8 fit in the remaining free computational space W_R between these shelves.*

Proof. The proof is similar to the one of Lemma 4.3.4. If we modify the starting time of the tasks of S_6, currently λ, so that all the working processors complete their tasks at

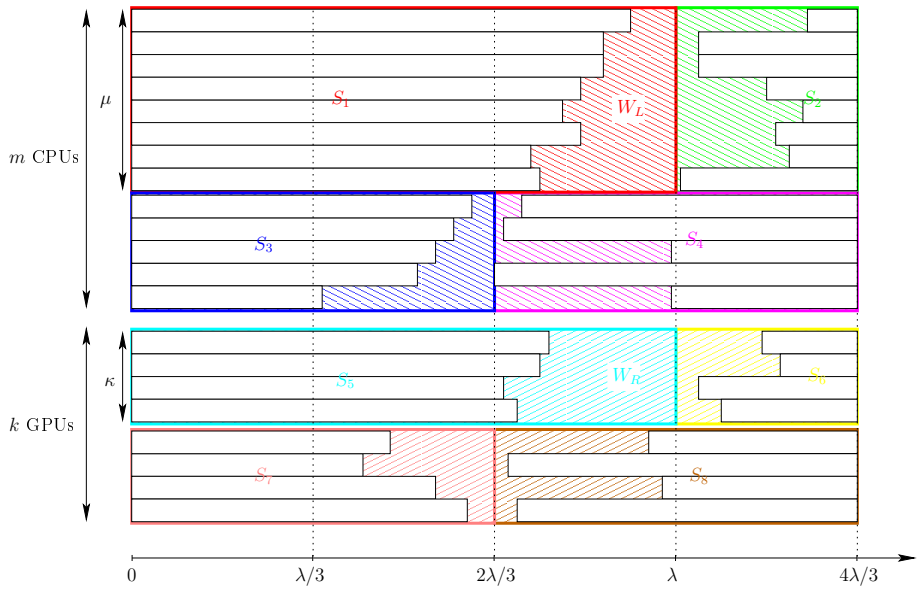


Figure 4.10: All the shelves on CPUs and GPUs.

$\frac{4\lambda}{3} + \frac{\lambda}{3k}$, creating an idle time interval between the end of S_5 and the starting time of S_6, the load of a GPU is equal to $\frac{4\lambda}{3} + \frac{\lambda}{3k}$ minus the length of the idle time interval.
With the same arguments as for Lemma 4.3.4, the only problem that may occur is if a task T_i remaining to be assigned cannot be completed before the starting time of the tasks of S_6. But at each step, the least loaded processor has a load at most $\lambda + \frac{\lambda}{3k}$ since the total work area of the tasks is bounded by $k\left(\lambda + \frac{\lambda}{3k}\right)$. Hence, the idle time interval on the least loaded GPU has a length at least $\frac{\lambda}{3}$ and can contain the task T_i. □

We can conclude that the approximation algorithm can be extended to the problem with $k \geqslant 2$ GPUs with a performance guarantee of $\frac{4}{3} + \frac{1}{3k}$. In order to solve each step of the binary search, we have to consider $\mathcal{O}\left(n^2k^3m^2\right)$ states, since $1 \leqslant j \leqslant n$, $1 \leqslant \mu \leqslant m$, $1 \leqslant \mu' \leqslant 2(m-\mu)$, $1 \leqslant \kappa \leqslant k$, $1 \leqslant \kappa' \leqslant 2(k-\kappa)$, and $0 \leqslant N \leqslant 3kn$.

4.5 Summary

In this chapter, we have presented two algorithms for problem $(P1, P1) \mid\mid C_{max}$, and two algorithms for the more generic problem $(Pm, Pk) \mid\mid C_{max}$, one with performance ratio of 2 and the other with a performance ratio of $\frac{4}{3}$ in the case with one GPU, and a ratio of $\frac{4}{3} + \frac{1}{3k}$ in the case of $k \geqslant 2$ GPUs, all of them being new contributions in scheduling theory, as summarized in Table 4.1.
The algorithm with a performance ratio of $\frac{4}{3}$ can be generalized into two families of approximation algorithms for problem $(Pm, Pk) \mid\mid C_{max}$ with different approximation

Problem	Algorithm optimality ratio	Algorithm cost
$(P1, P1) \mid\mid C_{max}$	$\frac{3}{2}$	$\mathcal{O}(n \log n)$
	$1 + \epsilon$	FPTAS
$(Pm, Pk) \mid\mid C_{max}$	2	$\mathcal{O}(n \log n)$
	$\frac{4}{3} + \frac{1}{3k}$	$\mathcal{O}\left(n^2 m^2 k^3\right)$

Table 4.1: Problems studied in this chapter and the algorithms developed for them.

ratios and different time complexities. These families are detailed in the following chapter.

Chapter 5

Two families of algorithms for $(Pm, Pk) \mid\mid C_{max}$ with ratios of $\frac{2q+1}{2q} + \frac{1}{2qk}$ and $\frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k}$

This chapter presents the generalization of the approximation algorithm presented in Chapter 4, Sections 4.3 and 4.4, into two families of approximation algorithms using dual approximation and dynamic programming. For the case of m CPUs and one GPU, we have a family of approximation algorithms that achieve ratios of $\frac{2q+1}{2q} + \epsilon$ for any integer $q \geqslant 1$, with computational costs in $\mathcal{O}(n^2 m^q)$ per step of dual approximation. This family is extended to the case of $k \geqslant 2$ GPUs, and the approximation ratios become $\frac{2q+1}{2q} + \frac{1}{2qk} + \epsilon$ (for any integer $q \geqslant 1$). The associated cost is in $\mathcal{O}(n^2 k^{q+1} m^q)$ per step of dual approximation. The other family of algorithms developed has approximation ratios of $\frac{2(q+1)}{2q+1} + \epsilon$ ($q \geqslant 0$) with one GPU and $\frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k} + \epsilon$ ($q \geqslant 0$) with $k \geqslant 2$ GPUs. The associated costs are respectively in $\mathcal{O}(n^2 m^{q+1})$ for a single GPU and in $\mathcal{O}(n^2 k^{q+2} m^{q+1})$ for $k \geqslant 2$, per step of the dual approximation.
These two families are a major contribution of this PhD.
In order to be as clear as possible, we choose to present the entire method for the families of algorithms, even if some steps are similar to the some used in the algorithms of Chapter 4 that were the basis for the construction of these families of algorithms. As a result, even if this chapter is practically self content, the notions already introduced in the manuscript are not always recalled.

5.1 Rationale of the Solving Method

The proposed algorithms are again based on the dual approximation technique (see Chapter 3, Section 3.2.2.2). For the $(Pm, Pk)||C_{\max}$ problem, we propose two families of algorithms that are developed in this chapter. Both families are complementary in the

sense that the sequence of approximation ratios are interleaved (see summary at the end of this chapter).

- In a first family of algorithms, we target $g = \frac{2q+1}{2q}$ (resp. $g = \frac{2q+1}{2q} + \frac{1}{2qk}$) in the case of one (resp. $k \geqslant 2$) GPU(s), for any given $q > 0$.
- In a second family of algorithms, we target $g = \frac{2(q+1)}{2q+1}$ (resp. $g = \frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k}$) for one (resp. $k \geqslant 2$) GPU(s), for any given $q \geqslant 0$.

For the sake of clarity, we will consider in what follows the problem with a single GPU ($k = 1$) and focus on the construction of the algorithms of the first family. Let λ be the current guess for the dual approximation. The key point is still to show how it is possible to build a schedule of length at most $\frac{2q+1}{2q}\lambda$, $q > 0$, starting from the assumption that there exists a schedule of length lower than λ.
The idea is to partition the set of tasks on the CPUs into several sets of two shelves, starting with a shelf of length λ and the other of length $\frac{\lambda}{2q}$ ($q > 0$), and for each new set, gradually lowering the length of the first shelf by $\frac{\lambda}{2q}$, the length of the second shelf is $\frac{2q+1}{2q}\lambda$ minus the length of the first shelf, while keeping the makespan on the GPUs lower than the target bound $\frac{2q+1}{2q}\lambda$. In Figure 5.1, an example is given for the CPUs in the case where $q = 2$: the first set has a shelf of length λ and the other of length $\frac{\lambda}{4}$, and the second set has a shelf of length $\frac{3\lambda}{4}$ and the other of length $\frac{2\lambda}{4}$.

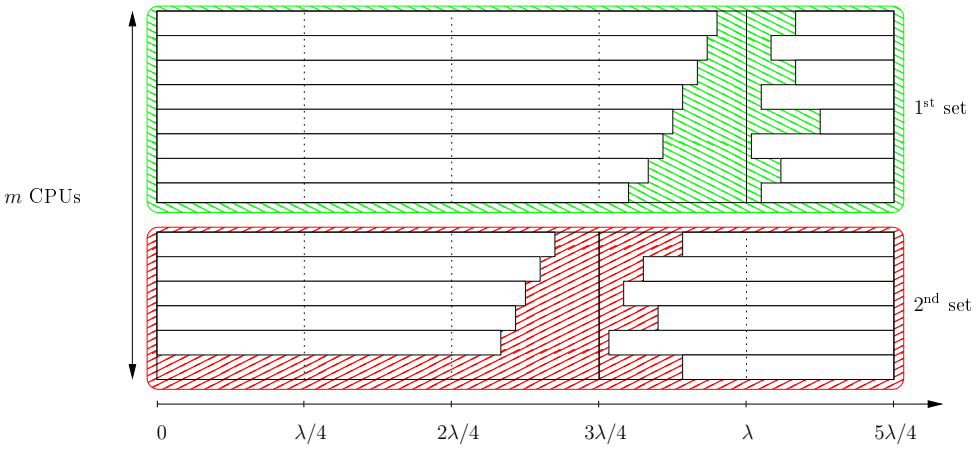


Figure 5.1: Two sets of two shelves for $g = 5/4$ ($q = 2$), with $m = 14$ CPUs: the first set with two shelves of length λ and $\lambda/4$, and the second one with two shelves of length $3\lambda/4$ and $2\lambda/4$.

The assignment of the tasks to the different shelves on the CPUs is done according to their processing time on CPU, each shelf corresponding to a time interval of length $\frac{\lambda}{2q}$ for the processing times.

- For the first shelf of each set, we have $\overline{p_j} \in \left(\frac{(2q-h)\lambda}{2q}, \frac{(2q-h+1)\lambda}{2q}\right]$ in the shelf of length $\frac{(2q-h+1)\lambda}{2q}$, $h \in \{1, \dots q\}$.

- For the second shelf of each set, we have $\overline{p_j} \in \left(\frac{(q-h)\lambda}{2q}, \frac{(q-h+1)\lambda}{2q}\right]$ in the shelf of length $\frac{(q-h+1)\lambda}{2q}$, $h \in \{1, \dots q\}$.

The partition ensures that the makespan on the CPUs is lower than $\frac{2q+1}{2q}\lambda$. Depending on the numbers of tasks with processing times in the time intervals of length $\frac{\lambda}{2q}$, there can be some CPUs left idle or some second shelves with gaps. These gaps are filled with tasks with smaller processing times and the details of this assignment are explained in the next section.
The tasks being independent, the scheduling strategy is straightforward when the assignment of the tasks has been determined and yields a solution of length at most $\frac{(2q+1)\lambda}{2q}$. The main issue is to assign the tasks in each shelf on the CPUs or on the GPU in order to obtain a feasible solution. This will be done using a dynamic programming algorithm. The main steps are summarized in the following algorithmic scheme:

1. Compute the guess $\lambda = \frac{B_{min}+B_{max}}{2}$ where B_{min} (resp. B_{max}) is a lower (resp. upper) bound of the optimal makespan.

2. Search for an assignment of the tasks such that:

   - the total load (work) on CPUs is at most $m\lambda$,
   - the makespan on the GPU is at most λ,
   - the tasks assigned to the CPUs whose processing time is in the time interval $\left(\frac{(2q-h)\lambda}{2q}, \frac{(2q-h+1)\lambda}{2q}\right]$, $h \in \{1, \dots, q\}$, occupy a maximum number of CPUs denoted by μ_h.
     The total number of processors used over all the intervals must not exceed m, i.e. $\sum_{h=1}^{q} \mu_h \leqslant m$,
   - the tasks assigned to the CPUs with processing time lower that $\frac{\lambda}{2}$ can be scheduled such that the induced makespan on CPU will be at most equal to $\frac{(2q+1)\lambda}{2q}$.

3. If such an assignment does not exist, adjust the bound B_{min} to λ and restart the process (Step 1).

4. If such an assignment exists, build the corresponding schedule with sets of shelves such that the makespan is lower than $\frac{2q+1}{2q}\lambda$, adjust the bound B_{max} to λ and restart the process.

In the following, we first analyze the structure of an optimal solution (Section 5.2.1), leading to a partition of the tasks into several shelves. Then, we show how the shelves are built (Section 5.2.2). The way to determine such a partition is finally described in Section 5.2.3.

5.2 Theoretical Analysis

In this section, we consider the problem with a single GPU ($k = 1$) to describe the first family of algorithms.

5.2.1 Structure of an Optimal Schedule of Length at most λ

As in Chapter 4, Section 4.3.2, we introduce an assignment function $\pi(j)$ of a task T_j which corresponds to the processor where the task is processed. The set $\mathcal{C}$ (resp. $\mathcal{G}$) is the set of all the CPUs (resp. the GPU). Therefore, if task T_j is assigned to a CPU, we can write $\pi(j) \in \mathcal{C}$. We define W_C as the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the processing times of the tasks assigned to the CPUs: $W_C = \sum_{j\,/\,\pi(j)\in\mathcal{C}} \overline{p_j}$. This corresponds to the computational load of all the CPUs.
Since we assume at each step of the dual approximation that there exists a schedule of length at most λ, we state below some straightforward properties of a feasible schedule of length at most λ. These properties will help in the construction of the solutions in the general solving framework.

Proposition 5.2.1. *In an optimal solution, the execution time of each task is at most λ, and the computational area on the CPUs is at most $m\lambda$.*

Proposition 5.2.2. *In an optimal solution, if there exist two tasks T_i, T_j executed on the same CPU, such that $\overline{p_i} > \frac{(2q-1)\lambda}{2q}$, then $\overline{p_j} \leqslant \frac{\lambda}{2q}$.*

Proposition 5.2.3. *If there exist two tasks T_i, T_j processed on the same CPU, such that $\frac{(2q-2)\lambda}{2q} < \overline{p_i} \leqslant \frac{(2q-1)\lambda}{2q}$, then $\overline{p_j} \leqslant \frac{2\lambda}{2q}$.*

This can be formulated more generally in the following property.

Proposition 5.2.4. *If there exist two tasks T_i, T_j executed on the same CPU, such that $\frac{(2q-h)\lambda}{2q} < \overline{p_i} \leqslant \frac{(2q-h+1)\lambda}{2q}$, then $\overline{p_j} \leqslant \frac{h\lambda}{2q}$, for $h \in \{1, \ldots, q\}$.*

Property 5.2.4 can be derived in specific properties similar to Properties 5.2.2 and 5.2.3 for each time interval of length $\frac{\lambda}{2q}$, i.e. $\overline{p_j} \in \left(\frac{(2q-h)\lambda}{2q}, \frac{(2q-h+1)\lambda}{2q}\right]$, $h \in \{1, \ldots q\}$.
Property 5.2.2 corresponds to the case $h = 1$, and Property 5.2.3 to the case $h = 2$. The case $h = q$ corresponds to the following property:

Proposition 5.2.5. *If there exist two tasks T_i, T_j processed on the same CPU, and if $\frac{\lambda}{2} = \frac{q\lambda}{2q} < \overline{p_i} \leqslant \frac{(q+1)\lambda}{2q}$, then $\overline{p_j} \leqslant \frac{\lambda}{2}$.*

In the following section, we describe how to build the shelves using the previous properties.

5.2.2 Building the Shelves

The properties presented in the previous section provide strong characteristics on the structure of optimal schedules. Based on these properties, we describe in what follows the partition of the tasks into shelves for the m CPUs.
Each schedule is composed of q sets of two shelves (S_i, S'_i) on the CPUs, for $i = 1, \ldots, q$. Without loss of generality, we consider that the tasks in S_i (resp. S'_i) are shifted to the left (resp. to the right), for $i = 1, \ldots, q$. Figure 5.2 illustrates the structure of the schedule for $q = 2$. We start by building $S_1, \ldots, S_q$ and continue with $S'_q, \ldots, S'_1$.

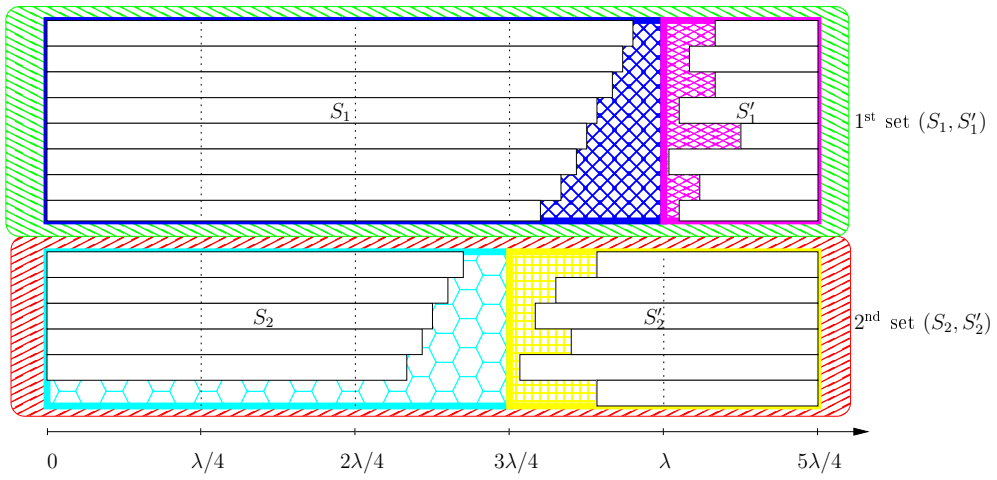


Figure 5.2: Example for $g = 5/4$ with two sets of two shelves (S_1, S'_1) and (S_2, S'_2).

Building S_1. From Property 5.2.2, the tasks assigned to a CPU whose execution times are strictly greater than $\frac{(2q-1)\lambda}{2q}$ do not use more than m CPUs, and hence can be executed concurrently. These tasks are assigned to the first set, in shelf S_1, of length λ. μ_1 denotes the number of CPUs occupied by these tasks (see Figure 5.3).

Building S_2. The tasks assigned to a CPU whose execution times are lower than $\frac{(2q-1)\lambda}{2q}$ and strictly greater than $\frac{(2q-2)\lambda}{2q}$ cannot be executed on the μ_1 CPUs occupied by S_1 from Property 5.2.1. Therefore, these tasks cannot be assigned to the first set, they are assigned to the second set, in shelf S_2, of length $\frac{(2q-1)\lambda}{2q}$. From Property 5.2.3, they

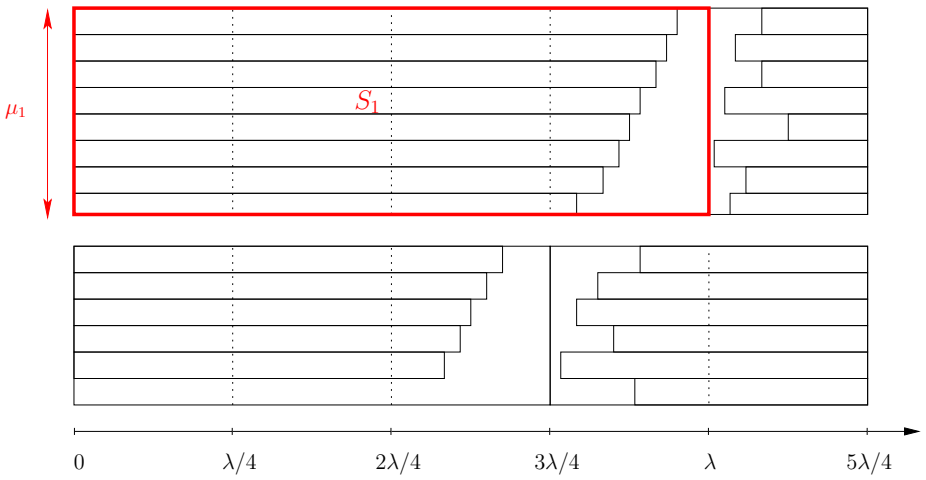


Figure 5.3: Example for $g = 5/4$ with $m = 14$, $\mu_1 = 8$ CPUs

are processed by at most $m - \mu_1$ CPUs. μ_2 denotes the number of CPUs used in S_2 (see Figure 5.4).

Building S_h, $h \in \{3, \ldots, q\}$. The tasks assigned to a CPU whose execution times are lower than $\frac{(2q-h+1)\lambda}{2q}$ and strictly greater than $\frac{(2q-h)\lambda}{2q}$ cannot be executed on the $\sum_{l=1}^{h-1} \mu_l$ CPUs occupied by $S_1, \ldots, S_{h-1}$ from Property 5.2.1. These tasks cannot be assigned to any of the first $h-1$ sets, they are assigned to shelf S_h of the h^{th} set, of length $\frac{(2q-h+1)\lambda}{2q}$. From Property 5.2.4, they occupy at most $m - \sum_{l=1}^{h-1} \mu_l$ CPUs. μ_h denotes the number of CPUs used in S_h.

Coupling Constraint. As the number of available CPUs is m, we have to ensure that:

$$\sum_{l=1}^{q} \mu_l \leqslant m \tag{5.1}$$

The tasks assigned to a CPU but not to shelves $S_1, \ldots, S_q$ have execution times lower than $\frac{\lambda}{2}$ and can be assigned to shelves $S'_1, \ldots, S'_q$ or to the remaining idle CPUs if $\sum_{l=1}^{q} \mu_l < m$. We describe in what follows the construction of the S'_i shelves starting from $i = q$ to 1, and the case of CPUs left idle.

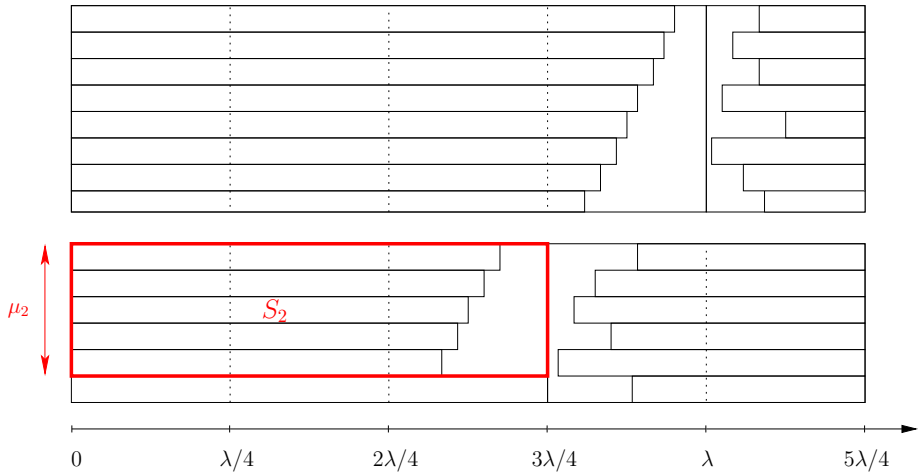


Figure 5.4: Example for $g = 5/4$ with $m = 14$, $\mu_1 = 8$, $\mu_2 = 5$ CPUs

Building S'_q. The tasks assigned to a CPU whose execution times are lower than $\frac{q\lambda}{2q}$ and strictly greater than $\frac{(q-1)\lambda}{2q}$ can only be executed on idle CPUs or after a task assigned to S_q, in the shelf S'_q of length $\frac{\lambda}{2}$ (see Figure 5.5). They satisfy the following constraint

$$\sum_{j\,/\,\pi(j)\in S'_q} \overline{p_j} \leqslant \lambda \left(m - \sum_{l=1}^{q} \mu_l \right) + \lambda\mu_q - \sum_{j\,/\,\pi(j)\in S_q} \overline{p_j}$$

that is equivalent to

$$\sum_{j\,/\,\pi(j)\in S'_q \cup S_q} \overline{p_j} \leqslant \lambda \left(m - \sum_{l=1}^{q-1} \mu_l \right) \tag{5.2}$$

When $\sum_{l=1}^{q} \mu_l$ is equal to m (Constraint (5.1)), then a task with a processing time greater than $\frac{\lambda}{2}$ is assigned to each CPU, implying that there can be at most μ_q tasks assigned to the CPUs with execution times lower than $\frac{q\lambda}{2q}$ and strictly greater than $\frac{(q-1)\lambda}{2q}$, since these tasks cannot be assigned to the $m - \mu_q$ CPUs that were assigned tasks from $S_1, \ldots, S_{q-1}$. However, if we have $\sum_{l=1}^{q} \mu_l < m$, this means that some CPUs are left idle, and therefore some tasks with execution times lower than $\frac{q\lambda}{2q}$ and strictly greater than $\frac{(q-1)\lambda}{2q}$ can be assigned to these $m - \sum_{l=1}^{q} \mu_l$ idle CPUs, finishing their execution before $\frac{q\lambda}{2q}$. In a sense, shelf S'_q now spreads across the μ_q CPUs of S_q and the idle CPUs. In Figure 5.5, we can see that S'_q occupies μ_{q+1} CPUs, and some tasks with execution times lower

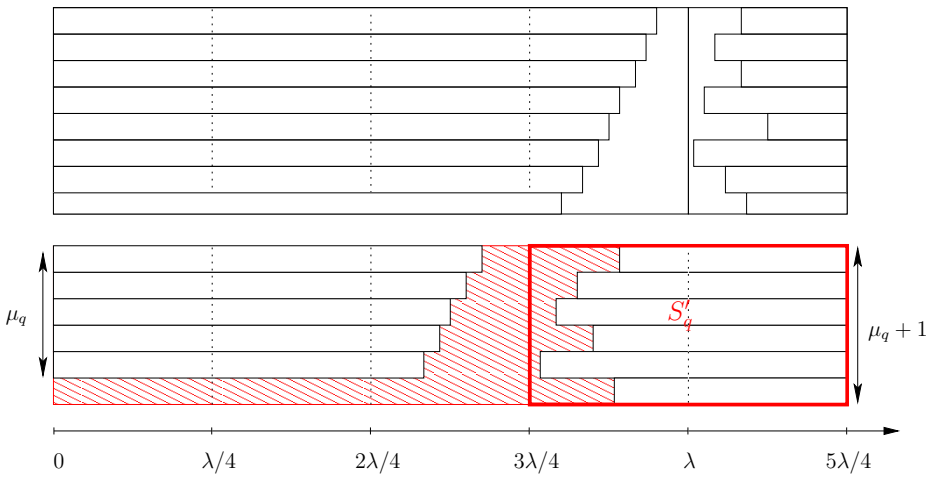


Figure 5.5: Example for $g = 5/4$. The shelf S'_q and where the tasks with processing time lower than $\frac{\lambda}{2q}$ can be assigned to (for $q = 2$).

than $\frac{q\lambda}{2q}$ and strictly greater than $\frac{(q-1)\lambda}{2q}$ could be assigned to the stripped area of the figure. These tasks with execution times lower than $\frac{q\lambda}{2q}$ and strictly greater than $\frac{(q-1)\lambda}{2q}$ all fit in S'_q and in the space remaining on the CPUs that were still idle after the construction of shelves $S_1, \ldots, S_q$. The proof of this assertion is done with a surface argument and is similar to the one of Lemma 5.2.6 described in what follows.

Building $S'_{q-1}, \ldots, S'_2$. The same process can be applied for building the shelves $S'_{q-1}, \ldots, S'_2$, assigning the tasks on CPUs whose execution times are lower than $\frac{h\lambda}{2q}$ and strictly greater than $\frac{(h-1)\lambda}{2q}$ to the shelf S'_h of length $\frac{h\lambda}{2q}$, or to the shelf S'_{h-1} if it is not filled, for $h = q-1, \ldots, 2$. With the creation of these $q-2$ additional shelves, $q-2$ constraints are imposed in addition to Constraints (5.1) and (5.2), the last one is:

$$\sum_{j \,/\, \pi(j) \in S'_q \cup S_q \cup \cdots \cup S'_2 \cup S_2} \overline{p_j} \leqslant \lambda \, (m - \mu_1) \tag{5.3}$$

Again, there can be only μ_h tasks with corresponding execution times assigned to CPUs in shelf S'_h, $h = q-1, \ldots, 2$, if the previous shelf to be built S'_{h+1} is full, i.e. occupies at least the μ_{h+1} CPUs also occupied by shelf S_{h+1}. However, if there are less than μ_{q+1} tasks in S'_{q+1}, it is possible to place some tasks normally assigned to S'_h in the remaining space in S'_{h+1} and therefore the number of tasks normally assigned to S'_h can be higher than μ_h. There can even be two tasks with processing times lower than $\frac{h\lambda}{2q}$ and strictly greater than $\frac{(h-1)\lambda}{2q}$ that might fit in the remaining space in between S_{h+1} and S'_{h+1}. The same surface argument as for shelf S'_q can be used to prove that all the tasks with

processing times lower than $\frac{h\lambda}{2q}$ and strictly greater than $\frac{(h-1)\lambda}{2q}$ fit in S'_h and in the space remaining between shelves $S_q, \ldots, S_{h+1}$ and $S'_q, \ldots, S'_{h+1}$ (see the proof of Lemma 5.2.6), for $h = q-1, \ldots, 2$.

Building S'_1. After the construction of shelves $S_1, \ldots, S_q$ and $S'_q, \ldots, S'_2$, the tasks remaining to be assigned to a CPU have a processing time lower than $\frac{\lambda}{2q}$. If shelf S'_2 is not filled, we assign the remaining tasks with the longest processing time to this shelf until it is completely filled. Then, we assign the μ_1 longest remaining tasks to shelf S'_1, of length $\frac{\lambda}{2q}$, if $\mu_1 \neq 0$.
W_L will denote the computational area on CPU remaining idle after this assignment in the schedule of length $\frac{(2q+1)\lambda}{2q}$. W_L corresponds to the stripped area in Figure 5.6.

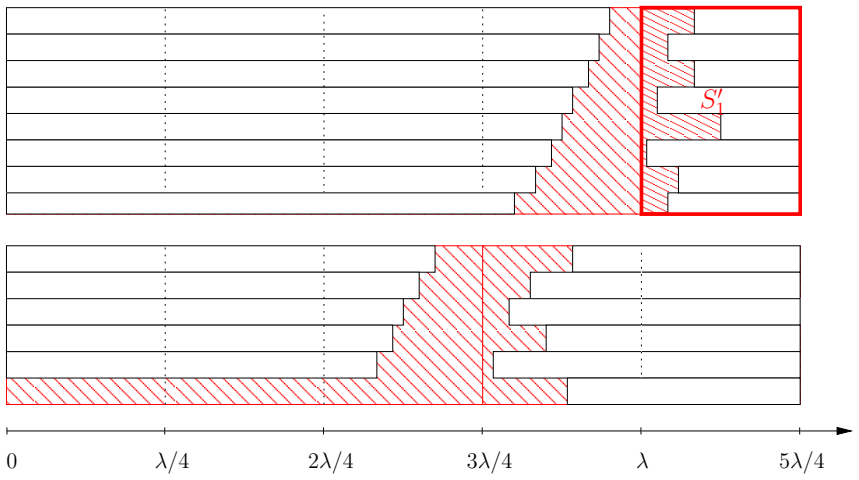


Figure 5.6: Example for $g = 5/4$. The free computational space W_L is represented by the stripped area.

Regarding the question of how the remaining tasks fit in the constructed schedule, we state the following lemma:

Lemma 5.2.6. *The tasks remaining to be assigned on CPUs after the construction of $S_1, \ldots, S_q$, $S'_q, \ldots, S'_1$ fit in the remaining free computational space W_L between these shelves.*

Proof. The proof is similar to the one of Lemma 4.3.4. The tasks remaining to be assigned after the construction of $S_1, \ldots, S_q$, $S'_q, \ldots, S'_1$ all have a processing time lower than $\frac{\lambda}{2q}$ by construction and they necessarily fit into the remaining computational space W_L, otherwise the schedule would not satisfy Property 5.2.1. The following algorithm can be used to schedule these tasks:

- Consider the remaining tasks ordered by decreasing order of processing time on CPU $T_1, \ldots, T_f$, f being the total number of tasks remaining to be assigned.

- At each step i, $i = 1, \ldots, f$, assign task T_i to the least loaded processor, at the latest possible date. Update its load.

At each step, the least loaded processor has a load at most λ; otherwise it would contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Property 5.2.1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{2q}$ and can contain the task T_i, which proves the correctness of the scheduling algorithm. □

When a shelf S'_h, $h = q \ldots, 2$, do not use the same number of processors as its corresponding shelf S_h, the same arguments as the ones from the proof of Lemma 5.2.6 can be used to prove that all the tasks with the execution times corresponding to the considered shelf S'_h, $h = q, \ldots, 2$, fit in the space remaining for their assignment. Otherwise it would contradict the fact that the total computational area on the CPUs is bounded by $m\lambda$ (see Property 5.2.1). Therefore all the tasks can be assigned following the construction described in the algorithm of Lemma 5.2.6's proof.

5.2.3 Assigning the Tasks to the Shelves

In this section, we detail how to fill the shelves on the CPUs and the GPU by specifying an initial assignment of the tasks to the processors according to Section 5.2.2. Determining if such an assignment exists reduces to solving a multi-dimensional knapsack minimization problem.

We use for each task T_j a binary decision variable x_j as defined previously, such that $x_j = 1$ if T_j is assigned to a CPU and 0 if T_j is assigned to the GPU. The problem can then be formulated as follows:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{5.4}$$

$$\text{s.t.} \sum_{j\,/\,\overline{p_j} > \lambda/2} x_j \leqslant m \tag{5.5}$$

$$\begin{cases} \sum\limits_{j\,/\,\pi(j) \in S'_q \cup S_q} \overline{p_j} x_j \leqslant \lambda \left(m - \sum\limits_{l=1}^{q-1} \mu_l \right) \\ \vdots \\ \sum\limits_{j\,/\,\pi(j) \in S'_q \cup S_q \cup \cdots \cup S'_2 \cup S_2} \overline{p_j} x_j \leqslant \lambda \left(m - \mu_1 \right) \end{cases} \tag{5.6}$$

$$\sum_{j=1}^{n} \underline{p_j} \left(1 - x_j\right) \leqslant \frac{(2q+1)\lambda}{2q} \tag{5.7}$$

$$x_j \in \{0,1\},\ j \in \{1, \ldots, n\} \tag{5.8}$$

Equation (5.4) represents the minimal workload on all the CPUs. Constraint (5.5) imposes that no more than m tasks can be executed on the CPUs with a processing time greater than $\frac{\lambda}{2q}$, considering all the first shelves of each set, $S_1, \ldots, S_q$. Constraints (5.6) represent the filling constraints related to the second shelves of each set, $S'_q, \ldots, S'_2$. Constraint (5.7) corresponds to the fact that the makespan on the GPU must be lower than $\frac{(2q+1)\lambda}{2q}$.

5.2.4 Dynamic Programming

We propose a dynamic programming algorithm in $\mathcal{O}\left(n^2 m^q\right)$ to solve the minimization knapsack problem. For this purpose, we first discretize the processing times of the tasks on the GPU, as it was described in Chapter 4, Section 4.3.3. We introduce $\nu_j = \left\lfloor \frac{\underline{p_j}}{\lambda/(2qn)} \right\rfloor$ to represent the number of integer time intervals of length $\frac{\lambda}{2qn}$ for a task T_j if it is executed on the GPU. For a graphical representation, see Figure 4.9. We denote by $N = \sum\limits_{\pi(j) \in \mathcal{G}} \nu_j$ the total number of these intervals on the GPU. We thus define the error on each task $\epsilon_j = \underline{p_j} - \nu_j \frac{\lambda}{2qn}$ induced by this time discretization.

This result allows us to consider only N states in the dynamic programming algorithm regarding the workload on the GPU. The error ϵ_j on each task is at most $\frac{\lambda}{2qn}$ so if all the tasks were assigned to the GPU, we would have underestimated the processing time on

the GPU by at most $n\frac{\lambda}{2qn} = \frac{\lambda}{2q}$. We have

$$\begin{aligned} N &= \sum_{\pi(j)\in\mathcal{G}} \nu_j \\ &= \sum_{j=1}^{n} \left\lfloor \frac{\underline{p_j}}{\lambda/(2qn)} \right\rfloor (1 - x_j) \\ &= \sum_{j=1}^{n} \frac{\underline{p_j} - \epsilon_j}{\lambda/(2qn)} (1 - x_j) \end{aligned}$$

Then we can rewrite Constraint (4.5), $\sum_{j=1}^{n} \underline{p_j}\,(1 - x_j) \leqslant \frac{(2q+1)\lambda}{2q}$, as

$$\sum_{j=1}^{n} \left(\underline{p_j} - \epsilon_j\right)(1 - x_j) + \sum_{j=1}^{n} \epsilon_j\,(1 - x_j) \leqslant \frac{(2q+1)\lambda}{2q},$$

$$\sum_{j=1}^{n} \left(\underline{p_j} - \epsilon_j\right)(1 - x_j) \leqslant \frac{(2q+1)\lambda}{2q} - \sum_{j=1}^{n} \epsilon_j\,(1 - x_j)\,,$$

$$\frac{\lambda}{2qn} N \leqslant \frac{(2q+1)\lambda}{2q} - \sum_{j=1}^{n} \epsilon_j\,(1 - x_j)\,.$$

In order to always satisfy Constraint (4.5), we have to consider that we are in the worst possible case, i.e. all the tasks are assigned to the GPU and the error for each task is $\frac{\lambda}{3n}$. We obtain

$$\frac{\lambda}{2qn} N \leqslant \frac{(2q+1)\lambda}{2q} - \frac{\lambda}{2q}.$$

Therefore Constraint (5.7) becomes:

$$N = \sum_{j\,/\,\pi(j)\in\mathcal{G}} \nu_j \leqslant 2qn \tag{5.9}$$

The approximated makespan on the GPU will be at most λ and so with the underestimation detailed above the makespan on the GPU will remain lower than $\frac{(2q+1)\lambda}{2q}$. Once this reduction is done, we define $W_C(j, \mu_1, \ldots, \mu_q, N)$ as the minimum sum of all the processing times of the tasks on the CPUs when the first j tasks are considered, where μ_l, $(l = 1, \ldots, q)$ denotes the number of processors occupied by the shelf S_l, and where N time intervals are occupied on the GPU.
We use a dynamic programming algorithm to compute the value of the objective function $W_C(j, \mu_1, \ldots, \mu_q, N)$.
If task T_j is assigned to the GPU, the sum of all the processing times of the tasks on the CPUs is then

$$F_{GPU}(T_j) = W_C\left(j-1, \mu_1, \ldots, \mu_q, N-\nu_j\right)$$

The dynamic programming algorithm is then based on the following recursive equation:

$$W_C\left(j, \mu_1, \ldots, \mu_q, N\right) = \min\left(F_{CPU}(T_j), F_{GPU}(T_j)\right)$$

If task T_j is assigned to a CPU, the resulting sum of all the processing times $F_{CPU}(T_j)$ of the tasks on the CPUs is then

$$F_{CPU}(T_j) = \overline{p_j} + W_C\left(j-1, \mu_1 - I_{\left(\overline{p_j} > \frac{(2q-1)\lambda}{2q}\right)}, \ldots, \mu_q - I_{\left(\frac{(q+1)\lambda}{2q} \geqslant \overline{p_j} > \frac{\lambda}{2}\right)}, N\right)$$

where $I_{\left(\overline{p_j} > \frac{(2q-1)\lambda}{2q}\right)}$, $\ldots$, $I_{\left(\frac{(q+1)\lambda}{2q} \geqslant \overline{p_j} > \frac{\lambda}{2}\right)}$ are indicating functions:

$$I_{\left(\overline{p_j} > \frac{(2q-1)\lambda}{2q}\right)} = \begin{cases} 1 & \text{if } \overline{p_j} > \frac{(2q-1)\lambda}{2q} \\ 0 & \text{otherwise} \end{cases}$$

$$\vdots$$

$$I_{\left(\frac{(q+1)\lambda}{2q} \geqslant \overline{p_j} > \frac{\lambda}{2}\right)} = \begin{cases} 1 & \text{if } \frac{(q+1)\lambda}{2q} \geqslant \overline{p_j} > \frac{\lambda}{2} \\ 0 & \text{otherwise} \end{cases}$$

In order to satisfy Constraints (5.6) and (5.9), we have the following border conditions:

$$W_C\left(j, \mu_1, \ldots, \mu_q, N\right) = \begin{cases} +\infty \text{ if } \mu_1 > m \\ \ldots \\ +\infty \text{ if } \mu_q > m - \sum_{l=1}^{q-1} \mu_l \\ +\infty \text{ if } \sum_{j\,/\,\pi(j)\in\mathcal{G}} \nu_j > 2qn \end{cases}$$

The optimal value of the computational area W_C on the CPUs, will be given by

$$W_C^* = \min_{0\leqslant\mu_1\leqslant m, \ldots,\ 0\leqslant\mu_q\leqslant m-\sum_{l=1}^{q-1}\mu_l,\ 0\leqslant N\leqslant 2qn} W_c\left(n, \mu_1, \ldots, \mu_q, N\right)$$

If W_C^* is greater than $m\lambda$, then there exists no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation. This means that the chosen guess λ is too small. Otherwise the guess λ is large enough, we construct a feasible solution with a makespan at most $\frac{(2q+1)\lambda}{2q}$, with the corresponding shelves on the CPUs and the corresponding $\mu_1, \ldots, \mu_q$ and N values.
This dynamic programming algorithm represents one step of the dual-approximation algorithm, with a fixed guess λ. A binary search is then used to try different guesses to approach the optimal makespan as explained in Section 5.1.

Cost Analysis. Solving the dynamic program for a fixed value of λ requires to consider $\mathcal{O}(n^2 m^q)$ states, since $1 \leqslant j \leqslant n$, $1 \leqslant \mu_1 \leqslant m$, $1 \leqslant \mu_h \leqslant m - \sum_{l=1}^{h-1} \mu_l$, for $h \in \{2, \ldots, q\}$ and $0 \leqslant N \leqslant 2qn$. Therefore, the time complexity of each step of the binary search is $\mathcal{O}(n^2 m^q)$, which is polynomial for a fixed value of q. We have to solve one of these problems at each step of the binary search.

5.3 Solving the problem with $k \geqslant 2$

The algorithm described in Section 5.1 can be extended to the problem with $k \geqslant 2$ GPUs, using the same structure for the GPUs as the one used for the CPUs in Section 5.2. The analysis of an optimal solution leads to the following properties:

Proposition 5.3.1. *In an optimal solution, the execution time of each task is at most λ, and the computational area on the GPUs is at most $k\lambda$.*

Proposition 5.3.2. *In an optimal solution, if there exist two tasks T_i, T_j executed on a GPU such that $\underline{p_i} > \frac{(2q-1)\lambda}{2q}$, then $\underline{p_j} \leqslant \frac{\lambda}{2q}$.*

Proposition 5.3.3. *If there exist two tasks T_i, T_j processed on a GPU such that $\frac{(2q-2)\lambda}{2q} < \underline{p_i} \leqslant \frac{(2q-1)\lambda}{2q}$, then $\underline{p_j} \leqslant \frac{2\lambda}{2q}$.*

These properties can be formulated more generally as

Proposition 5.3.4. *If there exist two tasks T_i, T_j executed on a GPU such that $\frac{(2q-h)\lambda}{2q} < \underline{p_i} \leqslant \frac{(2q-h+1)\lambda}{2q}$, then $\underline{p_j} \leqslant \frac{h\lambda}{2q}$, for $h = 1, \ldots, q$.*

We use the same notations as before:

- The set $\mathcal{G}$ is now the set of all the GPUs.
- k sets of $2qn$ integer time intervals will be considered.
- We create q sets of two shelves on the GPUs, with the shelves $G_1, \ldots, G_q$ similar to the shelves $S_1, \ldots, S_q$ on the CPUs, as well as the shelves $G'_1, \ldots, G'_q$ similar to the shelves $S'_1, \ldots, S'_q$.
- The number of GPUs used in each shelf G_l is denoted by κ_l.
- The free computational space remaining after the construction of the sets of shelves is still denoted by W_L on CPUs and is denoted by W_R on GPUs.

The problem can be formulated in the same way, with the following constraints on the GPUs:

$$\sum_{j\,/\,\underline{p_j}>\lambda/2}(1-x_j)\leqslant k$$

$$\begin{cases}\displaystyle\sum_{j\,/\,\pi(j)\in G'_q\cup G_q}\underline{p_j}(1-x_j)\leqslant\lambda\left(k-\sum_{l=1}^{q-1}\kappa_l\right)\\ \vdots\\ \displaystyle\sum_{j\,/\,\pi(j)\in G'_q\cup G_q\cup\cdots\cup G'_2\cup G_2}\underline{p_j}(1-x_j)\leqslant\lambda\,(k-\kappa_1)\end{cases}$$

$$N=\sum_{j\,/\,\pi(j)\in\mathcal{G}}\nu_j\leqslant 2qkn$$

Then, the problem becomes:

$$W_C^*=\min\sum_{j=1}^{n}\overline{p_j}x_j$$

$$\text{s.t.}\quad\sum_{j\,/\,\overline{p_j}>\lambda/2}x_j\leqslant m$$

$$\begin{cases}\displaystyle\sum_{j\,/\,\pi(j)\in S'_q\cup S_q}\overline{p_j}x_j\leqslant\lambda\left(m-\sum_{l=1}^{q-1}\mu_l\right)\\ \vdots\\ \displaystyle\sum_{j\,/\,\pi(j)\in S'_q\cup S_q\cup\cdots\cup S'_2\cup S_2}\overline{p_j}x_j\leqslant\lambda\,(m-\mu_1)\end{cases}$$

$$\sum_{j\,/\,\underline{p_j}>\lambda/2}(1-x_j)\leqslant k$$

$$\begin{cases}\displaystyle\sum_{j\,/\,\pi(j)\in G'_q\cup G_q}\underline{p_j}(1-x_j)\leqslant\lambda\left(k-\sum_{l=1}^{q-1}\kappa_l\right)\\ \vdots\\ \displaystyle\sum_{j\,/\,\pi(j)\in G'_q\cup G_q\cup\cdots\cup G'_2\cup G_2}\underline{p_j}(1-x_j)\leqslant\lambda\,(k-\kappa_1)\end{cases}$$

$$N=\sum_{j\,/\,\pi(j)\in\mathcal{G}}\nu_j\leqslant 2qkn$$

$$x_j\in\{0,1\},\ j\in\{1,\ldots,N\}$$

If all the tasks are assigned to GPUs, the error $\epsilon=\sum_{j=1}^{n}\epsilon_j$ is still $\frac{\lambda}{2q}$, so the computational area of the GPUs is lower than $\left(k+\frac{1}{2q}\right)\lambda$.

This problem covers the assignment of the tasks with a processing time greater than $\frac{\lambda}{2q}$. As in Section 5.2.2, we have the following lemma:

Lemma 5.3.5. *The tasks remaining to be assigned on CPUs (resp. GPUs) fit in the remaining computational space W_L (resp. W_R) on the CPUs (resp. GPUs).*

Proof. The proof that the tasks remaining to be assigned on CPUs fit in W_L is identical to the proof of Lemma 5.2.6. The proof that the tasks remaining to be assigned on GPUs fit in W_R is very similar to the one of Lemma 5.2.6. The tasks remaining to be assigned to the GPUs after the construction of $G_1, \ldots, G_q$ and $G'_q, \ldots, G'_2$ all have a processing time lower than $\frac{\lambda}{2q}$ by construction and they necessarily fit in W_R, otherwise the schedule would not satisfy Property 5.3.1. The following algorithm is used to schedule the tasks:

- Consider the remaining tasks ordered in decreasing order of processing time on GPU, $T_1, \ldots, T_f$, f being the total number of tasks remaining to be assigned.
- At each step i, $i = 1, \ldots, f$, assign task T_i to the least loaded GPU, at the latest possible date. Update its load.

At each step, the least loaded GPU has a load at most $\lambda + \frac{\lambda}{2qk}$; otherwise it would contradict the fact that the total work area of the tasks is bounded by $k\left(\lambda + \frac{1}{2qk}\right)\lambda$ (according to Property 5.3.1). Hence, the idle time interval on the least loaded GPU has a length at least equal to $\frac{\lambda}{2q}$ and can contain the task T_i, which proves the correctness of the scheduling algorithm. □

As for the CPUs, when a shelf G'_h, $h = q \ldots, 2$, do not use the same number of processors as its corresponding shelf G_h, the same arguments as the ones from the proof of Lemma 5.3.5 can be used to prove that all the tasks with the execution times corresponding to the considered shelf G'_h, $h = q, \ldots, 2$, fit in the space remaining for their assignment. Otherwise it would contradict the fact that the total work area on the GPUs is bounded by$k\left(\lambda + \frac{1}{2qk}\right)\lambda$ (see Property 5.3.1). Therefore all the tasks can be assigned following the construction described in the algorithm of Lemma 5.3.5's proof.

The family of approximation algorithms presented in the previous sections can be extended to the problem with $k \geqslant 2$ GPUs with a performance guarantee of $\frac{2q+1}{2q} + \frac{1}{2qk}$. In order to solve each step of the binary search, we have to add q parameters to W_C in the dynamic programming and parameter N now varies between 0 and $2qkn$, so we have a time complexity of $\mathcal{O}\left(n^2 m^q k^{q+1}\right)$.

5.4 Complementary Family of Approximation Algorithms

We can derive similarly another family of algorithms with ratios of $\frac{2(q+1)}{2q+1}$ for the case $k = 1$ and ratios of $\frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k}$ when $k > 1$. The lengths and numbers of the shelves have to be adapted but the idea is similar.

The same properties are verified and we construct the shelves $S_1, \ldots, S_q$ and $G_1, \ldots, G_q$ in the same way, except the length of the time intervals which is now $\frac{\lambda}{2q+1}$ instead of $\frac{\lambda}{2q}$. We have then considered the tasks with processing time strictly greater than $\frac{(q+1)\lambda}{2q+1}$. But some of the remaining tasks still have a processing time greater than $\frac{\lambda}{2}$. We have two additional properties:

Proposition 5.4.1. *If there exist two tasks T_i, T_j executed on the same CPU such that $\frac{\lambda}{2} = \frac{(q+1/2)\lambda}{2q+1} < \overline{p_i} \leqslant \frac{(q+1)\lambda}{2q+1}$, then $\underline{p_j} \leqslant \frac{q\lambda}{2q}$.*

Proposition 5.4.2. *If there exist two tasks T_i, T_j processed on the same GPU such that $\frac{\lambda}{2} = \frac{(q+1/2)\lambda}{2q+1} < \underline{p_i} \leqslant \frac{(q+1)\lambda}{2q+1}$, then $\underline{p_j} \leqslant \frac{q\lambda}{2q}$.*

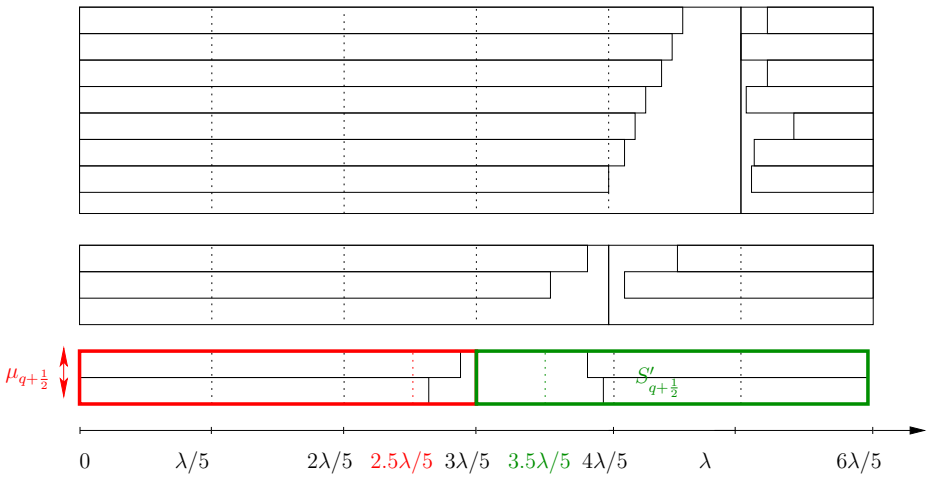


Figure 5.7: Example for $g = 6/5$, where λ is the guess.

These tasks are executed in a $\left(q + \frac{1}{2}\right)^{\text{th}}$ additional set, see Figure 5.7, in a shelf $S_{q+\frac{1}{2}}$ (resp. $G_{q+\frac{1}{2}}$), with $\mu_{q+\frac{1}{2}}$ CPUs (resp. $\kappa_{q+\frac{1}{2}}$ GPUs), such that $\mu_{q+\frac{1}{2}}$ and $\kappa_{q+\frac{1}{2}}$ satisfy the following constraints:

$$\mu_{q+\frac{1}{2}} \leqslant m - \sum_{l=1}^{q} \mu_l$$

$$\kappa_{q+\frac{1}{2}} \leqslant k - \sum_{l=1}^{q} \kappa_l$$

The execution times of the tasks remaining to be assigned on the CPUs (resp. GPUs) are lower than $\frac{\lambda}{2}$ and some of them can be executed on the same CPUs (resp. GPUs) used by the tasks with processing times greater than $\frac{\lambda}{2}$.

The tasks on CPUs (reps. GPUs) whose execution times are lower than $\frac{(q+1/2)\lambda}{2q+1}$ and strictly greater than $\frac{q\lambda}{2q+1}$ can only be executed on idle CPUs (resp. idle GPUs) or following a task from $S_{q+\frac{1}{2}}$ (resp. $G_{q+\frac{1}{2}}$). They form the shelf $S'_{q+\frac{1}{2}}$ (resp. $G'_{q+\frac{1}{2}}$), and they satisfy the following constraints

$$\sum_{\pi(j)\in S'_{q+\frac{1}{2}}\cup S_{q+\frac{1}{2}}} \overline{p_j} x_j \leqslant \lambda \left(m - \sum_{l=1}^{q} \mu_l \right)$$

$$\sum_{\pi(j)\in G'_{q+\frac{1}{2}}\cup G_{q+\frac{1}{2}}} \underline{p_j} \left(1 - x_j\right) \leqslant \lambda \left(k - \sum_{l=1}^{q} \kappa_l \right)$$

We iterate this process, as in the previous section, with each interval of processing times for the remaining tasks, $2(q-2)$ additional constraints have to be considered, the last two being:

$$\sum_{j\,/\,\pi(j)\in S'_{q+\frac{1}{2}}\cup S_{q+\frac{1}{2}}\cup\cdots\cup S'_2\cup S_2} \overline{p_j} x_j \leqslant \lambda \left(m - \mu_1\right)$$

$$\sum_{j\,/\,\pi(j)\in G'_{q+\frac{1}{2}}\cup G_{q+\frac{1}{2}}\cup\cdots\cup G'_2\cup G_2} \underline{p_j} \left(1 - x_j\right) \leqslant \lambda \left(k - \kappa_1\right)$$

We can also write lemmas similar to Lemma 5.2.6 and Lemma 5.3.5, stating that the tasks remaining to be assigned after the construction of the shelves all fit in the remaining free computational space so that we obtain the desired approximation ratios. The proofs would be nearly identical, except the number of shelves considered and the resulting parameters.

Cost Analysis In the case of $k = 1$, we have to consider the same inequalities as for the ratios of $\frac{2q+1}{2q}$, and one more for $\mu_{q+\frac{1}{2}}$. The time complexity of the algorithms with ratios of $\frac{2(q+1)}{2q+1}$ becomes $\mathcal{O}\left(n^2 m^{q+1}\right)$. If $k > 1$, we consider the same inequalities as for the ratios of $\frac{2q+1}{2q} + \frac{1}{2qk}$, and two additional ones for $\mu_{q+\frac{1}{2}}$ and $\kappa_{q+\frac{1}{2}}$, which gives a time complexity of $\mathcal{O}\left(n^2 m^{q+1} k^{q+2}\right)$ for the algorithms with ratios of $\frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k}$.

5.5 Summary

Tables 5.1 and 5.2 summarize the ratios achieved for different values of k and q. Figure 5.8 shows for $k = 1$ that the ratios of the two families of algorithms are intertwined and the ratios are closer together as q increases.

	Ratio	Cost
$k = 1$	$\frac{2q+1}{2q}$	$\mathcal{O}\left(n^2 m^q\right)$
	$\frac{2(q+1)}{2q+1}$	$\mathcal{O}\left(n^2 m^{q+1}\right)$
$k > 1$	$\frac{2q+1}{2q} + \frac{1}{2qk}$	$\mathcal{O}\left(n^2 m^q k^{q+1}\right)$
	$\frac{2(q+1)}{2q+1} + \frac{1}{(2q+1)k}$	$\mathcal{O}\left(n^2 m^{q+1} k^{q+2}\right)$

Table 5.1: Associated costs and ratios for different values of k.

q	Ratio	Cost
0	2	$\mathcal{O}\left(n^2 k\right)$
1	$\frac{3}{2} + \frac{1}{2k}$	$\mathcal{O}\left(n^2 m k^2\right)$
	$\frac{4}{3} + \frac{1}{3k}$	$\mathcal{O}\left(n^2 m^2 k^3\right)$
2	$\frac{5}{4} + \frac{1}{4k}$	$\mathcal{O}\left(n^2 m^2 k^3\right)$
	$\frac{6}{5} + \frac{1}{5k}$	$\mathcal{O}\left(n^2 m^3 k^4\right)$

Table 5.2: Associated costs and ratios for different values of q.

Now that we have established a complete set of approximation algorithms for problem $(Pm, Pk) \mid\mid C_{max}$, we can study other instances of the problem of scheduling independent tasks, such as more specific cases regarding the nature of the tasks, or other objectives than the makespan. This is studied in the following chapter.

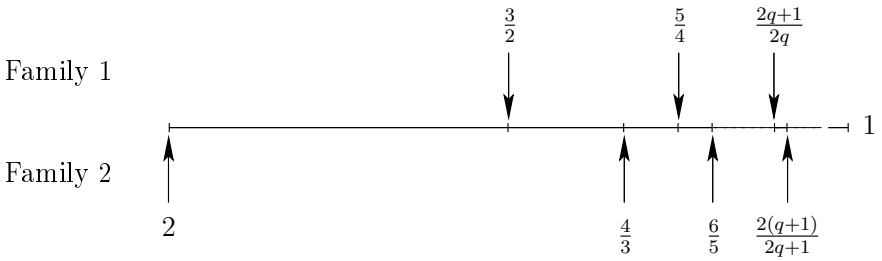


Figure 5.8: Different approximation ratios for the two families of algorithms for $k = 1$.

Chapter 6

Scheduling Other Instances with Independent Tasks

After studying in great details the $(Pm, Pk) \mid\mid C_{max}$ problem, we looked at possible variations of this core problem. In this chapter, we study some other instances of the problem of scheduling independent tasks on CPUs and GPUs. We first study the specific case of problem $(Pm, Pk) \mid\mid C_{max}$ where all the tasks of the instance have accelerated processing times when assigned to a GPU. Then we move on to the special case of $(Pm, Pk) \mid\mid C_{max}$ where preemption are allowed on the CPUs, which is a possibility that can be encountered in practice. Then, we look at the problem of scheduling on heterogeneous platform with a model of tasks designed to deal with communication issues between processors, which is a problem that can be encountered on computing platforms, the tasks being still independent but moldable on CPU, not sequential, meaning that they can be assigned to more than one CPU. Finally we briefly study the case of uniform CPUs and/or uniform GPUs.

6.1 All the tasks are accelerated on GPU

We consider, in this section, a version of the problem $(Pm, Pk) \mid\mid C_{max}$ where all the tasks are accelerated when assigned to a GPU, since this is the case in most applications, i.e $q_j = \frac{\overline{p_j}}{\underline{p_j}} \geqslant 1$ for $j = 1, \ldots, n$ (all the tasks do not have the same acceleration factor on GPU). This specific case if denoted $(Pm, Pk) \mid q_j \geqslant 1 \mid C_{max}$. No improvement in the time complexity of the algorithms presented in Chapters 4 and 5 is observed for the case where $q_j = \frac{\overline{p_j}}{\underline{p_j}} \geqslant 1$ for $j = 1, \ldots, n$. Therefore we present an new algorithm for this case, based on a scheme similar to the one for the $\frac{4}{3}$-approximation algorithm presented in Chapter 4, Sections 4.3 and 4.4, but with a much lower time complexity for an approximation ratio of $\frac{3}{2}$.
The algorithm is also based on the dual approximation technique (see Chapter 3, Section 3.2.2.2. At each step, we have a guess on the optimal makespan. Let us consider

one step of the dual approximation scheme and let as before λ be the current guess. The idea is to divide the set of tasks $\mathcal{T}$ into four sets of tasks, two of them whose tasks will be assigned to a CPU, $\mathcal{C}_1$, $\mathcal{C}_2$, and the other two whose tasks will be assigned to a GPU, $\mathcal{G}_1$ and $\mathcal{G}_2$. We denote the cardinality of set $\mathcal{C}_i$ (resp. $\mathcal{G}_i$) by $|\mathcal{C}_i|$ (resp. $|\mathcal{G}_i|$), $i = 1, 2$. The algorithm is as follows for one step of the dual approximation, λ being the current guess.

Algorithm 6.1.1.

1. *For each task T_j:*
   - *If $\overline{p_j} \leqslant \frac{\lambda}{2}$, task T_j is assigned to $\mathcal{C}_2$.*
   - *Otherwise, $\frac{\lambda}{2} < \overline{p_j}$, task T_j is assigned to $\mathcal{G}_1$.*
2. *Reorder the tasks of $\mathcal{C}_2$ by decreasing order of $\overline{p_j} - \underline{p_j}$.*
   Do the same reordering for the tasks or $\mathcal{G}_1$.
3. *While $W_G = \sum_{T_i \in \mathcal{G}_1 \cup \mathcal{G}_2} \underline{p_i} \leqslant k\lambda$,*
   (a) *Assign the first task of set $\mathcal{C}_2$ to $\mathcal{G}_2$.*
   (b) *If $|\mathcal{C}_1| < m$, assign the first task from $\mathcal{G}_1$ to $\mathcal{C}_1$.*

The first step of the algorithm consists in a preliminary assignment of the tasks of the whole set $\mathcal{T}$ to two of the sets: $\mathcal{G}_1$ and $\mathcal{C}_2$. This pre-assignment of each task T_j is done by considering the value of the processing time $\overline{p_j}$ of T_j on CPU.
Unfortunately, this pre-assignment does not guarantee that the resulting schedule will have a makespan lower than $\frac{3}{2}\lambda$, even if there exists a schedule of makespan lower than λ. However, we note that if $|\mathcal{G}_1| > k + m$, there are too many tasks with a processing time on CPU greater than $\frac{\lambda}{2}$ to fit in a schedule of makespan lower than λ. The dual approximation rejects this guess λ.
In the second step of the algorithm, we order the tasks of sets $\mathcal{C}_2$ and $\mathcal{G}_1$ in decreasing order of the computational surface change induced when a task T_j changes from a CPU to a GPU, $\overline{p_j} - \underline{p_j}$.
In order to achieve the desired performance ratio, we have to reassign some of the tasks assigned to $\mathcal{C}_2$ to $\mathcal{G}_2$ and some of the tasks assigned to $\mathcal{G}_1$ to $\mathcal{C}_1$ in the third step of the algorithm. In Step 3.b), one exception has to be made for set $\mathcal{G}_1$: some tasks can have a processing time on CPU larger than λ. These tasks are too big to fit on the CPUs with the current guess. They cannot be reassigned and are put at the end of $\mathcal{G}_2$, no matter the impact they can have on the computational surfaces. We can note that at most m tasks of set $\mathcal{G}_1$ can be reassigned to $\mathcal{C}_1$. The two substeps of Step 3 are therefore repeated at most $m + 1$ times, as long as we have $W_C > m\lambda + \frac{\lambda}{2}$ or $W_G > k\lambda + \frac{\lambda}{2}$.
With this assignment, the computational area on the CPUs has been reduced to a minimum with the constraint of keeping the computational area on the GPUs lower than

$k\lambda + \frac{\lambda}{2}$. Therefore, the value of W_C obtained by our algorithm is smaller than the value of the computational area on the CPUs of the optimal schedule, the most accelerated tasks having been assigned to the GPUs. Therefore, if $W_C > m\lambda + \frac{\lambda}{2}$, we conclude that the value of λ is too small and adjust the bounds of our binary search accordingly.
If $W_C \leqslant m\lambda + \frac{\lambda}{2}$, we can construct a feasible schedule with a makespan lower than $\frac{3}{2}\lambda$ with the previous algorithm. Indeed, the number of tasks in $\mathcal{C}_1$ is lower than m so we can build a shelf S_1 as we did in Chapter 4, Section 4.3, occupying $|\mathcal{C}_1|$ CPUs, with a length at most λ. The same arguments given in the proof of Lemma 4.3.4 can be used for building a shelf S_2 of length $\frac{\lambda}{2}$ and all the tasks from $\mathcal{C}_2$ can be fitted in the schedule as before. For the GPUs, the algorithm makes sure that the number of tasks in $\mathcal{G}_1$ is lower than k and that W_G does not go over the bound of $k\lambda + \frac{\lambda}{2}$ so shelves similar to S_1 and S_2 can be built easily. However, we did not have to make any discretization on the processing times of the tasks assigned to the GPUs here, so, contrary to Chapter 4, Section 4.4, we get the same performance ratio of $\frac{3}{2}$ for any number k of GPUs. The time complexity of an algorithm based on this principle is in $\mathcal{O}(mn \log n)$.

6.2 Partial Preemption

In the existing scheduling algorithms, a GPU is usually seen as a co-processor of a CPU, and, up to now, it is difficult and costly to interrupt the execution of a task on a CPU and resume it on a GPU or even to preempt a task on GPUs. No definitive solution has been given to the matter of preemption of the tasks on these platforms. However, the use of preemption could yield much better schedules for the CPUs. Thus, we investigate in this section how preemptions can be introduced in order to improve global computations. Here, preemption is allowed for the tasks scheduled on the CPUs and even between CPUs, but, due to the architecture of the GPUs, preemptions of the tasks are not allowed in the latter. Therefore only a "partial" preemption on CPUs, denoted *ppmtn*, is addressed in this section.
$(Pm, Pk) \mid ppmtn \mid C_{max}$ is NP-hard, since if we consider the problem with $m = 1$, $k = 1$ and only one type of tasks, i.e. $q_j = q$, the problem is equivalent to the classical $Q2 \mid\mid C_{max}$ problem, which is NP-hard. We develop a dual approximation algorithm running in $\mathcal{O}(n \log n)$. Depending on the value of k, the approximation ratio of the algorithm varies. As before, we first present the case with only one GPU.

6.2.1 Single GPU Case

For $(Pm, P1) \mid ppmtn \mid C_{max}$, the algorithm have the following steps for each guess λ of the dual approximation scheme:

- Extract from the set of tasks those which necessarily fit in the GPU ($\overline{p_j} > \lambda$), and complete them by the tasks with the largest acceleration factor $q_j = \frac{\overline{p_j}}{\underline{p_j}}$ up to the guess.

- Put all the remaining tasks on the m CPUs.

Lemma 6.2.1. *This algorithm has an approximation ration of* $1 + \frac{1}{m}$.

Proof. If at one step of the algorithm the current guess λ is lower than the optimal makespan of the schedule, then the workload of the CPUs cannot be lower than $m\left(1 + \frac{1}{m}\right) C^*_{max}$ with the task assignment given by the algorithm. If the guess is larger than C^*_{max}, the assignment of the tasks with the largest acceleration factors to the GPU ensures that the workload of the CPUs is lower than $m\left(1 + \frac{1}{m}\right) C^*_{max}$. Therefore, the dual approximation scheme narrows the value of λ down to C^*_{max}.
When $\lambda = C^*_{max}$, let us consider the last task assigned to the CPUs, T_{last}. If T_{last} was assigned to the GPU, the makespan of this processor would go over C^*_{max}, and therefore the remaining tasks on the CPUs would have a workload lower than the optimal one. Indeed, this workload cannot be lowered by swapping a task on the CPUs with a task on the GPU, the acceleration factors of the tasks assigned to the GPU being larger than the ones remaining on the CPUs. Therefore, we have

$$\frac{W_C^-}{m} \leqslant C^*_{max},$$

where W_C^- represents the workload of the CPUs without the last task assigned to the CPUs by the algorithm. We have $W_C^- = W_C - \overline{p_{last}}$ (W_C being the workload of the CPUs), and it follows that

$$\frac{W_C}{m} \leqslant \frac{\overline{p_{last}}}{m} + C^*_{max}.$$

$\frac{W_C}{m}$ corresponds to the makespan of the CPUs for the schedule determined by the algorithm, since preemptions are allowed on these processors. Since all the tasks too large to fit on one CPU have been assigned to the GPU, $\overline{p_{last}} \leqslant C^*_{max}$, hence leading to the approximation ratio of $1 + \frac{1}{m}$ for this algorithm. □

Remark. One sub-problem of $(Pm, P1) \mid ppmtn \mid C_{max}$ worth investigating is $(Pm, P1) \mid q_j = q, ppmtn \mid C_{max}$. It is a particular case of $Q2 \mid\mid C_{max}$, so the problem is still NP-hard, but, for this particular case, the dual approximation scheme is not necessary in order to obtain a similar approximation ratio. Here, a lower bound of the makespan of the schedule is $\sum_{i=1}^{n} \overline{p_i}/(m+q)$. The tasks with the largest processing times are assigned to the GPU up to this bound, and one more task is assigned to the GPU. This additional task plays the same part as T_{last} in the previous proof. Since the additional task is placed on the GPU here, the approximation ratio becomes $1 + \frac{1}{q}$, and the time complexity of the algorithm is still $\mathcal{O}(n \log n)$.

6.2.2 Multiple GPUs Case

For problem $(Pm, Pk) \mid ppmtn \mid C_{max}$, with $k \geqslant 2$, the algorithm proposed for $k = 1$ provides a ratio of $1 + \max\left(\frac{1}{m}, 1 - \frac{1}{k}\right)$: the computing area on the GPUs is filled up to

$k\lambda$, but for $k \geqslant 2$ the scheduling of the tasks assigned to the GPUs cannot be done as easily as before since the performance ratio of the scheduling algorithm on the GPUs is similar to the one of the classical list algorithm: $2 - \frac{1}{k}$.
We can also extend the approximation algorithm developed for problem $(Pm, Pk) \mid\mid C_{max}$ in Chapter 4, Section 4.4, to problem $(Pm, Pk) \mid ppmtn \mid C_{max}$, with an approximation ratio of $\frac{4}{3} + \frac{1}{3k}$ and a time complexity in $\mathcal{O}(n^2k^3)$.
If λ is the current guess of the dual approximation, the algorithm presented for $(Pm, Pk) \mid\mid C_{max}$ in Chapter 4, Section 4.4 partitions the set of tasks on CPUs into two sets, each set consisting of two shelves, and does the same partition on the GPUs: a first set with a shelf S_1 of length λ and the other S_2 of length $\frac{\lambda}{3}$, occupying κ GPUs and a second set with two shelves S_3, S_4 of length $\frac{2\lambda}{3}$, occupying $k - \kappa$ GPUs. In order to do so, the number of tasks on each shelf is constrained, for the CPUs and the GPUs.
However, in $(Pm, Pk) \mid ppmtn \mid C_{max}$, there is no need for the shelves on the CPUs, since preemption renders the objective of minimizing the makespan equivalent to the objective of minimizing the computational area. By construction of the shelves, the makespan on the GPUs does not go over $\frac{4\lambda}{3} + \frac{\lambda}{3k}$. Since preemptions are allowed on CPUs, the makespan on the CPUs equals to $\frac{W_C}{m}$, We obtain the following problem, using the same variables as in the previous chapters:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{6.1}$$

$$\text{s.t. } \frac{1}{2} \sum_{2\lambda/3 \geqslant \underline{p_j} > \lambda/3} (1 - x_j) + \sum_{\underline{p_j} > 2\lambda/3} (1 - x_j) \leqslant k \tag{6.2}$$

$$N = \sum_{T_j \ / \ x_j = 0} \nu_j \leqslant 3kn \tag{6.3}$$

$$x_j \in \{0, 1\} \tag{6.4}$$

This problem is solved by dynamic programming with a time complexity in $\mathcal{O}(n^2k^3)$ per step of the dual approximation.

Remark 6.2.2. We can note that the introduction of preemptions on CPUs achieves a saving of m^2 in the time complexity bound of the algorithm described in Chapter 4, Section 4.4.

6.3 Moldable Tasks

With the development of parallel and distributed systems came a new type of application, more complex than the previous programs: parallel application. The tasks of a parallel program can be considered as indivisible pieces of the application that are executed sequentially on a processor. Scheduling these tasks requires sophisticated

algorithms to determine a date for each task to start its execution together with a processor location. Then arises the question of considering the communications between tasks of the same application executed on different processors.
In the *moldable* tasks model (denoted MT) [24], a function represents the parallel execution time of a task with the penalty due to the management of the parallelism including communications between parallel processors, synchronization, etc. In this model, a moldable task is a computational unit which may be executed on several processors with a running time that depends on the number of processors assigned to it.

6.3.1 Problem Definition

We consider again 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, together with a set of n functions $\overline{p_i} : l \mapsto \overline{p_{i,l}}$ that represent the processing time of task T_i when executed on l CPUs and a set of n numbers $\underline{p_i}$ corresponding to the processing time of T_i when executed on a GPU. We assume that these processing times are known in advance (it is a common assumption in case of classical numerical codes like those considered in the experiments).
The problem consists in finding for each task T_i a starting time $\sigma(i)$ and a subset $\mathcal{P}_i$ of processors to execute it, under the constraints that a task T_i starts its execution simultaneously on all the processors of $\mathcal{P}_i$ and occupies them without interruption until its completion time $C_i = \sigma(i) + t_{i,\mathcal{P}_i}$, where

$$t_{i,\mathcal{P}_i} = \begin{cases} \overline{p_{i,|\mathcal{P}_i|}} & \text{if } \mathcal{P}_i \text{ corresponds to } |\mathcal{P}_i| \text{ CPUs} \\ \underline{p_i} & \text{if } \mathcal{P}_i \text{ corresponds to a GPU} \end{cases}$$

We define the CPU work function w_i of a task T_i, which corresponds to its computational area on the CPUs in the Gant chart representation of a schedule, as $w_i : l \mapsto w_{i,l} = l \times \overline{p_{i,l}}$ for $l \leqslant m$. According to the usual executions of parallel programs, we assume that the tasks assigned to the CPUs are monotonic: allocating more CPUs to a task usually decreases its execution time at a price of increasing its work (with some internal communications and synchronizations). There are two types of monotony, namely the *time monotony* which is achieved when $\overline{p_i}$ is a decreasing function for all the tasks and the *work monotony* which is achieved when w_i is an increasing function for the tasks. A set of tasks is said *monotonic* when it achieves both monotonies. This assumption may be interpreted by the well-known Brent's lemma [14], 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 monotony property, replacing function $\overline{p_i}$ by $\overline{p_i'} : l \mapsto \min\{\overline{p_{i,q}} \mid q = 1, \ldots, l\}$. 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 of such an hypothesis on the GPUs, since the tasks can only be processed on one GPU at the same time.
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 (denoted by C_{max}^{CPU}) and the makespan on the GPUs (C_{max}^{GPU}).
This study is restricted to algorithms that provide non-preemptive schedules with contiguous processor allocation. It is clear that the optimal assignment could use CPUs that are not consecutive ones. However, this restriction does not have a substantial impact on the achieved results [67].

6.3.2 Related Work

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. Classical scheduling (i.e. those with sequential tasks) are a particular case of this problem, and hence their complexity results apply directly to MT problems. It implies that scheduling independent moldable tasks is NP-hard [33], in the ordinary sense if the number of machines m is fixed.
Jansen and Porkolab [49] 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 high dependent of the accuracy of the approximation due to an exponential factor in the number of processors. Thus, even though 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 [86]. The basic idea here was 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, which 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 [5, 21].
It is clear that applying an approximation of guarantee λ for the rigid problem on the assignment of an optimal solution provides the same guarantee λ 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 [62, 63] improved the complexity of the assignment selection in the special case of monotonic tasks leading to a 2-approximation. 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 $(\sqrt{3}+\epsilon)$-approximation for monotonic tasks [66]. Then, Mounié *et al.* [67] focused on

the second approach and showed how a $(\frac{3}{2}+\epsilon)$-approximation algorithm can be obtained for any $\epsilon > 0$.

6.3.3 Building a feasible Schedule

The principle of the algorithm is again to use the dual approximation technique.
We target $g = \frac{3}{2}$. Let λ be the current real number input for the dual approximation. In the following, we assert that there exists a schedule of length lower than λ. Then, we have to show how it is possible to build a schedule of length at most $\frac{3\lambda}{2}$.
Given a real number h, we can define as in [67] for each task T_i its canonical number of CPUs $\gamma(i,h)$ as the minimal number of CPUs needed to execute task T_i in time at most h. If T_i cannot be executed in time less than h on m CPUs, we set by convention $\gamma(i,h) = +\infty$.
Notice 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. In addition $w_{i,\gamma(i,h)}$ is also the minimal work area needed to execute T_i on CPUs in time less than h.
From [67], we know that, given a real number h, if $\gamma(i,h) < +\infty$, the execution time of task T_i on its canonical number of CPUs satisfies the inequality

$$h \geqslant \overline{p_{i,\gamma(i,h)}} > \frac{\gamma(i,h)-1}{\gamma(i,h)}h. \tag{6.5}$$

This inequality is a consequence of the monotonic behavior of the tasks on the CPUs, and if the canonical number of CPUs for a task T_i is at least 2, Equation 6.5 can be simplified into

$$2\,\overline{p_{i,\gamma(i,h)}} \geqslant \overline{p_{i,\gamma(i,h)-1}} > h \geqslant \overline{p_{i,\gamma(i,h)}} > \frac{1}{2}\,h. \tag{6.6}$$

6.3.3.1 Structuring Tasks into Shelves

The idea of the algorithm is to partition the set of tasks on the CPUs into five sets, and the set of tasks on the GPUs into two sets, as depicted in Figure 6.1.
On the CPUs:

- (0): the set containing the tasks sequentially assigned to the CPUs with a processing time lower than $\frac{\lambda}{2}$;
- (1): the set containing the tasks sequentially assigned to the CPUs with a processing time strictly greater than $\frac{\lambda}{2}$ and lower than $\frac{3\lambda}{4}$; this set can be divided into 2 shelves: the left shelf (L in Figure 6.1) and the right shelf (R in Figure 6.1).
- (2): the set containing the tasks assigned to the CPUs with different canonical numbers of CPUs for the times λ and $\frac{3\lambda}{2}$. Task T_i is then assigned to $\gamma(i, 3\lambda/2)$ CPUs;

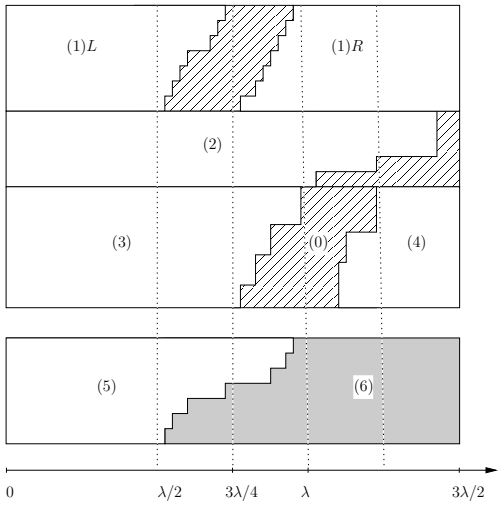


Figure 6.1: Structure of the schedule. For a better understanding, the processors are overloaded.

- (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 tasks assigned to their canonical number of CPUs for time $\frac{\lambda}{2}$, which is greater than 1.

On the GPUs:

- (5): the set containing the tasks assigned to a GPU with a processing time strictly greater than $\frac{\lambda}{2}$;
- (6): the set containing the tasks assigned to a GPU with a processing time lower than $\frac{\lambda}{2}$.

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

6.3.4 Analysis

6.3.4.1 Structure of a Schedule

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 6.3.1. *In an solution of makespan at most λ, the execution time of each task is at most λ and the computational area on the CPUs is at most $m\lambda$, as well as the computational area on the GPUs is at most $k\lambda$.*

We can note that for the problem of scheduling moldable tasks on identical processors [67], we only have 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 here.

Proposition 6.3.2. *In an solution of makespan at most λ, if there exist two consecutive tasks on the same processors such that one of them has an execution time greater than $\frac{\lambda}{2}$, then the other one has an execution time lower than $\frac{\lambda}{2}$.*

Proposition 6.3.3. *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 $\frac{3\lambda}{2}$.*

These properties allow us to write the following lemma.

Lemma 6.3.4. *If there exists an schedule S of makespan λ, then we can construct a schedule S' with the tasks partitioned into sets* (0), (1), (2), (3), (4), (5) *and* (6) *with a makespan at most $\frac{3\lambda}{2}$ and a CPU load lower than the CPU load of S.*

Proof. The tasks in the optimal schedule can be divided into two categories: those processed on the CPUs and those on the GPUs. The tasks are considered assigned to the canonical number of processors corresponding to their processing time in the optimal schedule. Assigning a task to more processors would only be a waste of resources and therefore would be suboptimal.
Let us start with the tasks assigned to the GPUs in the optimal schedule. The tasks are all sequential here, and can therefore be divided in two distinct sets, those with a processing time strictly greater than $\frac{\lambda}{2}$, and those with a processing time lower than $\frac{\lambda}{2}$, which corresponds exactly to the partition of sets (5) and (6) without changing anything to the optimal schedule.
Now we deal with the more complex case of the tasks assigned to the CPUs in the optimal schedule. We can classify the tasks into distinct categories:

- The tasks assigned to one CPU with a processing time lower than $\frac{\lambda}{2}$. These tasks corresponds to those assigned to set (0).
- The tasks assigned to one CPU with a processing time strictly greater than $\frac{\lambda}{2}$ and lower than $\frac{3\lambda}{4}$. These tasks corresponds to those assigned to set (1).
- The tasks assigned to one CPU with a processing time strictly greater than $\frac{3\lambda}{4}$ and the tasks assigned to their canonical number of CPUs for time λ, when this number is strictly greater than 1. These tasks corresponds to those assigned to set (3).
- The tasks assigned to their canonical number of CPUs for time $\frac{\lambda}{2}$, when this number is strictly greater than 1. These tasks corresponds to those assigned to set (4).

- The tasks assigned to their canonical number of CPUs for time h, where $\frac{\lambda}{2} < h < \lambda$ and for a task T_j, $\gamma(j,\lambda) < \gamma(j,h) < \gamma(j,\lambda/2)$. These tasks have no corresponding set in the partition we are aiming at. We consider them for now in a set (u).

It is clear that sets (0), (1) have no task in common and that they share no task with sets (3) and (4). What is unclear is the intersection of sets (3) and (4). If a task T_j had the same canonical number of processors for times λ and $\frac{\lambda}{2}$, this would mean that its processing time when assigned to λ processors, $\overline{p_{j,\gamma(j,\lambda)}}$, is lower than $\frac{\lambda}{2}$. However, we know, from Equation (6.6), that $\overline{p_{j,\gamma(j,\lambda)}} > \frac{\lambda}{2}$, which leads to a contradiction. We conclude that sets (3) and (4) have no tasks in common.
The only point that remains is the assignment of the tasks in set (u). From Proposition 6.3.2, the tasks whose execution times on CPUs are strictly greater than $\frac{\lambda}{2}$ do not use more than m CPUs, so we know that the tasks from sets (1), (3) and (u) cannot be executed on the same processors. Since there can only be one of these tasks on each CPU and all the tasks are independent, we can rearrange the order in which the tasks are processed so that the task starting its processing at time 0 on each CPU is a task from set (1), (3) or (u) if such a task was processed on this CPU. The only tasks that can be executed on the same processors after one of the tasks from sets (1), (3) and (u) are the tasks from sets (0) and (4) that fit in the remaining computational space. Let us consider only the CPUs occupied by a task from (1), (3) or (u) and denote their number by m_1. The time available in the optimal schedule to process tasks from sets (0) and (4) on each of these processors is lower than $\frac{\lambda}{2}$. In the schedule we aim to construct, the makespan is at most $\frac{3\lambda}{2}$, meaning that we add a time of $\frac{\lambda}{2}$ to the optimal schedule, which is enough to execute the tasks from sets (0) and (4) and still have a computation are of $m_1\lambda$ available to process the tasks from (1), (3) and (u). This means that each task T_j from (u) can now be processed in time λ, *i.e.* be assigned to $\gamma(j,\lambda)$ processors and therefore can be assigned to either set (1) or (3) depending on the value of $\gamma(j,\lambda)$. Therefore we have a partition of the tasks on the CPUs with sets (0), (1), (3) and (4). Set (2) is empty, but could be constructed easily if there are tasks from (3) with no other task executed on their processors and their canonical number of processors for time $\frac{3\lambda}{2}$ is different from the one for time λ. In that case, there is no obstacle to the processing of these tasks in time $\frac{3\lambda}{2}$ on a reduced number of processors and with a lower work. □

Now that we have proven that a schedule with our seven sets can be contracted from an optimal schedule, we look at exploiting the properties of said optimal schedule, in order to construct our sets.

- From Proposition 6.3.3, if we aim at a makespan of $\frac{3\lambda}{2}$, two tasks from (1) can be executed successively on the same CPU, occupying $\mu_{(1)}$ CPUs.

- From Proposition 6.3.2, the tasks whose execution times on CPUs are strictly 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 an optimal solution, 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 monotony 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 $\mu_{(3)} + \mu_{(2)} + \mu_{(1)} \leqslant m$ must be satisfied.
- The remaining tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU and those who 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 go on the CPUs that already process two tasks from (1), but if the number of tasks in (1) is odd, there is a CPU that only processes one task from (1) 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 $\mu_{(4)} + \mu_{(2)} + \mu_{(1)} - 1_{\mu_{(1)} \text{ uneven}} \leqslant m$ must be satisfied.
- The remaining sequential tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU and are executed in set (0).
- With the same reasoning, the tasks on GPUs whose execution times are strictly greater than $\frac{\lambda}{2}$ do not use more than k GPUs, and hence can be executed concurrently in set (5). We note κ the number of GPUs executing these tasks.
- 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, after a task from (5) or on the remaining free GPUs.

Thus, we are looking for a schedule on the CPUs in five sets and a schedule on the GPUs in two sets.

6.3.5 Formulation as a Linear Program

We define W_C as being 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 of the CPUs:

$$W_C = \sum_{T_j \in (0) \cup (1)} w_{j,1} + \sum_{T_j \in (2)} w_{j,\gamma(j,3\lambda/2)} + \sum_{T_j \in (3)} w_{j,\gamma(j,\lambda)} + \sum_{T_j \in (4)} w_{j,\gamma(j,\lambda/2)}.$$

In order to obtain a 5-set schedule on the CPUs and a 2-set schedule on the GPUs, we look for an assignment satisfying the following constraints:

(C_1) The total computational area W_C on the CPUs is at most $m\lambda$.

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

(C_3) Sets (1), (2) and (4) use a total of at most m processors, minus one if the number of tasks in set (1) is odd.

(C_4) The total computational area on the GPU is lower than $k\lambda$.

(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 sum of the numbers of tasks processed in each of its two shelves.

(C_8) The task of Set (1) are evenly shared between its two shelves, with at most one task less in the right shelf, which is processing tasks at the same time as Set (4).

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

Due to the monotonic assumption, we then have only five assignments to consider for a task: if it is selected to belong to (3), clearly $\gamma(i, \lambda)$ is a dominant assignment; if it is selected to belong to (2), $\gamma(i, 3\lambda/2)$ is a dominant assignment; if it is selected to belong to (4), $\gamma(i, \lambda/2)$ is a dominant assignment; if it is selected to belong to (1) or (0), the task is considered sequential and executed on a CPU. Otherwise it is selected to belong to (5) or (6), *i.e.* be on the GPU and the tasks scheduled on the GPU are considered sequential. According to Proposition 6.3.1, we note that $\gamma(i, \lambda)$ is at most m for all the tasks.

Determining if such an assignment exists reduces to solving a linear program (LP) that can be formulated as follows.

We define for each task T_j seven binary variables $x_j^{(q)}$, $q = 0, \ldots, 6$, such that $x_j^{(q)} = 1$ if T_j is assigned to Set (q) or 0 if T_j is assigned to another set. We also define for Set (1) the variable $left^{(1)}$ (resp. $right^{(1)}$), corresponding to the number of tasks assigned to the left (resp.right) shelf of Set (1) (see Figure 6.1).

$$\min\ W_C^{(LP)} = \sum_{j=1}^{n} \Big[w_{j,1}(x_j^{(0)} + x_j^{(1)}) + w_{j,\gamma(j,3\lambda/2)} x_j^{(2)} + w_{j,\gamma(j,\lambda)} x_j^{(3)} + w_{j,\gamma(j,\lambda/2)} x_j^{(4)} \Big] \quad (C_1)$$

$$\text{s.t.} \quad \sum_{j=1}^{n} \left(\gamma(j,\lambda) x_j^{(3)} + \gamma(j,3\lambda/2) x_j^{(2)} \right) + left^{(1)} \leqslant m \quad (C_2)$$

$$\sum_{j=1}^{n} \left(\gamma(j,\lambda/2) x_j^{(4)} + \gamma(j,3\lambda/2) x_j^{(2)} \right) + right^{(1)} \leqslant m \quad (C_3)$$

$$\sum_{j=1}^{n} \underline{p_j} \left(x_j^{(5)} + x_j^{(6)} \right) \leqslant k\lambda \quad (C_4)$$

$$\sum_{j=1}^{n} x_j^{(5)} \leqslant k \quad (C_5)$$

$$\sum_{q=0}^{6} x_j^{(q)} = 1 \quad \forall j \in 1, \ldots, n \quad (C_6)$$

$$\sum_{j=1}^{n} x_j^{(1)} = left^{(1)} + right^{(1)} \quad (C_7)$$

$$0 \leqslant left^{(1)} - right^{(1)} \leqslant 1 \quad (C_8)$$

$$x_j^{(q)} \in \{0,1\} \quad \forall j \in 1, \ldots, n \wedge \forall q \in 0, \ldots, 6 \quad (C_9)$$

$$left^{(1)}, right^{(1)} \in \mathbb{N} \quad (C_{10})$$

The first eight equations of this linear program correspond to the constraints listed above in order to obtain a 5-set schedule on the CPUs and a 2-set schedule on the GPUs. The last two equtations (C_9), (C_{10}) are integrity constraints for the variables of the linear program.
If we assume that there exists a schedule of makespan at most λ, and moreover that the condition of validation of the guess of the dual approximation is satisfied, i.e. if $W_C^{(PL)} \leqslant m\lambda$, we have the following lemmas:

Lemma 6.3.5. *With the assumption that $W_C^{(LP)} \leqslant m\lambda$, the tasks assigned to sets* (1), (2), (3) *and* (4) *occupy at most m CPUs, in a time at most $3\lambda/2$.*

Proof. From Constraints (C_2) and (C_3), we have the proof that the assignment of the tasks of these four sets is such that they occupy at most m CPUs when scheduled two by two in (1) and the tasks of (4) are scheduled after tasks from (3) or on remaining free CPUs, with the possibility of occupying one processor previously occupied by a task

from (1) if this set has an odd number of tasks. With this schedule, at most m CPUs are occupied and the makespan is lower than $3\lambda/2$. □

Lemma 6.3.6. *If $W_C^{(LP)} \leqslant m\lambda$, the tasks assigned to set* (0) *fit in the remaining free computational space, while keeping the makespan under* $3\lambda/2$.

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 $3m\lambda/2$, otherwise the schedule would not satisfy Proposition 6.3.1.
The following algorithm can be used to schedule these tasks:

- Consider the remaining tasks ordered by decreasing order of sequential processing time on CPU, $T_1, \ldots, T_f$, f being the total number of tasks remaining to be assigned.
- At each step i, $i = 1, \ldots, f$, assign task T_i to the least loaded CPU, at the latest possible date, or between Set (3) and Set (4) if relevant. Update its load.

At each step, the least loaded CPU has a load at most λ; otherwise it would contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Proposition 6.3.1). 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_i, which proves the correctness of the scheduling algorithm. □

Lemma 6.3.7. *If $W_C^{(LP)} \leqslant m\lambda$, the tasks assigned to sets* (5) *and* (6) *occupy at most k GPUs, in a time at most* $3\lambda/2$.

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, otherwise the dual approximation would reject the solution.
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 $3k\lambda/2$, otherwise the schedule would not satisfy Proposition 6.3.1 and Constraint (C_4).
The following algorithm can be used to schedule these tasks:

- Consider the remaining tasks ordered by decreasing order of processing time on GPU, $T_1, \ldots, T_f$, f being the total number of tasks remaining to be assigned.
- At each step i, $i = 1, \ldots, f$, assign task T_i to the least loaded GPU, at the latest possible date. Update its load.

At each step, the least loaded GPU has a load at most λ; otherwise it would contradict the fact that the total work area of the tasks is bounded by $k\lambda$ (according to Proposition 6.3.1 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_i, which proves the correctness of the scheduling algorithm. □

These three lemmas allow us to write the following theorem:

Theorem 6.3.8. *If $W_C^{(LP)} \leqslant m\lambda$, then, with the assignment of the tasks given by the solution of (LP), we can construct a schedule of length at most $\frac{3\lambda}{2}$.*

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

If the value of the guess of the dual approximation, λ, is rejected, then the computational area on the CPUs returned by the solution of (LP), $W_C^{((LP))}$, is greater than $m\lambda$. Since we minimize the computational area on the CPUs in the resolution of (LP), then if we had $\lambda \leqslant C_{max}^*$, we would get $W_C^{(LP)} \leqslant W_C^*$, which is impossible since we have $W_C^* \leqslant m\lambda$. Therefore in that case there exists no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation. Otherwise, we can construct a solution with a makespan at most $\frac{3\lambda}{2}$, with the corresponding sets on the CPUs and GPUs.

Binary Search We have described one step of the dual-approximation algorithm, with a fixed guess. A binary search will be used to try different guesses to approach the optimal makespan as follows.
We first take an initial lower bound B_{min} and an initial upper bound B_{max} of the optimal makespan. We start by solving the problem with λ equal to the average of these two bounds and then we adjust the bounds:

- If the previous algorithm returns “NO”, then λ becomes the new lower bound.
- If the algorithm returns a schedule of makespan at most $\frac{3\lambda}{2}$, then λ becomes the new upper bound.

The number of iterations of this binary search can be bounded by $\log(B_{max} - B_{min})$.

6.4 Looking at uniform CPUs and uniform GPUs

The problems we studied in this work were all dealing with a set of identical CPUs and a set of identical GPUs to schedule our tasks on. However, it can happen that on some platforms, the CPUs are not identical and the same can be said for the GPUs. Since the non-identical CPUs would have a similar architecture, it is safe to assume that the processing times of a set of tasks on a type of CPU would be proportional to the processing times these tasks would have on another type of CPU. Therefore, the CPUs

can be considered as uniform machines, as well as the GPUs. Computing platforms being composed of a great number of processors, it is clear that an occurrence of only one processor of a given type would highly unlikely. We therefore consider an instance of the problem with c different types of CPUs, composed of $m_1, \ldots, m_c$ processors for each type, and g different types of GPUs, with $k_1, \ldots, k_g$ processors for each type. The tasks are again considered sequential on both the CPUs and the GPUs.
Using the dual approximation technique, we can adapt the knapsack formulation of problem $(Pm, Pk) \mid\mid C_{max}$ we presented in Section 4.4, by replacing the constraints imposing a certain number of tasks in each shelf by additional constraints regarding the computational areas of the different types of processors. Indeed, if the objective to minimize is now the computational area of the first set of identical CPUs, all the other computational areas should be constrained to remain lower than the value of the objective function. If λ is the current guess of the dual approximation, and we introduce binary variables $x_j^{m_i}$, x_{k_h} corresponding to the type of processor a task T_j is assigned to $(i = 1, \ldots, c,\ h = 1, \ldots, g)$, s_{m_i}, s_{k_h} being the corresponding speeds, we have the following formulation:

$$
\begin{aligned}
W_C^* = \min & \sum_{j=1}^{n} \frac{\overline{p_j}}{s_{m_1}} x_j^{m_1} \\
\text{s.t.} \quad & \sum_{j=1}^{n} \frac{\overline{p_j}}{s_{m_2}} x_j^{m_2} \leqslant m_2 \lambda \\
& \vdots \\
& \sum_{j=1}^{n} \frac{\overline{p_j}}{s_{m_c}} x_j^{m_c} \leqslant m_c \lambda \\
& \sum_{j=1}^{n} \frac{p_j}{s_{k_1}} x_j^{k_1} \leqslant k_1 \lambda \\
& \vdots \\
& \sum_{j=1}^{n} \frac{p_j}{s_{m_g}} x_j^{k_g} \leqslant k_g \lambda \\
& x_j \in \{0, 1\} \quad j = 1, \ldots, n
\end{aligned}
$$

This problem can be solved with dynamic programming in polynomial time if we discretize the constraints with the same discretization technique that was used in Chapter 4, Section 4.3.3. With $m_c - 1 + k_g$ constraints to discretize, we obtain a time complexity of $\mathcal{O}\left(n^{m_c+k_g} m_1 \ldots m_c k_1 \ldots k_g\right)$ per step of dual approximation.
It is interesting to note that the power of the number of tasks in the time complexity of the algorithm is equal to the number of types of processors considered in the problem.

However, with such an exponent, it is clear that with several types of processors, such an algorithm would not be practical for a real-time implementation.

Chapter 7

Experiments

To assess the good behavior of the scheduling algorithms proposed in the previous chapters, we drive an experimental analysis based on various classes of instances. Some of them are obtained using a generation scheme with random values and others are derived from real data. The $\frac{4}{3}$-approximation algorithm presented in Chapter 4, Sections 4.3 and 4.4 is compared to other existing scheduling algorithm as well as a lower bound or an optimal value derived from the integer linear programming formulation of the problem. Experiments were also conducted to compare the 2-approximation algorithm from Chapter 4, Section 4.2.3 to the classical HEFT algorithm presented in Chapter 4, Section 4.2.1. Then, an implementation of both the 2 and $\frac{4}{3}$-approximation algorithms on a real run-time system were realized and tested on a classical Linear Algebra kernel. Finally, we present an application of our approximation algorithm with a performance ratio of $\frac{3}{2}$ (see Chapter 5) for the implementation of the Smith Waterman algorithm in the field of biological sequence comparison.

7.1 $\frac{4}{3}$-approximation Algorithm Experimental Analysis

We compare first the $\frac{4}{3}$-approximation algorithm presented in Chapter 4, Sections 4.3 and 4.4, denoted by DP for dynamic programming, with a ratio of $\frac{4}{3} + \frac{1}{3k}$ to two greedy list algorithms, namely, an arbitrary list algorithm (LIST) and the LPT algorithm, then, to the HEFT algorithm. All the algorithms are implemented in C++ programming language and run on a 3.4 GHz PC with 15.7 Gb RAM. All the experiments show that the CPU time of the DP algorithm is fast for small instances but it is limited for too large instances. This is not surprising since the time complexity of the $\frac{4}{3}$-approximation algorithm is $\mathcal{O}(n^2m^2k^3)$.

7.1.1 First experiments based on random simulations

We first run a series of experiments on random instances of various sizes: 10, 20, 40 and 80 tasks, 1, 2, 4, 8, 16, 32 and 64 CPUs, 1, 2, 4 and 8 GPUs. The processing times on

the CPUs are randomly generated using the uniform distribution U[10,100] so that $\overline{p_j} \in \{1, \ldots, 100\}$ for each task T_j. All the tasks have the same acceleration factor q chosen in $\{1, 5, 10, 50\}$. The resulting processing times on the GPU are thus $\underline{p_j} = \frac{\overline{p_j}}{q}$.
The aim of these preliminary experiments is to practically compare the quality of the solutions obtained by the DP algorithm with the ones obtained by the greedy algorithms. Indeed, even if a theoretical performance guarantee ratio has been provided, the objective here is to show that the algorithm outperforms the greedy algorithms in terms of mean/max deviations.
Figure 7.1 represents the mean deviations of the makespans of the solutions returned by our algorithm and both greedy algorithms (i.e. LIST and LPT) from the optimal makespan obtained with a linear program solved by Cplex [40] for different acceleration factors for 40-tasks instances with $m = 1$ and $k = 1$. The mean deviations of the DP algorithm remain very low for all instances with small acceleration factors while they are more important for the greedy algorithms. For these instances, the GPU can be viewed as a speeding up resource compared to the CPU. We can also notice that the highest mean deviations of the approximation algorithm compared to the optimal makespan is less than 10% for these classes of instances with identical acceleration factors.

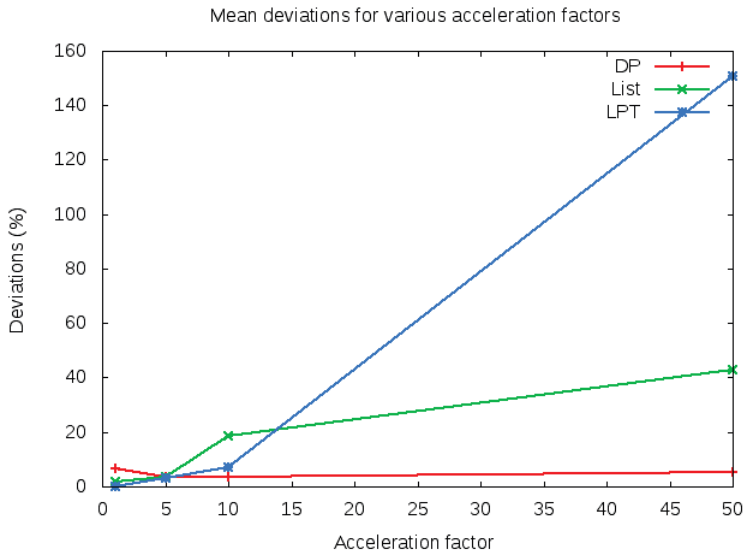


Figure 7.1: Gaps for various acceleration factors, $n = 40$, $m = 1$ and $k = 1$.

Tables 7.1 and 7.2 represent the mean deviations (Gap) of the makespan of our algorithm and the two greedy algorithms, i.e. List and LPT, compared to the last lower bound calculated by our algorithm during the binary search. Table 7.1 represents the mean deviations of the three algorithms with $m = 16$, $k = 1$ and $k = 4$ and Table 7.2 represents the mean deviations for different values of the common acceleration factor q. The small Gap values confirm the good behavior of our algorithm.

$m = 16, k = 1$

n	Gap DP	Gap List	Gap LPT
10	14,01%	317,96%	317,96%
20	12,68%	148,00%	148,72%
40	27,45%	113,17%	72,56%
80	19,82%	72,32%	33,11%

$m = 16, k = 4$

n	Gap DP	Gap List	Gap LPT
10	23,84%	1252,52%	1252,52%
20	18,93%	719,44%	750,07%
40	16,45%	309,34%	297,64%
80	16,98%	152,83%	129,28%

Table 7.1: Mean deviation for $m = 16$ and $k = 1, 4$ with different values of n

$m = 16, k = 1$

Acc. Fact.	Gap DP	Gap List	Gap LPT
1	10,77%	15,5%	3,87%
5	10,68%	38,29%	18,69%
10	18,88%	86,1275%	70,98%
50	33,63%	511,53%	478,82%

$m = 16, k = 4$

Acc. Fact.	Gap DP	Gap List	Gap LPT
0,02	24,47%	1906,74%	1944,81%
0,1	19,47%	369,91%	345,01%
0,2	15,56%	130,81%	123,27%
1	16,71%	26,69%	16,42%

Table 7.2: Mean deviation for $m = 16$ and $k = 1, 4$ with different acceleration factors

7.1.2 A more realistic benchmark

The second series of experiments were conducted using a more realistic benchmark. As we did not find adequate datasets, we constructed our own benchmark as follows: the execution time of the independent tasks have been extracted from the actual RICC log (the last one of the collection *Parallel Workloads Archive* at the time, May 2010) of Feitelson [30]. We extracted randomly 30 sets of 80 sequential tasks, among the sequential tasks with a running time between 5 seconds and 5 minutes (25% of the 6974 tasks of the RICC Log). The distribution of the acceleration factors on the GPU have been measured in [75] using the classical numerical kernels of Magma [2] in a multi-core multi-GPU machine hosted by the Grid'5000 infrastructure experimental platform [11]. We extracted a distribution of the acceleration factors q_j which reflects the qualitative speed-up on real kernels: we assign to each task an acceleration factor $q_j = \frac{\overline{p_j}}{\underline{p_j}}$ of 15 or 35 with a probability of 1/2. Then, we extract randomly the tasks by groups of size 10 to 70 from these sets.
Every point in Figures 7.2, 7.3, 7.4 and 7.5 represents the average value over 30 instances. In these experiments, we compared the performance of our $\frac{4}{3}$-approximation algorithm with only HEFT.

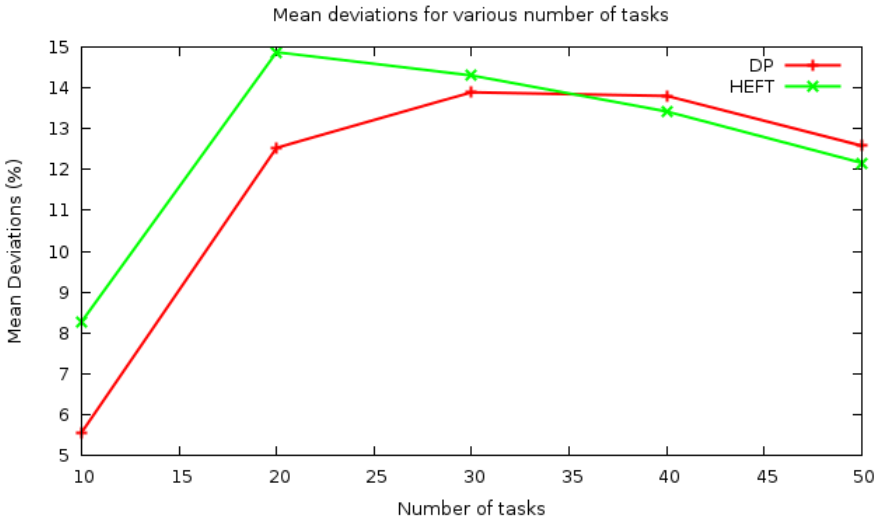


Figure 7.2: Gaps for various numbers of tasks, $m = 16$ and $k = 4$.

Figure 7.2 represents the mean deviations of the makespan compared to the last lower bound computed by the dual approximation in the binary search, for various numbers of tasks, and $m = 16$, $k = 4$. As we can see, our algorithm outperforms HEFT for small instances, and their performances are similar for larger instances.
We represented in Figure 7.3 the maximum deviation and minimum deviation in addition to the mean deviation of the previous figure for both algorithms, and we observe that the maximum deviation of HEFT often goes over the 33% limit of the $\frac{4}{3}$

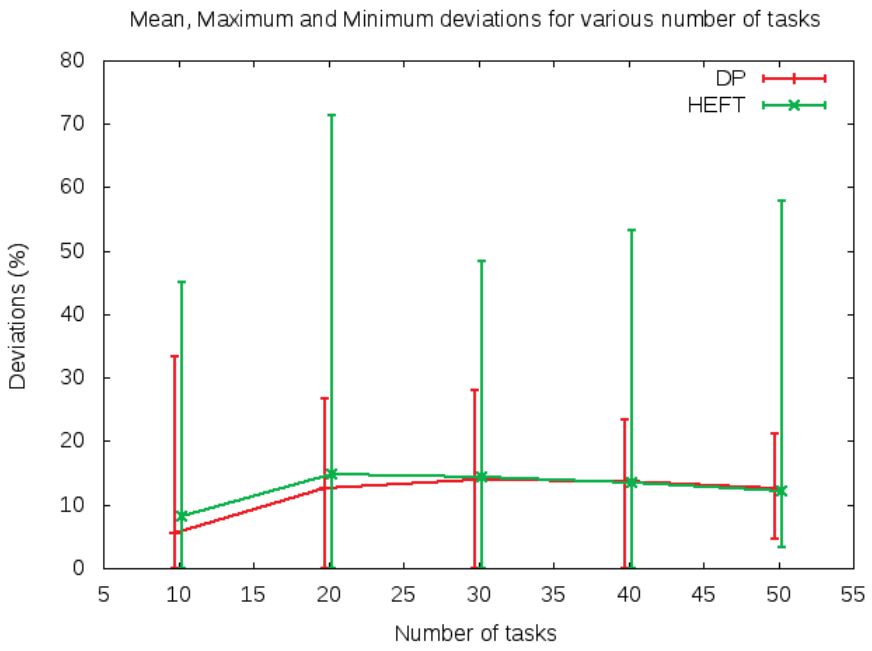


Figure 7.3: Maximun, mean and minimum deviations for various numbers of tasks, $m = 16$ and $k = 4$.

guarantee. When the number of processor is small, DP is better than HEFT even for the larger instances.

Figures 7.4 and 7.5 compare the algorithms for $m = 1$ and $k = 1$ in a similar fashion as Figures 7.2 and 7.3. At the cost of a larger complexity, the DP algorithm provides schedules consistently more stable than the algorithms used classically in the parallel programming environment for CPU-GPUs.

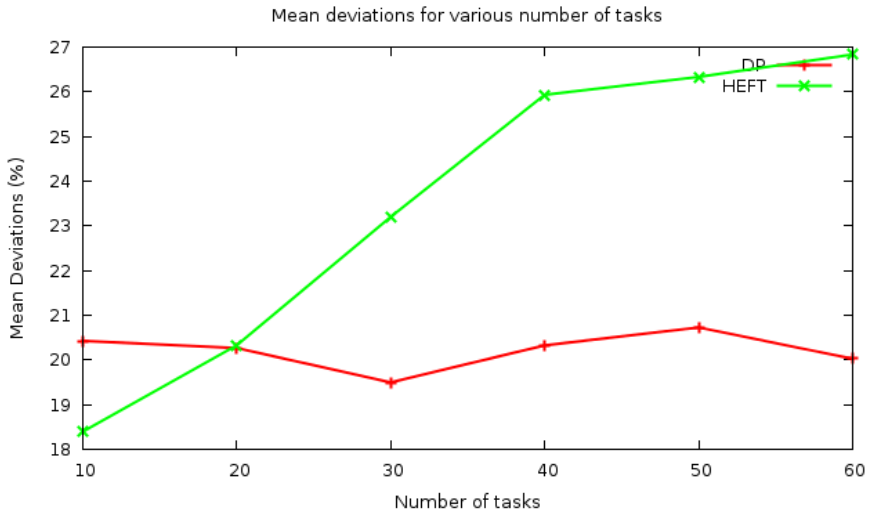


Figure 7.4: Gaps for various numbers of tasks, $m = 1$ and $k = 1$.

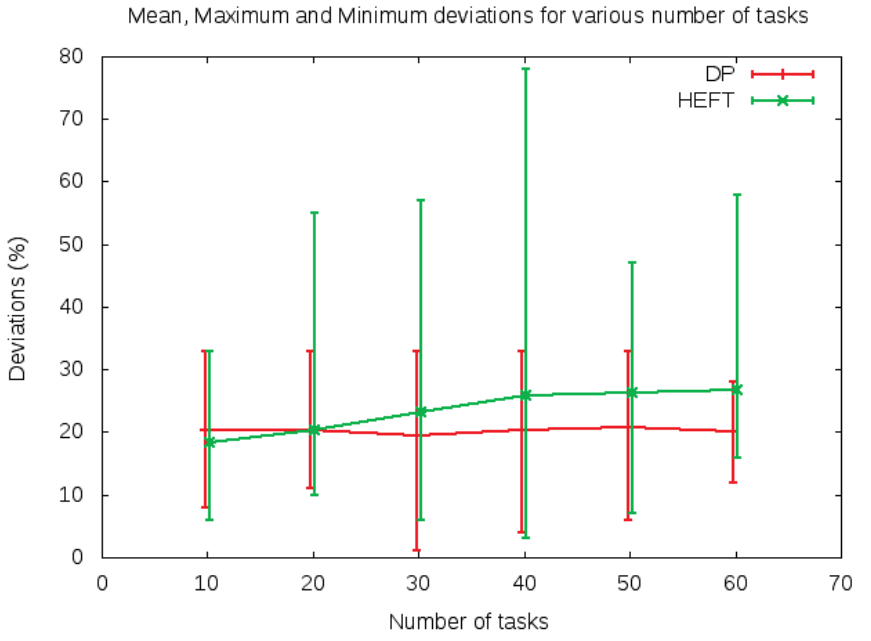


Figure 7.5: Maximun, mean and minimum deviations for various numbers of tasks, $m = 1$ and $k = 1$.

7.2 Experiments with the 2-approximation algorithm and the algorithm for the case when all the tasks are accelerated

The dual approximation based algorithm for problem $(Pm, Pk) \mid\mid C_{max}$ presented in Chapter 4, Section 4.4 provides a performance ratio of $\frac{4}{3} + \frac{1}{3k}$ with a reasonable time

complexity. However, this running cost is not comparable to the one of HEFT which basically only needs to sort the tasks ($\mathcal{O}(n \log n)$). But we also have a 2-approximation algorithm for problem $(Pm, Pk) \mid\mid C_{max}$ with a running time of $\mathcal{O}(n \log n)$ per step of dual approximation, presented in Chapter 4, Section 4.2.3. This algorithm is comparable to HEFT in terms of running time and still provides a performance guarantee. This 2-approximation algorithm, denoted by *Ratio2* in what follows, was implemented and compared to HEFT by simulations based on various classes of instances. Moreover, for the special case where all the tasks are accelerated, we implemented the algorithm presented in Chapter 6, Section 6.1, denoted by *Accel*, which provides a performance ratio of $\frac{3}{2}$ with a time complexity of $\mathcal{O}(mn \log n)$. All these algorithms were again implemented in C++ programming language and run on a 3.4 GHz PC with 15.7 Gb RAM.

We report below a series of experiments run on the same random instances as in Section 7.1.2: from 10 to 1000 tasks, with a step of 10 tasks, 2^a CPUs, a varying from 0 to 6, and 2^b GPUs, b varying from 0 to 3. For each combination of these sizes, 30 instances were considered, bringing us to a total of 10500 tested instances. The processing times on the CPUs are again randomly generated using the uniform distribution $U[10, 100]$. The distribution is the one based on the Magma kernels presented in the previous section. Since in this generation scheme all the tasks of these instances are accelerated on GPU, DP, HEFT and Accel were all compared on these instances. The running time of the three algorithms is always under one second, even for the largest instances. We calculated the mean and maximal deviations of the makespans of the solutions returned by these algorithms from the lower bound of the makespan derived from the binary search of the approximation algorithm, over all the instances. As we can see in Table 7.3, the maximal deviations of Ratio2 are usually below the maximal deviations of HEFT and more importantly these deviations respect the theoretical performance guarantee in the case of Ratio2 whereas the maximal deviations of HEFT sometimes go over the 100% barrier corresponding to a performance ratio of 2. The same can be said for Accel, with maximal deviations staying below the 50% barrier corresponding to a performance ratio of $\frac{3}{2}$.

n	120	160	220	260	360	380
Ratio2	76.88	72.73	70.37	69.14	70.00	70.00
HEFT	123.53	98.44	92.55	91.90	110.37	91.78
Accel	46.15	42.86	50.00	41.18	37.82	43.59

n	660	700	760	780	920	940
Ratio2	67.42	50.82	42.77	54.47	91.77	63.07
HEFT	113.48	98.10	98.77	103.15	116.46	96.31
Accel	36.36	40.91	32.52	37.04	48.24	34.65

Table 7.3: Maximal deviations (%) for Ratio2, HEFT and Accel.

Figure 7.6 shows that in average, Ratio2 even outperforms HEFT for large instances.

However, Accel remains slightly above HEFT, staying close to its $\frac{3}{2}$ bound but never going above it, contrary to HEFT which does not have a performance guarantee, as seen in Table 7.3. This better performance ratio of $\frac{3}{2}$ with its low cost make Accel preferable to Ratio2 for practical intensive use with still a better maximum performance guarantee.

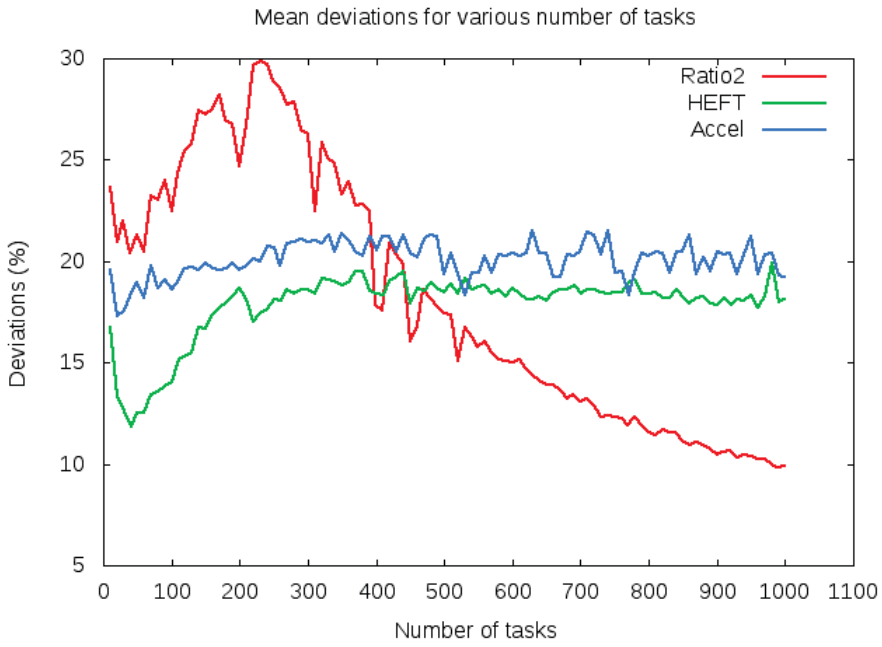


Figure 7.6: Mean deviations of Ratio2, HEFT and Accel for various n.

7.3 Experiments on a real run-time

We investigate in this section the practical use of the algorithms we previously conducted an experimental analysis on. We target classical linear algebra kernels, since they are extensively used and they generate a loop of independent tasks. The Ratio2, HEFT and DP algorithms were all implemented in the scheduler of the xKaapi run-time system [34].

7.3.1 Implementation of the $\frac{4}{3}$-approximation algorithm

In most linear algebra applications, the block size is the most important characteristic to take into account in order to maximize the performance. Indeed, the block size has a direct impact on memory transfers between the host and the accelerators, and on cache effects, all key players in the length of the processing time of an application.
We study the variation of the computation time as a function of the block size for the same matrix size of a Cholesky factorization extracted from the MAGMA library. The

results are presented in Figure 7.7. We use a single GPU and the matrix is decomposed over simple square blocks. The experiments have been conducted on a quad-core Intel i7-3840QM with hyper threading and a Nvidia Quadro K1000M GPU.

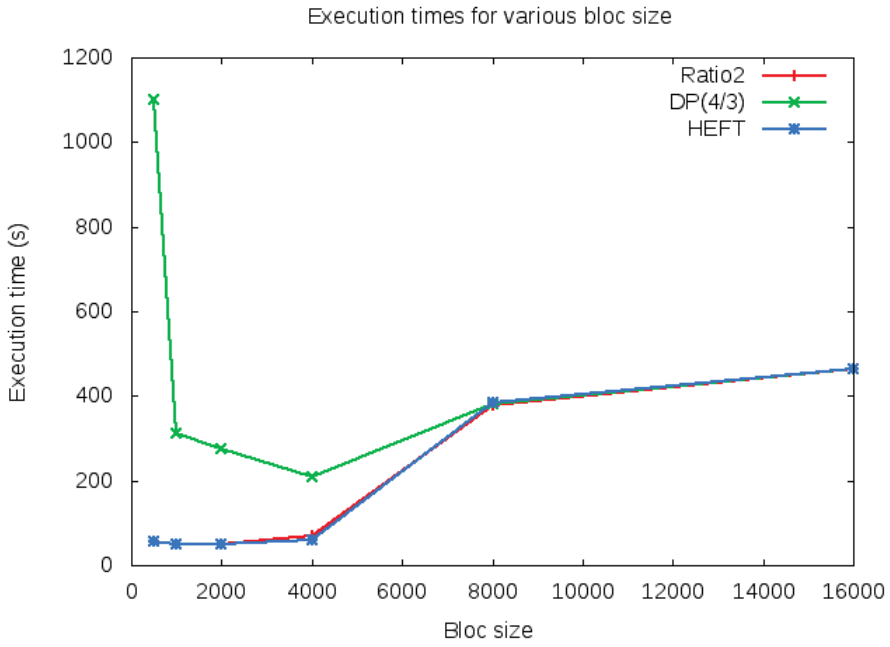


Figure 7.7: Execution time of a Cholesky factorization scheduled by Ratio2, DP (4/3) and HEFT for various block sizes, on 3 hyper threaded CPUs and a single GPU

Remark 7.3.1. We can observe that we only use 3 CPUs of the quad-core for the Cholesky factorization calculations because the 4^{th} one is used to control the GPU.

As the block size decreases, the number of independent tasks increases. Thus, the computation time of the scheduling using the $\frac{4}{3}$-approximation (DP) algorithm increases quadratically with the number of tasks too. As a result, the scheduling time dominates the execution time saved for a large block size (which corresponds to a small number of tasks). The $\frac{4}{3}$-approximation algorithm is therefore usable mostly for cases where the computation time is larger than the scheduling time. It is probably not the best suited algorithm for linear algebra kernels.

7.3.2 Practical issues: 2-approximation algorithm versus HEFT

We now compare our 2-approximation algorithm (Ratio2) with HEFT on a machine with several GPUs. These experiments have been conducted with the same calculations as in Section 7.3.1 on a heterogeneous, multi-GPU system composed of two six-core Intel Xeon X5650 CPUs running at 2.66 GHz with 72 GB of memory. This parallel system is enhanced with eight NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores

(scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB in total). It has 4 PCIe switches to support up to 8 GPUs. When 2 GPUs share a switch, their aggregated PCIe bandwidth is bounded by the one of a single PCIe 16x.
The structure of the 2-approximation algorithm allowed us to combine its implementation with an improved local mapping in order to minimize data transfers [9]. We studied the number of operations per second and the size of the memory transfers for both the Ratio2 algorithm and the HEFT algorithm. The results are shown in Table 7.4

Algorithm	Gflops	Memory transfer/GB
HEFT	535	2.62
Ratio2	565	1.91

Table 7.4: Performance of the 2-approximation algorithm and HEFT for Cholesky factorization with m=4 CPUs and k=8 GPUs

With 8 GPUs, the Ratio2 algorithm outperforms HEFT both in the raw performance and memory transfers. We can note that if the number of operations per second is not dramatically improved by the 2-approximation algorithm, the introduction of the procedure of local mapping allowed by the dual approximation algorithm leads to great results in terms of memory transfers. The execution times are close to each other in all cases, but our algorithm has the major advantages of having a performance guarantee on the makespan of the resulting schedule and providing a decrease in the volume of communication with the improved mapping.

7.4 An Application to Biological Sequence Comparison

7.4.1 Motivation

The family of approximation algorithms presented in Chapters 4 and 5 with different approximations ratios was applied to the implementation of a biological problem regarding the comparison of biological sequences with the performance ratio $\frac{3}{2}$.
Indeed, once a new biological sequence is discovered, its functional/structural characteristics must be established. In order to do that, the newly discovered sequence is compared against other sequences, looking for similarities. Sequence comparison is, therefore, one of the most crucial operations in Bioinformatics [68].
The most accurate algorithm to execute pairwise comparisons is the one proposed by Smith-Waterman (denoted by SW in short) [79], which is based on dynamic programming and run in quadratic time and space complexity in the length of the sequences. This can easily lead to very large execution times and huge memory requirements, since the size of biological databases is growing exponentially. Parallel implementations can be used to compute results faster, reducing significantly the time needed to obtain results with the SW algorithm. GPUs have been explored to speed-up the SW algorithm [23, 51, 61].

In [52], a new implementation of the Smith-Waterman algorithm, SWDUAL, on hybrid platforms composed of multiple processors and multiple GPUs, is proposed, with the scheduling of the calculations based on the $\frac{3}{2}$-approximation scheduling algorithm derived from the algorithms presented in Chapters 4 and 5.
Given a set of query sequences and a biological database, the strategy uses a one round master-slave approach to assign tasks to the processing elements according to the dual approximation scheduling algorithm.
First, a word on the sequence comparison problem and the classical SW algorithm.

7.4.2 Biological Sequence Comparison and Smith-Waterman Algorithm

A biological sequence is a structure composed of nucleic acids or proteins. It is represented by an ordered list of residues, which are nucleotide bases (for DNA or RNA sequences) or amino acids (for protein sequences). DNA and RNA sequences are treated as strings composed of elements of the alphabets $\sum = \{A, T, G, C\}$ and $\sum = \{A, U, G, C\}$, respectively. Protein sequences are also treated as strings which elements belong to an alphabet with, normally, 20 amino acids.
Since two biological sequences are rarely identical, the sequence comparison problem corresponds to approximate pattern matching. To compare two sequences, a good alignment between each other should be determined. This corresponds to placing one sequence above the other, making clear the correspondence between similar characters [68], creating two columns of two bases. Furthermore, in an alignment, some gaps (space characters) can be inserted in arbitrary locations such that the sequences end up with the same size. Given an alignment between sequences s and t, a score is associated to it as follows. For each two bases in the same column:

- a punctuation ma is associated if both characters are identical ($match$);
- a penalty mi, if the characters are different ($mismatch$);
- a penalty g, if one of the characters is a gap.

The score is obtained by the addition of all these values. The maximal score is called the similarity between the sequences. Figure 7.8 presents one possible global alignment between two DNA sequences and its associated score. In this example, $ma = +1$, $mi = -1$ and $g = -2$.

A	C	T	T	G	T	C	C	G
A	$-$	T	T	G	T	C	A	G
$+1$	-2	$+1$	$+1$	$+1$	$+1$	$+1$	-1	$+1$

$score = 4$

Figure 7.8: Example of an alignment and score

Smith-Waterman (SW) Algorithm
The SW algorithm [79] is an exact method based on dynamic programming to obtain the optimal pairwise local alignment in quadratic time and space in the length of the sequences.
The first phase of the SW algorithm starts by two input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ is the size of sequence s. The similarity matrix is denoted by $H_{m+1,n+1}$, where $H_{i,j}$ contains the score between prefixes $s[1..i]$ and $t[1..j]$. At the beginning, the first row and column are filled with zeros. The remaining elements of H are obtained from Equation (7.1). In addition, each cell $H_{i,j}$ contains the information about the cell that was used to produce the value. $S_{i,j}$ is a similarity score for the elements i and j

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i,j-1} + \mathrm{g} \\ H_{i-1,j} + \mathrm{g} \\ 0 \end{cases} \tag{7.1}$$

The SW algorithm assigns a constant cost to gaps. Nevertheless, in nature, gaps tend to appear in groups. For this reason, a higher penalty is usually associated to the first gap and a lower penalty is given to the following ones (this is known as the affine-gap model). Gotoh [36] proposed an algorithm based on SW that implements the affine-gap model by calculating three dynamic programming matrices, namely H, E and F, where E and F keep track of gaps in each of the sequences. The gap penalties for starting and extending a gap are G_s and G_e, respectively. These recursion formulas are given by the following equations:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases}$$

$$E_{i,j} = -G_e + \max \begin{cases} E_{i,j-1} \\ H_{i,j-1} - G_s \end{cases}$$

$$F_{i,j} = -G_e + \max \begin{cases} F_{i-1,j} \\ H_{i-1,j} - G_s \end{cases}$$

To parallelize the SW algorithm, the SWDUAL implementation uses a combination of classical parallelization approaches (see [52] for more details). Each of the platform computing processors compares one query sequence to one database sequence, in a more or less parallelized way depending on the type of processor used for the comparison. At the same time, other computing processors compare other sequences of the query set to the database in the same way.

7.4.3 SWDUAL implementation

In SWDUAL, the problem is to determine an allocation of the tasks to the computing CPUs and GPUs that minimizes the global completion time, i.e. the makespan. A master CPU uses the approximation algorithm from the family of algorithms described in Chapter 5 with a performance ratio of $\frac{3}{2}$ to schedule tasks to the computing processors, CPUs and GPUs. Each task is equivalent to the comparison of one sequence of a query set to a sequence of a database, i.e. a pairwise comparison. Additionally, all the sequences sizes are known beforehand, which simplifies the memory allocation process. This $\frac{3}{2}$-approximation algorithm has a time complexity in $\mathcal{O}(n^2mk^2)$ per step of the binary search, where n corresponds to the number of tasks to schedule, m and k are respectively the number of CPUs and GPUs available on the platform to execute the sequence comparisons.
This time complexity is important, but it can be lowered with special instances where all the considered tasks are accelerated when assigned to a GPU, which is the case for the sequence comparison problem addressed here. With the algorithm for this special case (see Chapter 6, Section 6.1), the time complexity reduces to $\mathcal{O}(mn\log(n))$, which is satisfactory for real implementations.

7.4.4 Experimental Results

The $\frac{3}{2}$-approximation scheduling algorithm was implemented in C++ with SSE extensions and CUDA. The SWDUAL strategy was implemented in C with SSE extensions and CUDA, and it integrates techniques from the classical SW methods CUDASW++ 2.0 [61] and SWIPE [73] into the code. This code was compiled with the CUDA SDK 4.2.9 and gcc 4.5.2. The operating system used was Linux 3.0.0-15 Ubuntu 64 bits. The tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000 amino acids, which were compared to 5 real genomic databases: Uniprot with 537,505 sequences (*www.uniprot.org*), Enbembl (*www.ensembl.org*) Dog with 25,160 sequences and Rat with 32,971 sequences and RefSeq (*www.ncbi.nlm.nih.gov/RefSeq*) Human with 34,705 sequences and Mouse 29,437 sequences.
The tests were executed in the Idgraf high performance computer located at Inria Grenoble. It contains 2 Intel Xeon 2.67GHz processors with 6 cores each (i.e. 12 CPUs in total), 74GB of RAM and 8 Nvidia Tesla C2050 GPUs.

Remark 7.4.1. We can note that even if 12 CPUs are available on the Idgraf platform, they cannot be all used for comparing sequences. Indeed, each GPU used for computations needs to be controlled by a CPU dedicated to this specific task, meaning for instance that if all the Idgraf GPUs are used to compare sequences, only 4 CPUs remain available for performing computations, leading to a total number of 12 processors available for calculations.

The Idgraf machine was reserved for exclusive use for the duration of the test to ensure that no other major process was running concurrently. All the sequences used were

available locally to minimize the influence of the network and file reading time. All combinations of programs, number of processors, query and database sequences were executed twenty-five times and the average total wall-clock execution time was recorded. Also, processor affinity was used to ensure that each process stayed in the same processor during the whole execution.

7.4.4.1 Comparison to other implementations

Table 7.5 shows the state-of-the-art implementations that were compared to SWDUAL, as well as their version number and command line options. For the commands, the variables were T for the number of threads, Q query sequence and D database sequence.

Table 7.5: Applications included in the comparison.

Application	Version	Command line
SWIPE	1.0	./swipe -a $T -i $Q -d $D
STRIPED		./striped -T $T $Q $D
SWPS3	20080605	./swps3 -j $T $Q $D
CUDASW++	2.0	./cudasw -use_gpus $T -query $Q -db $D

The SWDUAL implementation was compared against SWIPE, STRIPED, SWPS3 and CUDASW++.
SWIPE [73] was written mostly in C++ with some parts hand coded in assembly. It was compiled using the provided Makefile. The source code for the Farrar's STRIPED implementation of the SW algorithm [28] was compiled using the provided Makefile. It was written mainly in C with some parts also coded in assembly or Intel intrinsics. SWPS3 [82] was downloaded from the author's website and was written in C. It was compiled using the provided Makefile. CUDASW++ 2.0 [61] was also downloaded from the author's website and was written in C++ and CUDA. It was compiled using the provided Makefile. CUDA 4.1 was used in the compilation.
The tests were conducted using the UniProt database (*www.uniprot.org*) and 40 query sequences taken from it. Also, were used in this test up to four CPUs and four GPUs. For that reason the considered applications were executed with up to four processors, while SWDUAL, that uses both types of processors, CPUs and GPUs, was executed with a number of processors between two and eight: we start with one GPU and one CPU, then add one processor, alternating between types, starting with a GPU (i.e. three processors means two GPUs and one CPU).
The SWDUAL implementation was able to significantly reduce the execution time of the sequence database searches using the Smith-Waterman algorithm compared to earlier proposals that use only CPUs, i.e. SWPS3, STRIPED and SWIPE, as it can be seen on Figure 7.9 and Table 7.6. When executing with two processors, SWDUAL showed a reduction of 54.7%, 85% and 98% when compared to the same execution on SWIPE,

STRIPED and SWPS3, respectively. When executing with four processors, a reduction of 55.3% was obtained when compared to the execution on SWIPE, 73.5% when compared to STRIPED and 98.6% on SWPS3.

Application	Number of processors			
	1	2	3	4
SWPS3	69208.2	36174.09	25206.563	18904.31
STRIPED	7190	3615.38	1369.33	1027.28
SWIPE	2367.24	1199.47	816.61	610.23
CUDASW++	785.26	445.611	350.09	292.157
SWDUAL		543.28	472.84	271.98
Application	Number of processors			
	5	6	7	8
SWDUAL	266.69	239.04	183.12	142.98

Table 7.6: Execution times (s) for the compared implementations.

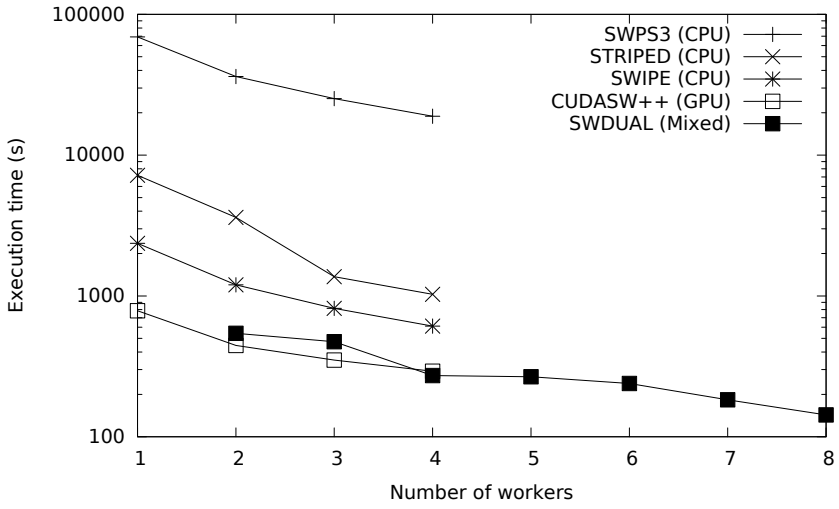


Figure 7.9: Execution times in seconds for the compared implementations.

The case of CUDASW++ is different. This classical implementation is designed to run on GPUs only, and when compared to SWDUAL using up to four processors, the execution times are comparable. This can be explained by the fact that CUDASW++ is the implementation we used to program the sequence comparison on GPU in SWDUAL. However, we can note that the hybrid implementation SWDUAL allow us to use more processors to perform the computations, and for the same number of computing

processors, SWDUAL actually uses less processors than CUDASW++: indeed, when CUDASW++ works on four GPUs, four CPUs are also working to control the GPUs, meaning a total of eight processors, whereas SWDUAL only uses two CPUs and two GPUs for the computations, and only two CPUs to control the GPUs, meaning only six processors in total.

7.4.4.2 Comparison to 5 genomic databases

In this case, the tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000 amino acids, which were compared to 5 real genomic databases, listed in Table 7.7.

Database	Number of database seqs	Smallest query seq	Longest query seq
Ensembl Dog Proteins	25,160	100	4,996
Ensembl Rat Proteins	32,971	100	4,992
RefSeq Human Proteins	34,705	100	4,981
RefSeq Mouse Proteins	29,437	100	5,000
UniProt	537,505	100	4,998

Table 7.7: Genomic Databases used on the tests.

In order to measure the benefits of using a hybrid platform, the wall-clock execution time and GCUPs (billion cell updates per second) obtained were measured when comparing 40 query sequences to the five genomic databases.

Nb CPUs/ Nb GPUs	1/1	2/2	4/4	4/8
Database	Time (s) GCUPS	Time (s) GCUPS	Time (s) GCUPS	Time (s) GCUPS
Ensembl Dog	78.36 18.91	39.63 37.39	20.45 72.45	12.87 115.13
Ensembl Rat	75.85 22.97	37.97 45.89	20.17 86.38	12.86 135.48
RefSeq Mouse	84.40 18.99	46.25 34.66	23.59 67.95	14.99 106.93
RefSeq Human	95.09 20.70	48.01 41.00	24.82 79.31	15.40 127.82
Uniprot	543.28 35.81	271.98 71.53	142.98 136.06	86.16 225.78

Table 7.8: Results running on CPUs and GPUs.

As can be seen on Table 7.8, SWDUAL was able to obtain good speedups while

combining CPUs and GPUs, reducing the execution time repeatedly while adding processing elements. For the Uniprot database the execution time was reduced from 543 seconds (approximately 10 minutes) to 86 seconds when executing on four CPUs and eight GPUs. Figure 7.10 shows the execution times obtained when comparing the databases.

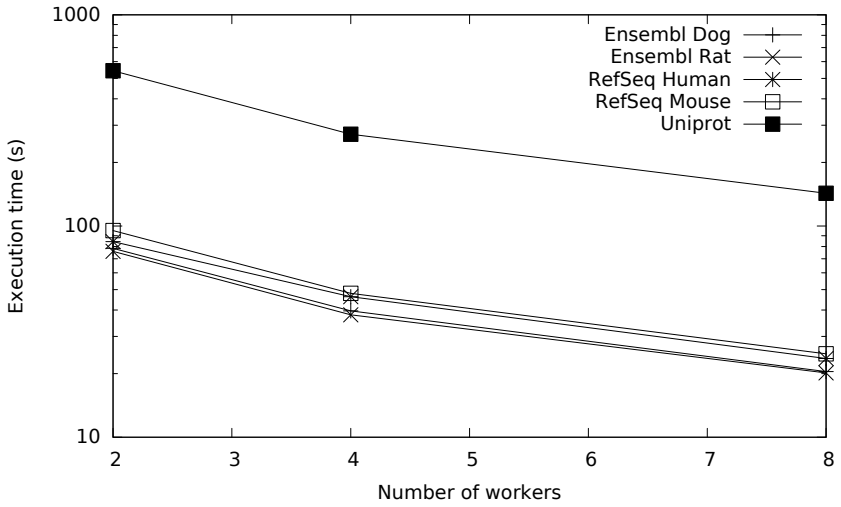


Figure 7.10: Execution times for the compared databases with SWDUAL.

7.4.4.3 Comparison of homogeneous and heterogeneous sets

For this test, two additional query sets were created from the Uniprot database. Each query set have, like in the previous tests, 40 sequences. In this case, the sequences in the homogeneous set range in size from 4500 to 5000 and the ones in the heterogeneous set have sizes between 4 (the smallest sequence in the database) and 35213 (the largest sequence in the database).
The idea is to verify that the allocation strategy and the application as a whole is equally able to work with sequences, and therefore tasks, that are similar in terms of size as well as tasks with very different sizes.
Table 7.9 shows the execution times and the GCUPs obtained when comparing these two sets to the UniProt database. In this case, SWDUAL was able to achieve good performance on both sets. Figure 7.11 also shows the results obtained in these comparisons.

7.5 Summary

In this chapter we presented the different experiments that were conducted in order to practically validate some of the algorithms presented in Chapters 4, 5 and 6. The

Nb CPUs/ Nb GPUs	1/1	2/2	4/4	4/8
Sets	Time (s) GCUPS	Time (s) GCUPS	Time (s) GCUPS	Time (s) GCUPS
Heterogeneous	3554.36 37.55	1785.73 74.74	908.45 146.92	528.26 252.67
Homogeneous	998.27 36.3	484.74 74.76	249.69 145.14	138.38 261.9

Table 7.9: Results running the homogeneous and the heterogeneous sets for SWDUAL.

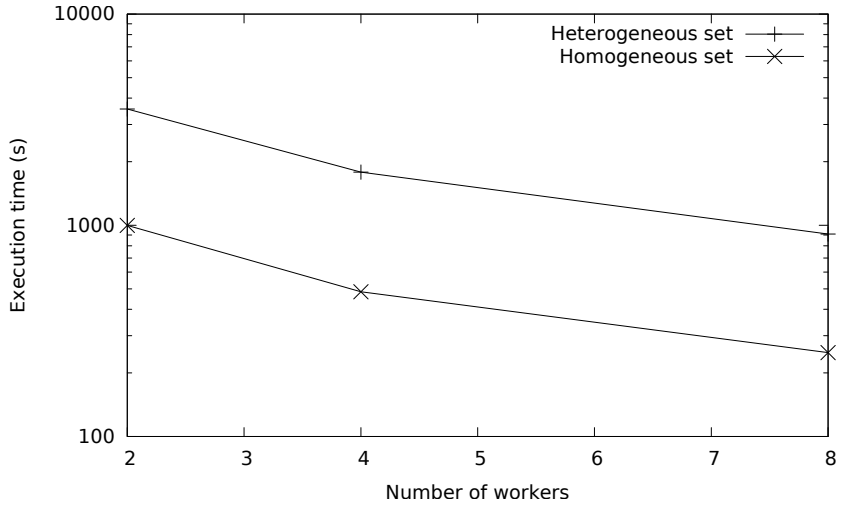


Figure 7.11: Execution times for the heterogeneous and homogeneous sets for SWDUAL.

experiments ranged from simulations on instances generated with random values to an actual implementation on a run-time and an application of our scheduling method to biological sequence comparison. Overall these experiments, the performance ratios of all the algorithms developed in this work were experimentally validated, and the dual approximation technique allowed good local mapping to minimize memory transfers, a point that can be crucial when performing calculations on GPUs. The time complexity of the algorithms with small approximation ratios may be too high for a generic use on a large-scale computing platform, however it may be of interest for some users who have very long calculations to schedule, for whom a mistake in the scheduling may result in an important delay in the acquisition of their results.

Chapter 8

Minimizing the Makespan with Dependent Sequential Tasks

In Chapters 4 to 6, we studied various instances of the problem of scheduling independent tasks on CPUs and GPUs with minimum makespan. However, it can happen that some tasks need the results of the execution of other tasks to start their processing. In that case, these tasks cannot be executed before the tasks whose results they need have finished their processing. The problem of interest in this chapter is the problem of scheduling dependent tasks on hybrid platforms.

8.1 Problem Definition

We consider again a multi-core parallel platform with m identical CPUs and k identical GPUs. An application is here composed of n sequential non-preemptive tasks denoted by $T = \{T_1, \ldots, T_n\}$, linked by precedence constraints. Let $G = (V, E)$ be a directed acyclic graph, where $V = \{1, \ldots, n\}$ represents the set of sequential tasks, and $E \subseteq V \times V$ represents the set of precedence constraints among the tasks. If there is an arc $(i, j) \in E$, then task T_j cannot be processed before the completion of task T_i. Task T_i is called a predecessor of T_j, while T_j is called a successor of T_i. We denote by $\Gamma^-(j)$ (resp. $\Gamma^+(j)$) the sets of the predecessors (resp. successors) of T_j. If a task has no predecessor, it is assigned a fictive predecessor T_0 which completes its execution at time 0. As in previous chapters, each sequential task has two processing times depending on which type of processor it is assigned to, $\overline{p_j}$ if T_j is processed on a CPU and $\underline{p_j}$ if it is processed on a GPU. We still assume that both processing times of a task are known in advance, or at least can be estimated at compile time. We will compute for each task T_j an associated completion time C_j and a starting time t_j. Again, we denote by $\mathcal{C}$ (resp. $\mathcal{G}$) the sets of tasks assigned to the CPUs (resp. GPUs).
With the notations introduced in Chapter 3, we denote by $(Pm, Pk)|prec \mid C_{max}$ the considered problem with m CPUs and k GPUs with dependent tasks. This scheduling problem is clearly more difficult to solve than its counterpart without precedence

constraints, $(Pm, Pk) \mid\mid C_{max}$, which is already NP-hard, therefore problem $(Pm, Pk)|prec \mid C_{max}$ is also NP-hard and we look for efficient approximation algorithms for this problem.
Since we only studied independent tasks in previous chapters of this work, we start with some related work on the subject on tasks linked by precedence constraints before diving into the heart of the matter.

8.2 Related Work

The problem considered here is more complex than the problem of scheduling tasks with precedence constraints on uniform machines, $Q \mid prec \mid C_{max}$ according to the classical scheduling notation [39] but easier than the same problem with unrelated machines, $R \mid prec \mid C_{max}$. There are very few results concerning the problem of scheduling tasks linked by precedence constraints on unrelated machines.
Chudak and Shmoys [20] developed a polynomial-time approximation algorithm for $Q \mid prec \mid C_{max}$ with worst-case performance guarantee $\mathcal{O}(\log m)$, where m is the total number of machines. Chekuri and Bender [17] gave another polynomial-time approximation algorithm with the same order of worst-case performance. They also proved that for the special case where the precedence graph of the problem is only constituted by chains of tasks, $Q \mid chain \mid C_{max}$, their algorithm is a 6-approximation algorithm. Woeginger [90] presented a 2-approximation algorithm for the same problem, based on a transformation of an instance of $Q \mid chain \mid C_{max}$ into other instances considered from the same problem without precedence constraints, $Q \mid\mid C_{max}$ and the one without precedence constraints but with preemptions, $Q \mid pmtn \mid C_{max}$ and the comparison of the respective optimal makespans.
When the tasks are considered malleable and the processors are all identical, i.e. problem $P \mid mal,\ prec \mid C_{max}$, Lepere et al. [58] developed an algorithm with an approximation ratio of $\left(2+\sqrt{5}\right) \sim 5.23606$. Jansen et al. [50] later improved this ratio to $100/62 + 100\left(\sqrt{6469}+13\right)/5481 \sim 3.291919$.

8.3 Approximation Algorithm

8.3.1 Preliminaries

It has been shown [81] that it is NP-hard to approximate the scheduling problem $P \mid prec \mid C_{max}$ within a factor strictly less than $2-\epsilon$, even in the case of unit processing times, making Graham's list scheduling algorithm [38] the best possible approximation approach, with a ratio of $2-\frac{1}{m}$, m being the number of identical processors considered. In the list scheduling paradigm, the set of tasks that are ready to be executed, i.e. the tasks whose predecessors have finished their processing, are kept in a priority list. When a computing resource becomes available, the task with the highest priority, and the earliest starting time, is scheduled on this resource. If no priority is specified, the tie is

broken randomly. If the processors of problem $(Pm, Pk) \mid prec \mid C_{max}$ were all identical, the ratio of the list scheduling algorithm for problem $P(m+k) \mid prec \mid C_{max}$ would be $2 - \frac{1}{m+k}$. The idea of the proof providing this ratio is based on the expression of the makespan of the resulting schedule: the Graham's bound has two consecutive terms, cumulative in the worst case, representing the critical path and the workload on the processors: $C_{max} = \frac{1}{m+k}\left(\sum_{l=1}^{n} p_l + \sum_{\phi\, idle} p_\phi\right)$, where ϕ represents a dummy task representing one idle time in the schedule, and p_ϕ the corresponding fictive processing time, representing the time a given processor remains idle. These two sums are bounded separately, and the sum of their upper bounds provides the expected ratio. In the list scheduling algorithm, a computing resource is never idle if one of the remaining tasks could be started on the resource at that time. This is the key point to achieve guaranteed performance ratio, even in the case of multiple resource constraints [32]. However, the use of the same strategy in a hybrid system, leads to a large value of the worst case performance ratio, even when there are no precedence constraints. Another technique has to be used.
If the acceleration ratios defined as $\frac{\overline{p_j}}{\underline{p_j}} = q_j$ were identical for all the tasks considered in $(Pm, Pk) \mid prec \mid C_{max}$, then the problem would reduce to $Q(m+k) \mid prec \mid C_{max}$. Liu and Liu [60] have shown that, when unforced idleness or preemption is allowed, the ratio of the makespan given by a list scheduling algorithm over the optimal makespan with unforced idleness is lower than $1 + \frac{\max_i\{q_i\}}{\min_i\{q_i\}} - \frac{\max_i\{q_i\}}{\sum_i q_i}$ and then the same inequality is valid for the ratio of the makespan obtained with a list scheduling algorithm and the optimal makespan with preemptions allowed.

8.3.2 Principle of the algorithm

We propose a two-phase approximation algorithm, aiming for a ratio of 6. In the first phase of the approximation algorithm, we solve an assignment problem. The goal of this assignment problem is to find an assignment $\alpha : V \to \{\mathcal{C}, \mathcal{G}\}$ deciding the type of processor (CPU or GPU) assigned to execute the tasks such that the makespan is minimized while the precedence constraints are satisfied. We solve the linear relaxation of this problem. The fractional solution of this linear program is then rounded to a feasible solution to the assignment problem. In the second phase, we apply a variant of list scheduling algorithm to generate a feasible schedule.

8.3.3 Linear Program

In the first phase, we develop a linear program. By rounding its fractional solution with a parameter $M = 2$, we are able to obtain a feasible assignment for the tasks such that each task T_j is assigned to either a CPU or a GPU. We introduce the binary variable x_j representing the assignment of task T_j:

$$x_j = \begin{cases} 1 & \text{if } T_j \text{ is assigned to a CPU} \\ 0 & \text{otherwise} \end{cases}$$

In any schedule, we know that the makespan is an upper bound of the critical path length L and the total works (i.e. the computational areas) on CPUs and GPUs divided by their respective number of processors, $\frac{W_C}{m}$ and $\frac{W_G}{k}$ respectively, i.e. $\max\left\{L, \frac{W_C}{m}, \frac{W_G}{k}\right\} \leqslant C_{max}$. In the first phase of the algorithm, the assignment problem to solve consists in the following problem (P):

$$(P) \quad \begin{aligned} &\min C \\ &\text{s.t. } C_i + \overline{p_j} x_j + \underline{p_j}\,(1 - x_j) \leqslant C_j, && \forall i \in \Gamma^-(j), \forall j && (8.1) \\ &\quad 0 \leqslant C_j \leqslant C && \forall j && (8.2) \\ &\quad \sum_{j=1}^{n} \overline{p_j} x_j \leqslant mC && && (8.3) \\ &\quad \sum_{j=1}^{n} \underline{p_j}\,(1 - x_j) \leqslant kC && && (8.4) \\ &\quad x_j \in \{0, 1\} && \forall j && (8.5) \end{aligned}$$

Constraints (8.1) are the precedence constraints. They impose that the predecessors of each task must be completed before its execution. Constraints (8.2) indicates that the completion time of every task is bounded by the makespan. The goal is to minimize the makespan C. Constraints (8.3) and (8.4) ensure that the computational areas on CPUs and GPUs do not exceed the makespan. Finally, Constraints (8.5) are the integrity constraints. The variables of (P) are x_j and C_j for $j = 1, \ldots, n$.
In order to have an easier problem, we relax Constraints (8.5) and allow any task to be assigned fractionally to a CPU and the rest to a GPU. This means that $x_j \in [0, 1]$. The new problem (P_R) is as follows:

$$(P_R) \quad \begin{aligned} &\min C \\ &\text{s.t. } C_i + \overline{p_j} x_j + \underline{p_j}\,(1 - x_j) \leqslant C_j, && \forall i \in \Gamma^-(j), \forall j && (8.6) \\ &\quad C_j \leqslant C && \forall j && (8.7) \\ &\quad \sum_{j=1}^{n} \overline{p_j} x_j \leqslant mC && && (8.8) \\ &\quad \sum_{j=1}^{n} \underline{p_j}\,(1 - x_j) \leqslant kC && && (8.9) \\ &\quad x_j \in [0, 1] && \forall j && (8.10) \end{aligned}$$

We denote by x_j^R the assignment of task T_j in an optimal solution of the linear program (P_R). The corresponding assignment of all the tasks is denoted by α_R for the optimal solution of (P_R). If x_j^R is an integer, it is a feasible assignment for scheduling task T_j in problem (P), otherwise, it has to be rounded to either 0 or 1. We apply the following rounding strategy to create another assignment denoted by x_j^A for T_j: if $x_j^R \geqslant \frac{1}{M}$, where M is a real number greater than 1, x_j^R will be rounded up to $x_j^A = 1$, otherwise it will be rounded down to $x_j^A = 0$. The optimal value of M for our problem is $M = 2$. The explanation of this choice will be given in Lemma 8.4.2. Here the total assignment of this solution is denoted by α^A. With this new assignment, the completion times C_j^R of all tasks T_j are not accurate anymore. In order to obtain a feasible schedule, the new completion times C_j^A are determined by the scheduling algorithm described in the following section, leading to a new value of the makespan.

8.3.4 Scheduling Algorithm

With the previous assignment α^A determined by the rounding of the solution α^R of (P_R) presented in the previous section, we schedule the tasks according to the following algorithm. We obtain a feasible schedule S^A for problem $(Pm, Pk) \mid prec \mid C_{max}$.

Algorithm 8.3.1.

1. *Compute assignment α^R by solving (P_R).*
2. *Compute a feasible assignment α^A by rounding α^R.*
3. *Build a feasible schedule S^A according to α^A.*
   - $S^A \leftarrow \emptyset$;
   - *While* $S^A \neq T$ *do*
     - $R \leftarrow \left\{T_j \mid \Gamma^-(j) \subseteq S^A\right\}$;
     - *Compute the earliest possible starting time for all tasks in R with respect to the precedence constraints according to α^A;*
     - *Schedule the task $T_j \in R$ with the smallest possible starting time;*
     - $S^A = S^A \cup \{T_j\}$;

The algorithm is composed of three steps. The first one is the resolution of the previously mentioned linear program (P_R), implying a polynomial time complexity $B(n, m, k)$. The second step can be done in linear time, and the third step consists in the scheduling algorithm of the assignment determined in the previous steps. This last step corresponds to a classical list scheduling algorithm. This can be executed in $\mathcal{O}(n^2)$, since the determination of the set R and the computations of the starting times of the tasks in R can be done in linear time for each iteration. Therefore, the algorithm has an overall polynomial time complexity in $\mathcal{O}\left(B(n, m, k) + n^2\right)$.

8.4 Analysis of the Algorithm

We shall determine the approximation ratio of Algorithm 8.3.1. We denote by L^A, W_C^A, W_G^A and C_{max}^A respectively the critical path length, the total works on CPUs and GPUs and the makespan of the final schedule S^A provided by Algorithm 8.3.1. Furthermore, we denote by C_{max}^R the optimal objective value of the linear program (P_R), and L^R, W_C^R, W_G^R respectively the (fractional) critical path length and the (fractional) works on CPUs and GPUs in the optimal solution of the linear program (P_R). We denote by C_{max}^* the optimal makespan (over all feasible schedules with integral number of processors assigned to all tasks), of the optimal solution of (P).

8.4.1 Properties resulting from the rounding phase

Let start by the following straightforward lower bounds:

$$\max\left\{L^R, \frac{W_C^R}{m}, \frac{W_G^R}{k}\right\} \leqslant C_{max}^R \leqslant C_{max}^*.$$

Lemma 8.4.1. *For any task T_j, in the assignment α^A derived from the rounding of the solution α^R of the linear program (P_R), its processing time satisfies the following inequalities:*

$$\overline{p_j}x_j^A \leqslant 2\overline{p_j}x_j^R, \qquad \underline{p_j}\left(1-x_j^A\right) \leqslant 2\underline{p_j}\left(1-x_j^R\right).$$

Proof. The two inequalities correspond to the two possible values for x_j^A.

- Suppose that $x_j^A = 1$. According to the rounding rule, this means that $x_j^R \geqslant \frac{1}{2}$, so that $\overline{p_j}x_j^R \geqslant \frac{1}{2}\overline{p_j}$, leading to $\overline{p_j} \leqslant 2\overline{p_j}x_j^R$, which is the first inequality of the lemma, since $x_j^A = 1$.
- If now $x_j^A = 0$, then we have, according to the rounding rule, $-x_j^R > -\frac{1}{2}$, so $\left(1-x_j^R\right)\underline{p_j} > \left(1-\frac{1}{2}\right)\underline{p_j}$, which becomes $2\left(1-x_j^R\right)\underline{p_j} > \underline{p_j}\left(1-x_j^A\right)$, since $x_j^A = 0$, which is the second inequality of the lemma.

□

Lemma 8.4.2. *The best value for the rounding strategy parameter M is 2 for a makespan minimization.*

Proof. If we look closely at the proof of the previous lemma, we can see why the rounding strategy parameter M was chosen equal to 2. Indeed, with an arbitrary value for M, the first inequality of the lemma becomes $\overline{p_j}x_j^A \leqslant M\overline{p_j}x_j^R$, and the second one is $\underline{p_j}\left(1-x_j^A\right) \leqslant \left(1+\frac{1}{M-1}\right)\underline{p_j}\left(1-x_j^R\right)$ since in that case we have $1-x_j^R > 1-\frac{1}{M}$ i.e $\underline{p_j} < \frac{1}{1-\frac{1}{m}}\left(1-x_j^R\right)$. The functions $f(x) = x$ and $g(x) = 1+\frac{1}{x}$ have opposite variations, so the best value for x in our case is when $f(x) = g(x)$ i.e. $1+\frac{1}{x-1} = x$. This equation

has one non-zero solution which is $x = 2$. This explains why the rounding parameter is chosen equal to 2. That way, the loads on the two types of processors are somewhat balanced. □

We have an immediate corollary to Lemma 8.4.1:

Lemma 8.4.3.

$$W_C^A \leqslant 2mC_{max}^*, \qquad W_G^A \leqslant 2kC_{max}^*.$$

Proof. We write down the definition of the work on CPUs:

$$\begin{aligned} W_C^A &= \sum_{x_j^A=1} \overline{p_j} = \sum_{x_j^R \geqslant \frac{1}{2}} \overline{p_j} \\ &\leqslant \sum_{x_j^R \geqslant \frac{1}{2}} 2\overline{p_j} x_j^R \\ &\leqslant 2W_C^R \leqslant 2mC_{max}^* \end{aligned}$$

Similarly, we can write the definition of the work on GPUs:

$$W_G^A = \sum_{x_j^A=0} \underline{p_j} \leqslant \sum_{x_j^R < \frac{1}{2}} 2\underline{p_j} x_j^R \leqslant 2W_G^R \leqslant 2kC_{max}^*.$$

□

8.4.2 A closer look at the schedule

In this section, we focus more on the structure of the schedule S^A built by the previous algorithm.

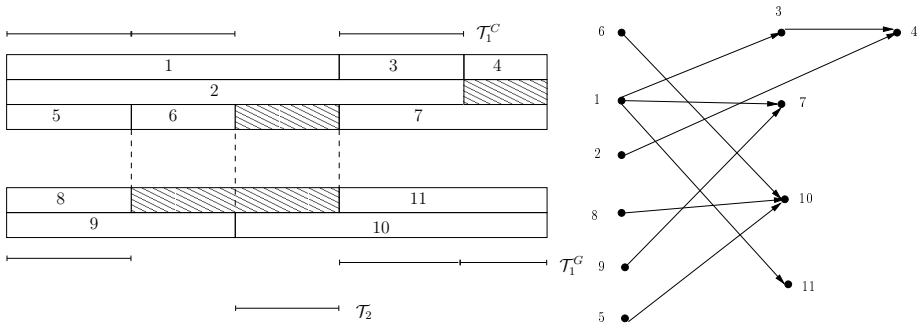
Time Interval Types. The time interval $[0, C_{max}^A]$ of schedule S^A can be divided into two subsets $\mathcal{T}_2$ and $\mathcal{T}_1$ (see Figure 8.1) defined as follows:

- $\mathcal{T}_2 = \left\{ t \in [0, C_{max}^A] \mid \text{at least one CPU and one GPU are idle at time } t \right\}$.
- $\mathcal{T}_1 = [0, C_{max}^A] \backslash \mathcal{T}_2$.

If we look more closely at subset $\mathcal{T}_1$, we note that at every time $t \in \mathcal{T}_1$, either all the CPUs are busy at time t, or all the GPUs are busy at time t. We note $\mathcal{T}_1^C$ (resp. $\mathcal{T}_1^G$) the subset of $\mathcal{T}_1$ where all the CPUs (resp. GPUs) are busy all the time (see Figure 8.1). The number of unitary time slots of type $\mathcal{T}_i$ is denoted by $|\mathcal{T}_i|$ for $i \in \{1, 2\}$.

Lemma 8.4.4.

$$|\mathcal{T}_1| \leqslant 4C_{max}^*.$$



(a) A schedule

(b) The corresponding precedence graph

Figure 8.1: An illustration of the different types of time intervals.

Proof. Since all the CPUs (resp. GPUs) are busy at any time t of $\mathcal{T}_1^C$ (resp. $\mathcal{T}_1^G$), we have the following inequalities:

$$\left|\mathcal{T}_1^C\right| \leqslant \frac{W_C^A}{m}, \qquad \left|\mathcal{T}_1^G\right| \leqslant \frac{W_G^A}{k}.$$

By combining these two inequalities, we get $|\mathcal{T}_1| \leqslant \frac{W_C^A}{m} + \frac{W_G^A}{k}$, and we know from Lemma 8.4.3 that $\frac{W_C^A}{m} \leqslant 2C_{max}^*$ and $\frac{W_G^A}{k} \leqslant 2C_{max}^*$, so we obtain:

$$|\mathcal{T}_1| \leqslant 4C_{max}^*.$$

□

Construction of a Directed Path. In order to estimate the length of the critical path of the final schedule S^A, we can construct a directed path $\mathcal{P}$ of tasks executed during the time slots in $\mathcal{T}_2$, where at least one CPU and one GPU are idle. The last task in the path $\mathcal{P}$ is any task T_{j_1} that completes at time C_{max}^A, the makespan of S^A.
As we have defined the last $i \geqslant 1$ tasks $T_{j_i} \to T_{j_{i-1}} \to \cdots \to T_{j_2} \to T_{j_1}$ on the path $\mathcal{P}$, we can determine the next task $T_{j_{i+1}}$ as follows: consider the latest time slot t in $\mathcal{T}_2$ that is before the starting time of task T_{j_i} in the final schedule. Let V' be the set of task T_{j_i} and its predecessor tasks that start after time t in the schedule. Since during time slot t at most $m-1$ CPUs and $k-1$ GPUs are busy, no task in V' is ready for execution during the time slot t. Therefore for every task in V' a predecessor is being executed during the time slot t. Then we select any predecessor of task T_{j_i} that is running during time slot t as the next task $T_{j_{i+1}}$ on the path $\mathcal{P}$. This search procedure stops when $\mathcal{P}$ contains a task that starts before any time slot in $\mathcal{T}_2$.

Lemma 8.4.5.

$$|\mathcal{T}_2| \leqslant 2C_{max}^*.$$

Proof. We examine the stretch of processing time for all tasks in $\mathcal{P}$ in the rounding procedure of the first phase. For any task T_j in $\mathcal{P}$ processed during any time slot in $\mathcal{T}_2$, the processing time of the fractional solution to the linear program (P_R) increases by at most a factor 2. The processing time does not change in the second phase as T_j is assigned to the type of processors determined in the first phase. Therefore, for such kind of tasks we have $\overline{p_j}x_j^A + \underline{p_j}\left(1 - x_j^A\right) \leqslant 2\left(\overline{p_j}x_j^R + \underline{p_j}\left(1 - x_j^R\right)\right)$ by combining the two inequalities from Lemma 8.4.1.
By construction, the directed path $\mathcal{P}$ covers all time slots in $\mathcal{T}_2$ in the final schedule. In addition , because of Lemma 8.4.1, the tasks processed in $\mathcal{T}_2$ in the final schedule contribute a total length of at least $\frac{1}{2}|\mathcal{T}_2|$ to $L^R(\mathcal{P})$, the length of the critical path $\mathcal{P}$ in the fractional solution of the linear program (P_R). Since the critical path $L^R(\mathcal{P})$ is not more than the makespan C_{max}^R, and that $C_{max}^R \leqslant C_{max}^*$ since the optimal solution of (P) is a solution of (P_R), we have proved the claimed inequality. □

By combining the two inequalities on the subsets forming $[0, C_{max}^A]$, we obtain the following bound on the makespan of the final schedule S^A:

Theorem 8.4.6. *The makespan of the schedule S^A delivered by our algorithm is bounded as follows:*

$$C_{max}^A \leqslant 6C_{max}^*.$$

Now that we have an approximation algorithm for the problem of scheduling dependent sequential tasks on CPUs and GPUs, we can recall what we observed in Chapter 2 concerning the importance of data transfers on GPUs, since their local memory was limited. While the tasks were considered independent in this work, the processing times were arbitrary and therefore we could assume that communication times were taken into account in these processing times. However, when the tasks are linked by precedence constraints, such an assumption cannot be made anymore, and a more accurate model taking into account these communications should be developed in order to be closer to the reality of hybrid platform computing. Such a model was not developed in this work, but we give below some perspectives we think are interesting to study further in the future.

8.5 A More Accurate Model for Communications

Communications between CPUs and GPUs or even between GPUs themselves are actually not without a cost, and sometimes the time delay created by these communications is not negligible when compared to the very short processing times of a GPU. Several models could be considered for integrating these communications into the studied problems.

One of these models consider the communication as a standard time delay that add up to the processing time. According to previous notations, the processing time of a task T_j on a GPU becomes $\underline{p_j} = \frac{\overline{p_j}}{q_j} + \beta_j$, β_j being the communication cost for the transfer of data. However, that model seems to be a little over simplistic, especially when considering that two tasks linked by precedence constraints and executed successively on the same GPU do not have that need for a communication time. A less systematic modeling of communications should be developed.
Another point to consider is that the GPU can at the same time process one task and communicate with another processor at the same time. With this other model there are again two possible configurations that are actually encountered on platforms: the first one is that there can be one communication channel for each GPU, and a complete communication/processing overlap is possible. However, sometimes there are hardware restrictions and some GPU have to share a communication channel: the case of partial communication/processing overlap has to be considered too. The simplest hypothesis that can happen in reality would be to consider the situation with complete communication/processing overlap.
It should also be noted that communications may not take the same amount of time when transferring data from a CPU to a GPU and when transferring data from this same GPU to the same CPU. Both ways should be considered separately.

Chapter 9

Conclusion

Synthesis

In this work, we presented and analyzed new algorithms for scheduling problems that occur in modern hybrid platform architectures. Most of the new computing platforms today are built with a hybrid structure constituted of multi-core CPUs coupled with several GPU accelerators. Several new applications as for example DNA assembling problem highly benefit from these hybrid architectures. These platforms create a need for generic scheduling algorithms on such heterogeneous systems. Some problems of scheduling on CPUs and GPUs can be linked to existing problems in the scheduling literature (Table 9.1 resumes these considered problems and the corresponding algorithm ratios and time complexities). However, for some problems, such an analogy is impossible.

Problem	Corresponding problem	Algorithm cost	Section
$(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$	$Q \mid p_j = 1 \mid C_{max}$	$\mathcal{O}\left((m+k)^2\right)$	3.2.1.2
$(Qm, Qk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$			
$(Pm, Pk) \mid q_j = q \mid C_{max}$	$Q \mid\mid C_{max}$	as $Q \mid\mid C_{max}$	3.2.2.1
$(Qm, Qk) \mid q_j = q \mid C_{max}$			
$(Pm, Pk) \mid\mid \sum C_j$	$R \mid\mid \sum C_j$	$\mathcal{O}\left(n^3\right)$	3.2.1.2
$(Pm, Pk) \mid ppmtn \mid \sum C_j$			

Table 9.1: Problems related to the classical ones and the corresponding algorithm costs.

We presented in this thesis original algorithms for these new scheduling problems on hybrid architectures using a generic methodology (in the opposite of specific *ad hoc* algorithms). We proposed several algorithms with constant approximation ratios in the case of independent tasks with a reasonable time complexity, the first algorithms combining performance guarantee and practical time complexity in this field of scheduling. 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. We

provided several algorithms with different performance ratios for the case of problem $(Pm, Pk) \mid\mid C_{max}$, that are summarized in the first lines of Table 9.2, so these families can be used by the programmers to choose which algorithm represents for them the best trade-off between a good performance guarantee and a suitable time complexity. If the time complexity is crucial (practical applications), the algorithm with a ratio of $2 = \frac{2(q+1)}{2q+1}$ when $q = 0$ is probably the best with a low lime complexity of $\mathcal{O}(n \log n)$, but the users can refine the performance by tuning parameter q, depending on the time complexity they are willing to allow for the scheduler. We also dealt with the special cases where all the tasks were accelerated when assigned to GPU, preemption was allowed on the CPUs, or when the tasks were considered malleable when affected to the CPUs.

The problem with dependent tasks was also studied in this work. We proposed a fast algorithm with a constant approximation ratio of 6 in the case of dependent tasks on a multi-core machines with GPUs, with precedence constraints being an arbitrary acyclic graph. The main idea of the approach is to determine a fractional assignment of the tasks to the CPUs and the GPUs via linear programming, round this fractional assignment to an integer assignment which is used with a list scheduling algorithm. Table 9.2 recapitulates the problems we studied and the different approximation ratios of the algorithms we developed along with their time complexities.

We also provided a simulation (based on realistic benchmarks) and experimental analysis on a real run-time system (xKaapi) in order to assess the computational efficiency of some of the proposed methods. The main conclusion is that these algorithms are stable because of their approximation guaranties, however, the high running time is often dominated by the cost of the scheduling itself, leading to inefficiency if the size of tasks is too small. According to our experimental setting, the algorithm with an approximation ratio equal to 2 was the best trade-off for arbitrary tasks. However, with long computations, we could argue that the scheduling time of an algorithm with a better performance ratio would be negligible compared to the gain in time on the schedule, because here the misplacement of one long task could have catastrophic consequences on the overall makespan of the schedule.

Problem	Algorithm optimality ratio	Algorithm cost	Section
$(P1, P1) \mid\mid C_{max}$	$\frac{3}{2}$	$\mathcal{O}(n \log n)$	4.1.3
	$1+\epsilon$	FPTAS	
$(Pm, Pk) \mid\mid C_{max}$	2	$\mathcal{O}(n \log n)$	4.2.3
	$\frac{4}{3}+\frac{1}{3k}$	$\mathcal{O}\left(n^2 m^2 k^3\right)$	4.3, 4.4
	$\frac{2r+1}{2r}+\frac{1}{2rk},\ r>0$	$\mathcal{O}\left(n^2 m^r k^{r+1}\right)$	5
	$\frac{2(r+1)}{2r+1}+\frac{1}{(2r+1)k},\ r \geqslant 0$	$\mathcal{O}\left(n^2 m^{r+1} k^{r+2}\right)$	
$(Pm, Pk) \mid q_j \geqslant 1 \mid C_{max}$	$\frac{3}{2}$	$\mathcal{O}(n \log n)$	6.1
$(Pm, P1) \mid ppmtn \mid C_{max}$	$1+\frac{1}{m}$	$\mathcal{O}(n \log n)$	6.2.1
$(Pm, P1) \mid q_j = q,\ ppmtn \mid C_{max}$	$1+\frac{1}{q}$	$\mathcal{O}(n \log n)$	
$(Pm, Pk) \mid ppmtn \mid C_{max}$	$1+\max\left(\frac{1}{m}, 1-\frac{1}{k}\right)$	$\mathcal{O}(n \log n)$	6.2.2
	$1+\max\left(\frac{1}{m}, \frac{1}{2r}+\frac{1}{2rk}\right),\ r>0$	$\mathcal{O}\left(n^2 k^{r+1}\right)$	
	$1+\max\left(\frac{1}{m}, \frac{1}{2r+1}+\frac{1}{(2r+1)k}\right),\ r \geqslant 0$	$\mathcal{O}\left(n^2 k^{r+2}\right)$	
$(Pm, Pk) \mid mall \mid C_{max}$	$\frac{3}{2}$	$\mathcal{O}(n \log n)$	6.3
$(Pm, Pk) \mid prec \mid C_{max}$	6	$\mathcal{O}(n \log n)$	8

Table 9.2: Problems with no equivalent counterpart in the literature studied in this work.

Perspectives

As we mentioned earlier, when a task is to be scheduled on a real life computing platform, both its execution time on CPU and on GPU are only estimated by either the user of the platform, or a program of the platform scheduler. However, depending on the method of estimation, sometimes measurement uncertainty could be enough to greatly affect the scheduling of the tasks. It would be interesting to test the robustness of the presented algorithms to some perturbations in the estimations of the execution times of the tasks.

In the dependent tasks problem we studied, we remained on a generic approach and considered an arbitrary directed acyclic graph to represent the precedence constraints linking the tasks of the instances. The resulting algorithm has a performance ratio of 6, which is quite high. An interesting point to investigate further would be to study the tightness of this performance ratio, or try to estimate a lower bound of the tight performance ratio. An experimental analysis could provide good insight into the impact of each phase of the algorithm on the resulting schedule, helping with the tightness analysis. Because of the rounding phase and the list scheduling algorithm on top of it, the ratio of 6 is probably not the tight bound, which may be around 3 or 4.
Another perspective for this work would be to refine our analysis of the dependent tasks problem to more specific precedence constraints, such as chains of tasks, trees (in an out) and see the improvements that could be made to the algorithm with these more specific graphs.
On the subject of dependent tasks, the introduction of the malleable tasks model would be an interesting perspective to study, since this model seems to take into account some communications between tasks in the malleable property of the tasks. This would be a first step to consider the problem of communications between tasks on different types of processors.

Communications constraints between the CPUs and the GPUs could also be added explicitly to the problem, as stated in the previous chapter. As it was mentioned in Chapter 2, even the geometry of large computing platforms has to be carefully planned in order to minimize the cable lengths to reduce communications delays between processors located far from each other. Since many technical problems can influence the communications between CPUs and GPUs, maybe the malleable tasks model may not provide enough flexibility to take into account the complex problem of communications on hybrid platforms. A completely new model may be needed in order to fully represent the material constraints on communications.

Another point that would be of interest is the consideration of other objectives for the scheduling problem. In High Performance Computing, the main objective is usually to execute the tasks as quickly as possible, as therefore the makespan was the obvious choice for a first study of the problem of scheduling on hybrid architectures. However, some platforms may allow users to assign priorities to the calculations they submit, or these priorities may be assigned to the users according to some quota, for instance.

Some calculations may also have due dates assigned to them, and the objective could become one of minimum lateness instead of minimum makespan.

The addition of the energy constraints required by the platform to the problem should also be investigated. Large computing platforms require a lot of power, for their calculations and for their cooling systems. It would be interesting to study the impact that the different architecture of the GPUs has on the power consumption of a platform, and if the scheduling of the calculations could be adapted accordingly.

This work was started three years ago, and, in the meantime, the computing platforms have evolved. We can wonder if the algorithms designed in this work are still valid for the new generation of platforms being built right now. Given that the first and second platforms in the Top500 list [83] are Tianhe-2 and Titan, respectively a platform with hybrid processors and a platform with CPUs and GPUs, it is safe to say that GPUs are not off the market just yet, and so the algorithms of this work are not as well.
In addition of CPUs, Tianhe-2 has Xeon Phi accelerator chips instead of GPUs. However, since we used a very generic model for the GPUs, with most of the time the hypothesis that the processing times of the tasks when assigned on CPUs or on GPUs are not related at all and are considered completely arbitrary, we could apply most of the algorithms presented in this work to computing platforms using two types of unrelated processors.
An extension of this work would be to see how far this adaptation could go. Another processor with a new architecture to consider could be the MIC processor. Some computing platforms may choose this type of processor, and we could see if the algorithms of this work could be adapted to this new type of processor, or if new algorithms are needed in this case.
It is important to keep in mind that the computing platforms of today tend to be more and more heterogeneous, and therefore our work is the first generic method for the ever-more complex field of scheduling on heterogeneous platforms.
The work started on the problem of scheduling on uniform CPUs and uniform GPUs could be further extended in order to take into account no just different models of GPUs, but also the MIC processors and all the other processors used in these new platforms.

Publications

Journals

- Concurrency and Computations: Practice and Experiments, Bleuse R., Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "Scheduling Independent Tasks on Multi-Cores with GPU Accelerators".
- Pending: Algorithmica, Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "A family of scheduling algorithms for Hybrid parallel platforms".
- Pending: Discrete Applied Math, Blazewicz J., Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "A Study of Scheduling Problems with Preemptions on Multi-Core Computers with GPU Accelerators".

Conferences

- Workshop New Challenges in Scheduling Theory 2012, Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "Scheduling Independent Tasks on Heterogeneous Platforms with GPUs", Fréjus, France.
- ECCO 2013, Blazewicz J., Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "Preemptive Scheduling with GPU", Paris, France.
- MAPSP 2013, Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "Scheduling on Multi-Cores with GPU", Pont-à-Mousson, France.
- HeteroPar 2013, Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "Scheduling Independent Tasks on Platforms with GPUs", Aachen, Germany. **Best paper award**.
- ICCP 2014, Kedad-Sidhoum S., Mendonca F., Monna F., Mounie G., Trystram D., "Fast biological Sequence Comparison on Hybrid Platforms", Mineapolis, USA.
- Pending: IPDPS 2015, Bleuse R., Hunold S., Kedad-Sidhoum S., Monna F., Mounié G., Trystram D., "The Power of Heterogeneity: Scheduling Independent Moldable Tasks on Multi-Cores with GPUs", Wroclaw, Poland.

Bibliography

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

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

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011.

[4] D. Lyla B. *The X86 Microprocessors: Architecture And Programming, 8086 to Pentium*. Pearson, 2010.

[5] B. S. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9:846–855, 1980.

[6] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in GPGPUs. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 287–296, July 2012.

[7] J. Blazewicz, M. Bryja, M. Figlerowicz, P. Gawron, M. Kasprzak, E. Kirton, D. Platt, J. Przybytek, A. Swiercz, and L. Szajkowski. Whole genome assembly from 454 sequencing output via modified DNA graph concept. *Computational Biology and Chemistry*, 33:224–230, 2009.

[8] J. Blazewicz, P. Formanowicz, F. Guinand, and M. Kasprzak. A heuristic managing errors for dna sequencing. *Bioinformatics*, 18:652–660, 2002.

[9] R. Bleuse, T. Gautier, J. F. Lima, G. Mounié, and D. Trystram. Scheduling data flow program in xKaapi: A new affinity-based algorithm for heterogeneous architectures. In *20th International European Conference on Parallel Processing*, ARCoSS/LNCS, Porto, Portugal, Aug 2014. Springer. to appear.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[11] R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4):481–494, 2006.

[12] V. Bonifaci and A. Wiese. Scheduling unrelated machines of few different types. *CoRR*, abs/1205.0974, 2012.

[13] A. Boukerche, J. M. Correa, A. Melo, and R. P. Jacobi. A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. *IEEE Transactions on Computers*, 59:808–821, 2010.

[14] R. P. Brent. The parallel evaluations of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

[15] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independant tasks to reduce mean finishing time. *Comm. ACM*, 17:155–178, 1974.

[16] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *IPDPS*, pages 557–568. IEEE Computer Society, 2012.

[17] C. Chekuri and M. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. In *Integer Programming and Combinatorial Optimization (IPCO)*, 1998.

[18] L. Chen, D. Ye, and G. Zhang. Online scheduling on a CPU-GPU cluster. *TAMC*, 7876:1–9, 2013.

[19] S.-J. Chen, G.-H. Lin, P.-A. Hsiung, and Y.-H. Hu. *Hardware software co-design of a multimedia SOC platform*. Springer, 2009.

[20] F. A. Chudak and D. B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30(2):323–343, February 1999.

[21] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9:808–826, 1980.

[22] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[23] E. F. de O Sandes and A. C. M. A. de Melo. Smith-Waterman alignment of huge sequences with GPU in linear space. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211, 2011.

[24] P.-F. Dutot, G. Mounié, and D. Trystram. Scheduling Parallel Tasks: Approximation Algorithms. In Joseph T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 26, pages 26–1 – 26–24. CRC Press, 2004.

[25] K. H. Ecker and R. Hirschberg. Task scheduling with restricted preemptions. *Proc. PARLE93 - Parallel Architectures and Langueges, Munich*, 1993.

[26] L. Epstein and L. M. Favrholdt. Optimal non-preemptive semi-online scheduling on two related machines. *ACM Journal of Algorithms*, 57(1):49–73, 2005.

[27] A.R. Hoffman et al. *Supercomputers: directions in technology and applications.* National Academies, 1990.

[28] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):15–161, 2007.

[29] K. Fatahalian and M. Houston. A closer look at GPUs. *Communication of the ACM*, 51:50–57, October 2008.

[30] D. Feitelson. Parallel workloads archive, 2010.

[31] D. K. Friesen. Tighter bounds for lpt scheduling on uniform processors. *SIAM Journal on Computing*, 16(3):554–560, 1987.

[32] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[34] T. Gautier, L. Ferreira, V. Joao, N. Maillard, and B. Raffin. xKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[35] T. Gonzalez, O. H. Ibarra, and S. Sahni. Bounds for LPT schedules on uniform processors. *SIAM Journal on Computing*, 6(1):155–166, 1977.

[36] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[37] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[38] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[39] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[40] D. M. Gray. *User's Manual for CPLEX*. IBM, 1999.

[41] GOThA group under the coordination of P. Baptiste, E. Néron and F. Sourd. *Modèles et Algorithmes en Ordonnancement, Exercices et Problèmes Corrigés*. Ellipses, 2004.

[42] D. Hochbaum. *Approximations algorithms for NP-hard problems*. Chapman and Hall, 1995.

[43] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[44] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.

[45] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23(2):317–327, 1976.

[46] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463–468, 1975.

[47] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24:280–289, 1977.

[48] C. Imreh. Scheduling problems on two sets of identical machines. *Computing*, 70:277–294, 2003.

[49] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 99)*, pages 490–498, Baltimore, MD, 1999.

[50] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. *Journal of Computer and System Sciences*, 78:245–259, 2012.

[51] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun. A reconfigurable accelerator for Smith-Waterman algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(12):1077–1081, 2007.

[52] S. Kedad-Sidhoum, F. Mendonca, F. Monna, G. Mounié, and D. Trystram. Fast biological sequence comparison on hybrid platforms. In *ICPP Proceedings*, 2014.

[53] M. Kierzynka, J. Blazewicz, W. Frohmberg, and P. Wojciechowski. G-MSA - GPU-based, fast and accurate algorithm for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 73:32–41, 2013.

[54] P. R. Lakhe. A technology in most recent processor is complex reduced instruction set computers (CRISC): A survey. *International Journal of Innovation Research and Studies*, 2(6):711–715, June 2013.

[55] P.-F. Lavallée. La programmation paralléle hybride MPI- OpenMP. *La lettre de l'IDRIS*, Février 2012.

[56] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, pages 451–460. ACM, 2010.

[57] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1988.

[58] R. Lepere, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *Internat. J. of Foundations of Computer Science*, 13(4):613–627, 2002.

[59] J.V.F. Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting concurrent gpu operations for efficient work stealing on multi-GPUs. In *24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Columbia University, New York, USA, oct 2012.

[60] J. W. S. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. *Information Processing, J. L. Rosenfeld, ed., North-Holland, Amsterdam*, 74:349–353, 1974.

[61] Y. Liu, B. Schmidt, and D. L. Maskell. Cudasw++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC research notes*, 3(1):93, 2010.

[62] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176, D. D. Sleator, ed., Arlington, VA, 1994.

[63] W. T. Ludwig. Algorithms for scheduling malleable and nonmalleable parallel tasks. Master's thesis, Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, 1995.

[64] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1st edition, 1990. Wiley Series in Discrete Mathematics and Optimization.

[65] R. McNaughton. Scheduling with deadlines and loss functions. *Management Sci.*, 6:1–12, 1959.

[66] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA 99)*, pages 23–32, New York, 1999. ACM Press.

[67] G. Mounie, C. Rapine, and D. Trystram. A 3/2 approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Computing*, 37(2):401–412, 2007.

[68] D. W. Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004.

[69] V. Nélis and G. Raravi. A ptas for assigning sporadic tasks on two-type heterogeneous multiprocessors. *RTSS*, 2012.

[70] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC*, 2008.

[71] F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel Distrib. Comput.*, 2012.

[72] E. D. Reilly. *Milestones in computer science and information technology*. Greenwood Publishing Group, 2003.

[73] T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelization. *BMC bioinformatics*, 12(1):221, 2011.

[74] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23:116–127, 1976.

[75] S. Seifu. Scheduling on heterogeneous cluster environments. Master's thesis, Grenoble university, June 2012.

[76] D. Shabtay and G. Steiner. A survey of scheduling with controllable processing times. *Discrete Applied Mathematics*, pages 1643–1666, 2007.

[77] E. V. Shchepin and N. Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*, 33:127–133, 2004.

[78] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[79] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[80] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *26th ACM International Conference on Supercomputing (ICS 2012)*, Venice, Italy, June 2012. ACM.

[81] O. Svensson. Hardness of precedence constrained scheduling on identical machines. *SIAM J. Computing*, 40(5):1258–1274, 2011.

[82] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. Swps3-fast multi-threaded vectorized Smith-Waterman for IBM Cell/BE and x86/SSE2. *BMC Research Notes*, 1(1):107, 2008.

[83] Top500. `http://www.top500.org/lists/2014/06/`.

[84] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS*, 13(3):260–274, 2002.

[85] D. Trystram. Les riches heures de l'ordonnancement. *Technique et science informatique*, 31(8):1021–1047, 2012.

[86] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

[87] C. Vaglio-Gaudard, K. Stoll, S. Ravaux, M. Lemaire, A. C. Colombier, J. P. Hudelot, D. Bernard, H. Amharrak, J. Di Salvo, and A. Gruel. Monte carlo interpretation of the photon heating measurements in the integral AMMON/REF experiment in the EOLE facility. *IEEE Transactions on Nuclear Science*, 61(1), February 2014.

[88] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[89] Wikipedia. `http://fr.wikipedia.org/wiki/Superordinateur`.

[90] G. J. Woeginger. A comment on scheduling on uniform machines under chain-type precedence constraints. *Operations Research Letters*, 26:107–109, 2000.