A study of scheduling problems with preemptions on multi-core computers with GPU accelerators

Jacek Blażewicz, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, Denis Trystram

A Study of Scheduling Problems with Preemptions on Multi-Core Computers with GPU Accelerators

Jacek Blazewicz[a], Safia Kedad-Sidhoum[b], Florence Monna[b,c,*], Grégory Mounié[c], Denis Trystram[c,d]

[a]*Poznan University of Technology, Poland*
[b]*Sorbonne Universités, UPMC Univ. Paris 06, UMR 7606, LIP6, F-75005, Paris, France*
[c]*Grenoble Institute of Technology, 51 avenue Kuntzmann, 38330 Montbonnot Saint Martin, France*
[d]*Institut Universitaire de France*

Abstract

For many years, scheduling problems have been concerned either with parallel processor systems or with dedicated processors - job shop type systems. With a development of new computing architectures this partition is no longer so obvious. Multi-core (processor) computers equipped with GPU co-processors require new scheduling strategies. This paper is devoted to a characterization of this new type of scheduling problems. After a thorough introduction of the new model of a computing system, an extension of the classical notation of scheduling problems is proposed. A special attention is paid to preemptions, since this feature of the new architecture differs the most as compared with the classical model. In the paper, several scheduling algorithms, new ones and those refining classical approaches, are presented. Possible extensions of the model are also discussed.

Keywords: Scheduling, Approximation algorithms, Parallel heterogeneous systems, Preemption

1. Introduction

This paper is concerned with scheduling problems arising in the new area of parallel processing. In several domains, like in real time finance applica-

*Corresponding author

tions for instance, complex computations are necessary, and the parallelism of processors of the same type is not always the best solution.

Another very important example, where different types of parallel processors are used is DNA assembling problem, where hundreds of millions of DNA chains are 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 (cf. [1, 2, 3] for a deeper analysis of the subject).

As a result, most of computing systems available today include parallel multi-core chips sharing a large memory with additional hardware accelerators [4]. There is an increasing complexity in the architecture of internal nodes of such parallel systems, mainly due to the heterogeneity of the computational resources. In order to take advantage of the benefits offered by heterogeneity, effective and automatic management of the resources will become more and more important for running any applications. These new hybrid architectures 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. A huge amount of work has been done for implementing algorithms on this new type of hybrid platforms, mostly for regular numerical kernels. However, most of these studies are dedicated to *ad hoc* analysis for specialized computations with specific applications like multiple alignments of biological sequences [5], or molecular dynamics [6], and few of them propose more generic high-level approaches, especially in the field of scheduling. All these approaches follow the main characteristics of these hybrid platforms in particular for managing communications from a standard computing unit (CPU) to a GPU as well as the memory organization. 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. Let us note here that the profitability of preemptions is a common feature of the classical systems [7], as well as multiprocessor task scheduling ones [8].

Thus, the objective of this work is to investigate 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 is addressed in this work. We consider several problems, all of them dealing with a heterogeneous computing platform constituted of mixed CPUs and GPUs, where n sequential tasks have to be executed. A task T_j has two distinct processing times, $\overline{p_j}$ if it is executed on a standard CPU and $\underline{p_j}$ if it is processed on a standard GPU.

Moreover, some CPUs may process the tasks faster than other CPUs. Due to the similar structure of all the CPUs, the speedup factor of one CPU over the other is the same here for all the tasks. The CPUs are said to be uniform. Then, the processing time of the task T_j on a given CPU is the processing time on the standard CPU, $\overline{p_j}$, multiplied or divided by the coefficient corresponding to the difference of speeds between the given CPU and the standard one. The same operations can be performed for uniform GPUs instead of identical ones. More details on this subject are given in Section 2.

The aim of the paper is to introduce a new model in the scheduling area, motivated by the practical applications of hybrid CPU/GPU systems, study the basic scheduling algorithms and point out possible refinements, moving ahead of the development of theses technologies.

In this work, some of the problems analyzed are matched with classical scheduling problems and their solving methods are adapted to the new hybrid problems, but some CPU/GPU problems cannot be restricted to standard problems, and therefore new algorithms are developed specifically for these new problems.

The structure of the paper is as follows. In the following section, a new notation for a new type of scheduling problems dealing with heterogeneity is introduced. Then, we provide in Section 3, a survey of classical problems whose scheduling algorithms can be adapted to solve some hybrid platform scheduling problems. In Section 4, a new method for scheduling efficiently parallel applications, where each task of the application can be processed either on a core (CPU) or on a GPU and are independent from each other, is proposed. Finally, in Section 5, some possible extensions of the model proposed are analyzed before the conclusion (Section 6).

2. Notation

We extend the traditional notation $\alpha \mid \beta \mid \gamma$ introduced by Graham et al. [9] in scheduling theory. We detail here only the proposed extensions for some scheduling problems with n independent sequential tasks.

2.1. Machines (α)

2.1.1. Sets of Identical CPUs and Identical GPUs

We denote by (Pm, Pk) the problem on a heterogeneous computing platform constituted of m identical CPUs (Pm) and k identical GPUs (Pk), where, as previously defined, 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. Since the CPUs are all identical, and so are the GPUs respectively, there is no need for a more complex notation involving the number of the CPU (or 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. Some tasks 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 even be slowed down when assigned to a GPU.

Let us consider for instance the problem $(Pm, Pk) \mid\mid C_{max}$, where the objective is to minimize the makespan (C_{max}). This objective is the most frequently investigated in HPC field. Other classical objectives found in the literature can also be integrated in this notation, as for example the mean flow time ($\sum C_j$), which is also investigated in this paper.

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.

2.1.2. Sets of Uniform CPUs and Uniform GPUs

With the same reasoning, 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 T_j on the slowest CPU, taken as the reference CPU. From there, the processing time of T_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 Q.

We introduce the same processing times for the GPUs, where $\underline{p_j}$ denotes the processing time of a task T_j on the slowest GPU. The processing time of

T_j on GPU i is then defined by $\underline{p_{ij}} = \frac{p_j}{\underline{s_i}}$, 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 standard 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 factor among the $\overline{s_i}$ is lower than the smallest acceleration factor q_j for a task T_j on the 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 can 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.

Let us remark that extending this 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.

2.2. Tasks (β)

2.2.1. 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 $(Pm, Pk) \mid q_j = q \mid C_{max}$. 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.

2.2.2. 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, stop its processing 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 [10].

We introduce the notion of "partial preemption", denoted by *ppmtn*, where preemption is only allowed for tasks remaining on the CPUs. In the rest of the article, we 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 is much easier. Nevertheless, efficient task migration between the CPU and the MIC remains an open problem.

3. Analysis of some CPU/GPU Problems

We start with some basic problems in order to analyze the complexity and potential resolution of the problems with CPUs and GPUs, and then see if the addition of the partial preemption imply a change in the order of complexity of these problems.

3.1. Minimization of the makespan with independent tasks and fixed speedup

The first problems considered have the objective of makespan minimization and have the additional simplifying constraint that all the tasks have the same speedup on the GPUs. We start with a problem which is close to a classical polynomial problem, $Q \mid p_j = 1 \mid C_{max}$ [9].

3.1.1. $(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$

Complexity. The problem can be solved polynomially. It can be formulated as a transportation problem in the same way as shown by Graham et al. [9] for the $Q \mid p_j = 1 \mid C_{max}$ problem.

Solving method. Below is the adaption of the method of Graham et al. [9] to the problem $(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$.

There are n sources $j = 1, \ldots, n$ each corresponding to a task T_j, and $(m + k)n$ sinks (i, v) for $i = 1, \ldots, m + k$, and $v = 1, \ldots, n$. The first m

machines correspond to the CPUs and the last k ones to the GPUs. The cost of arc $(j,(i,v))$ is

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

The arc flow is

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

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

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

The problem $(Pm,Pk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$ can therefore be solved in $\mathcal{O}\left(n^3\right)$ [9], since $m+k \leq n$.

Another method can be extended to solve $(Pm,Pk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$, based on an approach developed for the problem $Q \mid p_j = 1 \mid C_{max}$ by Sevastjanov [11] leading to a time complexity of $\mathcal{O}\left((m+k)^2\right)$.

The rationale of the solving algorithm is as follows: the minimum schedule length is

$$C^*_{max} = \sup\left\{t \mid m + k\left\lfloor\frac{t}{q}\right\rfloor < n\right\}.$$

A lower bound on the schedule length for the problem $(Pm,Pk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$ is $C' = \frac{n}{m+\frac{k}{q}} \leqslant C^*_{max}$.

If we assign $v_i = \begin{cases} \lfloor C' \rfloor, & i = 1,\ldots,m \\ \lfloor C'q \rfloor, & i = m+1,\ldots,m+k \end{cases}$ tasks to machine M_i, and then these tasks are processed in the interval $[0,C']$. However, $l = n - \sum\limits_{i=1}^{m+k} v_i$ tasks remain unassigned. We have $l \leqslant m-1$ since $C' - \lfloor C' \rfloor < 1$ and

$C'q - \lfloor C'q \rfloor < 1$. The remaining l tasks are assigned one by one to the machine M_i for which $\min_i \left\{ v_i + 1 \text{ for } i = 1, \dots m;\ \frac{v_i+1}{q} \text{ for } i = m+1, \dots, m+k \right\}$ is attained at a given stage, where, of course, v_i is increased by one after the assignment of a tasks to a particular machine M_i. This procedure is repeated until all the tasks are assigned, and we obtain the optimal schedule in time $\mathcal{O}\left((m+k)^2\right)$.

We can note that the time complexity of these methods has not varied from the classical case $Q \mid p_j = 1 \mid C_{max}$. The problem $(Pm, Pk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$ is polynomial and can be solved as efficiently as a classical platform of uniform cores.

3.1.2. $(Pm, Pk) \mid q_j = q \mid C_{max}$

Let us now study the problem when constraint $\underline{p_j} = 1$ is relaxed.

Complexity. $(P1, P1) \mid q_j = q \mid C_{max}$ is equivalent to the classical problem $Q2 \mid\mid C_{max}$, which is NP-hard [12]. Therefore, $(Pm, Pk) \mid q_j = q \mid C_{max}$ is also NP-hard.

Solving Method. We look at approximation methods from the literature that can be used for this problem. If we consider the $(Pm, P1) \mid q_j = q \mid C_{max}$ problem, we notice that it corresponds to a specific instance of $Q \mid\mid C_{max}$, where $m+1$ machines are considered with a very specific set of machine speedups: the first m machines have the same processing speed of 1 and the last one has a speedup of q. This special instance of $Q \mid\mid C_{max}$ will be denoted here by $Q(m+1) \mid\mid C_{max}$. Liu et al. [13] developed a list scheduling algorithm for $Q(m+1) \mid\mid C_{max}$: 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 machine becomes free, it gets the first non-assigned task of the list. If there are two or more free processors, the fastest is chosen. The performance ratio is $\begin{cases} \frac{2(m+q)}{q+2} & \text{for } q \leqslant 2 \\ \frac{m+q}{2} & \text{for } q > 2 \end{cases}$, with q the speedup of the GPU.

To address the more generic problem $(Pm, Pk) \mid q_j = q \mid C_{max}$, we can use the generalization of the LPT rule developed by Gonzalez et al. [14] for the problem $Q \mid\mid C_{max}$. The algorithm is as follows: assign each task, in the order of non-increasing (longest) processing time, to the machine it will be completed soonest. The ratio is then $2 - \frac{2}{m+k+1}$. Additionally, they gave examples for which $\frac{C_{max}(LPT)}{C^*_{max}}$ approaches $\frac{3}{2}$ as $m+k$ tends to infinity.

3.1.3. $(Qm, Qk) \mid q_j = q \mid C_{max}$

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 Sections 3.1.1 and 3.1.2 respectively 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 $(Qm, Qk) \mid q_j = q, \underline{p_j} = 1 \mid C_{max}$ and $(Qm, Qk) \mid q_j = q \mid C_{max}$ are also particular cases of $\overline{Q} \mid p_j = 1 \mid C_{max}$ and $Q \mid\mid C_{max}$, respectively. The proofs would be similar and the time complexities would remain unchanged.

3.2. Minimizing the Mean Flow Time with Independent Tasks

Another objective to consider on hybrid platforms is the Mean Flow Time.

3.2.1. $(Pm, Pk) \mid\mid \sum C_j$

Complexity. The problem $(Pm, Pk) \mid\mid \sum C_j$ is a specific case of the classical problem $R \mid\mid \sum C_j$ which is polynomial. Therefore $(Pm, Pk) \mid\mid \sum C_j$ is also a polynomial problem.

Solving method. We adapt an approach to the solution of $R \mid\mid \sum C_j$ [15] to $(Pm, Pk) \mid\mid \sum C_j$: the method is based on the observation that task T_j processed on machine M_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 mean flow time $\overline{F}$ for problem $(Pm, \overline{Pk}) \mid\mid \sum C_j$. The same task processed in the last but one position on the same processor contributes $2p_{ij}$ to $\overline{F}$ and so on. This reasoning allows us to construct an $(2n) \times n$ matrix $\mathcal{Q}$ presenting contributions of particular tasks processed in different positions on different processors to the value of $\overline{F}$:

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

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

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

under the constraints

$$\begin{aligned}
&\sum_{i=1}^{m+k}\sum_{l=1}^{n}x_{ijl}=1 && \forall j\in\{1,\dots,n\}\\
&\sum_{j=1}^{n}x_{ijl}\leqslant 1 && \forall i\in\{1,\dots,m+k\},\ l\in\{1,\dots,n\}\\
&x_{ijl}\geqslant 0 && \forall i\in\{1,\dots,m+k\},\ j,l\in\{1,\dots,n\}
\end{aligned}$$

where

$$x_{ijl}=\begin{cases}1 & \text{if } T_j \text{ is executed on } M_i \text{ in the } l^{\text{th}} \text{ position, starting counting from the end}\\ 0 & \text{otherwise}\end{cases}$$

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

3.2.2. $(Pm, Pk) \mid ppmtn \mid \sum C_j$

Since $R \mid pmtn \mid \sum C_j$ is NP-hard and $R \mid\mid \sum C_j$ is polynomial, we analyze the complexity of $(Pm, Pk) \mid ppmtn \mid \sum C_j$, to see if this also increases the complexity of the problem or if it remains easy to solve.

Complexity. In the problem $(Pm, Pk) \mid ppmtn \mid \sum C_j$, preemptions are only allowed on the CPUs that are considered identical. If we look at the problem $P \mid pmtn \mid \sum C_j$, it is polynomially solvable since preemptions are not profitable when minimizing the mean flow time. The scheduling of the tasks of $P \mid pmtn \mid \sum C_j$ is done in the same way as those of $P \mid\mid \sum C_j$. With that in mind, 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. Therefore, the problem remains easy to solve when partial preemptions are allowed.

In Section 3, all the studied problems can be linked to classical scheduling problems. However, if we relax some of the considered constraints, problems

on hybrid platforms cannot be seen as classical problems. Hence, new algorithms need to be developed for these problems.

4. New Algorithms to Minimize the Makespan with Independent Tasks and Task-dependent speedups

In this section, we study some problems that have no classical counterpart in the existing literature. The algorithms proposed in this section are all based on the dual approximation technique [16]. A g-dual approximation algorithm for a generic problem takes a real number λ (guess) as an input 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 and then the bounds are updated as follows:

- If the algorithm returns "NO", then λ becomes the new lower bound.
- 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.

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.

4.1. $(Pm, Pk) \mid ppmtn \mid C_{max}$

Complexity. $(Pm, Pk) \mid ppmtn \mid C_{max}$ 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.

Solving Method. 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.

4.1.1. 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 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 $\mathcal{O}(n \log n)$.

4.1.2. 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: $2 - \frac{1}{k}$.

Another dual approximation algorithm with dynamic programming has been developed for the problem $(Pm, Pk) \mid\mid C_{max}$ in [17] and can be extended to the problem $(Pm, Pk) \mid ppmtn \mid C_{max}$, with an approximation ratio of $\frac{4}{3} + \frac{1}{3k}$ with a time complexity in $\mathcal{O}\left(n^2 k^3\right)$.

Assuming that there exists a schedule of length lower than λ, λ being the current guess of the dual approximation, the idea is to partition the set of tasks on the GPUs into two sets, each consisting in two shelves: 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 as depicted in Figure 1.

The partition ensures that the makespan on the GPUs is 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. Preemption being allowed on the CPUs, the makespan of the CPUs is $\frac{W_C}{m} \leqslant \lambda$, yielding directly a solution of length at most $\frac{4\lambda}{3} + \frac{\lambda}{3k}$. The main problem is to assign the tasks to the CPUs or to each shelf on the GPUs in order to obtain a feasible solution. This is done using dynamic programming, solving the following

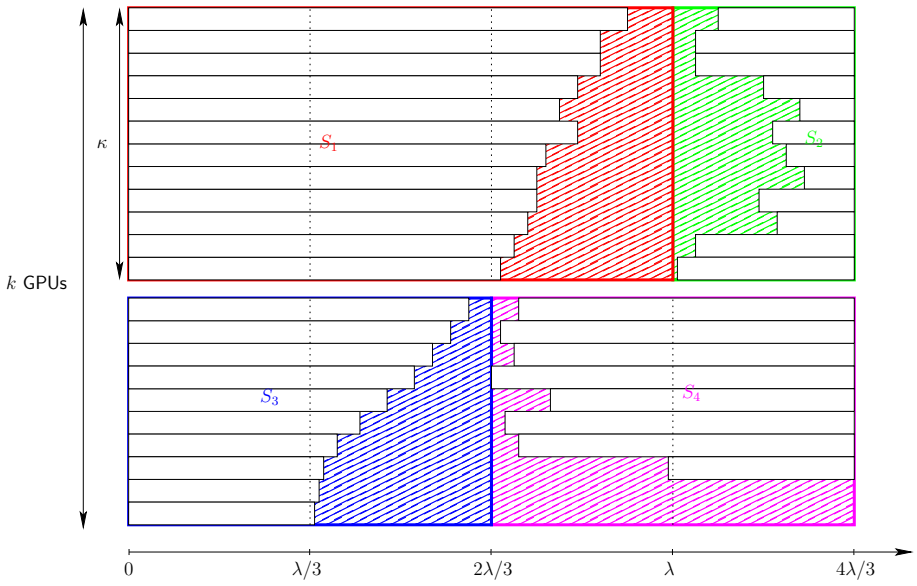

Figure 1: Partitioning the set of tasks on the GPUs into two sets of two shelves, the first one occupying κ GPUs, the second $k - \kappa$ GPUs.

problem:

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

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

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

$$x_j \in \{0, 1\} \tag{4}$$

where x_j is a binary decision variable such that $x_j = 1$ if task T_j is assigned to a CPU or 0 if T_j is assigned to a GPU.

Equation (1) represents the minimal workload on all the CPUs. Constraint (2) imposes that no more than k tasks can be executed on the GPUs with a processing time strictly greater than $\frac{2\lambda}{3}$ (occupying S_1), we note $\kappa = \sum_{\underline{p_j} > 2\lambda/3} (1 - x_j)$ their number; and that there cannot be more than $2(k-\kappa)$ tasks on the GPUs with a processing time lower than $\frac{2\lambda}{3}$ and strictly

greater than $\frac{\lambda}{3}$ (occupying S_3 and S_4). Constraint (3) imposes an upper bound on the computational area on the GPUs which is $k\lambda$.

In order to obtain a time complexity in $\mathcal{O}(n^2k^3)$, the processing times of the tasks on the GPUs are discretized. 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 GPUs, as shown in Figure 2. $N = \sum_{T_j \ / \ x_j=0} \nu_j$ denotes the total integer number of these intervals on the GPUs. 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.

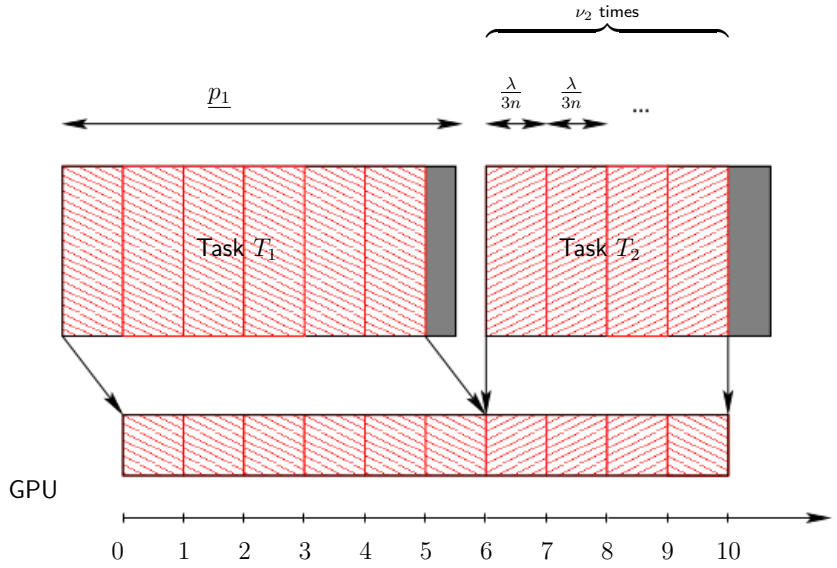

Figure 2: Rounded allocation of two tasks T_1 with $\underline{p_1} = 6.5$ and T_2 with $\underline{p_2} = 4.7$ on a GPU

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

$$N = \sum_{T_j \ / \ x_j=0} \nu_j \leqslant 3kn \tag{5}$$

The approximated computational area of the GPUs is at most $k\lambda$. Thus, the full computational area on GPU remains lower than $k\lambda + \frac{\lambda}{3}$. The tasks

remaining to be assigned on the GPUs after the construction of S_1, S_3, S_3, S_4, fit in the remaining free computational space W_R between these shelves. If all the working processors complete their tasks at $\frac{4\lambda}{3} + \frac{\lambda}{3k}$, with an idle time interval between the end of S_1 and the starting time of S_2, the load of a GPU is equal to $\frac{4\lambda}{3} + \frac{\lambda}{3k}$ minus the length of the idle time interval. If a task with processing time lower than $\frac{\lambda}{3}$ remains to be assigned, the least loaded processor has a load at most $\lambda + \frac{\lambda}{3k}$ since the total work area is bounded by $k\left(\lambda + \frac{\lambda}{3k}\right)$, so the idle time interval on the least loaded GPU has a length of at least $\frac{\lambda}{3}$ and can contain the task to be assigned.

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 $\frac{W_C}{m}$, where W_C is the computational area on the CPUs.

We define $W_C(j, \kappa, \kappa', N)$ 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 on the GPUs, κ of them having processing times greater than $\frac{2\lambda}{3}$ and κ' with $\frac{\lambda}{3} < \underline{p_j} \leqslant \frac{2\lambda}{3}$, and where N time intervals are occupied on the GPUs.

We use a dynamic programming algorithm to compute the value of $W_C(j, \kappa, \kappa', N)$. The optimal value of the computational area W_C on the CPUs will be given by $W_C^* = \min_{0,\ 0\leqslant\kappa\leqslant k,\ 0\leqslant\kappa'\leqslant 2(k-\kappa),\ 0\leqslant N\leqslant 3kn} W_C(n, \kappa, \kappa', N)$. If W_C^* is greater than $m\lambda$, then there exists no solution with a makespan at most λ, and the algorithm answers "NO" in the dual approximation framework. Otherwise, the guess λ is large enough, we construct a feasible solution with a makespan at most $\frac{4\lambda}{3} + \frac{\lambda}{3k}$, with the shelves and the corresponding κ, κ' and N values.

Solving the dynamic programming algorithm for a fixed value of λ requires to consider $\mathcal{O}(n^2k^3)$ states, since $1 \leqslant j \leqslant n$, $1 \leqslant \kappa \leqslant k$, $1 \leqslant \kappa' \leqslant 2(k-\kappa)$, and $0 \leqslant N \leqslant 3kn$. Therefore, the time complexity of each step of the binary search is $\mathcal{O}(n^2k^3)$.

4.2. $(Pm, Pk) \,||\, C_{max}$

Complexity. If we look at problem $(P1, P1) \mid q_j = q \mid C_{max}$, all tasks T_j have a processing time $\underline{p_j}$ on the GPU and a processing time $\overline{p_j} = q\underline{p_j}$ on the CPU. This is equivalent to consider the CPU and the GPU as two uniform machines with different speeds: a speed of 1 for the CPU, and a speed of q for the GPU. So $(P1, P1) \mid q_j = q \mid C_{max}$ is equivalent to the classical problem $Q2 \,||\, C_{max}$, which is NP-hard. Therefore problem $(Pm, Pk) \,||\, C_{max}$ is NP-hard.

Solving Method. The dual approximation algorithm running in $\mathcal{O}(n \log n)$ presented in the previous section can be used for this problem without partial preemption. An approximation ratio of 2 can be achieved, since we face here on the CPUs the same problem which was encountered on the $k \geqslant 2$ GPUs for the previous problem $(Pm, Pk) \mid ppmtn \mid C_{max}$, only solved with the performance ratio of the classical list scheduling algorithm.

Using dual approximation and dynamic programming, as mentioned in the previous section, an algorithm was developed in [17] to reach a ratio of $\frac{4}{3}$ in time $\mathcal{O}(n^2 m^2 k^3)$. Here, the CPUs have to be filled with shelves similar to the ones used on the GPUs in the previous section, since preemption is no longer allowed. There is however no need to count discretize the processing times of the tasks assigned to the CPUs since the computational area on CPUs is the objective to minimize and not a constraint, as it is the case for the GPUs. Therefore we introduce a new factor of m^2 in the time complexity of the algorithm. In [17], it has been shown that the approximation ratio of $\frac{4}{3}$ can be improved with an increase in time complexity. The different possible ratios and their corresponding time complexities are given in Table 2 of Section 6.

5. Further Extensions

The scheduling model discussed so far from Sections 2 to 4 can be further refined, taking into account the forthcoming development of computer architecture. The most important extensions are listed below.

- Preemptions are not allowed between CPUs and GPUs because of too many differences in the programming and the execution of the tasks. However, a recent study [10] has shown that a granular preemption is possible on the latest models of GPUs.

  The idea of a granular preemption $(g - pmtn)$ is the following: the preemption should be allowed on the GPU only after the tasks have been processed continuously for a given amount g of time. The value for g (preemption granularity) should be chosen large enough so that the costs are negligible. Note that if $g = 1$, this is equivalent to the regular preemption. However, when speaking of granular preemption, we have $g \geqslant 2$.

The problem $P \mid g - pmtn \mid C_{max}$ is NP-hard except for the case of two machines that can be solved in $\mathcal{O}(n)$ [18]. Our problem is therefore NP-hard.

Let us notice that the trend of the computer companies is to create GPUs that are closer to usual processors (both at the language level and resource manager), therefore it is likely that the future GPUs will allow preemptions.

- We are working on integrating the dual approximation algorithm described in Section 4.2 into a search procedure for the critical path of the problem $(Pm, Pk) \mid prec \mid C_{max}$. We are aiming at a time complexity of $\mathcal{O}\left(n^4\left(1+n+|E|\right)\right)$, E being the number of edges in our precedence graph, and an approximation ratio of 2.

  If we allow preemptions on the CPUs, we may reduce the ratio to $1+\frac{1}{m}$ in the special case of $k=1$, but the ratio would remain 2 for $k \geqslant 2$.

  For the problem $(Pm, Pk) \mid q_j = q, prec \mid C_{max}$, only polynomial optimization algorithms are known for the preemptive case. For $Q \mid prec \mid C_{max}$, Chudak and Shmoys [19] give an $\mathcal{O}\left(\log m\right)$-approximation algorithm similar to the list scheduling algorithm of Graham for the problem $P \mid prec \mid C_{max}$ [20].

- The communications between the CPUs and the GPUs or even between the GPUs themselves incur a cost, and sometimes the time delay created by these communications is not negligible compared to the very short processing times of a GPU. We are currently considering several models for integrating these communications into our problems.

  One of these models considers the communication as a standard time delay that adds up to the processing time. The processing times become $\underline{p_j} = \frac{\overline{p_j}}{q_j} + \beta_j$, β_j being the communication cost for the transfer of data.
  Another possibility is that the GPU can at the same time process one task and communicate with another processor. With this 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 GPUs have to share a communication channel: the case of partial communication/processing overlap has to be considered too. We

are currently working assuming a complete communication/processing overlap.

6. Conclusion

In the paper, a new computer architecture and related scheduling problems were presented and analyzed. Most of the new computing platforms today are built with a hybrid structure constituted of multi-core 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 CPU/GPU problems can be linked to existing problems in the literature but for other problems, this is impossible. For simple criteria, as for example makespan, some algorithms based on classical approaches could be used. For more complicated cases, new approaches had to be considered. Below, we summarize the results presented in the paper: those related to algorithms equivalent to the classical model are listed in Table 1, those being new are given in Table 2. Possible refinements of the approaches presented, have been discussed in Section 5.

Problem	Hardness	Algorithm cost	Reference
$(Pm, Pk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$	P	$\mathcal{O}\left((m+k)^2\right)$	3.1.1
$(Pm, Pk) \mid q_j = q \mid C_{max}$	NP-hard	considered as $Q \mid\mid C_{max}$	3.1.2
$(Qm, Qk) \mid q_j = q,\ \underline{p_j} = 1 \mid C_{max}$	P	$\mathcal{O}\left((m+k)^2\right)$	3.1.3
$(Qm, Qk) \mid q_j = q \mid C_{max}$	NP-hard	considered as $Q \mid\mid C_{max}$	3.1.3
$(Pm, Pk) \mid\mid \sum C_j$	P	$\mathcal{O}\left(n^3\right)$	3.2.1
$(Pm, Pk) \mid ppmtn \mid \sum C_j$	P	$\mathcal{O}\left(n^3\right)$	3.2.2

Table 1: Problems related to the classical ones and the corresponding algorithm costs.

Problem	Algorithm optimality ratio	Algorithm cost
$(Pm, Pk) \mid\mid C_{max}$	2	$\mathcal{O}(n \log n)$
	$\frac{4}{3} + \frac{1}{3k}$	$\mathcal{O}(n^2 m^2 k^3)$
	$\frac{2r+1}{2r} + \frac{1}{2rk},\ r > 0$	$\mathcal{O}(n^2 m^r k^{r+1})$
	$\frac{2(r+1)}{2r+1} + \frac{1}{(2r+1)k},\ r \geqslant 0$	$\mathcal{O}(n^2 m^{r+1} k^{r+2})$
$(Pm, P1) \mid ppmtn \mid C_{max}$	$1 + \frac{1}{m}$	$\mathcal{O}(n \log n)$
$(Pm, P1) \mid q_j = q,\ ppmtn \mid C_{max}$	$1 + \frac{1}{q}$	$\mathcal{O}(n \log n)$
$(Pm, Pk) \mid ppmtn \mid C_{max}$	$1 + \max\left(\frac{1}{m}, 1 - \frac{1}{k}\right)$	$\mathcal{O}(n \log n)$
	$1 + \max\left(\frac{1}{m}, \frac{1}{2r} + \frac{1}{2rk}\right),\ r > 0$	$\mathcal{O}(n^2 k^{r+1})$
	$1 + \max\left(\frac{1}{m}, \frac{1}{2r+1} + \frac{1}{(2r+1)k}\right),\ r \geqslant 0$	$\mathcal{O}(n^2 k^{r+2})$

Table 2: Problems with no equivalent counterpart in the literature.

References

[1] J. Blazewicz, P. Formanowicz, F. Guinand, M. Kasprzak, Bioinformatics 18 (2002) 652–660.

[2] J. Blazewicz, M. Bryja, M. Figlerowicz, P. Gawron, M. Kasprzak, E. Kirton, D. Platt, J. Przybytek, A. Swiercz, L. Szajkowski, Computational Biology and Chemistry 33 (2009) 224–230.

[3] M. Kierzynka, J. Blazewicz, W. Frohmberg, P. Wojciechowski, Journal of Parallel and Distributed Computing 73 (2013) 32–41.

[4] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, in: A. Seznec, U. C. Weiser, R. Ronen (Eds.), ISCA, ACM, 2010, pp. 451–460.

[5] A. Boukerche, J. M. Correa, A. Melo, R. P. Jacobi, IEEE Transactions on Computers 59 (2010) 808–821.

[6] J. C. Phillips, J. E. Stone, K. Schulten, in: SC.

[7] R. McNaughton, Management Sci. 6 (1959) 1–12.

[8] J. Blazewicz, P. Dellolmo, M. Drozdowski, M. Speranza, Information Processing Letters 49 (1994) 269–270.

[9] R. L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Annals of Discrete Mathematics 5 (1979) 287–326.

[10] C. Basaran, K.-D. Kang, Euromicro Conference on Real-Time Systems (ECRTS) (2012) 287–296.

[11] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Handbook on Scheduling, From Theory to Applications, International Handbooks on Information Systems, Springer, 2007.

[12] J. K. Lenstra, A. H. G. Rinnooy Kan, P. Brucker, Ann. of Discrete Math. 1 (1977) 343–362.

[13] J. W. S. Liu, C. L. Liu, Information Processing, J. L. Rosenfeld, ed., North-Holland, Amsterdam 74 (1974) 349–353.

[14] T. Gonzalez, O. H. Ibarra, S. Sahni, SIAM Journal on Computing 6 (1977) 155–166.

[15] J. Bruno, E. G. Coffman, R. Sethi, Comm. ACM 17 (1974) 155–178.

[16] D. S. Hochbaum, D. B. Shmoys, J. ACM 34 (1987) 144–162.

[17] S. Kedad-Sidhoum, F. Monna, G. Mounié, D. Trystram, Proc. HeteroPar 2013, Aachen (2013).

[18] K. H. Ecker, R. Hirschberg, Proc. PARLE93 - Parallel Architectures and Langueges, Munich (1993).

[19] F. A. Chudak, D. B. Shmoys, Journal of Algorithms 30 (1999) 323–343.

[20] R. L. Graham, Bell System Technical Journal 45 (1966) 1563–1581.