Impact of communication times on mixed CPU/GPU applications scheduling using KAAPI

David Beniamine

To cite this version:

David Beniamine. Impact of communication times on mixed CPU/GPU applications scheduling using KAAPI. Distributed, Parallel, and Cluster Computing [cs.DC]. 2013. hal-00924020

HAL Id: hal-00924020

https://inria.hal.science/hal-00924020

Submitted on 23 Jan 2014

Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option PDES

Impact of communication times on mixed CPU/GPU applications scheduling using KAAPI

David Beniamine

June 20th, 2013

Research project performed at INRIA

Under the supervision of:
Dr Guillaume Huard, INRIA

Defended before a jury composed of:
Prof Noël De Palma
Prof Martin Heusse
Dr Sihem Amer-Yahia
Prof Olivier Gruber
Dr Arnaud Legrand
Dr Sylvain Huet

June 2013

Abstract

High Performance Computing machines use more and more Graphical Processing Units as they are very efficient for homogeneous computation such as matrix operations. However before using these accelerators, one has to transfer data from the processor to them. Such a transfer can be slow.

In this report, our aim is to study the impact of communication times on the makespan of a scheduling. Indeed, with a better anticipation of these communications, we could use the GPUs even more efficiently. More precisely, we will focus on machines with one or more GPUs and on applications with a low ratio of computations over communications.

During this study, we have implemented two offline scheduling algorithms within XKAAPI's runtime. Then we have led an experimental study, combining these algorithms to highlight the impact of communication times.

Finally our study has shown that, by using communication aware scheduling algorithms, we can reduce substantially the makespan of an application. Our experiments have shown a reduction of this makespan up to 64% on a machine with several GPUs executing homogeneous computations.

Contents

Abstract **i**

1 Introduction **1**

1.1 Parallel computers 1

1.2 Parallel programming environments 1

1.3 Communication aware scheduling algorithms 2

1.4 Key issues 3

1.5 Outline 3

2 State of the art **5**

2.1 Writing parallel applications 5

2.2 Scheduling algorithms 6

3 Implementation and analysis of communication aware scheduling algorithms **11**

3.1 Implementation of a static scheduler 11

3.2 Experiments and results 16

4 Conclusions and perspectives **27**

4.1 Main contributions 27

4.2 Future work 27

Bibliography **29**

— 1 —
Introduction

Nowadays, computers are able to run more than one instructions at a time, this capacity can be used to reduce the execution time of computations. To exploit this possibility, we need to write parallel code. However, as it is not straightforward to transform a sequential algorithm into a parallel one, there exists numerous environments to program and execute this kind of applications.

In the remaining of this section, we are going to present briefly the peculiarities of parallel computers and the environments designed to write efficient code on these machines. Then we will focus on one of the main difficulty encountered by these middlewares: the scheduling problem. Finally we will detail more precisely the objectives and the main axes of this study.

1.1 Parallel computers

A desktop computer is parallel machine as well as a supercomputer. Indeed, there are different levels of parallelism inside one computer. For example, modern personal computers have only one chip with two or more cores, whereas some machines such as computing servers have more than one processors which are linked together by an interconnection network. Moreover the chips of these machines can contain many cores also linked by a network. Although these networks are fast, if two processors (on the same chip or not) have to exchange data, the communication will not be instantaneous as it will have to go through one or more interconnections. Since scientific applications works on huge set of data, the processors might have to communicate a intensively. Hence these communication times are not negligible.

Additionally, most computers embed a Graphical Processing Unit, which is designed for graphical rendering and have a totally different architecture than classical processors. Nevertheless this architecture is very efficient for highly parallel and regular computations (such as matrix and vector operations). Thus they are often used in High Performance Computing applications. However as the network between CPU and GPU are slower than the one used between two CPUs, it is not necessary worthwhile to move all the computation tasks onto the GPU.

1.2 Parallel programming environments

Various programming environment have been designed to help developers writing efficient parallel code. Although they target different architectures, some of them have in common the capacity to describe a program as a set of tasks and communications between them. The tasks correspond to the actual work (the functions) and use a set of data. If one task read a shared data written earlier by another one, there is a flow dependency between them. This dependency implies a communication if the two tasks are executed on different resources. Some middlewares let the user express these dependencies and generate the induced communications automatically whereas others let the programmer manage them. In the same way, the runtime can decide on the placement of the tasks on the resources or not.

The easiest way to determine this placement is to statically map each task onto a processor, however, as soon as the tasks have different execution times, some resources might be unused while the others have still some work to do. To avoid this, some middlewares use a dynamic scheduling algorithm such as workstealing which enable an idle resource to steal a task (or a group of task) from a busy one. Even though the workstealing offers a good load balancing, this algorithm does not take into account the communications induced by a steal. Consequently, it can take bad decisions on heterogeneous machines as communications can be costly.

1.3 Communication aware scheduling algorithms

In XKAAPI[11] and StarPU[3], an application is represented as a Directed Acyclic Graph where the nodes of the graph represent the tasks and an edge $x \rightarrow y$ means that the task y needs a piece of data produced by the task x. Additionally, the edges can be labelled with the volume of data exchanged by the two vertices.

The scheduling problem on a DAG consists in finding a mapping of the tasks to the resources and an execution date such that if there is a dependency $x \rightarrow y$, the tasks y cannot be executed before x has finished. The objective of a schedule is often to reduce the makespan ω (i.e. the time needed to execute all the tasks). The optimal makespan is denoted ω^*. As this problem is NP complete[22] , the ratio $\frac{\omega}{\omega^*}$ is an interesting indicator of the quality of a schedule produced by an heuristic.

In this study, we focus on two types of scheduling heuristics: list scheduling, and clustering. The algorithms of the first type keep the tasks ready to be executed in a list sorted according to some priority, and as soon as one resource is free, they execute the first task of the list on it. They ensure that if we disregard the communication times, the makespan is in the order of twice the optimal. This warranty is already good and the schedule is often better in practice. However as soon as we take the communications into account, the bound becomes a factor depending on their cost [14]. Furthermore, on heterogeneous machines and without communications, this limits depends on the number of different resources[9]. Thus these algorithms can be very inefficient on heterogeneous platforms when the communication times grow as shown in[7].

Although these heuristics are designed offline, and with a full knowledge of the execution times and communication times of all the tasks on each resources, they can also be implemented online (at the run time) without any information about the tasks which will be available in the future.

The second kind of heuristics, clustering, considers the whole DAG and group tasks into clusters. All the tasks of one cluster are executed on one resource. Therefore, there are no communications between them. Despite the fact that these algorithms do not provide any theoretical bound relative to the optimal makespan, they can be more efficient than list algorithms when the DAG is large and contains an important volume of communications. As these heuristics work on the structure of the DAG, they need to be implemented offline.

To improve the scheduling obtained by the workstealing algorithm, some runtimes such as XKAAPI[11] and StarPU[3] combine workstealing with more sophisticated algorithm. Among them, HEFT which is an adaptation of the list heuristic to a model with communications and heterogeneous resources. Nevertheless, this algorithm assigns tasks without considering the volume of incoming communications of their children, hence it can make costly mistakes when there is a large volume communications at the end of the execution.

1.4 Key issues

Although the GPUs are able to reduce substantially the execution time of some tasks, we need to transfer data from the main memory to them before the execution. Previous work[7] have shown that it is possible to reduce the impact these communications costs by overlapping them with computation and by using sophisticated scheduling algorithms such as HEFT. However, this work have also highlighted the fact that some contention can occur on machine with many GPUs linked to the processors by an interconnection network.

In this study, we will use machines with one or several GPUs, aiming to show and to measure the impact of the communication times on the scheduling of HPC applications. Moreover we will try to improve this schedule by according more importance to the communication times.

To reach that goal, we have implemented a static scheduler and several heuristics within the parallel runtime XKAAPI. We have combined list scheduling algorithms and clustering heuristics to reduce the impact of communications. Using our scheduler, we have led an experimental study showing that the offline version of HEFT can give a gain of 59% compared to the makespan of the online one when the volume of communications is large enough. Moreover, using a clustering algorithm before HEFT, we have reduced the makespan of one scheduling up to 64% compared to the online HEFT.

1.5 Outline

We are going to present and discuss the state of the art in the chapter 2, then we will present our contribution in chapter 3. This contribution is composed of the implementation of an offline scheduler composed of a clustering and a list heuristic in XKAAPI detailed in section 3.1. And of a set of experiments using this scheduler which shows the impact of communications presented in section 3.2. Finally the chapter 4 will conclude this report and give some perspectives of work.

— 2 —

State of the art

Numerous programing environment have been designed to help programmers writing efficient parallel code. Although these middlewares are conceived for various architectures and have different programing models, most of them need to find a suitable scheduling.

To present the related work, we are going to explain first the various ideas carried out by the existing parallel programing environments in section 2.1. Then in section 2.2 we will present several scheduling algorithms more or less suited to our problem and discuss their pros and cons.

2.1 Writing parallel applications

When we need to write parallel code, before choosing the middleware or the API we will use, we have to think about the architecture that we target. For example, if we want to run some computation on a GPU, we can use a vendor's interface such as CUDA[**?**] or an open one like OpenCL[20]. We can combine this code with a classical API for parallelization on Symetric MultiProcessing such as OpenMP[6]. And if we need to work on a cluster of machines and send messages through a network, we can add some MPI[17] code.

However each of these environments are designed for a specific kind of architecture, if we just combine them, we still have to manage all data transfers and synchronizations. Other runtimes such as KAAPI[10], StarPU[3] and OmpSS[5] have been designed to provide a higher level of abstraction to the programmer. Therefore he does not have to manage the data transfers and synchronizations which can be deduced by the middleware, from the dataflow graph of the application. The user only need to specify for each task the access mode of the data (read and/or write).

However this classification is not the only one possible, indeed all these softwares have evolved since their first release, hence it is interesting to discuss another difference between them such as the policy used for the scheduling of the tasks onto the resources.

The simplest policy consist in dividing the work into equally sized chunks (i.e. cut a loop into as much chunks as many resources we have) and give the same number of chunks to each processors. This policy is used by default in OpenMP, however when either the resources or the work are heterogeneous, it leads to workload imbalances. To avoid that, and following the example of Cilk[8], several middlewares implement a dynamic algorithm such as workstealing[4]. Among them we can find the latest revision of OpenMP[5], XKAAPI[11], its former release KAAPI[10] and StarPU[3].

Although the workstealing produce a good work balancing, it does not take into account the outgoing communications of the stolen tasks, hence it can take bad decisions regarding communication times and contention. Therefore some runtimes such as XKAAPI[11] and StarPU[3] embed more sophisticated scheduling algorithms such as HEFT which take both heterogeneity and communication times into account. These middlewares also have an API to develop other schedulers.

It's important to note that formalisms which have been designed initially for different purposes are converging to a dataflow model with a very sophisticated runtime. For example OmpSS which is aimed at providing OpenMP's model for heterogeneous platforms allow the user to express the dependencies between tasks. As these middlewares manages heterogeneous resources, they are all concerned by the limitation of the workstealing algorithm explained previously.

2.2 Scheduling algorithms

As we mentioned earlier, the task scheduling is a NP complete problem[22]. The most widely used scheduling heuristic is the list scheduling[12, 13]. The original idea is to forbid any processor to be idle when at least one task is ready (i.e. all its predecessors are finished) to be executed. When there are more than one ready tasks, they are sorted according to some priorities, and the first task of the list is executed as soon as possible. On a model with identical processors and without communication, this algorithm give us a 2-approximation. The workstealing algorithm is based on the same idea and is indeed quite efficient for SMP architecture.

However when we work on heterogeneous machines (without communication) the list scheduling can give bad results as the bound becomes $\frac{\omega}{\omega^*} \leq \min(s+2-\frac{2s+1}{n}, \frac{n+1}{2})$ [9] where s is the number of type of resources and n the number of processors. Thus each times we add a different resource, we add one time ω^* to the worst case of the algorithm. Furthermore with identical processors and communication times we have a bound of $\omega \leq (2-\frac{1}{n})\omega^* + C$ [14] where C is the maximum chain of communications through the graph. Although in this bound C is an additive constant, if there are some communication all along the critical path, $C \leq c\omega^*$ where c is the maximal cost of one communication. Hence the worst case is proportional to the cost of a communication. Consequently the list scheduling does not scale for heterogeneous machines and we need to take into account the communications.

In this section we will first describe some adaptations of the list scheduling to heterogeneous model, then we are going to present some clustering algorithms which works on the structure of the graph and group the tasks together, zeroing the communication cost inside a cluster.

2.2.1 List scheduling

Many searchers have revisited the list scheduling, adapting it to various architectures and testing several heuristics. Nevertheless, as our aim is to study the impact of data transfers, we will only present works related to scheduling with communication times.

Earliest Tasks First[14] is designed for a finite number of identical processors with a given unitary cost of communication for each couples of processors and without communication contention.

For a tasks T we defined the earliest starting time $e_s(T) = max(CM, min_{p \in I}(r(T,P)))$ where CM is the Current Moment, I is the set of free processors, and $r(T,P)$ is the minimum time when all message needed by T are received by P. ETF always execute the task with the smallest earliest starting time.

Although this algorithm find a scheduling with a makespan $\omega_{ETF} \leq (2-\frac{1}{n})\omega^* + C$, it is designed for homogeneous processors while this study target heterogeneous machines.

The algorithms Heterogeneous Earliest Finish Time and Critical Path On a Processor[21] target heterogeneous clusters where both communication times and execution times are not the same on all processors.

The HEFT algorithm sort the ready list by non increasing upward rank, which is defined recursively for a task t by the following:
$rank_u(t) = \overline{\omega_t} + \max\limits_{t' \in succ(t)} (\overline{C_{t,t'}} + rank_u(t'))$ where $\overline{\omega_t}$ is the mean execution time of t and $\overline{C_{t,t'}}$ is the mean communication time from t to t'. Then, for each task of the ready list, HEFT assign it to the processors which minimize its finish time.

CPOP is a variant of HEFT where the rank is the sum of upward and downward rank. Moreover CPOP compute the critical path and its execution time for each processor. The best processor is dedicated to the tasks of this path. All the other tasks are assigned the processor which minimize their finish time among the remaining processors.

As the HEFT algorithm assumptions correspond to our case and as it is already implemented in XKAAPI[11] and StarPU[3], we will use this algorithm in our study. However HEFT assigns tasks without considering the incoming communications of their children which can be costly while the clustering algorithms focus on the structure of the graph, and can avoid this kind of mistakes.

2.2.2 Clustering algorithms

Clustering algorithms are designed for an unbounded number of processors, they consider the structure of the DAG and group tasks into clusters. All the tasks which belong to one cluster are assigned to the same processor (i.e. there are no communication between them).

One of the most known clustering heuristic is the Dominant Sequence Clustering[23], it assigns to each task a priority corresponding to the sum of the longest path from a root node to this task and the longest path from this task to a sink node.Then the algorithm sort the tasks by decreasing priority and try to affect each task to the cluster which minimize its start time among the cluster containing its predecessors. However if a task can start earlier when we assign it to a new cluster instead of assigning it to an existing one, we create a new cluster for this task.

The idea of CLANS [2, 18, 19] is to decompose the DAG into a tree of clans representing a recursive expression of the relationship between tasks. For a graph $G = (V,E)$, a set of nodes $X \subset V$ is a clan of the graph G if and only if $\forall (x,y) \in X, \forall z \in V - X$, z is an ancestor (or successor) of x if and only if z an ancestor (resp. successor) of y. There are three types of clans: Independent, linear and primitive (see figure 2.1). Once the graph is decomposed into clans, the algorithm traverse the tree bottom up using the communication and execution times to determine if two clans have to be merged or not.

A theoretical comparison[15] between the algorithms DSC, MCP (Modified Critical Path, similar to DSC), MH, HU (lists algorithms) and CLANS have shown that CLANS is robust and efficient for a large and diverse set of graphs. All the others algorithms give very bad performances for graphs with a low ratio computation over communications (also called grain).

The convex clustering algorithm[16] produces an acyclic graph of clusters. Indeed a cluster C is convex if and only if $\forall\ (x,z)\ \in\ C$ all task y, such that $x\ \prec\ y\ \prec\ z$, is in C (see figure 2.2).

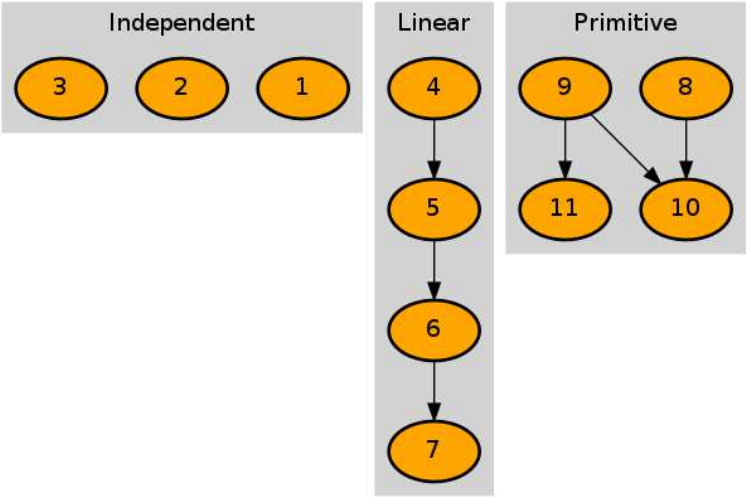


Figure 2.1 – Different type of CLANS.

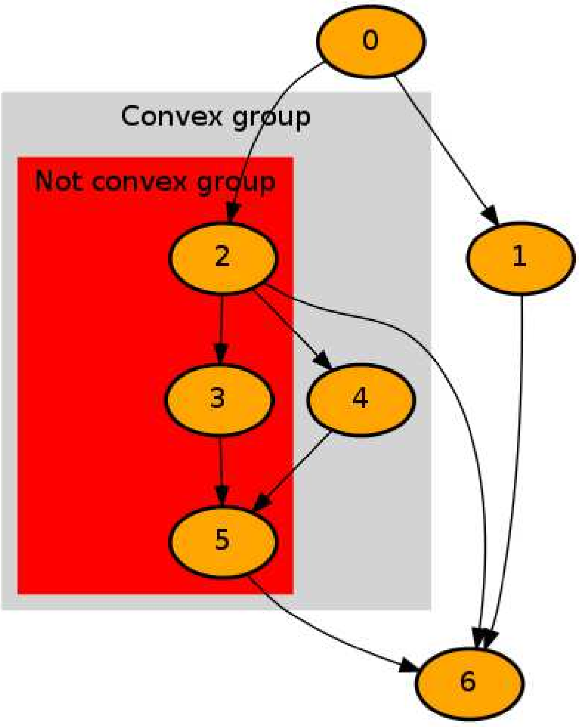


Figure 2.2 – Convex and non convex group of tasks.

By this property, we know that there are no two way communications between two clusters. Moreover as the graph obtained by merging the tasks inside a cluster is a Direct Acyclic Graph, it allow us to run any scheduling algorithm on it.

To study the impact of communication times, we want to reduce the grain of the DAG by managing clusters as if they were large tasks. As the dataflow model implemented by XKAAPI doesn't allow cyclic dependencies between tasks, the only algorithm which fits our needs is the convex clustering. Hence we will implement this algorithm inside XKAAPI, and we are going to combine it with the HEFT algorithm to understand the impact of communication times on a scheduling.

— 3 —

Implementation and analysis of communication aware scheduling algorithms

By default, XKAAPI uses an online scheduler and compute data dependencies in a lazy way, nevertheless to study the impact of communications and to run structural algorithms, we have decided to work with an offline scheduler. Moreover an offline scheduler is more suitable for regular problems with large amount of data. However the online heuristic work with actual time when the offline one uses predicted times thus only the first one is able to adapt to unexpected effects such as contention. XKAAPI's scheduler provides various heuristics, among them, we can find an online adaptation of the HEFT algorithm. The main difference between this online version and the original one is that the first one cannot sort the ready tasks as the DAG is not fully known.

At first, we have compared the online adaptation of the HEFT algorithm with an offline version, using linear algebra applications, as this heuristic do not consider the communication times as much as the original one. Then we have tried the static scheduler with and without a clustering phase before the HEFT algorithm to take even more the communications into account.

We will start this section by explaining the implementation choices of our scheduler and by giving theoretical analysis of its costs. Then we will detail the objectives of the experiments that we have conduct, their results and our interpretations.

3.1 Implementation of a static scheduler

XKAAPI has an API for writing scheduling heuristics, however as it is designed for online algorithms this interface is quite limited. First, we will present this API and explain its limits, then we are going to discuss the implementation of our online scheduler, and we will give a theoretical analysis of its cost. Finally we will give an illustration of our scheduler on a small example.

3.1.1 XKAAPI's scheduling API

To design a scheduler for XKAAPI, one has to write three callback functions `kaapi_affinity_init_mySched`, `kaapi_affinity_push_mySched` and `kaapi_affinity_steal_mySched`. The first one is called by the runtime during the initialization phase, when the ready tasks list has been computed. The second is called when a task finishes and activates another one to decide which processor should execute the newly available task. As a task can finish at anytime on any processor, this function can be called asynchronously. Hence to avoid useless synchronizations, which could slow down the execution, one must limit the use of shared data in this function. The last one is called when a processor try to steal another one to decide which task he can steal. Obviously this function is also asynchronous.

Although this API is minimalistic, as we can extract the DAG from the ready tasks list during the initialization, we are able to run an offline scheduler during this phase and apply its decision when the runtime calls the `push` and `steal` functions. However this interface do not enable us to force the order of execution of two tasks. Thus we need to add some (empty) internal tasks to enforce dependencies between tasks which belong to different clusters.

3.1.2 Offline Scheduling

To run offline scheduling algorithms we need to obtain the DAG representing the application, XKAAPI is able to construct this graph during the execution. Furthermore for the applications which do not create tasks recursively, this graph is known before the beginning of the computations. As an example some Linear algebra problems can be solved by this kind of programs. In this study, we have studied a Cholesky and a Lower/Upper factorization by tiles.

By default, in XKAAPI, each processor has a part of the work in his queue of tasks, nevertheless the XKAAPI's C++ API allow to spawn tasks with the keyword `SetStaticSched` which force XKAAPI to maintain a centralized list of ready tasks. As each task has a pointer to all its successors, the dependencies graph is known within the runtime. However as the convex cluster algorithm require a test of the relationship between almost every pairs of vertices of the DAG, with this pointer based representation, computing these relationships can be costly. Therefore the first step of our static scheduler is to extract the DAG into an adjacency matrix and to run a Warshall Algorithm which compute all the parenthood relations. Although this algorithm have a worst case in $O(|V|^3)$ for a graph $G = (V,E)$, and a mean case in $O(|V|^2)$ it is worthwhile as it allow us to know in constant time if two nodes are related or not.

Once these dependencies are computed, we are able to run our implementation of the convex cluster algorithm (see algorithm 1). The original convex cluster algorithm select two independent random nodes and use them to divide the group into four partitions A and $A^{\sim}$ which are independent, their predecessors $A^{<}$ and successors $A^{>}$. Once these partitions are computed, either the smallest of A and $A^{\sim}$ is large enough and all the partition are recursively divided, or they are all grouped into one cluster.

Our implementation is close to this algorithm however as it is designed for a theoretical machine with an unbounded number of identical processors, it requires a few adjustments, for example in the original version the recursion stops when the partition size is less than twice the communication cost. As we do not have a unitary communication cost, we have decided to fix a maximal size for a cluster and to stop the recursion when a cluster is smaller than this size (see line 20). Then, the clusters are mapped on the processors using the HEFT algorithm.

In addition, some experiments described in section 3.2.2 showed that the original criterion used to determine which clustering was the best among the random test gave unstable performances. Hence we have decided to change it to keep always the clustering giving the largest cluster (see line 14), and we have shown that this heuristic gave better results. Finally, we choose to recursively divide each individual clusters if it is large enough (line 20) while the

Algorithm 1 Convex cluster

```
function EXTRACT_CLUSTERS(Graph $G = (V,E)$)
    $Part \leftarrow NULL$
    $CurMax \leftarrow 0$
    for $rdt \in [0..MaxRandomTry]$ do
        $pivot \leftarrow RandomNode(G)$
        $< A, A^{\sim}, A^{<}, A^{>} >= Decompose\_4\_parts(G, pivot)$
                    ▷ Decompose G into four partitions pivot, nodes independent from pivot,
                    ▷ pivot's ancestors and pivot's successors
                    ▷ see figure 3.1b, Cost: $O(|V|)$
        $Update\_partitions(< A, A^{\sim}, A^{<}, A^{>} >)$
                    ▷ $A^{<}$ and $A^{>}$ becomes predecessors (respectively successors)
                    ▷ of all nodes from $A$ and $A^{\sim}$, see figure 3.1c
                    ▷ Cost:$(|A^{<}| + |A^{>}|) * |A^{\sim}| = O(|V|^2)$
        if $MAX(|A|, |A^{\sim}|) > CurMax$ then    ▷ We have found a better decomposition
            $CurMax \leftarrow MAX(|A|, |A^{\sim}|)$
            $Part \leftarrow < A, A^{\sim}, A^{<}, A^{>} >$
        end if
    end for
    for all $p \in Part$ do
        if $|p| > MaxClusterSize$ then
            $Extract\_Clusters(p)$    ▷ Divide the partition if big enough
        else
            $Add\_Cluster(p)$    ▷ Save the cluster
        end if
    end for
end function
```

original algorithm divide either all the clusters or none of them.

The two important (and costly) steps of this algorithm are the initial decomposition into four clusters, and their update to keep the clusters balanced. Starting from the graph shown on figure 3.1a, we choose randomly a pivot (the node 7) and we put all the others nodes in one of the three groups its predecessors $A^{<}$, its successors $A^{>}$ all the others $A^{\sim}$. At the end of this step we have the clustering displayed on figure 3.1b. As we can know in $O(1)$ the relationship between two nodes, this step costs only $O(|V|)$ comparisons. During the second step, we compare each nodes of $A^{<}$ and $A^{>}$ with the nodes of $A^{\sim}$ to determine the common predecessor (resp successors) of A and $A^{\sim}$ keep their initial placement, the others are moved in the group A. By this way we obtain the clustering presented on figure 3.1c. The second step has a cost of:

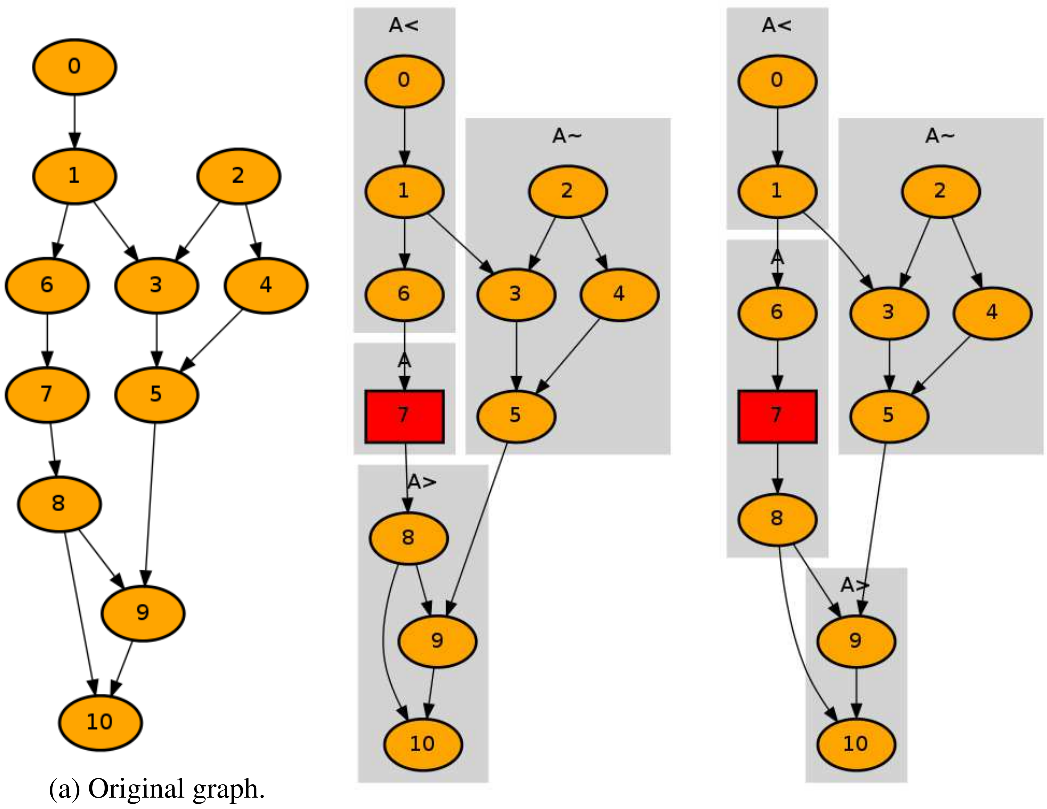
$$C = (|A^{>}| + |A^{<}|) * |A^{\sim}|$$

we know that at the end of the first step $|A| = 1$ which implies:

$$|A^{>}| + |A^{<}| + |A^{\sim}| = |V| - 1$$

thus:

$$C = (|V| - 1 - |A^{\sim}|) * |A^{\sim}|$$



(a) Original graph.

(b) `Decompose_4_parts()`

(c) `Update_Partitions()`

Figure 3.1 – Decomposition and update step of convex cluster algorithm.

The two terms are bounded by $|V|$ so we have:

$$C \leq |V|^2$$

and so if $|A^{\sim}| = \frac{|V|}{2}$,

$$C = (|V| - 1 - \frac{|V|}{2}) * \frac{|V|}{2} = \left(\frac{|V|}{2}\right)^2 - \frac{|V|}{2}$$

hence we have:

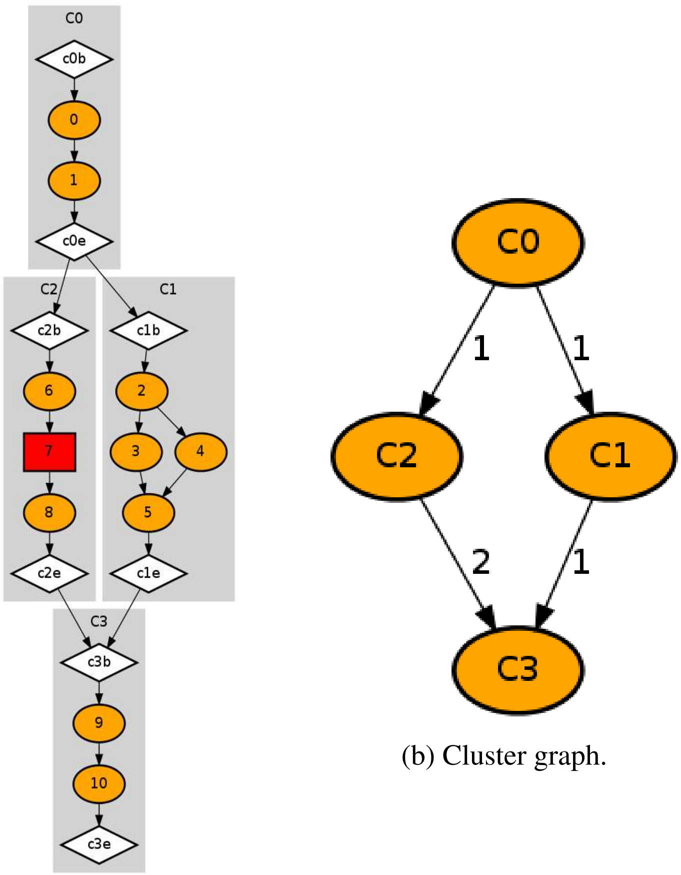
$$C = O(|V^2|)$$

Finally, if we assume that the clusters are balanced it takes $log\left(\frac{|V|}{MaxClusterSize}\right)$ calls to stop the recursion, as the two first step are repeated $MaxRandomTry$ times, the cost of our implementation is:

$$O\left(MaxRandomTry * |V|^2 * log\left(\frac{|V|}{MaxClusterSize}\right)\right)$$

All the important parameters which can influence the performances of a schedule such as the number of random tries (line 4) and the maximum size of a cluster (line 20) can be set by the user, and their influence is studied in section 3.2.

As the clustering phase aim of increasing the grain of the graph, we need to be sure that each cluster will behave as one big task. Therefore in the runtime, we have to add some empty synchronization tasks at the beginning and at the end of each clusters as shown in figure 3.2a



(b) Cluster graph.

(a) Graph after adding the internal tasks.

Figure 3.2 – Final transformations from a graph of tasks to a graph of clusters.

(white diamond shaped nodes). Indeed with the clusters of the figure 3.1c, without these tasks, as XKAAPI use a work stealing algorithm to execute a tasks as soon as it is ready, if the clusters $A^{<}$ and $A^{\sim}$ are executed on the same processor, the second one might start before the end of the first, delaying the execution of the cluster A. Although these synchronization tasks are empty, they increase the size of the actual DAG and they force the runtime to execute more instructions, hence we will discuss their impact and show that the overhead induced is negligible in section 3.2.2.

As we need a final $O(|V|^2)$ loop to convert the tasks graph into a clusters graph (as represented on figure 3.2b), we us it also to compute the source and sink nodes of each cluster and to add the internal tasks. Moreover, during this loop, we annotate the edges of the graph with the volume of communications induced by the inter cluster dependencies.

When the DAG of clusters has been computed, we can find the mapping between the clusters and the processors according to the HEFT algorithm for instance. This heuristic has a cost of $O(ncl^2)$ where ncl is the number of nodes in the graph of clusters. We know that:

$$ncl \leq \frac{|V|}{MaxClusterSize}$$

hence the cost of our implementation of HEFT is:

$$O\left(\left(\frac{|V|}{MaxClusterSize}\right)^2\right)$$

Then, the scheduling is saved as a mapping from clusters to processors. As each task is annotated with the cluster to which it belongs, when a processor calls the function `kaapi_affinity_push` for a specific task, we can find in constant time the processor on which we have to push it.

Finally, our offline scheduler has a theoretical worst case cost in $O(|V|^3)$ due to the Warshall algorithm, however the mean case cost is still $O(log(|V|) * |V|^2 * MaxRandomTry)$ (due to the convex cluster algorithm). Although this cost is quite high, in some domains such as physical simulation or signal analysis, applications are often iterative and regular. This means that the same parallel code has the opportunity to be executed several times in a row using the same schedule. Hence we can amortize that initial cost. As our aim is to study the impact of communication times, we will not try to amortize this initial cost in the following experiments and we will separate it from the execution times in our results.

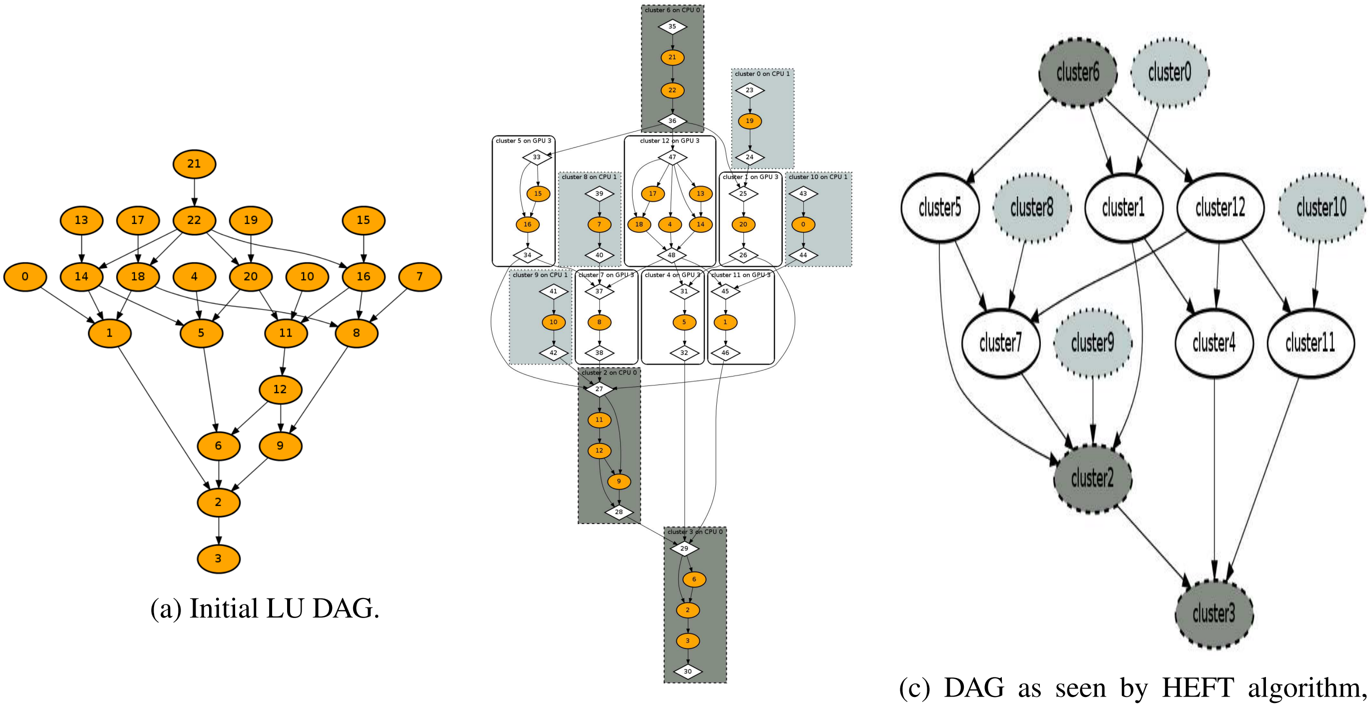
3.1.3 Simple example

The figure 3.3 depicts the transformations operated by our scheduler to the DAG of a LU decomposition by block. For this run the number of random tries is set to $\sqrt{|V|}$, and the maximum size of a cluster is 5. The initial DAG which contains 23 nodes and 30 (figure 3.3a) edges is reduced to a DAG of 13 nodes and 18 edges (figure 3.3c). In order to keep the DAG simple, the example is small (a few communications) hence the clustering phase might not be efficient. Anyway, using the DAG of clusters, HEFT algorithm produces the mapping shown in figure 3.3b, each cluster of the same color and shape belong to the same computing resource. We can see that a lot of communications happens between clusters mapped on the same processor (for instance on the GPU, white rounded clusters).

With this DAG, the clusters and the schedule have been computed in $2.4e^{-5}$ seconds while the execution takes $2.3e^{-2}$ seconds. However as explained earlier it is not useful to cluster this DAG as it does not contain a lot of communications. Indeed by scheduling the original graph with the offline HEFT algorithm, we obtained an execution time of $8.0e^{-3}$ seconds.

3.2 Experiments and results

Before showing the impact of communication times on a scheduling, we are going to present our experimental setup. The machines used and the applications tested for our study are described in section 3.2.1. First we will confirm experimentally the theoretical analysis in section 3.2.2. After this validation, we present the results obtained by combining the HEFT heuristic and the convex clustering algorithm compared to the online adaptation of HEFT in section 3.2.3. Finally in the section 3.2.4 we are going to give conclusions on these experiments.



(a) Initial LU DAG.

(b) Clustered DAG mapped on the processors.

(c) DAG as seen by HEFT algorithm, mapped on the processors.

Figure 3.3 – DAG representing a LU decomposition.

Name	#CPU	#Core / CPU	CPU vendor	CPU model	CPU freq (Ghz)	Memory (Gio)	#GPU	GPU vendor	GPU model	GPU memory	GPU clock freq (Ghz)	CUDA capability
Mapuche	1	4	Intel	Core I7-2600	3.4	8	1	Nvidia	NVS 300	512 MB DDR3	1.23	1.2
IdGraf	2	6	Intel	Xeon X5650	2.67	72	8	Nvidia	Tesla C2050	3GB GDDR5	1.15	2.0

Table 3.1 – Hardware used for the experiments.

3.2.1 Experimental setup

Hardware and Operating Systems

All the experiment presented in this study, including the example in the previous section have been run on one or the other of the two following machines: Idgraf[1] a HPC workstation with 12 cores and 8 GPUs managed by Grid5000 and Mapuche a personal workstation with 4 CPUs and one GPU. The hardware of these machines is described more precisely in table 3.1, for both machines we hyper-threading and thermal throttling are turned off.

It is important to note that Idgraf has a hierarchical topology as shown in figure 3.4, the eight GPUS are split over two PCI Bridge which are linked together. Each bridge manage 4 GPUs separated in two groups which belong to two different switches. With this particular

[1] http://digitalis.inria.fr/index.php/Usage#idgraf

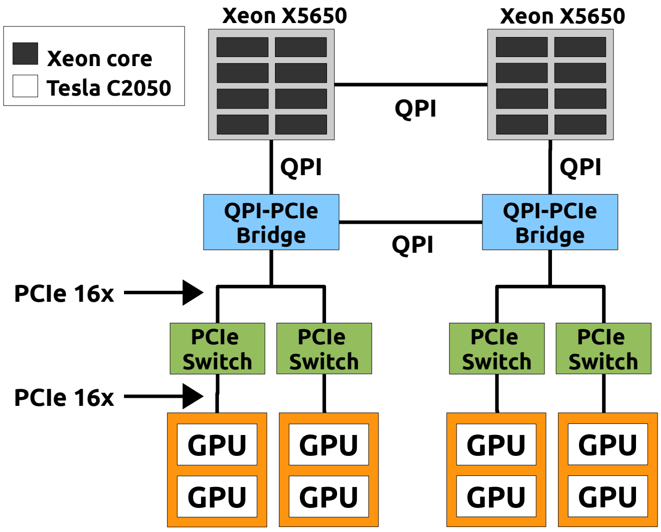


Figure 3.4 – Idgraf's topology[11].

topology, simultaneous communications involving several GPUs might led to contention on the switches, this contention will be highlighted in section 3.2.3.

For our experiments, we use Idgraf in deployment mode[2], in this mode we can be sure that no other user can use the machine at the same time (exclusive access), and we can install our own system which is a Debian squeeze with a Linux kernel 3.2.0.2-amd64. Mapuche's Operating System is an Ubuntu 12.10 LTS with a Linux kernel 3.2.0-40.

Applications

The applications used for the experiments are part of the example proposed by XKAAPI 1.0.4. They use the CBLAS, LAPACK(E), PLASMA and QUARK libraries which are all installed thanks to the plasma installer (version 2.4.6)[1]. These programs also use the CUBLAS library provided by Nvidia with CUDA 5.0, which is installed using the latest installer available on Nvidia's website (released the 01/10/2013).

We can see in figure 3.5 that the DAG of a Cholesky decomposition by bloc starts with several independent tasks and finishes with a lot of communications. This particular configuration is very interesting for our study, because, as explained in section 2.2.1 it can trick the HEFT algorithm which will not anticipate well these final communications. Hence we are going to use this application to illustrate our point in the following sections. Moreover, as scheduling algorithms can easily overlap the communications when the grain is higher than 1 (i.e. more computations than communications), we will work with small size of matrices ($\leq 2000 * 2000$ floats).

The three scheduling algorithms presented in this chapter use a performance model already implemented in XKAAPI. This model aim to give a theoretical prevision of execution time

[2] https://www.grid5000.fr/mediawiki/index.php/Deploy_environment-OAR2
and
http://digitalis.inria.fr/index.php/Usage#I_want_to_change_the_system_.28OS.2C_software.29_on_the_machine

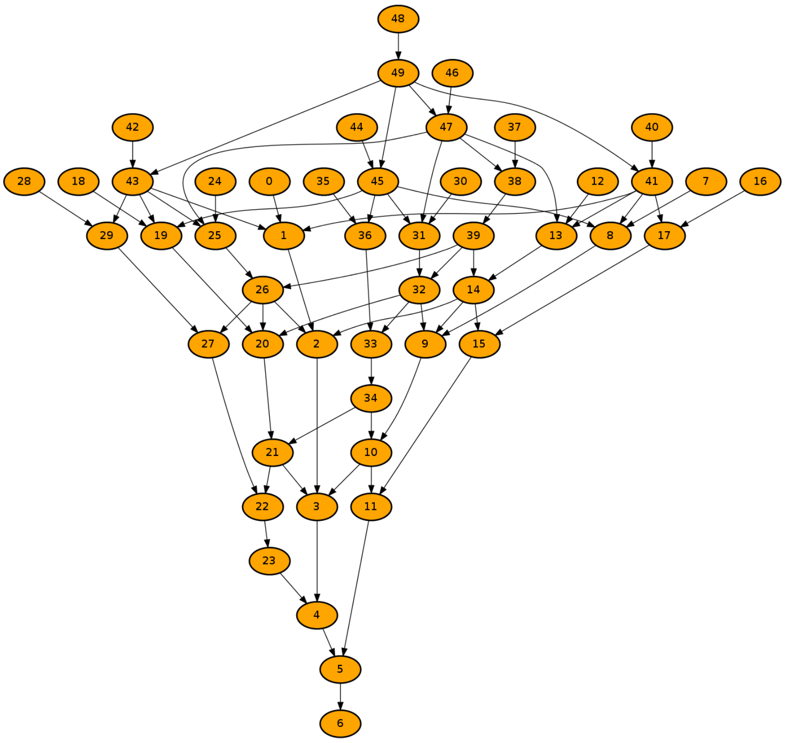


Figure 3.5 – DAG of a Cholesky decomposition by tiles.

of each tasks on each processor, and predict the time needed to transfer n bytes of data from one processor to an other. However these predictions are based on previous execution for the task and the estimation of communications does not take into account contention effects. To calibrate completely this performance model, one need to calibrate it, i.e. to run the application at least twice: only on CPUs and only on GPUs. During the calibration, XKAAPI save the execution times of the tasks. However as GPUs do not embed a complete Operating System, we measure this execution time on the CPU, hence it include a few communication times. We will discuss the impact of this bias in section 3.2.3.

Concerning our algorithms, unless stated otherwise, we use the following parameters: $NbRandomTries = \sqrt{|V|}$, $MaxClusterSize = 10$, and for the studied applications $MatrixSize = 1000$, $BlocSize = MatrixSize/10$.

For each curve presented in the following sections, each points represent the mean runtime of 15 different runs with the same parameters, and a confidence interval of 95% is displayed as an error bar, excepted for figure 3.8 where each run is represented by one point.

Finally, by default the execution time presented for the algorithms HEFT offline, and convex cluster correspond to the total time minus the overhead induced by the static scheduler.

3.2.2 Validation of the theoretical analysis

Before using our implementation in an experimental study, we need to validate our theoretical analysis. The first point of this validation concern the impact of the additional synchronization tasks we add to enforce clusters execution order, in terms of middleware overhead. To show this overhead, we have studied the worst case: when each task is a cluster, although in that case the internal tasks do not modify the execution order, we create two empty tasks for each

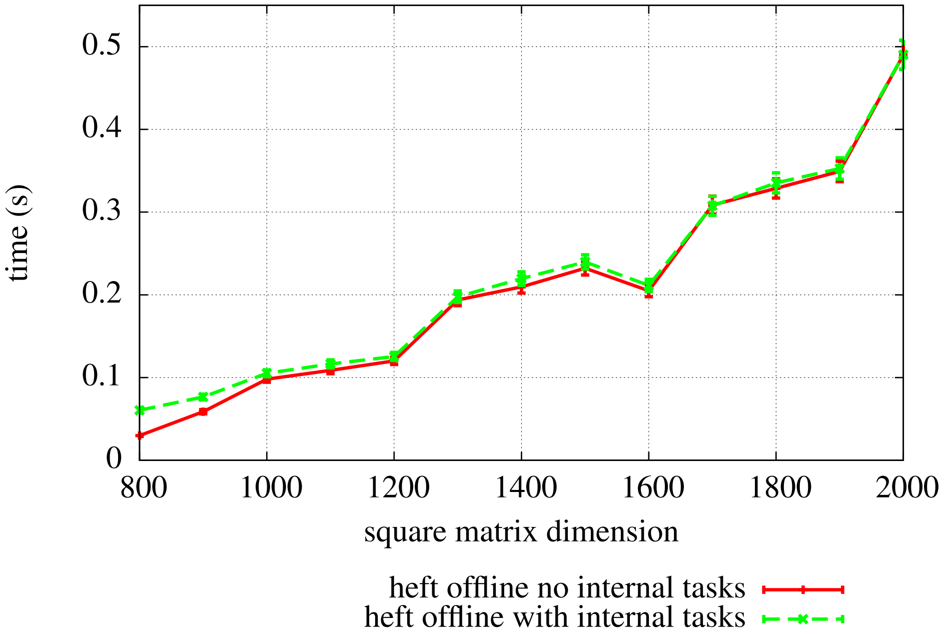
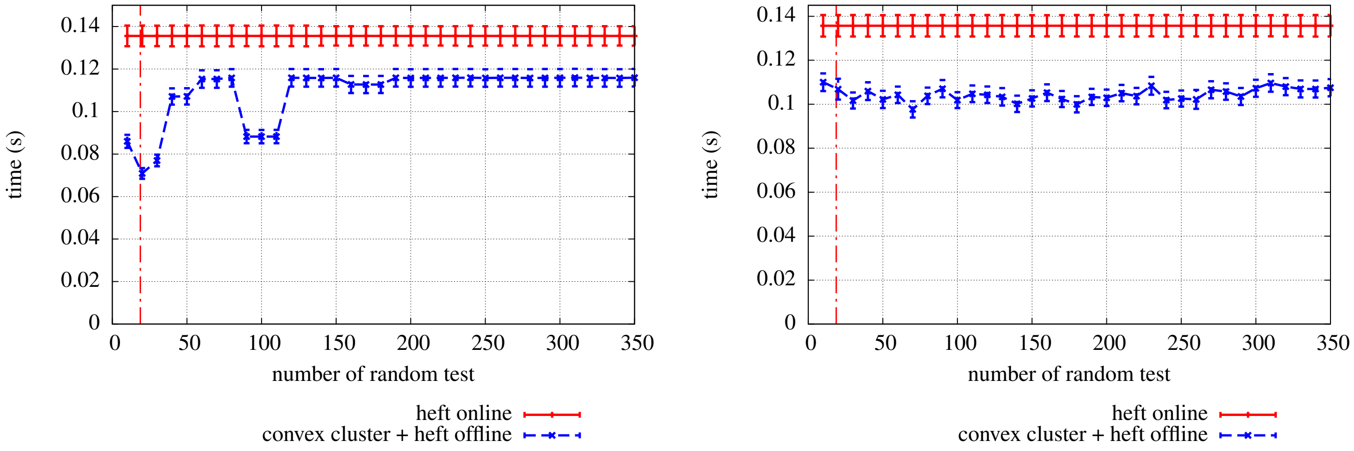


Figure 3.6 – Worst case overhead caused by internal tasks, on a Cholesky reduction, on Mapuche.

existing node of the graph. In the general case, these tasks add also another overhead due to the new synchronizations between tasks of dependent clusters. However, we do not evaluate this overhead as it will be included in the execution time of the clustered graph. The figure **??** shows that if these tasks can have a huge impact on very small instances (+48% for a matrix of size 800), this impact is quickly compensated: for a matrix of size 1000 the overhead is only +10%, and +1% for a size of 1300.



(a) Original heuristic.

(b) Our heuristic.

Figure 3.7 – Heuristic used to choose the best clustering, on a Cholesky reduction, on Mapuche.

Before evaluating the global overhead, as it depends on the number of random tries that we are testing to find our clustering, we have to decide on a reasonable value. While we were studying this parameter, we have realized that with the heuristic proposed by the original convex cluster paper to choose the best clustering, we need a lot of tries to stabilise the results. Moreover, the stabilisation occurs on the worst execution time provided by this clustering as it is shown by figure 3.7a.

In the original algorithm, we keep the clustering which maximize the size of the smallest of the two partition A and $A^{\sim}$. Either that partition is big enough and all the four partition are recursively divided or the four partitions are merged into one big cluster. Using this method, our clusters were never well balanced. Thus we have have proposed another heuristic: we choose the graph partition which maximize the maximum of $|A|$ and $|A^{\sim}|$. The objective of this heuristic is to be able to decompose this large cluster once again. This way we should find a more balanced global clustering. Indeed, figure 3.7b shows that with this approach the stabilisation occurs faster with a better execution time. Nevertheless, we have to note that the best clustering found by our algorithm results in a slower execution than the one found by the first heuristic.

Our algorithm converges quickly to stable results, thus we can set the number of random tries to $\lfloor\sqrt{|V|}\rfloor$ (in our example, we have $|V| = 352$, so $\lfloor\sqrt{|V|}\rfloor = 18$). We have represented this threshold by a dashed vertical line in figure 3.7.

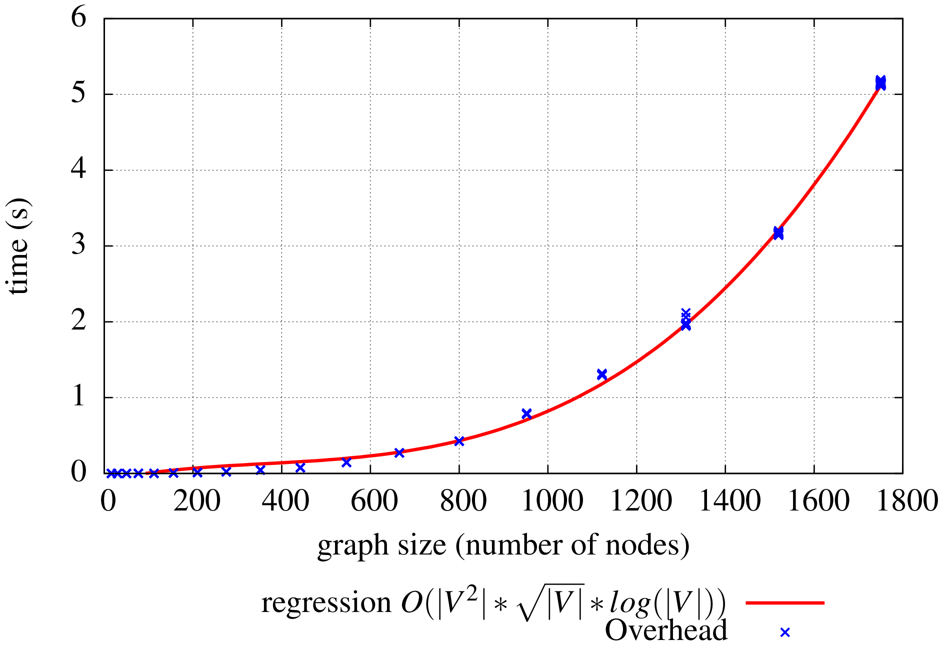


Figure 3.8 – Overhead induced by the computation of the clusters and the HEFT algorithm, on a Cholesky decomposition, on Idgraf.

Finally, using this setting, we can evaluate the overhead of our scheduler. Figure 3.8 represent the time needed to compute the clustering and execute the HEFT algorithm for different size of graph, and a regression to a polynomial function $a * |V|^2 * log(|V|) * \sqrt{|V|} + b * |V|^2 + c * |V| + d$. The regression fit with a standard error of 1.067% on the parameter a. Hence we can accept the hypothesis that for this particular application the overhead is in $O(log(|V|) * |V|^2 * \sqrt{|V|})$.[3]

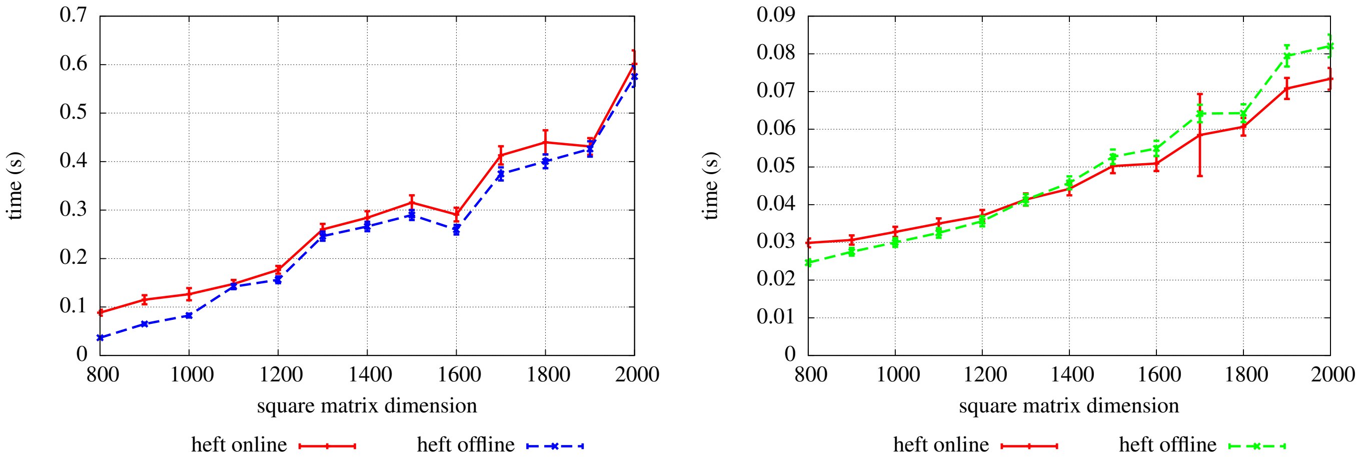
[3] The standard deviation is increasing with the size of the graph, despite the fact that we have also increased

Although this overhead is high, we can hope to amortize it if we use the same schedule for many instances of a problem. However, as this is not the objective of our study, we will ignore this overhead in the experiments presented on the next section.

3.2.3 Analysis of communications

Impact of the performance model

There are two important differences between the online and the offline version of the HEFT algorithm, the first one is that the offline algorithm sorts the tasks by upward rank, which mean that a task priority is related to the length of the chain of tasks that depends on it. As the online version does not have a global vision of the DAG, it maps tasks to processors as soon as they are ready, without sorting them. However the second difference is that the offline algorithm only uses theoretical execution and communication times while the online one can react on the fly to varying execution times. Hence if the model is erroneous, he will be able to adapt the next decisions.



(a) Comparison on Mapuche.

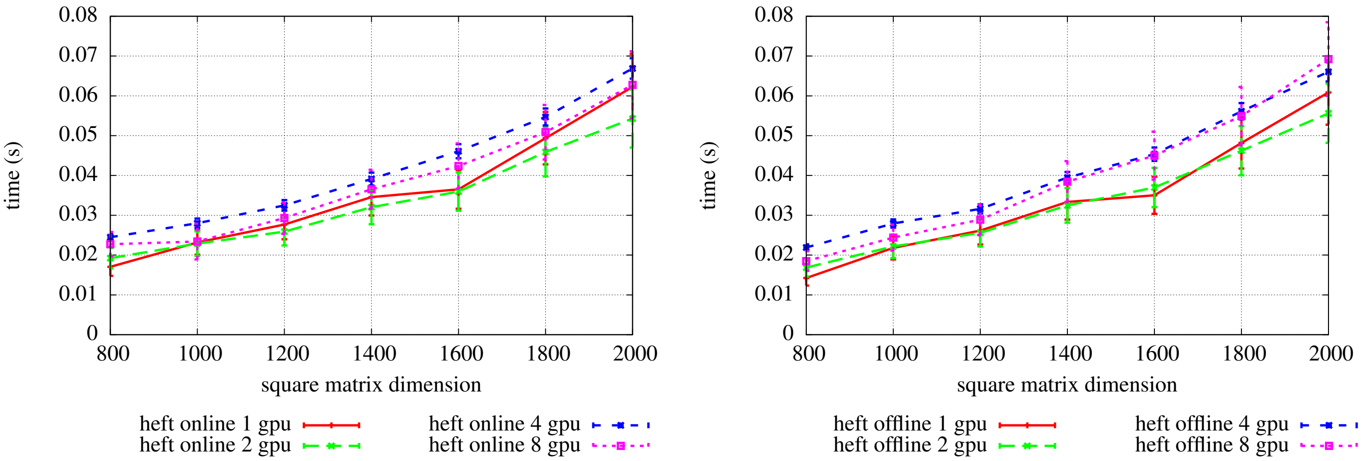
(b) Comparison on Idgraf.

Figure 3.9 – Online vs offline HEFT scheduling, on Cholesky reduction, on both machines.

To show these differences, we have compared the makespan of a Cholesky reduction on our two machines using all the available resources with the two versions of HEFT. The results are displayed in figure 3.9. The first important point is that, on Mapuche (figure 3.9a), the offline algorithm provide always a better makespan than the online one. Indeed we have observed a gain from 1 to 59%. This result confirms our first idea: we are in a configuration where there are many communications and not enough computations to overlap them. Hence, we can obtain a substantial gain by using an heuristic aware of these communications.

Nevertheless, the same experiment on Idgraf gives a different result, indeed we can see in figure 3.9b that if we still have a gain of 17.7% with a matrix of size 800. However, when we increase this size, the gap shrinks and finally, when it exceed 1300 the online algorithm gives

the number of run with the size of the graph to compensate this effect. Hence, it can influence the regression. Therefore, to validate definitively this hypothesis, we should run the same experiment on different graphs looking carefully this standard deviation.



(a) Online algorithm.

(b) Offline algorithm.

Figure 3.10 – Effect of the number of GPU used, on the execution time of a Cholesky reduction, on Idgraf.

better results up to 10.2%. As the main difference between the two machines is the number of GPUs and as the advantage of the online scheduler compared to the offline one is the capacity to adapt to a biased model, our first hypothesis, is that a contention which cannot be anticipated by the model appears on the PCI buses (see figure 3.4) when we use all the GPUs.

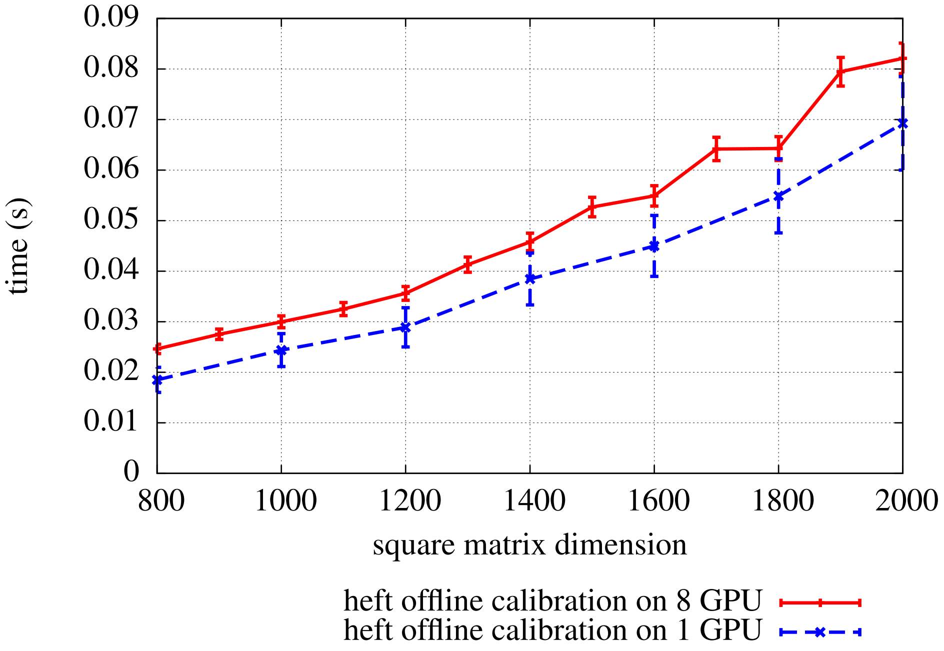


Figure 3.11 – Effects of the number of GPUs used for calibration on the execution time of a Cholesky reduction on Idgraf.

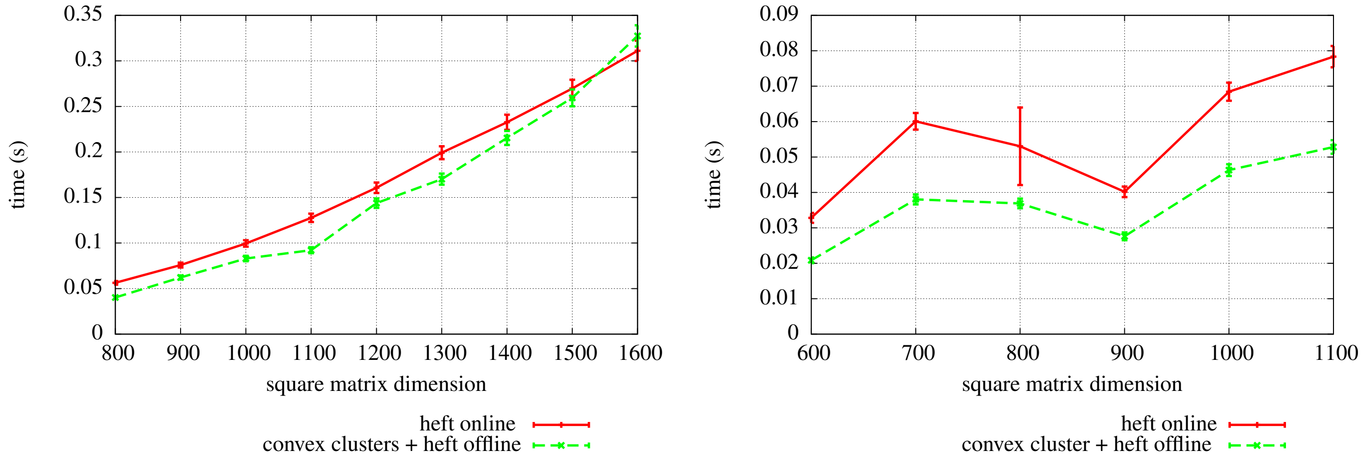
To prove this hypothesis, we have run the same experiment on one, two, four and eight GPUs with both algorithms. The results, shown in figure 3.10a and figure 3.10b shows that for both algorithms, when we use four or more GPUs, the makespan increase although our logs shown that we are using all the available GPUs.

In addition, we have found by looking at our experiments logs that for many tasks, the predicted execution times (i.e. the execution time extracted from the calibration runs) were increasing as we were using more GPUs. As the eight GPUs are identical, this rise means that our performances model include some communications in the execution times and these communications takes more time when we use all the available GPUs. Moreover as this contention appears in our execution times, our algorithms were biased. However, we can easily avoid this bias by performing our calibration run with only one GPU. Doing this, we are able to reduce the execution time of 24.7% on 8 GPUs as we can see on figure 3.11. All these results validate our hypothesis, there is some contention on the PCI buses.

Finally, we have clearly shown that the offline algorithm is able to give better results (up to 59% in our experiments) by avoiding some communications. However, if the model does not fit to the reality (for example when there is some contention), an online algorithm is more adaptated. Hence we can think about implementing an hybrid scheduler which takes initial decisions statically and adapts (or recompute them) if the actual execution times does not match to the expected ones.

Reducing the communication times' impact using a clustering algorithm

In the following experiments, we have increased the grain of the DAG using the convex cluster algorithm before running the offline HEFT algorithm. Using this combination, we will show that we can have even a better gain than with the offline or the online algorithm only.



(a) Mapuche.

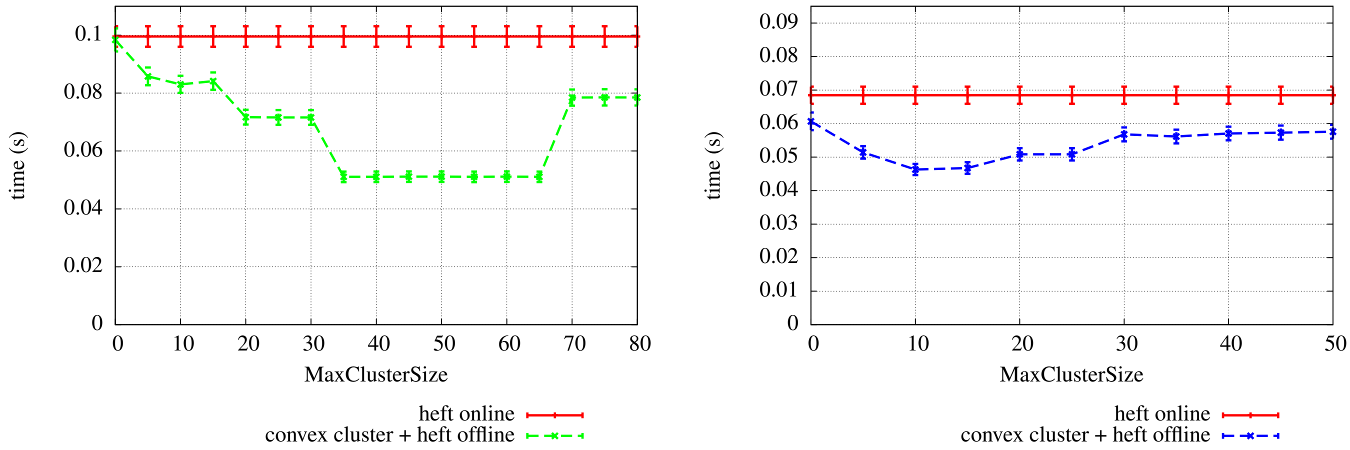
(b) Idgraf, $BlocSize = MatrixSize/20$.

Figure 3.12 – Square matrix dimension impact on convex cluster and HEFT algorithm, on a Cholesky reduction, on both machines.

Our first experiment on the convex cluster algorithm aim at showing for which matrix size this heuristic can be useful. We can see in figure 3.12a that as for the offline HEFT, the convex cluster algorithm give better results for small matrices, with a gain up to 28.5% on Mapuche with a matrix size of $800 * 800$. Nevertheless, there is still one parameter of the convex cluster algorithm that we have not tested yet which is the maximum cluster size. Following experiments will show that with an appropriate size of cluster, we can get a even better makespan.

On idgraf, to obtain a better scheduling, we had to increase the volume of communications by decreasing the bloc size. With this parameter, the convex clustering give a better schedule than the online HEFT as we can see in fiugre 3.12b. The best case tested in our experiment is for a matrix size of $600*600$ where we reduce the makespan of 64%.

From this experiment, it is clear that with small matrices, we can highly improve the scheduling obtained by the online algorithm, hence we will use a small matrix $1000*1000$ for the next experiment and we will keep the same bloc sizes as for this experiments. Finally, we are going to study the impact of the maximal size of one cluster on the scheduling.



(a) Mapuche.

(b) Idgraf, $BlocSize = MatrixSize/20$, 4GPUs.

Figure 3.13 – Effect of the maximum size of clusters on the convex cluster algorithm, on a Cholesky reduction, on both machines.

By increasing the maximum size of a cluster (see algorithm page 13, 1, line 20), we increase the grain of the graph and we reduce the amount of communications that HEFT has to manage.

Figure 3.13a show the results of our last experiment on Mapuche, we can see that we need large clusters ≥ 30 to obtain the maximal gain. However if we use a cluster size larger than 70, it seems that we loose to much informations and the HEFT algorithm cannot give an efficient scheduling anymore. Finally, we can notice that with $MaxClusterSize = 35$ we reduce the execution time by 48.7% in comparison to the online HEFT algorithm.

In figure 3.13b, we displayed the results of the experiment on Idgraf using 4 GPUs, we can see that the best cluster size is quite different than for Mapuche. This difference comes from the fact that as Idgraf embed more processors than Mapuche, hence it needs a larger graph (i.e. smaller clusters) to use them efficiently. Still, with the convex clustering, we can obtain a gain of 32.3% on Idgraf compared to the online one, while the offline algorithm only gave a gain of 11.4% (fig 3.13b, $MaxClusterSize = 0$).

3.2.4 Conclusions

In this chapter, we have observed experimentally that the overhead of our scheduler seems to fit the theoretical analysis which gives a cost in $O(|v|^2 * \sqrt{|V|} * log(|V|))$. Additionally we have justified some design choice such as the use of internal tasks to enforce synchronizations between dependent clusters.

Then, we have shown that using an offline algorithm can help to reduce the impact of communication times on the makepan. Nevertheless, offline algorithms rely on models and we have highlighted that when some unpredicted event, such as contention, occurs we might loose all the benefits of an offline algorithm.

Finally, our study have pointed out that by combining a list algorithm such as HEFT with a clustering algorithm, we can reduce even more the makespan of a schedule. Our experiments have shown a gain up to 64% using 4 GPUs on a machine with 8 GPUs.

— 4 —
Conclusions and perspectives

During this study, we have worked on XKAAPI[10, 11], an existing middleware for parallel programing in which we have implemented a static scheduler with several heuristics. Then, we have used these heuristics to evaluate the impact of communication times on the decisions a scheduler should take.

4.1 Main contributions

Our first contribution has been the implementation of a static scheduler using a classical list algorithm, Heterogeneous Earliest Finish Time combined to a clustering algorithm, convex clusters. We have analysed the cost of our scheduler theoretically and we have verified it experimentally.

We have focused on applications with a fine grain, where the communication times have a huge impact on the total execution time. Using an offline algorithm, we have been able to reduce the makespan of an application by 59%. We have also highlighted some effect of contention on an heterogeneous machine and the fact that because of this effect, an online algorithm becomes more efficient than the offline ones.

Our study has also shown some limits of the performance model used by XKAAPI and we have proposed one technique to reduce their effects on the offline algorithms.

Finally we have increased the grain of a DAG using the clustering algorithm before running HEFT to obtain a schedule. This way, we have observed a substantial gain up to 64% on the makespan, compared to the online HEFT.

However, these results do not take into account the overhead needed to run scheduling algorithms. Hence, we still need to work on the ability to reuse a schedule to amortize its initial cost.

To conclude, this study has highlighted the fact that the communication times are not negligible when we use some GPUs, moreover we have demonstrated that some algorithms from the literature can be used efficiently and combined together to reduce the effect of these communications.

4.2 Future work

It would be interesting to let the user save and reuse a schedule and to experiment our scheduler with iterative applications which runs several times in a row the same routine. Hence we should be able to amortize the initial cost of our scheduler and to evaluate a global gain.

Additionally, we would like to complete our study by a comparison with some others algorithms designed to reduce the impact of communication times such as DSC or CLANS. However as explained in section 2.2.2, these algorithms can create a cyclic DAG of clusters which is incompatible with XKAAPI's runtime. Hence we might need either to adapt them or to run another algorithm after them to merge or cut some clusters to remove the cyclic dependencies.

Finally An interesting perspective would be to design an adaptive algorithm able to update the different parameters of the heuristic tested in this study, and which combine an initial (offline) schedule with an online tool allowed to modify it if the actual times are not close enough to the predictions. Moreover using such an algorithm, we could avoid the initial calibration run required by our perfomance model.

Acknowledgments

Thanks Thierry Gautier for the figure 3.4 extracted from the paper XKAAPI IPDPS 2013 [11].

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

Bibliography

[1] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jackub Kuzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. *PLASMA Users' Guide, Parallel Linea Algebra Software for Multicore Architecture*, 2010.

[2] A. Aubum and V. McLean. Efficient exploitation of concurrency using graph decomposition. 1990.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[5] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, pages 555–566. Springer, 2011.

[6] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[7] Joao Vicente Ferreira Lima, Thierry Gautier, Nicolas Maillard, and Vincent 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)*, October 2012.

[8] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[9] Michael R Garey and Ronald L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.

[10] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[11] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. *IPDPS*, 2013.

[12] Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[13] Ronald L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[14] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[15] AA Khan, C.L. McCreary, and MS Jones. A comparison of multiprocessor scheduling heuristics. In *Parallel Processing, 1994. ICPP 1994. International Conference on*, volume 2, pages 243–250. IEEE, 1994.

[16] Renaud Lepère and Denis Trystram. A new clustering algorithm for large communication delays. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 68–73. IEEE, 2002.

[17] Ewing Lusk, S Huss, B Saphir, and M Snir. Mpi: A message-passing interface standard, 2009.

[18] C McCreary, J Thompson, H Gill, T Smith, and Y Zhu. Partitioning and scheduling using graph decomposition. *Department of Computer Science and Engineering CSE-93-06, Auburn University*, 1993.

[19] CL McCreary and A Reed. A graph parsing algorithm and implementation. *Tech. Rpt. TR-93-04, Dept. of Comp. Sci and Eng., Auburn U*, 1993.

[20] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

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

[22] Jeffrey D. Ullman. *NP*-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[23] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *Parallel and Distributed Systems, IEEE Transactions on*, 5(9):951–967, 1994.