Hierarchical Work-Stealing

Jean-Noel Quintin, Frédéric Wagner

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Hierarchical Work-Stealing

Jean-Noël Quintin — Frédéric Wagner

N° 7077 — version 1

initial version Octobre 2009 — revised version Novembre 2009

Thèmes COM et NUM

ISSN 0249-6399 ISRN INRIA/RR--7077--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Hierarchical Work-Stealing

Jean-Noël Quintin* †, Frédéric Wagner †

Thèmes COM et NUM — Systèmes communicants et Systèmes numériques
Équipe-Projet Moais

Rapport de recherche n° 7077 — version 1 — initial version Octobre 2009 —
revised version Novembre 2009 — 21 pages

Abstract:

In this paper, we study the problem of dynamic load-balancing on heterogeneous hierarchical platforms. In particular, we consider here applications involving heavy communications on a distributed platform. The work-stealing algorithm introduced by Blumofe and Leiserson is a commonly used technique to distribute load in a distributed environment but it suffers from poor performances in some cases of communications-intensive applications. We present here several variants of this algorithm found in the literature and different grid middlewares like *Satin* and *Kaapi*. In addition, we propose two new variations of the work-stealing algorithm : HWS and PWS. These algorithms improve performances by taking the networking structure into account within the scheduling. We conduct a theoretical analysis of HWS in the case of fork-join task graphs and present experimental results comparing the most relevant algorithms. Experiments on Grid'5000 show that HWS and PWS allow us to obtain performance gains of up to twenty per cent when compared to the standard algorithm. Moreover in some case, the standard algorithm reaches worse performances on the distributed platform than on a single machine while PWS and HWS achieve some speedup.

Key-words: Online scheduling, Work-stealing, Hierarchical platforms

* This work has been supported by the minalogic project CILOE
† INRIA Moais research team Laboratoire CNRS LIG Grenoble University France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Vol de travail hiérarchique

Résumé : Dans ce papier, nous étudions le problème d'équilibrage de charge dynamique sur plate-forme hétérogène et hiérarchique. En particulier, nous considérons ici des applications engendrant d'important transferts de données sur des plate-forme distribuées. L'algorithme de vol de travail introduit par Blumofe et Leiserson est une technique communément utilisée pour répartir la charge dans les environnements distribués. Mais cette méthode peut être inefficace dans le cas où les applications transfèrent beaucoup de données. Nous présentons ici plusieurs modifications des algorithmes de vol de travail existants dans la littérature et dans différentes bibliothèques comme *Satin* et *Kaapi*. De plus, nous proposons deux nouvelles variations du vol de travail : HWS et PWS. Ces algorithmes augmentent les performances en prenant en compte la hiérarchie du réseau pour ordonnancer les tâches. Nous réalisons une analyse théorique de HWS dans le cas des graphes de type "Fork-Join". Nous comparons les algorithmes proposés avec les algorithmes de la littérature proposant les meilleures performances. Ces expériences mettent en évidence que HWS et PWS nous permettent d'obtenir un gain de performances supérieur à vingt pour cent en comparaison avec l'algorithme de vol de travail classique.

Mots-clés : ordonnancent dynamique, vol de travail, plate-forme hiérarchique

1 Introduction

Nowadays, a great number of parallel libraries use work stealing as their scheduling engine. The work-stealing algorithm [2] is a distributed version of list scheduling allowing to achieve a good load-balancing on distributed platforms. However, it may suffer from communication times on applications transferring large amount of data or platforms with complex topologies.

The objective of our studies is to analyze the behavior of work-stealing on distributed architectures and to propose modifications allowing for greater locality of data.

Section 2 we present the classical work-stealing algorithm together with variants from the literature. We provide an analysis of the different possibilities of modification within the algorithm. Section 3 we propose two new algorithms HWS and PWS designed to reduce long distance communications. We present Section 4 a theoretical analysis of the HWS algorithm showing an efficient load-balancing and a reduced number of communications. Section 5 describes a set of experiments validating the new algorithms and comparing them with standard algorithms from the previous sections. Finally, we conclude Section 6 by summing up the obtained results.

2 Work-Stealing Algorithms

The work-stealing is one way to achieve an efficient dynamic load-balancing. This algorithm has many good assets :

- Scalable,
- Distributed algorithm,
- Theoretical execution time is bounded,
- Theoretical number of steal attempts is bounded.

In this Section, we detail the mechanism of work-stealing and assorted steal policy.

Section 2.1, we present the Classical Work-stealing algorithm. Section 2.2 details the work around *Satin* on hierarchical work-stealing algorithms [8]. Finally, the steal policy generally applied within the distributed middle-ware *Kaapi* [6] is detailed in Section 2.3.

2.1 Classical Work-Stealing

Several libraries like Cilk [4], TBB, *Satin* [9], *Kaapi* [5] ... suggest an implementation of the work-stealing algorithm. All of them rely on a common mechanism. The programmer describes the execution with a set of tasks. Each task can spawn/fork other tasks, and/or execute some instructions. Depending on the library the programmer has more or less work to realize a description of dependencies between tasks. The set of all tasks forms a DAG (directed acyclic graph) which is generated on-line during the execution. Since running on a distributed architecture the DAG is itself distributed : To store these tasks,

each processor store his part of the DAG in a stack of tasks. Since the DAG is discovered on-line, all processors can have various amount of tasks.

In function of this amount,two possible states for each processor could be distinguished :

Worker :

The processor has some work to do. Its stack may be empty or not. During the execution of a task, the worker creates some tasks. All tasks are pushed into the stack.

Thief :

The processor become a thief when it has no work to do. Therefore, its stack is empty. It tries to steal another one.

Usually, at beginning of the execution, one processor is a worker, and other processors are thieves.

During the execution, processors have to make choices depending on of their state. These choices lead to variations of the work-stealing algorithm. Each worker must choose which task will be executed. When a processor is or becomes a thief, it must choose which processor will be stolen and which task will be taken. In practice, these questions are the main issue as in this paper. In theory, there are some proved propositions to answer them.

How the processor chooses the next task in his stack

Blumofe and Leiserson [2] suggested to follow the sequential execution. This keeps optimizations done by the programmer. When a processor executes a task without stealing, the processor executes instructions in the same order as the sequential computation. To achieve this, we need to execute firstly the last task created.

How the processor chooses the stolen processor

In [2] the authors demonstrate how the random choice is fair. A large portion of the "steal-able" work, is stolen with a high probability, on a limited number of attempts, if the stolen processor is chosen randomly. Note that random choices present the advantage that the choice of the target does not require more information than the total number of processors in the executive platform. Thus, the random choice is the simplest choice with bounded performances.

How the thief chooses the stolen task

During a steal request, the thief should not slow down the worker because it could be working on the critical path. Therefore, any disruption might raise the execution time. Many papers in the literature explain how we can implement a lock free or wait free algorithm [7]. Furthermore, steal attempts are realized to balance the load and when the thief has no work. Consequently to obtain the same amount of work on the thief and the stolen processor, half of the work on the worker should be stolen. The task which contains very likely the largest amount of work, is the oldest task in the stack. To implement this, the thief chooses the task on top of the stack.

If the implementation of a work-stealing algorithm follows these choices, Blumofe and al [1] prove that the execution time and the number of steal attempts is bounded. We present here some notations which are used to conduct their theoretical analysis.

In this analysis, the application is modeled by a DAG of unit-sized tasks [2]. This DAG is characterized by $W_{(work)}(or\ T_1)$ the number of nodes and $D_{(depth)}(or\ T_\infty)$ the number of nodes on the critical path. Figure 1 illustrates these notations. The execution time is bounded by $\frac{W}{p} + O(D)$, where p is the number of processors. The number of steal requests during the execution is bounded by $O(p * D)$.

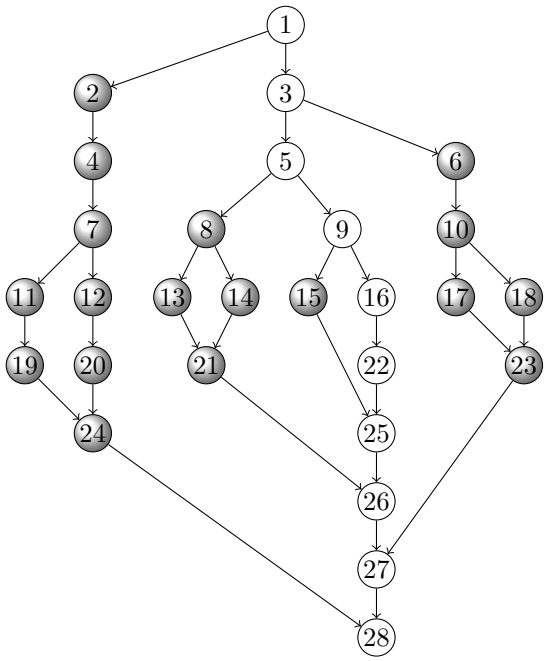

Figure 1: A representation of an application.

While these results are impressive, the model in use here might not be accurate enough for large scale distributed systems : the communication cost is not taken into account as, the possible saturation of network links. For this reason some hierarchical work-stealing algorithms were developed by Nieuwpoort and al [8].

2.2 Existing hierarchical work-stealing algorithms

Three hierarchical work-stealing heuristics [8] have been introduced in the library *Satin* [9]. These heuristics take into account the executive platform in different ways. These algorithms change only the choice of the stolen processor. The stolen task is still the oldest in the stack.

These heuristics are :

CHS :

Each cluster is represented by a tree. At time of steal attempts, the thief asks its children. In the case of setback on the whole sub tree, the request is transferred to its father. This heuristic tries to favor the locality of the work and decrease the number of remote steal requests.

CLS :

The platform is divided in several clusters. On each cluster, a leader is elected. Each computer sends information about the amount of work on itself, to its leader. Each leader takes into account this information to decide to steal a computer in another cluster. This heuristic decreases the number of data transfers between all clusters.

CRS :

Computers perceive two levels of hierarchy: processors in the same cluster, and others. Each computer is able to realize two types of steal attempts : asynchronous and synchronous. Asynchronous steal requests are restricted to one computer in other clusters at the same time. On the opposite, synchronous steal attempts are restricted to the computer in the same cluster. Steal attempts are realized only when the stack is empty, like in the classical policy.

In [8], Nieuwpoort and al. compare these heuristics on several applications. On most applications, CRS yields the best results. Moreover, this heuristic present the advantage that it does not require tunable parameters.

Experiments which are realized with *Satin* in [10], show that the random policy has not the best performances. If an algorithm takes into consideration the hierarchy of the platform, it could take the advantage over the classical heuristics. While CRS presents an increase in the performances, it suffers from several problems :

- No theoretical analysis,
- Some tasks might be transferred several time in advance for nothing.

Still since CRS outperforms other heuristics, we will not consider CLS and CHS in the remainder of this paper.

2.3 The *Kaapi* library

Kaapi is a middle-ware developed by our research team at INRIA, which implements a distributed work-stealing algorithm. The first part of this section deals with the work-stealing algorithm of *Kaapi*. Then, details on the data transfer and on the description of the DAG are described.

In the *Kaapi* library, a short description of the hierarchy is known. Each processor knows two levels of hierarchy :

- Processors on the same computer. Their stacks could be accessed directly concurrently.
- Other computers. A network communication is required to steal them.

The thief can choose to steal a processor on the same computer, or another computer. When a computer receives a steal request, it must choose how many processors of his computer will be stolen and which ones. In *Kaapi*, the processor tries at first to steal one processor on the same computer, then tries to steal another computer. The remote stolen computer chooses to steal randomly one processor. If the steal request failed, the thief restarts this algorithm.

This algorithm could easily fit to several levels of hierarchy. While it is not studied theoretically, this algorithm outperforms standard work-stealing algorithm in practical cases.

In this paper all experiments are going to be realized with *Kaapi*. We therefore give more details about the implementation of *Kaapi*.

When the programmer implements a program for *Kaapi*, he describes his application as a set of different tasks. Each task to be executed requires to have access to some read and written data. During a steal attempt, read data are transferred before the beginning of the task. At the end of the stolen task, write data are sent directly to the stolen processor. Thus, steal attempts lead to some data transfers. The data transfers are realized by only one thread on each computer.

When the programmer describes the application, he gives a recursive cutting of the work. The main issue associated to this, is to choose when the work could be executed sequentially. This choice is important for the execution time, it change the amount of work and the critical path. Thus, it is an important parameter. In our experiments, we have done our choice to obtain a few extra cost compared to the sequential time with the larger number of tasks.

3 Proposed algorithms

In previous section, we presented an overview of different work-stealing algorithms from the literature. Each of them has some interesting characteristics :

- The classical one benefits, from a theoretical analysis.
- CRS has good performances in practice on hierarchical platforms.
- *Kaapi* introduces its own algorithm which also outperforms the classical work-stealing algorithm [6].

Each algorithm has however some weaknesses :

- The classical is outperformed in practice.
- CRS and *Kaapi* work-stealing has also no theoretical analysis.

In addition, we believe load-balancing algorithms can benefit from the use of knowledge of the platform. We therefore introduce two new variations of the work-stealing algorithm.

The first one, “PWS” (Probability work-stealing) is a simple algorithm which takes into account several levels of the hierarchy. The second “HWS” (Hierarchical work-stealing) takes into account the hierarchy of the platform and knowledge of the application.

3.1 Probability Heuristics : PWS

In PWS, we suggest to reduce the time spent to steal by stealing in priority close processors. We require a description of the hierarchy to estimate the distance between the thief and the stolen processor.

We then apply the standard work-stealing algorithm with the following modifications : the probability to choose a target machine for steal attempts is not uniform anymore but instead inversely proportional to the distance between the stealer and the target.

This strategy presents the advantage to increase the locality of data transfers while reducing the latency of steal requests.

3.2 HWS

We also propose an algorithm based on a completely different approach : HWS. HWS has been created after observing the behavior of the classical work-stealing algorithm on several applications.

The work-stealing algorithm balance the load between the thief and the stolen processor. We want to realize this mechanic at the cluster level. Thus, we suggest to steal on one steal attempt half the amount of work in the target cluster.

To realize this HWS changes the choice of the stolen task along with the selection of the stolen processor. To change the stolen task we require some information about the application. The modification of the stolen processor takes advantage of our knowledge of the platform. Thus, HWS takes advantage of more information than classical algorithms.

We aim here to reduce the amount of transferred data on slow links which are shared with other users. Processors connected with a fast link, are "brought" together. We call this a processors group. For example, it could be a cluster, the set of cores in one processor... The risk of congestion between groups, arises with the amount of transferred data. To limit this risk, we have chosen to limit in each group, the number of processors which could steal another group. In each group, only one processor can realize remote steal requests in HWS. We call this processor the group leader.

This modification may however have a strong impact on load-balancing. Since the number of remote steal requests is decreased, we want remote thieves to steal a larger amount of work. Therefore, we suggest to restrict steal attempts between groups to the largest tasks. As seen in section 2.1, in most applications tasks close to the root node of the dag usually contains a large amount of work while this amount decreases as the recursion develops.

For this reason, we set up a limit to distinguish tasks. The set of tasks is represented by a dag. The level of a task in a dag could have several values. We therefore define the level of a task in function of the fork tree : the level of a task is defined as its number of parent tasks up to the root task. We need here to modify the middle-ware to store with each task its current level. When new tasks are created they define their level as the one of their parent augmented by one.

We define two types of tasks :

Global tasks :

Tasks above the limit on the forking tree which can be stolen between groups.

Local tasks :

Tasks under the limit, which stay inside one group.

We artificially limit the number of global tasks in any application. For example an application which divides the work in halves like in some divide and conquer problems has 2^l global tasks where l is the chosen limit. To avoid a huge number of global task l is chosen small. Global tasks are centralized on the leader.

To sum up the situation we have two types of processors :

Leaders :

which balance the load between groups and manage the load inside their groups.

Slaves :

which execute the work provided by their leader. They can only steal inside their groups.

To balance the load between groups and leaders, each leader has two stacks : the global stack for global tasks accessed by leaders, the local stack for local task accessed by slaves of the group.

Algorithm executed by leaders

At beginning of the execution, all tasks are on leaders. Leaders which have some tasks, execute them. When a task is created, the leader chooses to push the task in the local stack or the global task. This decision is taken in function of the task depth in the fork tree. The whole tasks of the sub fork tree is called a block of tasks. Figure 2 shows the set of blocks for an application, the limit being chosen equal to three.

The leader provides some work to its group by pushing a local task in its own local stack. In practice, we suggest to detect the amount of work inside the group. If this amount is insufficient to exploit the whole power of the group, the leader provides another local task to its group by executing a global task or stealing a global task. The amount of work inside the group can be evaluated as proposed in the CHS algorithm [8] by sending additional messages. However The amount of messages transferred inside the group by this detection method is large. We propose to avoid these messages : In practice the leader can detect a lack of work by evaluating the number of steal attempts he receives.

4 Theoretical analysis

We provide here a theoretical analysis of the HWS algorithm presented in previous section.

Section 4.1, we show a limit on the execution time and the number of steal requests of HWS. Section 4.2, we analyze in detail the obtained results.

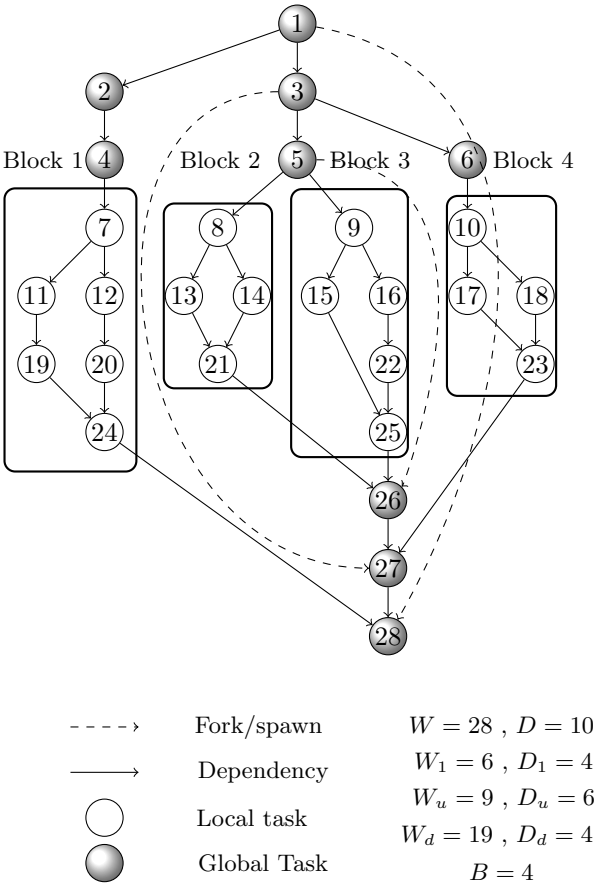

Figure 2: Representation of the work done by slaves and the leaders.

4.1 Analysis of HWS

During the presentation of HWS, several implementations are proposed to evaluate the amount of work in groups. This evaluation is difficult to model and we propose to restrict ourselves in our analysis to a simpler version of HWS.

In this analysis, we limit the leader to wait for the end of the provided block. Note that this restriction is degrading the performances of the practical version of HWS by considering worse case waiting times.

During this waiting time, the slaves execute all tasks of the block. Under this assumption block execution then behaves like classical work-stealing.

For this analysis, the waiting leaders could introduce some dead-locks in case of dependencies between different blocks. Therefore, we restrict our proof to applications without such dependencies. All "Fork-Join" applications respect this constrain.

The proof is divided in several parts. First, we explicit a model of the application, introducing some additional notations. Then, we begin by limiting the execution time of each block by a group of slaves. We then conclude by completing a global analysis.

Modeling the application

For our analysis we explicit the dependency dag G representing the application. The edges of G describe the dependency between tasks, for example the forking of a task or precedence with another task. On G we use the classical notations presented Section 2.1. We recall that for the classical work- stealing Blumofe and Leiserson proved that :

- Execution time is less than : $\frac{W}{p} + O(D)$
- Number of steal requests is limited by : $O(p * D)$

In HWS we use a limit l to separate global tasks and local tasks. Local tasks are packed into blocks as shown Figure 2. We introduce some additional notations :

$W_{g(lobal)}$ **:**
Number of global tasks. It is also the amount of work done by all leaders.

$D_{g(lobal)}$ **:**
Length of the critical path of the dag G without local tasks.

$W_{l(ocal)}$ **:**
Number of local tasks. It is also the amount of work executed by slaves.

$D_{l(ocal)}$ **:**
Length of the critical path of the dag G without global tasks.

W_i **:** Number of tasks inside the block i.

D_i **:** Length of the critical path of the dag representing the block i.

B **:** Number of blocks.

To understand exactly the signification of additional notations, Figure 2 illustrates an application with them. By definition of blocks, we have :

$$W_l = \sum_i W_i \tag{1}$$

$$D_l = \max_i D_i \tag{2}$$

We also introduce some notations to characterize the platform : The platform has p processors which are hierarchically organized. As described Section 3, there are some groups of processors. For this proof, we limit groups to the same number of processors (homogeneous case). We define p_g the number of processors in groups and g the number of groups. We hope to remove this limitation in future work in order to take into account heterogeneous groups.

$$\forall k : p_k = cste = p_g \tag{3}$$

$$g = \frac{p}{p_g} \tag{4}$$

Theoretical Proof

Theorem 1.
The execution time of an application with the HWS algorithm is less than

$$\frac{W_g}{g} + \frac{W_l}{p} + O(D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$$

In addition, the number of steal attempts between groups is fewer than

$$O(g * (D + \max_i(\frac{W_i}{p_g})))$$

Proof. We begin by studying the execution of blocks on a group. Each block is executed by one group only. Moreover only one block can be executed at the same time because the leader is waiting for the end of the block. Thus, the execution time of the block is limited by the bound of the classical work stealing algorithm. Here, the execution time of the block i is lower than $\frac{W_i}{p_g} + O(D_i)$. Therefore the leader of the group waits at most $\frac{W_i}{p_g} + O(D_i)$.

To continue with the global analysis, we are going to model the waiting times of the leaders. To achieve this, we build a new graph G' modeling the different activities of leader nodes. We replace the block executed by a group with a chain of tasks inducing the same waiting time for the leader. When the leader starts this chain with the work-stealing algorithm, nobody can steal the remaining of the chain.

We define G' as follows : Each global task of G is copied identically inside G'. Each block i of G is changed in a chain of tasks. The length of each chain is equal to $\frac{W_i}{p_g} + O(D_i)$ for the block i. Figure 3 presents the dag G' for an application.

Since the length of chain is an upper bound of execution time, the execution time of G' by leaders is greater than the execution of G with HWS on the whole platform. We analyze the new dag G' to bound its execution time by leaders. Characteristics of G' are :

$$\begin{aligned} D' &= D'_g + D'_l \\ &= D_g + \max_i (O(D_i) + \frac{W_i}{p_g}) \\ &\Rightarrow D' \leq O(D) + \max_i \frac{W_i}{p_g} \end{aligned} \tag{5}$$

$$\begin{aligned} W' = W'_g + W'_l &= W_g + \sum_i (\frac{W_i}{p_g} + O(D_i)) \\ &\leq W_g + \frac{W_l}{p_g} + O(B * D_l) \\ &\Rightarrow W' \leq W_g + \frac{W_l}{p_g} + O(B * D_l) \end{aligned} \tag{6}$$

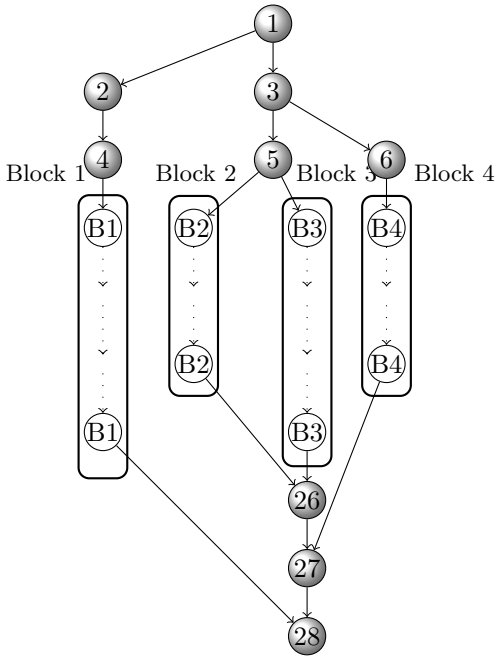

Figure 3: A representation of the graph executed by leaders

Execution time of G is less than the execution time of G' by leaders with the classical work-stealing algorithm. Thus, we can limit the execution time by : $\frac{W_g}{g} + \frac{W_l}{p} + O(D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$. In addition, the number of steal requests is lower than $O(g * (D + \max_i \frac{W_i}{p_g}))$.

4.2 Interpretation of results

In previous section we proved that the execution time of HWS is bounded by $\frac{W_g}{g} + \frac{W_l}{p} + O(D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$. This bound is rather complex and we might wonder if this expression reflects reality or if it is just a mathematical artefact. We therefore propose to study the signification of each of the terms of this expression :

$\frac{W_g}{g}$ **:**

The work above the limit is executed by leaders. HWS implies this expression because of its design. Each global task can only be executed by leaders. If the limit is chosen small enough, this expression is not significant when compared to others.

$\frac{W_l}{p}$ **:**

This shows that local tasks are balanced efficiently between slaves. Thus, most of the work is balanced on the whole platform.

D **:**

Each tasks on the critical path have to be executed one after one. This term comes directly from the bound of classical work-stealing algorithm.

$\max_i \frac{W_i}{p_g}$ **:**

This expression derives from the main constraint of HWS: each block is restricted to be executed by only one group. This constraint is imposed by HWS to limit communications. Thus the execution time cannot be lower than the time of the execution of the largest block on one group. This term is sizable if the amount of work is too imbalanced between each blocks. In practice with "fork-join" applications, this problem rarely occurs.

$D\frac{B}{g}$ **:**

In the demonstration, we can understand the origin of this term. It derives from the fact that all blocks are executed one by one. Since blocks are executed sequentially, their critical paths may add up during the execution. Note that this term comes from the waiting constraint artificially added to the leaders and might to be insignificant in practice.

To conclude this analysis, we highlight several of the obtained results. In this proof we show that the work is efficiently balanced under some conditions : the limit is chosen small and the work is not too imbalanced between the different blocks. What is more, the number of global steal attempts is limited by $O(g * (D + \max_i \frac{W_i}{p_g}))$ which fulfills the main goal of the HWS algorithm. We therefore expect the experimental validation of HWS to show an increase in performances over the standard work-stealing algorithm.

5 Experimental validation

We compare here PWS and HWS with standard algorithms from the literature : the classical work-stealing (denoted by WS), *Kaapi*work-stealing and CRS. These experiments stand as a practical test of our algorithms and enable us to explicit in particular cases the amount of time we can win over standard algorithms.

Section5.1, we describe the main characteristics of the experimental platform and we present the applications chosen to compare the heuristics. Then Section5.2 presents results from the experiments.

5.1 Experimental Setup

Our experiments are designed not only to compare existing heuristics to PWS and HWS. We wish also to quantify the impact of the network on different kind of applications. We therefore propose to compare the execution time on a distributed platform to the execution time on a similar platform with shared memory.

In our experiments, we use the Grid'5000 [3] platform. On this platform, the largest shared memory computer has two 2.5 GHz Intel Xeon E5420 of four cores each. We choose here to compare execution on a single node with eight cores to the execution on two nodes with four cores each. The different machines are here connected by an infiniBand network.

We propose to compare here all scheduling algorithms using two different applications. The first one is an implementation of the merge sort algorithm. We sort here an array of four Gigabytes of data. This algorithm is chosen because it induces a large amount of communication.

The second application proposed here solves the Nqueen problem. This application spawns a large number of tasks but each task transfers only a little amount of data. Thus we hope to estimate the additional costs of the scheduling algorithms.

5.2 Experiments

5.2.1 Merge Sort

This section deals with the execution time of each work-stealing algorithm to sort an array of four Gigabytes on a distributed platform. This implementation of the merge sort algorithm uses a recursive splitting in two by spawning two tasks each sorting half of the data. The fusion of each half of the array is done in place with the algorithm of the stl library. When the size of the array is under four Kilobytes, the work is executed in sequential with the merge sort of the standard library stl. We evaluated the overhead of tasks creations by comparing a pure stl sequential execution with an execution of the parallel code on one core. The time of execution of the stl merge sort is equal to one hundred and four seconds. The same execution with parallel version sees an execution time equal to one hundred and seven seconds. The extra cost of tasks creations is therefore less than three per cent.

For each parameters of the executions, we ran one hundred experiments. The error bars on all figures represent the confidence interval. Note that here, more than ninety five per cent of experiments have an execution time within the confidence interval.

We start by comparing the performances of CRS and the classical work-stealing algorithm on two nodes with one processor used on each node. We compare the result obtained with the performances of the classical work-stealing on one processor only. Figure 4 shows that CRS and WS achieve a slow down in more than half of the executions when compared with the one processor setup. We have also monitored the amount of data transferred during the executions, which is around of :

- Ten Gigabytes for CRS,
- Eight Gigabytes for the classical work stealing.

These amounts exceed the actual data size because data is being sent back and forth during the execution.

This result is not in contradiction with the result in [8]. In this paper CRS is shown to achieve bad performances in several experiments. For example CRS performed weakly on matrix multiplications. We believe this result is similar

to the results we obtain and is due to the high amount of data in use in both setups.

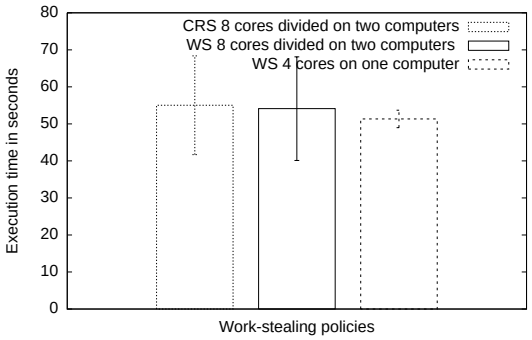

Figure 4:

We then compare the *Kaapi* work stealing with the performances of the classical work-stealing algorithm. Figure 5 shows that *Kaapi* achieves better results with an average 10 per cent execution time improvement over WS. *Kaapi* exhibits a small speedup in more than 60 per cent of all executions over WS on one processor. The amount of data transferred for *Kaapi* varies greatly between the different executions : from four to eight Gigabytes.

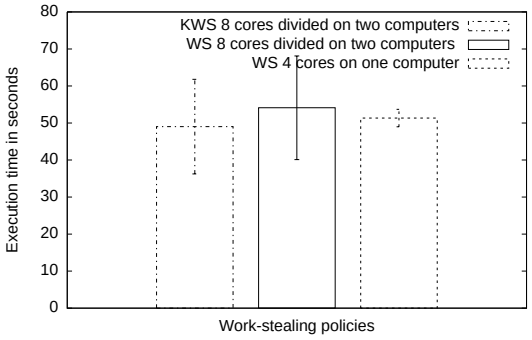

Figure 5:

To continue, we present experiments with PWS and HWS. PWS has a tunable parameter which is the probability to steal a processor in another cluster.

Figure 6 shows the evolution of the execution time in function of this probability together with the execution time of HWS. This figure also shows the execution of WS for comparison purposes.

We first analyze the behavior of PWS.

If the probability is chosen less than 5 per cent, PWS achieves a speedup of 20 per cent on all executions compared to the classical work-stealing algorithm on one processor. For greater values of this probability the behavior of PWS becomes similar to the one of WS.

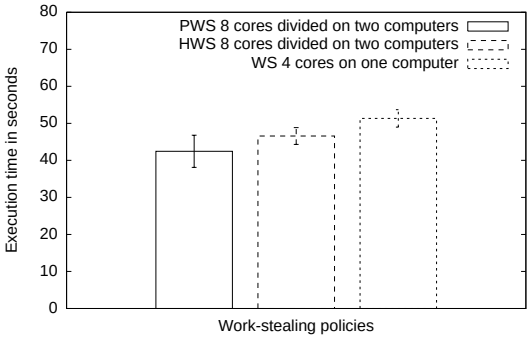

Figure 6:

We also compare HWS with PWS and the execution of WS on one processor. For HWS, the limit l has been chosen at two. We have not changed this limit because of an issue in the implementation of *Kaapi* with HWS for the transfer of data. We believe nonetheless that the obtained results are pertinent enough to be presented here. We affix the probability of PWS to the best possible value. Figure 6 shows the obtained execution time. We can see that HWS achieves a speedup on all experiments compared to WS on one processor. PWS achieves a greater speedup in most of cases if the parameter is well-chosen. We can see in Figure 7 that the probability to steal a remote processor must be smaller than 0.2.

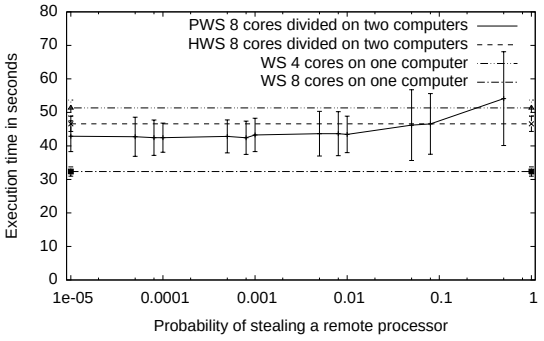

Figure 7:

In these experiments the amount of data transferred is :

- around four Gigabytes for HWS.
- under four Gigabytes for PWS.

The time needed to transfer four Gigabytes (thirty two Gigabits) is around forty seconds within *Kaapi*. This time takes into account the cost of using sockets over IP on infiniBand and the cost of send operation inside *Kaapi*. On Figure 7 the differences of the execution times of both PWS and HWS on two nodes

and WS with two processors on one node is less than these forty seconds. We therefore conclude that achieving better execution times can only be achieved by covering communications with computations.

5.2.2 Nqueen

In our second set of experiments we use a recursive algorithm to solve the Nqueens problem. This algorithm has already been tested within *Kaapi*at large scale during the third and fourth Plugtest Contest. The program spawns a task for each possible position of the queen on the first line. Then with the choice of the first queen, the program spawns a task for each possible position of the queen on the second line and so on. When the level of recursion is more than four, the work is executed with the same algorithm without spawning tasks. The program gives the number of possibility to solve the problem.

In our experiments we solve the problem for eighteen queens. Figure 8 shows the execution time of all scheduling algorithms considered executed on two nodes and the time of WS one node with one or two processors. We can see that all strategies show a roughly equivalent execution time. We conclude from these experiments that while hierarchical scheduling algorithms do not reach greater speedups when the amount of data to communicate is low they also do not decrease the execution times. The implementation of the different algorithms bear therefore no great costs.

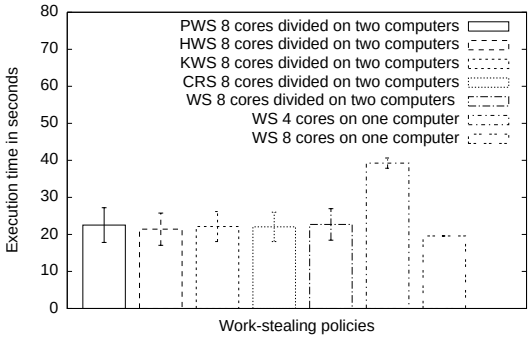

Figure 8:

We also study the impact of the limit chosen within HWS. Figure 9 show the evolution of the execution time in function of the limit value. We can see the execution degrades with a limit value increasing above two. This is because the sequential execution begins starting from the fourth level of the recursion. With a limit of three not enough tasks are generated in only one level of recursion to take advantage of all available cores.

Our experiments also show that the probability to steal a remote processor can have an influence on the scheduling of the tasks. Figure 10 shows that the execution time grows on the Nqueen problem if the probability to steal a remote processor is under of 5×10^{-4}. Nonetheless the execution time of PWS is pretty stable for a large scope of probabilities.

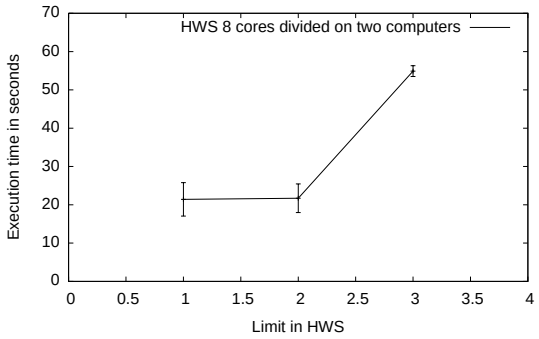

Figure 9:

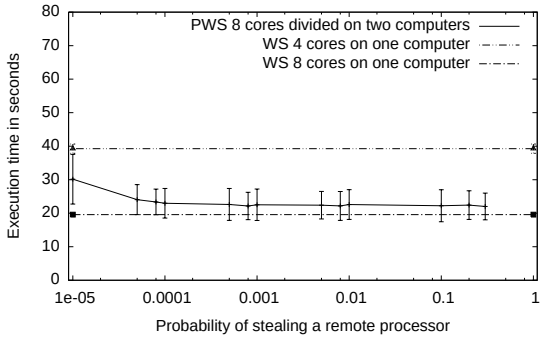

Figure 10:

5.2.3 Conclusion

To conclude these experiments, we draw attention to the main results observed.

First of all, we have been achieving speedup with both PWS and HWS on experiments with large data transfers. With an improvement reaching 20 per cent of the execution time over standard algorithms, a distributed execution becomes a viable option achieving speedup in all executions while classical algorithms increase execution time when using remote resources. PWS and HWS divide the amount of data transferred by two. Both algorithms show a very weak overhead over the classical work stealing algorithm.

PWS achieves the best performances and the probability parameter has been shown to accept a very wide range of values all giving the best performances.

6 Conclusion

This paper has been studying different work-stealing algorithms for use on hierarchical platforms. We presented a survey on the various algorithms available in the literature and introduced two new algorithms : PWS and HWS. We provided a theoretical analysis of HWS and showed that under reasonable as-

sumptions it would exhibit an efficient load balancing while greatly reducing the number of remote communications. PWS and HWS have then been validated experimentally on two different kind of applications. Both of them show an increase in performances strong enough to justify for a distributed execution, with PWS outperforming HWS.

These works may be extended in several directions. First, on the theoretical side, we would like to perform an analysis of PWS and remove the actual constraints on the analysis of HWS. On a practical side we hope to conduct experiments at larger scales with several levels of hierarchy.

We believe that while our work is not the optimal solution for many distributed applications (which could take advantage of more standard techniques like partitioning and static scheduling) it can increase the reach of existing applications which rely on work stealing middle-wares by enabling them to run on a wider range of platforms.

References

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *IN PROCEEDINGS OF THE TENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES (SPAA), PUERTO VALLARTA*, pages 119–129, 1998.

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

[3] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages 99–106, Seattle, USA, November 13-14 2005. IEEE/ACM.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Langua ge Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, june 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[5] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23, New York, NY, USA, 2007. ACM.

[6] Thierry Gautier, Jean-Louis Roch, and Frédéric Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part II*, pages 593–600, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] John D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.

[8] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer application s. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM.

[9] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient Java-based grid programming. In *AGridM Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, USA, September 2003.

[10] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Thilo Kielmann, and Henri E. Bal. Adaptive load balancing for divide-and-conquer grid applications. *Journal of Supercomputing*, 2006.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399