Hierarchical Work-Stealing

Jean-Noël Quintin[⋆] and Frédéric Wagner

INRIA Moais research team, CNRS LIG Labs, Grenoble University, France
`jeannoel.quintin@imag.fr` and `Frederic.wagner@imag.fr`

Abstract. We study the problem of dynamic load-balancing on hierarchical platforms. In particular, we consider applications involving heavy communications on a distributed platform. The work-stealing algorithm introduced by Blumofe and Leiserson is a commonly used technique to balance load in a distributed environment but it suffers from poor performance with some communication-intensive applications. We describe here several variants of this algorithm found in the literature and in different grid middle-wares like *Satin* and *Kaapi*. In addition, we propose two new variations of the work-stealing algorithm : HWS and PWS. These algorithms improve performance by considering the network structure. We conduct a theoretical analysis of HWS in the case of fork-join task graphs and prove that HWS reduces communication overhead. In addition, we 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 classical work-stealing algorithm. Moreover in some cases, PWS and HWS achieve speedup while classical work-stealing policies result in speed-down.

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

1 Introduction

With the recent increase of interest in parallel programming, the number of parallel libraries using work stealing as their scheduling engine raises. The work-stealing algorithm [2] is a distributed version of list scheduling achieving a good load-balancing on distributed platforms. However, it may suffer from high communication costs for applications transferring large amounts of data or platforms with complex network topologies.

In this paper we propose to improve the performance of work stealing by taking the data locality into account.

Section 2 presents the standard work-stealing algorithm together with variants from the literature. We provide a detailed description of the work-stealing algorithm and emphasize different implementation choices leading to different variants of the algorithm. In Section 3, we introduce two new algorithms HWS

[⋆] This work has been supported by the minalogic project CILOE and PWS, designed to reduce long distance communications. We present in 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 some existing policies. Finally, we conclude in Section 6.

2 Work-Stealing Algorithms

Work stealing is a scheduling algorithm which achieves an efficient dynamic load-balancing. This algorithm has many good assets:

- it is a scalable distributed algorithm,
- the execution time is theoretically bounded,
- the number of steal attempts is theoretically bounded.

The classical work-stealing algorithm is presented in Section 2.1. The work around the library *Satin* [9] on hierarchical work-stealing algorithms [8] is detailed in Section 2.2. Finally, the steal policy commonly applied within the distributed middle-ware *Kaapi* [5] is described in Section 2.3.

2.1 Classical Work-Stealing

Several libraries and languages like Cilk [3], TBB [11], *Satin* [9], *Kaapi* [4], Charm++ [6], X10 [7] implement in their own way a work-stealing algorithm. In each of them, the execution is described with a set of tasks. Each task can spawn/fork other tasks, and/or execute some instructions. Depending on the library, the programming work to describe dependencies between tasks can be more or less tedious.

The set of all tasks forms a DAG (directed acyclic graph), which is generated online during the execution. Since tasks are running on a distributed architecture, the DAG is itself distributed: each processor stores its part of the DAG in a stack of tasks. As the DAG is discovered online, the work is unbalanced between processors. Each processor is in one of two possible states depending on the amount of tasks in its stack:

A **worker** is a processor which has some work to do. Its stack may be empty or not. During the execution of a task, the worker creates some tasks. All such tasks are pushed onto the stack.

A **thief** is a processor which has no work to do. It therefore tries to steal work from another one (the victim).

Usually when the execution starts one of the processor is a worker while the others are thieves. During the execution, processors have to make choices depending on their state. Each worker must choose which task will be executed. When a processor is a thief, it must choose which processor will be the victim and which task will be stolen. The performance of the work-stealing algorithm depends heavily on these choices. The theoretical bounds of Blumofe and Leiserson [2] are valid for particular choices.

How does a processor choose the next task in its stack? Blumofe et al. [2] suggested that on each processor the parallel schedule executes instructions in the same order as in the sequential execution. This choice preserves sequential optimizations designed by the programmer. Thus when a processor executes a task without stealing, the processor performs instructions in the same order as in the sequential computation. To achieve this, it needs to execute the latest task created.

How does an idle processor choose which processor to steal? In [2] the authors prove that random choice is efficient. Within a bounded number of attempts workers receive a steal attempt with a high probability. The main advantage is that the victim choice does not require more information than the total number of processors. Thus, the random choice is the simplest choice which leads to theoretical bounds.

How does the thief choose the stolen task? During a steal request, any disruption on the worker might raise the execution time. Many papers in the literature as [12,13] explain how a lock/wait free algorithm can be implemented to limit overheads. As tasks are forked recursively, tasks close to the root of the DAG potentially contain more work than tasks forked farther away. Using the above guidelines, the oldest task which is on the top of the stack, is chosen.

Blumofe et al. [1] prove under these choices that the execution time and the number of steal attempts is bounded. We introduce here some notation, 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 $W(or\ T_1)$ the number of nodes and $D(or\ T_\infty)$ the number of nodes on the critical path. With a high probability the execution time is bounded by $\frac{W}{p} + O(D)$, where p is the number of processors in [1]. The number of steal requests during the execution is bounded by $O(p * D)$.

While these results are impressive, the model in use here might not be accurate enough for large-scale distributed systems since the communication cost is not taken into account. The time of each steal attempt is theoretically bounded by the same constant although the cost of communications is not uniform on the platform. For this reason hierarchical work-stealing algorithms were introduced by Nieuwpoort et al. [8].

2.2 Existing hierarchical work-stealing algorithms

Three hierarchical work-stealing heuristics [8] called CHS, CLS, CRS have been introduced in the library *Satin* [9]. These heuristics take into account the execution platform in different ways but only change the victim choice, the stolen task remaining the oldest one in the stack. In [8], Nieuwpoort et al. compare these heuristics on several applications. Since CRS largely outperforms CLS and CHS on most application, we will not consider CLS and CHS in the remainder of this paper.

In **CRS**, computers perceive two levels of hierarchy : processors in the same cluster and others. Each computer is able to send two types of steal attempts : asynchronous and synchronous. Asynchronous steal requests are restricted to one computer in other clusters at the same time. Conversely, synchronous steal attempts are restricted to computers in the same cluster. Steal attempts are sent only when the stack is empty, like in the classical policy. Each processor is able to send at any time one asynchronous steal and one synchronous steal.

Experiments using the *Satin* library show in [10] that the classical policy can be outperformed. An algorithm which considers the hierarchy of the platform, can take the advantage over existing heuristics. While CRS presents a performance improvement, it can be further improved since:

- No theoretical analysis is provided,
- Tasks might be transferred several times in advance for nothing.

To illustrate this last point, consider a processor working on a task received by a synchronous request. While working, it receives a task from a previous asynchronous request. This task can eventually be stolen once more before being consumed by the ongoing processor.

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 *Kaapi* work-stealing policy takes advantage of a short description of the hierarchy. Each processor perceives two levels of hierarchy :

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

The thief can choose to send a request to a processor on the same computer, or another one. In the *Kaapi* work-stealing algorithm a processor becoming a thief first sends a steal request to a processor on the same computer, then sends a steal attempt to a processor on another computer. If the steal request fails, 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 many practical cases.

In this paper, the implementation of work-stealing policies is done with the *Kaapi* library. We have implemented the classical work-stealing algorithm and the CRS algorithm to compare them with our policies.

3 Proposed algorithms

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

- The classical algorithm benefits from a theoretical analysis.
- CRS has experimentally good performances on hierarchical platforms.
- *Kaapi* introduces its own algorithm, which also outperforms the classical work-stealing algorithm [5].

Each algorithm has however some weaknesses.

- The classical one is experimentally outperformed by CRS and *Kaapi* policies.
- However CRS and *Kaapi* work-stealing have not been theoretically studied.

In addition, we believe that load-balancing algorithms can improve performance by using some 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 considers several levels of hierarchy. The second “HWS” (Hierarchical work-stealing) takes into account the hierarchy of the platform and the knowledge of the application.

3.1 Probability Heuristics : PWS

In PWS, we suggest reducing the steal time by picking up nearby processors in priority. This requires a description of the hierarchy to estimate the distance between the thief and the target processor.

We then apply the classical work-stealing algorithm with following changes: the probability to choose a target computer for steal attempts is not uniform anymore but instead proportional to the inverse of the distance between the thief and the target processor. This strategy has the advantage of increasing the data locality and of reducing the average latency of steal requests.

3.2 HWS

We also introduce a second algorithm called HWS based on a completely different approach. HWS has been designed after observing the behavior of the classical work-stealing algorithm on several applications. The work-stealing algorithm balances the load between the thief and the target processor. The main idea behind HWS is to use the same mechanism at the cluster level. Thus, we suggest stealing in a single attempt, a large amount of work inside the target cluster.

To this end, HWS changes the choice of the stolen task as well as the target processor. HWS needs information about the platform. The platform is divided in some processor groups which are sets of processors connected with a fast link. For example, it could be a cluster or 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 chose to restrict in each group, the number of processors which can steal another group. In each group, only one processor sends remote steal requests in HWS. This processor is called a leader.

This change 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, each leader gives some work to its cluster when there is not enough work, and keeps the large tasks to balance efficiently the load between leaders. As seen in Section 2.1, in most applications, tasks close to the root node of the DAG usually contain a large amount of work while this amount decreases as the recursion develops.

To distinguish tasks with a large amount of work, a limit is given by the user. The level of a task is determined according to the fork tree : the level of a task is equal to the number of its parent tasks up to the root task. When new tasks are created, their level is their parent level incremented by one. Two types of tasks are defined :

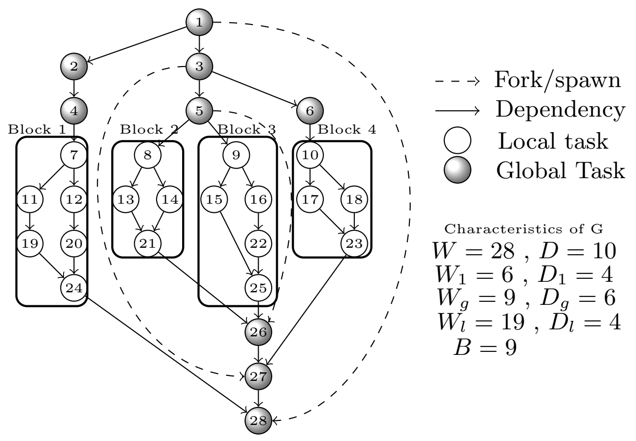


Fig. 1. Representation of the work done by slaves and leaders.

Global tasks have a level below the limit. They can be stolen between groups.
Local tasks have a level higher than the limit. They belong to a single group.

We artificially limit the number of global tasks in any application. For example an application recursively dividing 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 tasks, l is chosen small. Global tasks are centralized on the leader. In summary we have two types of processors :

Leaders : they execute only global tasks. And they balance the load between groups and manage the load inside their groups.
Slaves : they perform the classical work-stealing algorithm within their group.

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 tasks accessed by slaves from its group.

Leader algorithm : When the execution starts, all tasks are located on leaders stacks. Leaders which have some tasks execute them. When a task is created, the leader can choose to push the task in the local stack or in the global task. This decision depends on the task depth in the fork tree. Task pushed in the local stack by a leader is called a slave task. All tasks belonging to the fork subtree of a single slave task is called a block of tasks. Figure 1 shows the set of blocks for an application, the limit l being set arbitrarily to three.

The leader provides some work to its group by pushing a local task in its own local stack. We suggest detecting the amount of work inside the group. If this amount is not sufficient to exploit the group processing power, 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 estimated as proposed in the CHS algorithm [8] by sending additional messages. Another solution that can avoid such extra messages is for the leader to detect a lack of work by evaluating the number of steal attempts it receives. We chose to implement this latest option for this study.

4 Theoretical analysis

We provide here a theoretical analysis of the HWS algorithm. Section 4.1 shows that the execution time and the number of steal requests of HWS are bounded. Section 4.2 analyzes in detail the obtained results.

4.1 HWS analysis

In this analysis, we restrict ourselves to a simpler variant of HWS where each leader waits for the end of each provided block. Note that this restriction increases the waiting time of the implemented version of HWS by considering the worst case. During this waiting time, all slaves execute block tasks. For this analysis, the waiting leaders could introduce some deadlocks in the case of dependencies between different blocks. Therefore, we restrict our proof to applications without such dependencies. All "Fork-Join" applications satisfy this constraint.

The proof is divided in several parts. First, we present a model of the application. Then we begin by bounding the execution time of each block in a single group of slaves. We then conclude by completing a global analysis.

Modeling the application: for our analysis we introduce the dependency DAG G representing the application presented Figure 1. The edges of G describe the dependencies between tasks, for example the forking of a task or precedence with another task. On G we use classical notation presented in Section 2.1. We recall that for the classical work stealing, Blumofe and Leiserson proved that :

- The execution time is less than : $\frac{W}{p} + O(D)$
- The number of steal requests is bounded by : $O(p * D)$

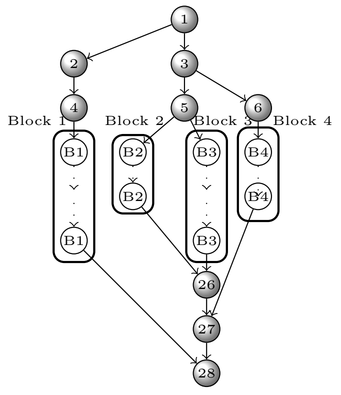
In HWS we use a limit l to separate global tasks and local tasks. Local tasks are packed into blocks as shown in Figure 1. We introduce some additional notation :

W_g	:	Number of global tasks and amount of work done by all the leaders.
D_g	:	Length of the critical path of the DAG G without local tasks.
W_l	:	Number of local tasks and amount of work executed by the slaves.
D_l	:	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.

An illustration of the above quantities is given in Figure 1. By definition of blocks, we have :

$$W_l = \sum_i W_i \qquad (1) \qquad\qquad D_l = \max_i D_i \qquad (2)$$

We also introduce some notation to characterize the platform: the platform has p processors divided in number of groups, as described in Section 3. For this proof, the groups are considered of equal size (homogeneous case). We define p_g the number of processors in a group and g the number of groups. The case of heterogeneous groups will be addressed in a future work.



Characteristics of G' are :

$$\begin{aligned} D' &\le D'_g + D'_l \\ &\le D_g + \max_i (O(D_i) + \frac{W_i}{p_g}) \\ &\le O(D) + \max_i \frac{W_i}{p_g} \qquad (3) \\ W' &= W'_g + W'_l \\ &= W_g + \sum_i (\frac{W_i}{p_g} + O(D_i)) \\ &\le W_g + \frac{W_l}{p_g} + O(B * D_l) \qquad (4) \end{aligned}$$

Fig. 2. A representation of the graph G' executed by leaders

Theorem 1 *With high probability, the execution time of an application with the HWS algorithm is less than* $\frac{W_g}{g} + \frac{W_l}{p_g * g} + O(D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$. *In addition, the number of steal attempts between groups is smaller 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 bounded 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 $\frac{W_i}{p_g} + O(D_i)$.

We now 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 $\frac{W_i}{p_g} + O(D_i)$. Figure 2 presents a DAG G' and its characteristics.

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

Note : the execution time of work-stealing algorithm is bounded with a given probability. Thus chain lengths are here in fact random variables. It is possible to show that these probabilities have little impact in the remaining results presented here. We do not present these computations here due to space limitations.

4.2 Results interpretation

In the previous section, we proved that the HWS execution time was bounded by $\frac{W_g}{g} + \frac{W_l}{p_g * g} + O(D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$. This bound is rather complex and we might wonder whether this expression is out of touch with the real world or not. We offer an interpretation of the above terms:

$\frac{W_g}{g}$ **:** The work above the limit l is performed by leaders. This term results from the HWS design. Each global task can only be executed by leaders. If the limit l is chosen small enough, this term is negligible compared to others.

$\frac{W_l}{p_g * g}$ **:** This shows that local tasks are balanced efficiently between slaves. Thus, most of the work is balanced on the whole platform.

D **:** The tasks on the critical path have to be executed one after the other. This term comes directly from the classical work-stealing algorithm bound.

$\max_i \frac{W_i}{p_g}$ **:** This expression derives from the main HWS constraint : each block is restricted to be executed by only one group. This constraint is imposed by HWS to restrict communications. The execution time cannot be lower than the execution time of the largest block on one group. This term is sizable if : $\max_i W_i < \frac{W_d}{g}$. In practice, this problem rarely occurs.

$D\frac{B}{g}$ **:** The origin of this term can be understood from the proof. 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 for the analysis and might be negligible in practice.

To conclude this analysis, we highlight two observations. We have shown that the work is efficiently balanced under some conditions : the limit l is small and the work is not too unbalanced between the different blocks. In addition, the number of global steal attempts is bounded by $O(g * (D + \max_i \frac{W_i}{p_g}))$ which fulfills the main goal of the HWS algorithm. We therefore expect experimental validations of HWS to outperform the standard work-stealing algorithm.

5 Experimental validations

We compare PWS and HWS with three standard algorithms from the literature : the classical work-stealing (denoted by WS), *Kaapi* work-stealing and CRS. These experiments stand as a practical test for our algorithms and enable us to exhibit a significant performance gain.

Our experiments are designed not only to compare existing heuristics to PWS and HWS but also to show the disruption of the network on different kind of applications.

In our experiments, we use the Grid'5000 [1] platform. On this platform, the largest shared memory computer has two 2.5 GHz Intel Xeon E5420 of four

[1] 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).

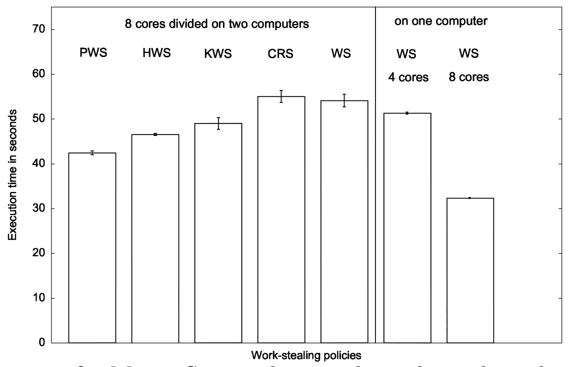


Fig. 3. Execution time for Merge-Sort with severals work-stealing algorithms on shared and distributed platform.

cores each. The baseline is an execution on a single node with eight cores to the execution on two nodes with four cores each. The different computers are here connected by an *InfiniBand* network. The execution platform is hierarchical and enable us to understand the behaviour of heuristics without many disruptions. These first experiments achieve a distributed execution in a controlled environment.

We analyze all scheduling algorithms for two different applications. The first one is an implementation of the Merge Sort algorithm. The goal is to sort an array of four Gigabytes. This algorithm was selected because it induces a large amount of communications. The second application solves the N-queen problem. This application spawns many tasks but each task transfers only a little amount of data.

Sections 5.1 and 5.2 report respectively the results from the experiments with the Merge Sort and the N-queen example.

5.1 Merge Sort

We analyze the execution time of each work-stealing algorithm to sort an array of four Gigabytes on a distributed platform. The 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 is done in place with the merge algorithm from the STL library. When the array size is less than four Kilobytes, the work is performed sequentially with the sort of the standard library STL. We evaluated an overhead lower than three per cent by comparing a pure STL sequential execution with an execution of the parallel code on one core. The error bars on all figures represent the confidence interval computed with one hundred experiments per point.

Figure 3 shows the comparison between PWS, HWS and existing policies. CRS and WS obtain a speed-down with respect to the execution time on four cores on one node, on about ninety per cent of all executions. This result is consistent with the analysis provided in [8]. In this paper, CRS achieves bad performances for matrix multiplications. The results are due to the high amount of data in use in both setups. *Kaapi* work-stealing policy exhibits a speed-up for more than 60 per cent of all executions over WS on one processor.

Figure 3 shows performances of PWS and HWS with a fixed value of ten per cent for the probability to send a steal request to the other node in PWS and

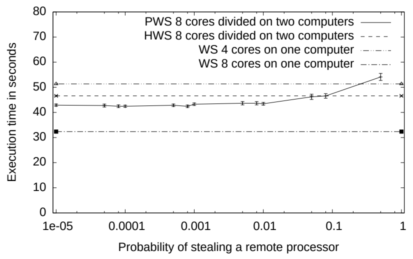


Fig. 4. Execution time for Merge-Sort contingent on the probability in PWS

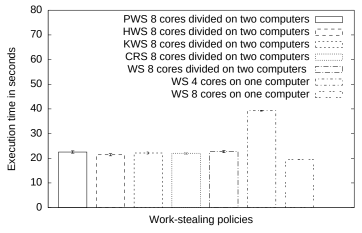


Fig. 5. Execution time for N-queen with severals work-stealing algorithms on shared and distributed platform.

the limit l is set to two in HWS. All executions with HWS and PWS achieve a significant speed-up compared to the classical policy on four cores of one node. Figure 4 shows the evolution of the execution time as the PWS probability varies. This probability can be chosen in a very wide range of values. Thus, the probability could be easily chosen.

We have also monitored the amount of data transferred by the Merge Sort application, which helps explaining these results. These amount is :

– Around eight Gigabytes for the classical work stealing CRS,
– Less than four Gigabytes for HWS and PWS.

5.2 N-queen

In our second set of experiments, we use a recursive algorithm to solve the N-queens problem. The program spawns a task for each possible position of the queen on the first line. Then given 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 sequentially. The program gives the number of possibilities to solve the problem. Many tasks are forked but little data is transferred for each task. Thus, this example requires a heuristic without overhead to get a linear speed-up.

In our experiments, the problem was solved for eighteen queens. Figure 5 shows the execution time of all scheduling algorithms executed on two nodes and the WS on one node with one or two processors. These experiments indicate that all strategies exhibits a roughly equivalent execution time. We conclude from these experiments that when the amount of data to transfer is low, hierarchical policies do not decrease performances. Moreover, these heuristics do not incur any measurable overhead.

Conclusion

To conclude this section, we emphasize that we were able to achieve significant speed-up with both PWS and HWS on experiments with large data transfers. With an improvement reaching twenty per cent of the execution time over standard algorithms on the same platform, a distributed execution becomes a viable option. It achieves speedup in all executions while classical algorithms increase execution time when using remote resources. PWS and HWS reduce the amount of data transferred. With PWS, the programmer can tune the probability parameter on a very wide range and obtain the best performance.

6 Conclusion

In this paper, we introduce two new work-stealing algorithms HWS and PWS for hierarchical platform, and compare them to the state of the art. We provide a theoretical analysis for HWS and show that under reasonable assumptions it exhibits an efficient load balancing while greatly reducing the number of remote communications. PWS and HWS are then validated experimentally on two different kinds of applications. Both of them show an increase in performance strong enough to justify a distributed execution, with PWS outperforming HWS.

This work may be extended in several directions. First, on the theoretical side, we would like to perform an analysis of PWS and relax the current assumptions on the analysis of HWS. On a practical side we plan to conduct experiments at larger scales with several levels of hierarchy.

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

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: ACM SPAA (1998)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM (1999)
3. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN. pp. 212–223 (june 1998)
4. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO (2007)
5. Gautier, T., Roch, J.L., Wagner, F.: Fine grain distributed implementation of a dataflow language with provable performances. In: ICCS (2007)
6. Kale, L.V., Krishnan, S.: Charm++: Parallel Programming with Message-Driven Objects. Parallel Programming using C++ (1996)
7. Lee, J.K., Palsberg, J.: Featherweight x10: A core calculus for async-finish parallelism. In: PPoPP (2010)
8. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: PPoPP (2001)
9. van Nieuwpoort, R.V., Maassen, J., Hofman, R., Kielmann, T., Bal, H.E.: Satin: Simple and efficient Java-based grid programming. In: AGridM Workshop (2003)
10. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Kielmann, T., Bal, H.E.: Adaptive load balancing for divide-and-conquer grid applications. SC (2006)
11. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: IPDPS (2008)
12. Shivali, A., Rajkishore, B., Dan, B., Vivek, S., K., S.R., Katherine, Y.: Deadlock-free scheduling of x10 computations with bounded resources. In: SPAA (2007)
13. Valois, J.D.: Implementing lock-free queues. In: PDCS (1994)