On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing

Guilherme P. Pezzi Márcia C. Cera Elton Mathias Nicolas Maillard
Philippe O. A. Navaux
Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{pezzi, mccera, enmathias, nicolas, navaux}@inf.ufrgs.br

Abstract

MPI (Message Passing Interface) is the de facto *standard in High Performance Computing. By using some MPI-2 new features, such as the dynamic creation of processes, it is possible to implement highly efficient parallel programs that can run on dynamic and/or heterogeneous resources, provided a good schedule of the processes can be computed at run-time. A classical solution to schedule parallel programs on-line is Work Stealing. However, its use with MPI-2 is complicated by a restricted communication scheme between the processes: namely, spawned processes in MPI-2 can only communicate with their direct parents. This work presents an on-line scheduling algorithm, called* Hierarchical Work Stealing*, to obtain good load-balancing of MPI-2 programs that follow a Divide & Conquer strategy. Experimental results are provided, based on a synthetic application, the N-Queens computation. The results show that the* Hierarchical Work Stealing *algorithm enables the use of MPI with high efficiency, even in parallel dynamic HPC platforms that are not as homogeneous as clusters.*

1 Introduction

Most of today's high-performance architectures aggregate heterogeneous resources, with different levels of parallelism: multi-core processors share a common memory in a node, and different nodes are interconnected by possibly different, hierarchical networks. Programming such machines is a challenging task: portability must be achieved by the use of state-of-the-art libraries or languages, in order to evolve at the same pace as the architectures. On the other hand, high-level languages do not provide the expected performance on HPC machines.

The *Message Passing Interface* (MPI) [8, 16] is a recognized standard for HPC programming in a distributed memory environment. The MPI-2 norm [9] has added new features to MPI like dynamic processes creation, remote memory access and parallel I/O. With MPI-2, it is possible to implement dynamic applications which adapt, at run-time, to heterogeneous or dynamic resources in the computing platform.

Many projects have proposed other programming interfaces or languages, with dynamic adaptation to the underlying architecture. One of the key features is the control of the parallelism, *i.e.* of the number of tasks that may be run concurrently (depending of the architecture and of the language, a task may be run by a heavy process or by a thread). A major proposal to that end is to use a recursive-like programming model, for instance Divide and Conquer (*D&C*). Besides its algorithmic interest (see for instance [4]), *D&C* allows to adapt the degree of parallelism since recursive calls may easily be performed in parallel or be run sequentially, on-demand, in function of the processing resources [7]. Languages that use this approach include Cilk [1] for shared-memory systems and based on C; Satin [17], a Java extension which runs on distributed-memory clusters of clusters; and Charm++ [12], a C++ interface for parallel programming. Yet, they either are limited to shared-memory systems, either do not provide performances as good as MPI.

The parallel performance of *D&C* programs relies on the scheduling algorithm. Theoretical and practical results have shown that such *D&C* programs can be efficiently scheduled, at run-time, by *Work Stealing* [2]. *Work Stealing* can be seen as a distributed version of list-scheduling. It implies that the processors, when they are idle, may steal tasks from any randomly chosen victim processor. However, MPI-2 does not allow a spawned process to communicate with a process other than its parent (the one that spawned it) or its children. Thus, using MPI-2 to implement *Work Stealing* is not a trivial task.

This article proposes a variation of *Work Stealing*, called *Hierarchical Work Stealing*, suited to MPI-2 implementations of dynamic parallel programs. *Hierarchical Work Stealing* generalizes Satin's scheduling algorithm by using a tree-like hierarchy of *manager processes* that route the stealing requests. The results that are presented show that this generalization allows to improve the run-time of a dy-

namic parallel program by removing the bottleneck on the tasks list. This on-line scheduling algorithm also enables the efficient use of a dynamically increasing platform.

The article is structured as follows: the section 2 details how dynamic parallel programs can be obtained with *D&C*, as well as some parallel programming environments that support it. Then, Sec. 3 presents the main contribution, the *Hierarchical Work Stealing* scheduling algorithm, that is usable with MPI-2 *D&C* programs. In the next section, an experimental study is performed with timings that show the performance of such MPI implementations and the impact of the hierarchical tree of managing processes on the global run-time. Finally, Sec. 5 sums up our contribution and points to future work.

2 Dynamic Parallel Programs

One of the main techniques to unfold parallelism in an application at run-time is Divide and Conquer. Subsection 2.1 details it and lists theoretical arguments and real implementations that justify its use. The scheduling of the processes that are created during the "divide" phase is usually computed by *Work Stealing*, which has been shown to be the best possible solution. Subsec. 2.2 explains this algorithm.

2.1 Parallel *D&C* Programs

Given an instance of a problem, *D&C* consists in partitioning it recursively in δ sub-problems until its resolution turns to be trivial. Each partial solution has then to be merged with the other one(s) in order to aggregate the results into the output of the initial problem. Thus, it can be formally described (here for $\delta = 2$) as:

$$\text{Solution}(i): \begin{cases} \text{if trivial}(i) \rightarrow \text{direct}(i) \\ \text{else} \rightarrow \text{merge}\begin{pmatrix} \text{Solution}(\text{part}_1(i)), \\ \text{Solution}(\text{part}_2(i)) \end{pmatrix} \end{cases}$$

When more than one processors are available to run a parallel *D&C* program, two characteristics of this model grant a good efficiency:

1. Since the algorithm is recursive, the depth of the call tree is a logarithm of the size of the input. If there are enough processors to execute all tasks of a given depth of the tree, the run-time will be logarithmic. When the number of processors is bounded by a polynomial of n, the problem belongs to the NC class of "highly parallel" problems [11, 13]. It is for instance the case of various sorting algorithms, of the computation of the prefix, or of the iterated sum. Notice that the time complexity is logarithmic only if the merge operation can also be performed efficiently in parallel.
2. Since there is recursive division in sub-problems, each division offers the option to be run by a separate thread (or process), or to be performed as a sequential, recursive function call. Thus, the exact degree of parallelism in the algorithm can be controlled.

   In that sense, the parallel *D&C* program is adaptive to a dynamic and heterogeneous environment.

Many parallel programming environments have been designed to offer such a *D&C* programming model to users.

Charm++ [12] is a parallel programming environment which is not restricted to, but supports, the *D&C* model. The programmer can use a C++ API to define the sequential tasks, as well as their input and output parameters, with a special object called *chare*. On top of Charm++, the user can use Adaptive-MPI (AMPI) [10], an implementation of MPI with possibilities of dynamic load balancing. However, providing adaptability to any MPI program implies a relevant overhead in the portability layer. The approach in this paper is to limit the programming model to *D&C*, to avoid the need of a middleware such as Charm++ and to use pure MPI programming.

Cilk [1] has been developed in the MIT and increases the C language with three keywords: *(i)* `cilk`, inserted in front of a function, declares it as "spawnable", *i.e.* asynchronously executed. *(ii)* `spawn`, before a call to a `cilk` function, will dynamically create a new execution flow to execute the function. *(iii)* `sync` forces a synchronization between a parent task and its children. The combined use of `spawn` and `sync` implies that the programmer uses a *D&C* model. A compiler, provided by the Cilk team, transforms the C code and these three keywords into a threaded code. Cilk has been used to obtain very efficient codes for SMP architectures. One of its major results has been the chess program called Socrates [5], which won a prize. The main limitation of Cilk is its lack of an efficient implementation for distributed architectures.

Satin [17, 19] is, to our best knowledge, the only environment that supports adaptability in the sense that has been defined above on a large scale distributed platform. It is a Java implementation of the Cilk model, that provides an efficient distributed implementation of Java. In [19], reports are given of efficient executions of several *D&C* benchmarks and applications, including the N-Queens program that is used for our validation in Sec. 4.

2.2 Scheduling *D&C* Parallel Programs with *Work Stealing*

Closely related to *D&C* parallel programs, *Work Stealing* has been devised by Blumofe and Leiserson [2] as an efficient distributed algorithm to maintain the balance of the load. Each processor maintains a deque (*doubly ended*

queue): when a new task has to be computed, it is pushed on top of the deque. When a processor gets idle, it pops a task to process from the top of the deque. If it is empty, then the processor issues a stealing request, aimed at a randomly chosen victim processor. When a processor receives such a stealing request, it pops a task from the bottom of its deque, to send it to the stealing processor.

The authors mentioned above proved that *Work Stealing* is efficient in a homogeneous environment, both in time and in space, for *strict computations*, *i.e.* those where no task is executed before its parents have been executed. *D&C* programs belong to this category. The proof relies on the random choice of the victim and on the homogeneous distribution of the stealing requests that this implies.

Satin's team proposed a variation of standard *Work Stealing* in [19], to be efficient in a cluster of cluster with heterogeneous network. In this case, a processor can either try to steal another one in the same, local cluster, or in a remote one. In Satin's two-level *Work Stealing*, a processor which has to steal tasks first issues a non-blocking stealing attempt to a processor in a remote cluster. In parallel, it tries to steal other tasks from processors in its local cluster. The first answered request is served. Usually, it is the local one, but the other request still get processed, and eventually the stealer will receive some tasks to process from a remote cluster, without having had to wait idly for it. In simulations and experimentations, this version of *Work Stealing* has proven to be more efficient than a random choice of the victim in an environment with heterogeneous network connection.

The dynamic creation of processes in MPI-2 can be used to implement parallel applications that spawn processes on the fly. Such MPI-2 programs, following the *D&C* model, should be scheduled by *Work Stealing*. Next section shows how this can be achieved.

3 Scheduling *D&C* MPI-2 Programs with *Hierarchical Work Stealing*

A classical MPI program is executed with a fixed number of processes, defined before the execution. With MPI-2, the `MPI_Comm_spawn` procedure enables the spawning of another MPI program during the execution of a MPI process. These newly spawned processes may communicate by message passing (*e.g.* with standard Send/Receive), with their parent and their children processes. A previous work [3] has shown how to schedule, at run-time, these spawned processes, with a centralized daemon. Pezzi *et al.* [15] showed how to program *D&C* applications with MPI-2. This section explains how a *D&C* program implemented with MPI-2 can be scheduled on-line by *Work Stealing*.

3.1 Using *Work Stealing* with MPI

With MPI, the main problem is that the processes can only communicate if they have a parent/child relationship. Namely, the message passing is possible between processes that share an Intercommunicator, which is initialized by the `MPI_Comm_spawn` call. One solution would be to merge all the processes in the same communicator with `MPI_Intercomm_merge`. Yet, this would mean a collective synchronization, at each creation of a process.

Since collective synchronization should be spared, another way to implement *Work Stealing* in MPI is as follows: a tree hierarchy of processes (called *management processes*, or *managers*) is built, which will be in charge of the tasks deques and of routing the stealing requests. Let δ be the number of branches of each node in the management tree, and d be its depth. Each manager of depth d spawns l *leaf processes*, which will be in charge of computing the tasks.

A leaf process that does not have any task to run issues a stealing request to its parent. Either the parent (which is a management process) has some tasks in its deque to send to the leaf; either it has not, and then it forwards the request to its own parent. This routing of the requests does not respect the uniform distribution of the stealing requests, that is a key for the good theoretical properties of *Work Stealing* in Blumofe's work. On the other hand, it is a generalization of the two-levels request that Satin's group proposed and respects the parent/child communicators of MPI. This variation of *Work Stealing* is called *Hierarchical Work Stealing*.

The number of tasks to be sent (if there some) from a manager to its child is a parameter that can be set; in the following discussion it is supposed to be set to 1.

The size of the management tree is determined by the user in function of the topology of his executing environment. There are $1 + \delta + \ldots + \delta^{d-1} = \frac{\delta^d - 1}{\delta - 1}$ managers and $l\delta^{d-1}$ leaves. Setting the number of leaves equal to the number p of processors, one gets $l\delta^{d-1} = p$, and therefore the number of managers is $\frac{p\delta - l}{l\delta - l}$ (Sec. 4.4 shows the influence of this factor).

3.2 The *Hierarchical Work Stealing* Algorithm

The *Hierarchical Work Stealing* algorithm starts as shown in Algorithm 1. In all algorithms, `this` is supposed to return an identifier of the running process. `ROOT` is the first process to be run, the others being recursively spawned by it. The deque data-structure provides the methods `empty()`, `push()`, `size()` and `pop()`.

In Algorithm 1, one process starts executing and if it is a leaf process, it just has to receive the tasks from it parent, process them and return the partial result; then, it exits (lines 3–6). Else, the process pushes on its deque all the tasks, and gets the list of available processors (the depth

d of the management tree will be controlled by the number of available processors); if the process is `ROOT`, it has to obtain this information by the O.S. and the launcher of the program (lines 9–10); in the other case, it receives the tasks and the list of processors from its parent (lines 12–14). Then, it calls the creation of managers or leaves depending of the number π of available processors. Both functions `create-managers` and `create-leaves` (presented below) will eventually spawn new processes, that will recursively run the same algorithm 1. However, since they will not be `ROOT`, they will receive tasks from their parent (line 12), push them on their deque and receive the list of processors. Then, they proceed with the recursive behavior of calling `create-managers` or `create-leaves` (the latter will end the recursion, see Algorithm 3).

In all the cases, the process will finally either return the total result of the computation if it is the `ROOT`, either simply send its partial result to its parent. The corresponding receive is issued in the `create-managers` below (Algorithm 2, line 6). After this send, the process exits.

Notice that the `recv` in this algorithm must deal with messages of unknown size. They can be implemented with MPI as a sequence of two messages, a first one that indicates the number of elements to be received in the second message.

Algorithm 1 Hierarchical *Work Stealing* for MPI.

```
 1: Input: an integer π.
 2: Output: the result of the program.
 3: if this == leaf then
 4:    recv(parent, tasks)
 5:    part-res = Compute( tasks );
 6:    send(parent, part-res);
 7: else
 8:    if this == ROOT then
 9:       deque.push( new_tasks );
10:       Processors-l = getProcessorsList();
11:    else
12:       recv(parent, tasks);
13:       deque.push( tasks );
14:       recv(parent, Processors-l);
15:    end if
16:    if Processors-l.size() > π then
17:       create-managers();
18:    else
19:       create-leaves();
20:    end if
21:    if this == ROOT then
22:       return(result);
23:    else
24:       send(parent, part-res);
25:    end if
26: end if
```

Algorithms 2 and 3 present the (recursive) creation of managers (`create-manager`) and leaves (`create-leaf`).

Algorithm 2 Creation of Management Processes (create-manager).

```
 1: for i = 1 ... δ do
 2:    spawn(child[i]);
 3:    tasks = deque.pop();
 4:    send(child[i], tasks);
 5:    send(child[i], processor-l);
 6:    irecv(child[i], part-res);
 7: end for
 8: live-children = δ;
 9: while live-children > 0 do
10:    wait-any();
11:    if part-res.tag == END then
12:       live-children - - ;
13:    end if
14:    if ! deque.empty() then
15:       tasks = deque.pop();
16:       send(part-res.src, tasks);
17:       irecv(part-res.src, part-res);
18:    else
19:       if father-has-tasks then
20:          send(parent, part-res);
21:          recv(parent, tasks);
22:          if tasks.size > 0 then
23:             deque.push( tasks );
24:          else
25:             father-has-tasks = false;
26:          end if
27:       end if
28:    end if
29: end while
```

In Algorithm 2, the running process first spawns δ children (line 2) and gets a reference on each one of them, that will be used to communicate (actually, this is implemented with the intercommunicator mechanism of MPI-2). Notice that this procedure `create-manager` will be called only when management processes must be spawned.

The manager sends to each child some tasks of its deque (lines 3–5) and starts receiving any partial result that may arrive. It then enters into a loop (line 9), waiting for any incoming message that could have been received by the non-blocking `irecv` of line 6. The `wait-any` call of line 10 blocks until receiving whatever MPI non-blocking message (from any source). When a process ends up with its tasks, it sends a message with the partial results that it has computed if it is a leaf process, or received from its children if it is a management process, to its parent (see Algorithm 3,

line 21). This message has a tag `end`. The parent manager, upon receiving a message with tag `end` (line 11 of Algorithm 2), updates the number of live children that remain. In all other cases, a received message is a stealing request, which is treated in lines 14–17: if there is still tasks to be processed in the deque of the manager, then one is sent to the process which asked for work (as informed in the `msg.src` field of the received message). Else, the steal request will be forwarded to the parent of the management process (lines 19–27). When the own parent does not have any task left in its queue, and since the stealing requests are always routed bottom-up, and then answered from `ROOT` down to the leaves, there can not be any more task available up to the `ROOT` process. At this point the manager just waits for the results from its children which are already executing, sends the merged results to its parent and finalizes.

The variable `processors-l` of line 5 (in Algorithm 2) is the list of processors that the process is managing. It is initialized by the `ROOT`, which sends it to each manager. It controls the recursive creation of managers or leaves.

Algorithm 3 Creation of Leaf Processes (create-leaf).

```
 1: live-children = 0;
 2: local-result = 0;
 3: for i ∈ processors-l do
 4:    spawn(child[i]);
 5:    live-children++ ;
 6:    tasks = deque.pop();
 7:    send(child[i], tasks);
 8:    irecv(child[i], part-res);
 9: end for
10: while live-children > 0 do
11:    wait-any();
12:    local-result += part-res;
13:    live-children– ;
14:    task = deque.pop();
15:    if task != ∅ then
16:       spawn(child);
17:       live-children++ ;
18:       send(part-res.src, task);
19:       irecv(part-res.src, part-res);
20:       if father-has-tasks then
21:          send(parent, local-result);
22:          recv(parent, tasks);
23:          if tasks.size > 0 then
24:             deque.push( tasks );
25:          else
26:             father-has-tasks = false;
27:          end if
28:       end if
29:    end if
30: end while
```

In the `create-leaf` procedure (Algorithm 3), the current process starts spawning one child process by processor. Each child is now a leaf process. The current process sends to each child a task to be processed and issues a non-blocking reception of some partial result. When it receives a partial result, it aggregates it to other that may have been previously received in `local-result`. Since partial results that have been received mean that the leaf process is done with his tasks (and has exited, see Algorithm 1), the current process spawns a new leaf (line 16), pops a task and sends it to the leaf, if it is not NULL (*i.e.* if its deque was not empty). When sending the task, it tests if its deque is now empty, in order to issue a steal request to its parent (lines 20–28) if necessary.

The *Hierarchical Work Stealing* algorithm has been implemented and tested on a synthetic application. The next section gives practical insights on its effective performance.

4 Experimental Evaluation of a Dynamic MPI-2 Application Scheduled by *Hierarchical Work Stealing*

The performance of a dynamic MPI program, scheduled by *Hierarchical Work Stealing*, is demonstrated in this section. In 4.1, the N-Queens application is quickly presented. Then, Sec. 4.2 presents timing results for the N-Queens program and compares the run-time of the MPI adaptive implementation with Satin's, on a 20 nodes cluster. Sec. 4.3 presents its behavior in a dynamic environment: during the execution of the program, new processors are included. As expected, the MPI dynamic program reacts well and *Hierarchical Work Stealing* schedules new spawned processes on the new resources. Finally, Sec. 4.4 illustrates the main interest of *Hierarchical Work Stealing*: by adapting the number of manager processes (*i.e.* the depth of the management tree), the load balance improves and the run-time gets lower.

4.1 The Test Application: N-Queens

The N-Queens problem consists in placing N queens on a $N \times N$ chessboard, in such a way that no queen may capture any other. That is to say, one has to find all the board configurations in which there exists at most one queen in a given row, column or diagonal. Although direct applications of the N-Queens problem are limited, this problem is often used as a benchmark because it represents a large class of problems, known as Constraint Satisfaction Problems (CSPs) [14].

The standard backtracking algorithm used to solve the N-Queens problem consists in placing recursively and exhaustively the queens, row by row. With MPI, each placement consists in a new spawned task. The algorithm backtracks whenever a developed configuration contains two

queens that threaten each other, until all the possibilities have been considered. A maximum depth is defined, in order to bound the depth of the recursive calls.

4.2 Performance of MPI *vs.* Satin

In this section, the performance of the MPI N-Queens program is compared to a Satin implementation.

The same *D&C* algorithm has been implemented in both MPI and Satin. First, the constant overhead due to Java (in Satin) should be evaluated, when compared to the MPI implementation. Table 1 presents the run-times of each one of the two implementations, executing on one processor, for different sizes N of the problem.

	MPI run-time (sec.)		Satin run-time (sec.)	
N	Average	Std. Dev.	Average	Std. Dev.
16	11.31	0,01	15.74	0.01
17	76.79	0,28	108.75	0.06
18	552.88	0.14	790.75	0.42
19	4252.88	0.57	6085.02	0.38

Table 1. Run-time of the N-Queens program with MPI-2 *vs.* Satin in the sequential case.

As can be seen, in all sequential executions, Satin is substantially slower than MPI, as expected. This is because of this overhead that MPI is a clear interesting solution if it can be used to implement and schedule dynamic programs.

All further parallel executions have been made on a Linux cluster of 20 nodes bi-processors Pentium III. Each program has been run 5 times and the mean run-time is presented. The standard deviation has been small enough to consider the mean to be relevant. Figure 1 presents the gain of run-time with the MPI implementation of N-Queens, for $N = 18$, expressed as a percentage of the run-time of the Satin implementation, when the number n of computing nodes increases: $1 - T_{MPI}(n)/T_{Satin}(n)$.

In the parallel executions, the MPI program has never been slower than the Satin one. In this configuration, the sequential run-time is around 265 sec. with MPI, and 400 sec. for Satin. On 20 nodes, MPI runs in 18.2 sec., *vs.* 26.7 sec. for Satin. Actually, when the size N of the problem increases, MPI turns out to outperform Satin even more. Thus, for $N = 20$, MPI solved the N-Queens problem in 846.1 sec., *vs.* 1604.6 sec. for Satin (gain of 47.3 %).

As can be seen, the MPI implementation is consistently faster than Satin; but this gain actually comes from the sequential overhead measured in the former table: for $N = 18$, it constantly remains around 30%. The interest in this comparison is to show that the MPI program, scheduled by

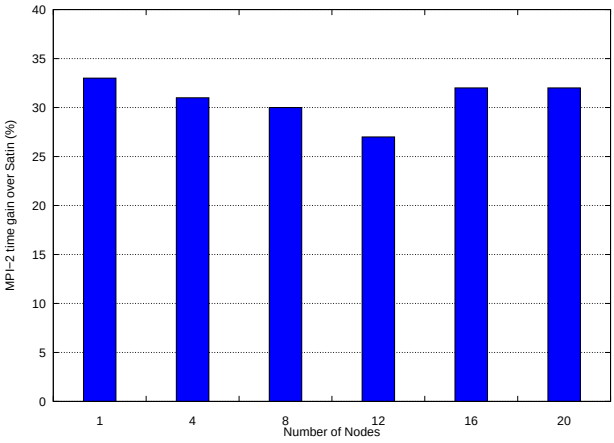

Figure 1. Run-time gain of the N-Queens program with MPI-2 over Satin.

Hierarchical Work Stealing, is as scalable as the Satin implementation (up to 20 nodes, 40 CPUs), already known to be scalable in real Grid environments [18].

4.3 Scheduling a MPI Program in a Dynamic Environment

Figure 2 shows the behavior of the N-Queens application, scheduled by *Hierarchical Work Stealing*, with $N = 18$, implemented with MPI. In this experiment, new processors are dynamically added to the processing environment.

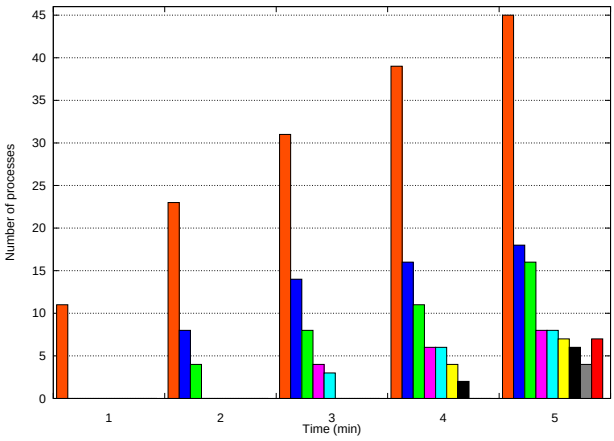

Figure 2. Number of processes on processors *vs.* time during an execution of N-Queens with MPI. Each minute, a new node is added. The new processes are scheduled on-line on the new nodes due to *Hierarchical Work Stealing*.

In the experiment, LAM's `lamgrow` facility has been used, in order to add a new processing node (made of 2 CPUs) each minute. The bar diagram shows how many processes are run on each CPU, as a function of the time: at the beginning, all processes are run on the only node available. During the first minute, only one bar appears, which shows that all the 11 processes are run on one node. Each time that a 2-CPUs node becomes available, the N-Queens programs adapts itself by spawning new processes that are scheduled, on-line, by *Hierarchical Work Stealing* on the newly available CPUs, thanks to the stealing requests that are issued. For instance, in the third minute of the execution, 5 CPUs are used with different numbers of processes running on each one. At the end of the computation, all the CPUs of the 5 nodes are running processes of the N-Queens computation. Since no migration is allowed, the initial nodes run, until the end, more processes than those that were added later on. Yet, this experiment shows that the MPI program adapts itself to the appearance of new nodes in the computing environment.

4.4 Performance *vs.*. Number of Managers in *Hierarchical Work Stealing*

The *Hierarchical Work Stealing* algorithm enables to choose the number of management processes and their topology: for instance, it can build a flat tree with only one manager, which balances the load between n leaves. When a leaf gets idle, it will ask the centralized manager for more tasks. This centralized strategy clearly leads to a bottleneck. It is also possible to increase the depth of the management tree and change the degree δ of each management node to manage the same number of processes as in the centralized solution, yet without bottleneck.

Clearly, The performance of the *Hierarchical Work Stealing* algorithm depends of the granularity: the finer the grain, more tasks there will be to balance the load and the better the result. With coarse tasks, the processors do not have much opportunities to steal work from other. In this evaluation, the granularity has been fixed to two tasks by stealing request. The study of the impact of the granularity on the *Hierarchical Work Stealing* algorithm is left for another article.

Figure 3 shows the execution time of N-Queens, for $N = 19$, with different numbers of managers, on a 20 nodes (dual) cluster. Each measure is the mean of 10 executions, and the standard deviation is represented in the graph. As in previous experiences, it is insignificant.

Using the notations of Sec. 3.1, with $p = 40 = 2^3 \times 5$ and since l, δ, d are integers, the choices are limited for these parameters. Table 2 indicates the possibilities and the corresponding number of managers ($\frac{p\delta - l}{l\delta - l}$).

As can be seen, $1, 3, 5, 6, 7, 9, 11, 15$, and 21 managers

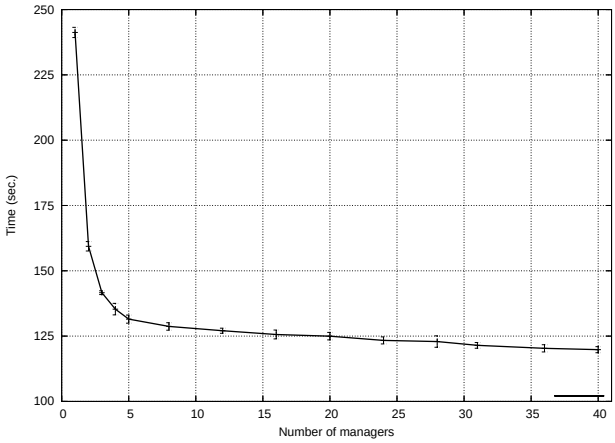

Figure 3. Run-time of the N-Queens MPI program, with 40 CPUs, *vs.* number of managers in *Hierarchical Work Stealing*.

Depth d	l	δ	Nb of managers
1	40	0	1
2	2	20	21
	4	10	11
	8	5	6
	5	8	9
	10	4	5
	20	2	3
3	2	10	7
4	5	2	15

Table 2. Tree hierarchy and number of managers for $p = 40$ processors.

can be obtained with different topologies for the management tree. Other values can be obtained if the tree is not totally balanced. Fig 3 has been obtained, with these values, of with depth 2 but an adapted number l of leaves. Fig. 3 clearly shows the decreasing run-time when the number of managers increases. The worst case is the flat hierarchy with only 1 manager (centralized management of the steal requests). An interesting point is that a limited number of managers (5) already lowers significantly the run-time. Notice also that the experimental platform only included 40 CPUs: on hundreds of nodes, *Hierarchical Work Stealing* with more managers should do even better.

To sum this section up, the MPI *D&C* implementation of N-Queens is adaptable to a growing number of CPUs, thanks to *Hierarchical Work Stealing*; its global run-time outperforms the Satin implementation on a distributed plat-

form, as expected from MPI, yet remains as scalable. Finally, the use of a hierarchical scheme to manage the stealing requests allows for a clear improvement in the run-time of the application, when compared to a centralized load balancing scheme, or to non-hierarchical Work Stealing.

5 Conclusion: Contribution and Future Work

This article has presented how the *D&C* parallel model and *Work Stealing* can be used to obtain dynamic, efficient MPI-2 programs. The *Work Stealing* algorithm has had to be adapted, to use a hierarchical routing of the stealing requests along the process tree. For an instance of such a program, the response to a changing environment, the quality of the load balance and the overall timing performance have been shown to be good, and anyway better than the Satin implementation, one of the references in Java-based distributed computing environments for Grid and Clusters. The use of a hierarchical scheme in *Hierarchical Work Stealing* allows to avoid bottlenecks, thus improving even more the run-time.

These programming techniques have been proposed in other environments, yet their effective use in MPI, in a dynamic context, allows new high-performing code to be easily developed. We think that this parallel programming model should be fostered in MPI and used with HPC traditional applications. Our *D&C* MPI implementation of N-Queens is very generic and can be generalized to any *D&C* application. It can be applied for instance to Branch & Bound problems, to sorting algorithms, to Game Theory (*e.g.* chess), or to Scientific Computations. The LU factorization or the matrix product are both part of the benchmarks of Cilk and Satin. Since the best known algorithms for these operations are block-recursive [6], they are well suited to this parallel programming model.

One of the most interesting possibilities to continue this work is to explore the control of the parallelism, in function of the resources that can be dynamically integrated in the environment: for instance, instead of spawning a MPI process, a light-weight process could be launched. Another important study would be the impact of the hierarchical routing of the stealing requests on the mean number of steals.

Acknowledgement: this work has been partially supported by CAPES scholarships.

References

[1] R. Blumofe and et al. Cilk: An efficient multithreaded runtime system. In *PDOPP'05*, pages 207–216, 1995.

[2] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[3] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the Dynamic Creation of Processes in MPI-2. In *LNCS - 13th European PVMMPI Users Group Meeting*, volume 4192/2006, pages 247–255, Bonn, Germany, Set. 2006.

[4] T. H. Cormen, C. E. Leiserson, and R. L. R. ans Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[5] D. Dailey and C. E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[6] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of Parallel Computing*. Morgan Kaufmann, first edition, 2003.

[7] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, oct 1994.

[9] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, USA, 1999.

[10] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[11] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[12] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[13] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, chapter 17, pages 871–941. Elsevier Science Publishers, 1990.

[14] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[15] G. P. Pezzi, M. C. Cera, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Escalonamento dinâmico de programas mpi-2 utilizando divisão e conquista. In *VII Workshop em Sistemas Computacionais de Alto Desempenho*, pages 71–79, Ouro Preto, Brazil, Oct. 2006.

[16] M. Snir and et al. *MPI: the Complete Reference vol. 1 and 2*. The MIT Press, 1998.

[17] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-Par 2000 Parallel Processing*, number 1900 in LNCS, pages 690–699, Munich, Germany, aug 2000. Springer.

[18] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based grid programming. In *AGridM 2003 Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, USA, 2003.

[19] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Adaptive load balancing for divide-and-conquer grid applications. *J. of Supercomputing*, 2006.