Scheduling Dynamically Spawned Processes in MPI-2

Márcia C. Cera[1], Guilherme P. Pezzi[1], Maurício L. Pilla[2],
Nicolas B. Maillard[1], and Philippe O. A. Navaux[1]

[1] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil,
{mccera, pezzi, pilla, nicolas, navaux}@inf.ufrgs.br,
WWW home page: http://www.inf.ufrgs.br
[2] Universidade Católica de Pelotas, Pelotas, Brazil, WWW home page:
http://esin.ucpel.tche.br/

Abstract. The Message Passing Interface is one of the most well known parallel programming libraries. Although the standard MPI-1.2 norm only deals with a fixed number of processes, determined at the beginning of the parallel execution, the recently implemented MPI-2 standard provides primitives to spawn processes during the execution, and to enable them to communicate together.
However, the MPI norm does not include any way to schedule the processes. This paper presents a scheduler module, that has been implemented with MPI-2, that determines, on-line (i.e. during the execution), on which processor a newly spawned process should be run, and with which priority. The scheduling is computed under the hypotheses that the MPI-2 program follows a Divide and Conquer model, for which well-known scheduling algorithms can be used. A detailed presentation of the implementation of the scheduler, as well as an experimental validation, are provided. A clear improvement in the balance of the load is shown by the experiments.

1 Introduction

The Message Passing Interface (MPI) [11] has imposed itself since 1996 as the library for parallel programming in High Performance Computing (HPC). MPI's clean definition of messages, as well as the natural and efficient extension that it provides to classical sequential languages (C/Fortran), make it the most encountered parallel programming interface for clusters and dedicated parallel machines. Virtually all the distributed benchmarks in HPC have been ported to MPI (*e.g.* Linpack [7], NAS [6]); and nowadays the most challenging HPC applications are programmed in MPI (*e.g.* weather forecast, astrophysics, quantum chemistry, earthquakes, nuclear simulations… [14]).

The MPI 1.2 norm builds upon PVM (Parallel Virtual Machine) [15] to define a SPMD (Single Program, Multiple Data) programming approach, based on a fixed number of processes that can communicate through messages. MPI 1.2 defines groups of processes, as well as a communication space (communicator) to isolate the communication within a group. In a group, each process is identified by a rank. Messages are defined by a source and destination process, a basic type and a number of elements of this type. The data is packed by the programmer into a buffer of appropriated size. Communication may be synchronous or not, blocking or not. For non-blocking communications, a set of primitives allows to test the completion and to wait for it.

In spite of the success of MPI 1.2, one of PVM's features, not implemented in MPI 1.2, has long been missed: the dynamic creation of processes. The success of Grid Computing and the necessity to adapt the behavior of the parallel program, during its execution, to changing hardware, encouraged the MPI committee to include the dynamic management of processes (creation, insertion in a communicator, communication with the newly created processes…) in the MPI-2 norm. Other features have also been added, such as Remote Memory Access - RMA (one-sided communication) and parallel I/O. Although it has been defined in 1998, MPI-2 has taken some time to be implemented, and was included in a few MPI distributions only recently.

Neither MPI 1.2 nor MPI-2 define a way to schedule the processes of a MPI program. The processor on which each process will be executed, and the order in which the processes could run, is left to the MPI runtime implementation and is not specified in the norm. In the static case, for a regular application on homogeneous platforms, the schedule is trivial, or can be guided by some information gathered on the program [13]. Yet, in the dynamic case, a scheduling module should be developed to help decide on which processor each process should be physically started, during the execution. Since MPI-2 implements the dynamic creation of processes, the scheduling decision has to be taken on-line. As will be shown in Sec. 5, the native LAM solution is far from being efficient and may lead to very poor run-times.

This paper presents an on-line scheduler which targets dedicated platforms and attempts to minimize the execution time, regardless of other criteria. This contribution is organized as follows: Section 2 presents the dynamic process creation part of the MPI-2 norm as well as the distributions of MPI that implement it, and how MPI-2 programs can scheduled. Section 3 details the implementation of a scheduler for MPI-2 programs. In Sec. 4, the programming model, used in our test-cases with MPI-2, is presented, and Sec. 5 shows how the scheduler manages the balance of the load among the processors, with two distinct benchmarks. Finally, Sec. 6 concludes this article and hints at the following work to be done.

2 Dynamic Creation of Processes in MPI

Since 1997, MPI-2 has provided an interface that allows the creation of processes during the execution of a MPI program, and the communication by message passing. Although MPI-2 provides more functionalities, this article is restricted to the dynamic creation of processes. Sec. 2.1 details the `MPI_Comm_spawn` primitive which creates new MPI processes, and show how they may exchange messages. Section 2.2 presents how to schedule such spawned processes.

There is an increasing number of distributions that implement MPI-2 functionalities. LAM-MPI is the first distribution of MPI to have implemented MPI-2. LAM also ships some tools to support the run-time in a dynamic platform: the `lamgrow` and `lamshrink` primitives allow to pass to the runtime information about newly entering or leaving processors in the MPI virtual parallel machine. MPI-CH is the most classical MPI distribution, yet its implementation of MPI-2 dates back only to January 2005 only. This distribution aims at high-performance and scaling up to tens or hundreds of thousands of processors. Open-MPI is a brand new MPI-2 implementation based on

the experience gained from the developments of the LAM/MPI, LA-MPI, and FT-MPI projects [8]. HP-MPI is a high-performance MPI implementation delivered by Hewlett-Packard. It was announced in December, 2005, that it now implements MPI-2.

2.1 MPI-2

MPI_Comm_spawn is the newly introduced primitive that creates new processes after a MPI application has been started. It receives as arguments the name of an executable, that must have been compiled as a correct MPI program (thus, with the proper MPI_Init and MPI_Finalize instructions); the possible parameters that should be passed to the executable; the number of processes that should be created to run the program; a communicator, which is returned by MPI_Comm_spawn and contains an inter-communicator so that the newly created processes and the parent may communicate through classical MPI messages. Other parameters are included, but are not relevant to this work. MPI_Comm_spawn is a collective operation over all processes of the original communicator since it needs to be updated with the data about the children.

In the rest of this article, a process (or a group of processes) will be called *spawned* when it is created by a call to MPI_Comm_spawn, where the process that calls the primitive is the *parent* and the new processes are the *children*.

MPI_Comm_connect / MPI_Comm_accept With MPI-2, it is possible to establish a connection among dynamically created processes to exchange information in a client/server model. To do this, a process (the server) creates a port with MPI_Open_port, to which another process can connect afterwards. After the creation, the port name is published by MPI_Publish_name. Once the port is open and its name is published, the process allows connections by MPI_Comm_accept which returns an inter-communicator. This primitive is blocking and each process in the input communicator (MPI_Comm_accept's fourth argument) will be connected to a specific process using the same port name.

On the other hand, the client process looks the name up of the port previously published with MPI_Lookup_name. Afterwards, the client establishes connection to the server through MPI_Comm_connect. The output of this primitive is an inter-communicator to communicate with the server. When all communications are done, the process can disconnect calling MPI_Comm_disconnect, and the server can close the port with MPI_Close_port. More details about these primitives can be found in [12].

2.2 On-line Scheduling of Parallel Processes

The extensive work on scheduling of parallel programs has yielded relatively few results in the case where the scheduling decisions are taken on-line, *i.e.* during the execution. Yet, in the case of dynamically evolving programs such as those considered with MPI-2, the schedule must be computed on-line. The problem is crucial, since a good, on-line, schedule may grant both efficient run-time and portability.

The most used technique is to keep a list of ready tasks, and to allocate them to idle processors. Such an algorithm is called *list scheduling*. The description of the tasks must

be such that it allows to compute, at runtime, which tasks are ready. Thus, the programming environment must enable the description of the tasks and of their dependencies, typically the input and output data for each task [9]. The theoretical grounds of list scheduling relies on Graham's analysis [10]. Let T_1 denote the total time of the computation related to a sequential schedule, and T_∞ the critical time on an unbounded number of identical processors. If the overhead O_S induced by the list scheduling (management of the list, process creation, communications) is not considered, then $T_p \leq T_1/p + t_\infty$, which is nearly optimal if $T_\infty \ll T_1$. This bound is extended to non identical processors by Bender and Rabin [1].

Workstealing is a distributed version of list scheduling that has been proven to be optimal for a class of programs called fully strict. In this case, with a high probability, each processor makes $O(T_\infty)$ steal attempts [4]. The total number of steal attempts made by p processors is bound by $O(p.T_\infty)$, which yields: $T_p \leq \frac{T_1}{p} + O(p.T_\infty)$. The fully strict model implies that a parent process be blocked until all of its spawned tasks return their results. It includes all Divide and Conquer parallel programs for example. Some parallel programming environment that implement a "Divide & Conquer" programming interface are for example Cilk [2, 3] and Satin [16, 17].

Three important characteristics motivate the use of this programming model:

1. some of the most rated parallel programming interfaces are based on this model;
2. its use allows to have some performance bounds on the schedules (using workstealing);
3. a large set of important applications can be efficiently programmed with such a model. The LU factorization, Branch and Bound search, or sorting are examples.

Workstealing (and list scheduling) only uses a basic information of "load" about the available processors in order to allocate tasks to them when they turn idle (or underloaded). Typically, workstealing uses the number of processes in the local waiting list of each processor to estimate its load.

Our scheduler is based on the assumption that the MPI-2 program is using a Divide and Conquer programming model: basically, the idea is to use a Cilk-like program, where the 'fork' construct would be substituted by the `MPI_Comm_spawn`, and the 'synch' by the `MPI_Finalize`. Processes migration is not allowed in this model, which is also non-preemptive.

3 A Scheduler for MPI-2 Programs

The scheduler is constituted of two main parts: a set of header files that re-define some of MPI-2's constructs at compile-time; and a scheduler daemon that runs during the execution of the application (the `mpirun` script has been tampered in order to run this extra process along with the "normal" application MPI processes). The overloaded primitives are used to enable the communication between the MPI processes and the scheduler, so that the latter may update its data-structure about the MPI computation and take the scheduling decisions.

The scheduler must maintain a task graph, in order to compute the best schedule of the processes. It is implemented in two modules: `sched` which is in charge of updating

the task graph; and `libbetampi` which implements the internal routines corresponding to the overloaded MPI-2 routines.

3.1 The Scheduler

The task graph is maintained as a generalized tree, where a node may have p children, p being the number of processes spawned by a parent. The implementation is made in the `graph` module. Each node in the tree points to an internal data-structure, `struct process_desc`, that represents a MPI process. Each process has a state, which can be `Blocked`, `Ready` or `Running`. To control the states of processes, the scheduler maintains lists that represent each state; it moves the processes from one list to another when the parallel program executes. In the current version, the scheduler does not control the states of processes but this functionality will be included in a future version. The overloaded MPI-2 primitives send (MPI) messages to the scheduler process to notify it of each event regarding the program. The scheduler waits for these messages, and when it receives one, it proceeds with the necessary steps: update of the task graph; evolution of the state of the process that sent the message; possible scheduling decision.

The scheduling decisions are to be taken:

- At process creation (as a result of a `MPI_Comm_spawn` call): the newly created process(es) has to be assigned a processor where it will be physically forked;
- At process termination (`MPI_Finalize`), since an occupied processor will be freed; an already existing process may start running;
- When new processor(s) get(s) available. In the current version, this is not contemplated.

Since neither preemption nor migration are used, no other event may require a scheduling decision between the creation and the termination of a process.

3.2 The Overloaded Primitives

To be consistent with the scheduling decisions, the MPI-2 primitives that require overloading are:

- `MPI_Comm_spawn`: the overloaded version has the following action: the parent process first sends a MPI message to the scheduler, informing the number n of processes that it wants to spawn, and its own pid. It then waits (with a blocking `MPI_Recv`) for a return from the scheduler.
  At this point, there is an important issue about the physical creation of processes (physical spawn), that may be done either by the parent process or by the scheduler. In the first case, the scheduler will decide of the location of the children and return the information to the parent process. After the creation of the children, the parent process can determine their pids and send them back to the scheduler, so that it may, later on, issue remote system call in order do priorize them. Thus, in this approach there are two communications between the parent and the scheduler.
  On the other hand, if the physical creation is done by the scheduler, it will decide the location of the children, physically create them, and use the inter-communicator

returned by MPI_Comm_spawn to locally determine the children's pids. Thus, the scheduler can definitely update its task graph. But then, it has to send the MPI_Comm_spawn return code back to the parent process, as well as the inter-communicator. This second option needs only one communication between the scheduler and the parent.

The current version of the scheduler has been implemented with the first option, where the physical spawn is done by the parent process. Figure 1 shows the steps of the overloaded MPI_Comm_spawn. First, the parent process will create new processes (children) through the MPI_Comm_spawn primitive (step 1). The overloaded primitive will establish a communication (step 2) between the parent and the scheduler, to notify the creation of the processes and the number of children that will be created (in the diagram, only one process is created). The scheduler updates the task graph structure (step 3), decides on which node the children should physically be created, and returns this physical location of the new processes (step 4). The parent process, that had remained blocked in a MPI_Recv, receives the location and physically spawns the children (step 5). It then enters into a blocking receive of a message from the scheduler, until all his children complete, so that the computation may be fully strict.

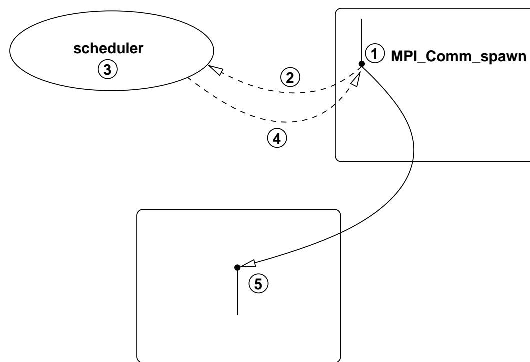

Fig. 1. MPI_Comm_spawn overload.

Notice that the creation of new processes is delayed until the scheduler decides where to execute them. This enables the manipulation of the (light) process descriptor data-structure, until there is some idle processor. Then, the scheduler may decide to allocate the created processes to this processor, and only then will the physical creation occur. Thus, the overhead of the heavy process creation is delayed until an otherwise idle processor may do it.

- MPI_Finalize: this serves to notify the scheduler that a process has terminated, and therefore that a processor will be idle. The MPI_Finalize just sends a message to the scheduler.

Figure 2 shows the MPI_Finalize overload. Step 1 represents the call of MPI_Finalize, where is send a message to scheduler (step 2) notifying the scheduler of the process completion. The scheduler updates the task graph structure (step 3) and, if there are processes waiting for a processor, it will unblock a process (shows in step 4).

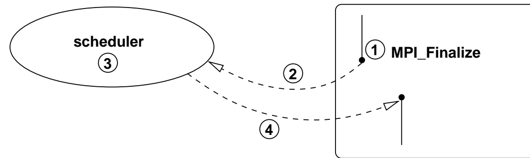

Fig. 2. MPI_Finalize overload.

- MPI_Init: in order to know if a MPI program is called as an "entry point" of the computation, *i.e.* directly run by mpirun or mpiexec, or as a spawned program (*i.e.* through MPI_Comm_spawn calls), the MPI_Init function is overloaded and tests the size of the MPI_Parent_group. It is zero if and only if the program has been "mpirun". In the other case, this call serves to get the parent communicator and merge it together with the program's MPI_Comm_world, so that all processes may communicate through an unique communicator.

From the scheduler point of view, the decisions taken are:

- when it receives a message from a parent process, the scheduler updates its task graph, associating the parent's pid to n processes children (the pid and n are the information contained in the message). It then decides on which nodes the n children will be created (the heuristics are detailed in Sec. 3.4), and send their locations to the parent. Afterwards, the scheduler will receive another message from the parent, with the pids of the children that have been created, in order to store them in the task graph.
- when it receives a message from a terminating process, the scheduler updates its task graph to delete the terminated process, and can take the appropriate scheduling decision; for instance, it could remotely contact the source processor of the message, to notify the process with the new highest priority that it can use the processor. Finally, it sends a message to the parent process, that was blocked in a receive that would notify it that its children had completed their computation.

3.3 The Task Graph Structure of the Scheduler

The scheduler needs to update the task graph of the application dynamically. This graph must allow for an arbitrary number of children for each element that will be known at execution time. To support this feature, the scheduler uses a rooted tree data-structure, with left-child, right-selling representation [5]. Each graph node has a pointer that will cast to a process_desc structure with the information about the MPI processes.

3.4 Scheduling Heuristics

The scheduler can apply scheduling heuristics in two levels: to schedule processes into resources and to priorize the execution of the processes that are ready to run. In the first level the heuristics find a good distribution of processes among the available resources. In the other level it can change the processes priority to get a better resource utilization and performance.

The LAM MPI-2 implementation provides a Round-Robin mechanism to distribute processes on the nodes through a special key, `lam_spawn_sched_round_robin`, that can be set into `MPI_Comm_spawn`'s `MPI_Info` argument. In order to specify the value of this information, the `MPI_Info_set` primitive is used. But this mechanism is only efficient when more than one process are created by the same `MPI_Comm_spawn` call. If only one process is created by the call into a loop structure (for example into a while), all the children processes will be allocated in the same resource. To bypass this restriction, our scheduler implements its own Round-Robin mechanism that is able to distribute the processes in the available resources. With this mechanism, when only one process is spawned by the call, the scheduler maintains information about the last resource that has received a spawned process and allocates the new process to the next available resource in the process topology ($new_resource = (last_resource + 1)\%total_resources$). If more than one process is spawned, then the MPI-2 standard solution is used. The advantage of this approach is that the distribution occurs transparently, without any change in the implementation of the application.

The second level of scheduling isn't implemented in the current version of the scheduler. The priority of the processes is left under the responsibility of the operating system's scheduler, on each node. But it is important to notice that it aims to execute fully strict applications. To make it possible to enforce a coherent execution, one has to provide a blocking mechanism to make the parent processes wait for the execution of their children. This is made through a blocking `MPI_Recv` into the overloaded `MPI_Comm_spawn`, that will wait until the scheduler sends a message (one by child), triggered by the children's `MPI_Finalize`. This approach guarantees a hierarchical execution where new processes have higher priority.

4 Programming with MPI-2: the Fibonacci Example

This section presents an example of how to program an MPI-2 application that dynamically spawns new processes. The example computes *Fibonacci* numbers and is programmed in a recursive way following this definition:

$$\text{fib}(n)\text{: } \begin{cases} \text{if } n < 2 \rightarrow \text{fib}(n) = n \\ \text{else fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \end{cases}$$

Although the Fibonacci sequence may seem somewhat artificial, its main interest is in the recursive computational scheme. It is frequently used to test Divide and Conquer parallel programs. The recursive calls will be implemented, in MPI-2, with the `MPI_Comm_spawn` primitive. The most technical decision when programming this recursive application is about the synchronization at the start and the termination of the processes. The MPI-2 primitive that spawns new processes takes as argument, besides

other information, the executable file name and the command line parameters. These parameters may be used to pass data to the starting process without exchanging additional messages, but this may not be convenient for complex data-types. In this case, the most portable way is to use normal message passing: the data is packed using a classical MPI data-type and sent as a message. On the Fibonacci example, the first method has been chosen, since only an integer has to be transmitted from the parent to the children.

The communication in MPI may be synchronous or not. In the contemplated case, if synchronous send or receives were used, deadlock could occur: for example, a synchronous send, in the parent, before spawning the children, would obviously prevent them from being created and therefore from receiving the data and match the parent's send. In the case of the receives in the parent from the children, one wants them to be synchronous, in order to implement a fully strict computation: the parent has to be blocked until all its children end up their computation and send their output back.

From the children's point of view, all they have to communicate is the result of their computation. They have to send it back to their parent, and this communication must be asynchronous in our implementation of the scheduler: remember that in order to block the parent process until the return of its children, the overloaded `MPI_Comm_spawn` blocks the parent into a receive. If the child process uses a synchronous send, it will never complete, since it would wait for the matching receive from the parent's side, who is busy waiting for a message from the scheduler.

Figure 3 presents the example code that shows how the synchronization was implemented, and this synchronization prevents any deadlock. `MPI_Comm_spawn` calls the executable `Fibo`, that includes the code segment of the figure 3. Notice that the `MPI_Comm_spawn` is a collective operation which imposes a synchronization among all processes in a same communicator (since the latter must be updated with the descriptors of the children processes). This feature does not influence the scheduling decisions, but may impact the overhead imposed by the scheduler. Yet, in the case of Divide and Conquer parallel programs, the children processes are recursively created from one unique parent an its communicator. Thus, in the context of this work, the synchronization occurs between one parent and each one of its children without any global synchronization.

5 Experimental Evaluation of the Scheduler

This section presents and analyzes the executions of two example programs with three different schedulers: the LAM scheduler, an scheduler directly embedded in the application and the proposed scheduler, discussed in Sec. 3. All tests have been made on a cluster of up to 20 Pentium-4 nodes dual, each one with 1 GB de RAM. The main purpose of these tests is to find out how the spawned processes are distributed on the processors, with each one of the three schedulers. Our claim is that the use of the proposed scheduler enables a good distribution of the spawned processes.

In the following, the section 5.1 presents a Fibonacci test-case designed with MPI-2 and some results and conclusions about this experiment. Afterwards, Sec. 5.2 shows a second benchmark that demonstrates the behavior of the schedulers in a situation that is more CPU-involved and which is highly irregular.

```
if (n < 2) {
   MPI_Isend (&n, 1, MPI_LONG, 0, 1, parent, &req);
}
else{
   sprintf (argv[0], "%ld", (n - 1));
   MPI_Comm_spawn ("Fibo", argv, 1, local_info, myrank,
                MPI_COMM_SELF, &children_comm[0], errcodes);
   sprintf (argv[0], "%ld", (n - 2));
   MPI_Comm_spawn ("Fibo", argv, 1, local_info, myrank,
                MPI_COMM_SELF, &children_comm[1], errcodes);
   MPI_Recv (&x, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
                      children_comm[0], MPI_STATUS_IGNORE);
   MPI_Recv (&y, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
                      children_comm[1], MPI_STATUS_IGNORE);
   fibn = x + y;
   MPI_Isend (&fibn, 1, MPI_LONG, 0, 1, parent, &req);
}
MPI_Finalize ();
```

Fig. 3. Part of MPI-2 code from the Fibonacci example.

5.1 The Fibonacci Test-Case with MPI-2

This implementation of the Fibonacci program is not designed for speed measurements, since it implies two recursive calls (following the exact definition) and could be implemented using only one recursion. Thus, the number $N(p)$ of spawned processes to compute fib(p) is exponential (it is trivial to obtain that $N(p) = 1 + N(p-1) + N(p-2)$, with $N(2) = N(1) = 1$, and thus $N(p) \geq \text{fib}(p) = \lceil \frac{\Phi^p}{\sqrt{5}} \rceil, \Phi = \frac{1+\sqrt{5}}{2}$.

In all experiments have been used the LAM-MPI distribution. To run the Fibonacci test-case, three different configurations have been used:

1. Simple calls to `MPI_Comm_Spawn` were issued, using only LAM's embedded scheduling mechanism. With the default provided `MPI_Info`, LAM uses the Round-Robin policy.
2. The `MPI_Info_Set` primitive has been issued before each spawn, not with the `lam_spawn_sched_round_robin` key, but directly with the hard-coded ID of the node onto which should run the process. This is the internal mechanism directly written in the source code. The node ID is computed to implement a simple Round-Robin allocation to the nodes. Notice that each process that issued a spawn computes the round-robin allocation from the node ID on which it is executing.
3. A proposed scheduler has been used, with the scheduling heuristic as described in Sec. 3.4 (Round-Robin), yet this time the scheduling decision is external to the source application.

First, Table 1 presents the schedules obtained when computing the 6th Fibonacci number with the three configurations using 5 nodes.

Table 1. Comparing different schedules: number of processes spawned on each node.

Environment	Node 1	Node 2	Node 3	Node 4	Node 5
fib(6) with LAM standard scheduler	25	0	0	0	0
fib(6) with embedded scheduler	8	4	8	2	3
fib(6) with proposed scheduler	5	5	5	5	5

In the first case (LAM's native schedule) all processes were spawned in the same node. The second case just changed the starting node and this is reflected by a non-constant number of processes allocated to each node. In the last case, our scheduler provides an effective Round-Robin distribution of processes among the nodes and a perfect load balance.

The question that remains is about the first case: if the LAM scheduler uses a Round Robin algorithm, should it not spawn processes on all nodes? The reason why this does not happen is that LAM does not keep scheduling information between two spawns. That means that LAM will always start spawning on the same node and only if multiple processes are spawned in the same call the processes will be balanced. This situation gets clearer observing Table 2 with an experiment that compares the result of spawning 20 processes in a single call, *vs.* in a loop of multiple, individual spawns (`MPI_Comm_Spawn`).

Table 2. Spawning 20 processes in 5 nodes using single and multiple spawn calls with LAM scheduler.

Environment	Node 1	Node 2	Node 3	Node 4	Node 5
20 spawns of 1 process	20	0	0	0	0
1 spawn of 20 processes	4	4	4	4	4

In order to stress the scheduler with a higher number of spawned processes, the execution of the computation of fib(13) has been used. It results in 753 processes. Table 3 shows the distribution of the processes among 5 nodes, obtained with our scheduler.

Table 3. Computing the 13th Fibonacci number with the new scheduler.

	Node 1	Node 2	Node 3	Node 4	Node 5	Total Number of Processes
fib(13)	151	151	151	150	150	753

Table 3 shows again the effect of our scheduler: besides the good load balance that has been reached, the proposed scheduler makes it possible to compute the 13th Fibonacci number, which is not practicable with the standard LAM mechanism: on our

experimental platform, LAM tries to run all the processes on a single node, reaches an internal upper bound on the number of processes descriptors that it can handle, and fails.

5.2 Computing Prime Numbers in an Interval

In this test-case, the number of prime numbers in a given interval (between 1 and N) is computed by recursive search. As in the Fibonacci program, a new process is spawned for each recursive subdivision of the interval. Due to the irregular distribution of prime numbers and irregular workload to test a single number, the parallel program is natively unbalanced.

Table 4 presents the distribution of the processes among 5 nodes when executing the computation in an interval between 1 and 20 millions, using LAM's native scheduler and the proposed one.

Table 4. Comparing LAM's standard scheduler and the proposed one: number of processes spawned on each node.

Environment	Node 1	Node 2	Node 3	Node 4	Node 5
LAM's standard scheduler	39	0	0	0	0
proposed scheduler	8	8	8	8	7

Table 4 shows, once more, the good load balance that has been reached with the proposed scheduler. Measuring the execution time, the average duration of the parallel program has been $181.15s$ using LAM's standard scheduler and $46.12s$ with the proposed scheduler. Clearly, the good load balance with our solution has a direct consequence about the performance of the application.

In this kind of application where the tasks are irregular, a solution that gathers information about the load on each node in order to decide where to run each process should be more efficient. Future work on the proposed scheduler should tackle this issue.

6 Conclusion and Future Work

The implementation of MPI-2 is a new reality in distributed programming, which permits the use of MPI's based HPC codes with new infrastructures such as computational grids. However, the diversity of programming models that can be supported by MPI-2 is difficult to match with efficient scheduling strategies. The approach presented in this paper is to restrict MPI-2 programs to fully strict computations, which enable the use of Workstealing.

This article has shown how MPI-2 can be used to program with such a model, and how it can be coupled with a central scheduler. Some preliminary tests have been presented, that show that LAM MPI's native scheduling functionalities are clearly outperformed by such a solution. Although a distributed solution would be much more

scalable, this centralized prototype results in a simple implementation and already validates the interest in such a scheduler of dynamic spawned processes in MPI.

It is therefore interesting to continue the development of such a scheduler, to implement a real workstealing algorithm: an easy way to do it is to decide on which processor to execute the processes, based on information about their respective loads. Future work could also include altering the priority of the processes on each node, through remote system calls, to control the execution of the parallel, dynamic program.

Special thanks: this work has been partially supported by HP Brazil.

References

1. A. M. Bender, , and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. In *Theory of Computing Systems, Special Issue on SPAA '00*, volume 35, pages 289–304, 2002.
2. M. A. Bender and M. O. Rabin. Scheduling cilk multithreaded parallel programs on processors of different speeds. In *Twelfth annual ACM Symposium on Parallel Algorithms and Architectures - SPAA*, pages 13–21, Bar Harbor, Maine, USA, 2000.
3. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. C. E. Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, Aug. 1995.
4. R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.
5. T. H. Cormen, C. E. Leiserson, and R. L. R. ans Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.
6. D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, Jan. 1991.
7. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.
8. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
9. F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, October 1998.
10. R. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.
11. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, Oct. 1994.
12. W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, USA, 1999.
13. N. Maillard, R. Ennes, and T. Divério. Automatic data-flow graph generation of mpi programs. In *SBAC'05*, Rio de Janeiro, Brazil, November 2005.
14. S. Moore, F. Wolf, J. Dongarra, S. Shende, A. D. Malony, and B. Mohr. A scalable approach to mpi application performance analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 309–316. Springer, 2005.

15. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, Dec. 1990.
16. 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 Lecture Notes in Computer Science, pages 690–699, Munich, Germany, Aug. 2000. Springer.
17. 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.