On the Advantages of an Alternative MPI Execution Model for Grids[*]

A. C. Sena[†], A. P. Nascimento[‡], J. A. da Silva[§], D. Q. C. Vianna, C. Boeres and V. E. F. Rebello
Instituto de Computação, Universidade Federal Fluminense (UFF),RJ, Brazil
e-mail:{*asena,depaula,jacques,dvianna,boeres,vinod*}*@ic.uff.br*

Abstract

The MPI message passing library is used extensively in the scientific community as a tool for parallel programming. Even though improvements have been made to existing implementations to support execution on computational grids, MPI was initially designed to deal with homogeneous, fault-free, static environments such as computing clusters. The typical programming approach is to execute a single MPI process on each resource. However, this may not be appropriate for heterogeneous, non-dedicated and dynamic environments such as grids. This paper aims to show that programmers can implement parallel MPI solutions to their problems in an architectural independent style and obtain good performance on a grid by transferring responsibility to an application management system (AMS). A comparison of program implementations under a traditional MPI execution model and a fine-grain model highlight the advantages of using the latter.

1. Introduction

In the last decade, exploiting parallelism in computing has become essential as applications seek more computational power to obtain solutions in acceptable run times. Although computers have became exponentially faster, application demands have also increased in terms of data storage, memory, networks bandwidth and processors. As a consequence of technological advances, the design of more complex applications is permitting research in new areas of science to be undertaken. The maturity of cluster computing, together with the advent and rapid development of grid computing, has caught the eye of the scientific community. By providing more cost effective computing power, they are now within the reach of nearly every researcher and have become the computing platforms of choice.

[*]This work is funded by FAPERJ and CNPq.
[†]Funded by a postgraduate scholarship from CNPq.
[‡]Funded by a postgraduate scholarship from FAPERJ.
[§]Funded by a postgraduate scholarship from CAPES.

Programmers typically view computing clusters as consisting of a fixed number of fault-free homogeneous resources interconnected on a fast local network. Since the objective is to maximize performance, resources are usually dedicated to the execution of one parallel application at a time. In addition, it is commonly presumed that failures are unlikely during execution so little or no provision is made for applications to tolerate faults. In consideration of these characteristics, numerous tools have been developed to facilitate the development of parallel programs. The MPI library [11] has become the *de facto* message passing standard among parallel application developers. In most MPI programs, a fixed number of processes are created at startup, one per resource, and each execute for the entire duration of the application [6]. Thus, designing programs for regularly structured problems can be relatively effortless. However, many parallel applications tend to be irregular, have dynamic computing demands, and thus require scheduling/load balancing and latency hiding techniques in order to execute efficiently.

On the other hand, grid environments are composed of numerous heterogeneous resources typically dispersed across wide area networks. Belonging to different organizations, resources may have different access policies and unpredictable availability causing them to enter or leave the grid without prior notice. The computing power available to an application can even be variable if resources are being shared with local or other grid users. Many grid applications are designed to run for days or weeks at a time and, being longer than the mean-time between failures of grid resources, faults are not only probable but should be expected. In comparison with cluster environments, all these issues make computational grids extremely more complex for parallel applications to utilize effectively. If parallel applications are to harness the potential of grids, the applications designers must be aware of these particularities, and create applications that adapt efficiently. Addressing these issues using the predominant MPI programming model requires significant programming effort.

Implementations of MPI for grids are currently limited to the Globus Toolkit-enabled MPICH-G2 [12], LAM/MPI [10] and MPICH-GF [19] libraries. While these aim to facilitate the uptake of grid computing by allowing existing cluster-based MPI programs to run on the grid without modification (recompilation is necessary), they lack the capability to deal with dynamic heterogeneous environments efficiently. MPICH-G2 [12], being an extension of the cluster-based MPICH implementation, does not support dynamic process creation. While this is addressed in LAM/MPI, MPICH-GF uses dynamic process creation to implement fault tolerance transparently.

This paper compares the performance of a typical MPI program implementation based on an architecture dependent execution model (where the number of MPI processes is proportional to the number of processors available) with one based on an architecture independent execution model (where the number of MPI processes is proportional to the degree of parallelism in the application). The advantages of this fine-grain model for MPI applications when running on a grid under the management of an AMS is highlighted. The EasyGrid AMS (based on an existing standard MPI implementation) is embedded into the application when the user's cluster-based MPI program is recompiled, allowing it to execute in an efficient and robust manner. Even though the EasyGrid AMS incurs an overhead to manage the execution of significant larger number of MPI processes, the execution model offers the middleware the opportunity for better performance and more cost effective robustness in dynamic and heterogeneous grid environments.

2. Related Work

In order to develop their parallel programs, designers must follow a suitable parallel programming model. Theoretical models, like PRAM [5] whose goal is to help compare and evaluate the design of parallel algorithms, allow designers to abstract away details of the system architecture. However, this very abstraction makes the model inadequate for practical use in real world applications. In an attempt to be more general, the *Bulk-Synchronous Parallel* (BSP) model [18] addresses both theoretical and practical issues. For programming simplicity, the model assumes synchronous execution across all resources, but captures the costs associated with computation and communication. In the same vein, the *Coarse Grained Multicomputer* (CGM) model [1] enforces an additional restriction that the computational phase must be larger than communication one. Both a BSP and CGM algorithms consist of a sequence of *steps* or rounds, which are made up of well defined local computation and global communication phases.

Interestingly, a key characteristic of these and most other models is that they specify or implicitly imply that each process is mapped to a virtual processor for the duration of the execution of the parallel application. This is perfectly acceptable for homogeneous clusters with a sufficient number of processors. However, in the case where virtual processors outnumber physical processors or heterogeneous resources are employed, the best allocation of processes to processors may not be clear. In dynamic grid environments, this execution model may not be the most appropriate.

Due to technological motivations, experience and simplicity, the majority of parallel program designers chose the messaging passing programming approach, where multiple tasks can be defined, each associated with local data, and which interact amongst themselves through the exchange of messages. The fact that the SPMD model is more commonly used in the scientific community is indicative of the need to simplify the exploitation of parallelism.

Charm++ [9] is a parallel programming language based on C++, in which programs are decomposed into a number of cooperating message-driven objects. These objects are mapped to physical processors by an adaptive runtime system. The runtime system transparently supports load balancing and fault tolerance strategies through the automatic checkpointing and migration of objects. Due to the popularity of MPI, researchers developed an implementation, AMPI [7], based on Charm++. MPI processes are implemented as light-weight migratable objects (or threads), which are then managed by Charm++ runtime system. Converting an MPI program to an AMPI one however may require some modifications.

The execution model adopted by the EasyGrid AMS follows a similar application centric philosophy, but goes one step further, dividing the sequence of *steps* of an individual process into tasks (which are implemented as MPI processes). This model permits dynamic re-scheduling and fault tolerance without the need for process migration and checkpointing, respectively. For deployability, the EasyGrid AMS is based on the widely available standard implementation of LAM/MPI.

3. A MPI Execution Model for Grids

Being heterogeneous, shared and dynamic environments, all of these characteristics combined make grids extremely difficult programming targets for scientists entirely inexperienced in the nuisances of these systems. In order to relieve programmers of this burden, and fundamental to the success of grid computing, middlewares are being developed to hide these complexities [2, 3, 8, 17]. Given the dynamic behavior of both grids and applications, these middlewares are designed to manage the execution of applications, determining their requirements, and decide on the most appropriate allocation of resources. During execution, they are able to adapt the application, for example by rescheduling processes to improve performance and recovering or restarting processes when failures occur.

With the availability of efficient grid management systems, programmers can concentrate their efforts on designing parallel programs that execute efficiently in a hypothetical ideal environment [6]. In this way, only one solution that maximizes the parallelism exposed in the problem needs to be designed, independent of the computing platforms on which it will execute on both now and in the future.

SPMD MPI applications are typically designed to execute identical long running processes, one per homogeneous processor [6]. In dynamic grid environments, this *one process per processor* (1PProc) execution model becomes inappropriate since the granularity of each process will require adjustment. The fact that resources maybe heterogeneous, shared with local jobs, and fault-prone not only makes this model utterly inefficient but also makes managing the execution of applications extremely complex and expensive in terms of computation and storage.

If the maximum parallelism is exposed when designing a parallel program, the application may become *fine grained*. An application is said to be fine grained if the average process computation time is smaller than the average communication time, otherwise, the application is coarse grained. Although fine-grained processes are sequences of code that cannot be parallelized further, they can be divided into *basic blocks* delimited by communications (initiated by receives and finalized by sends) and thus defining application *tasks*. Typically, due to the high(er) communications costs in a distributed environment, the performance of fine grained application may be poor, especially if the allocation of tasks is inappropriate. Clustering mechanisms can be used to map tasks to the same processor in accordance with their dependencies, thus decreasing communication costs. For grid management systems, clustering processes (tasks) appropriately is easier than either extracting parallelism automatically from an application or adjusting process granularity, at runtime. Under a *one process per task* (1PTask) execution model, even though managing a large number of processes can incur a significant additional overhead, grid management systems can manipulate with greater ease tasks to improve performance given the need to migrate processes and to recover from failures in this dynamic environment. It should be noted, however, that the 1PTask execution model is hampered by the fact that the number of processes that can be created statically in standard installations of MPI is limited to a couple of hundred [14].

The size of an MPI process directly affects the design of the scheduling and fault tolerance mechanisms that should be employed. Coarse grained applications with long running processes must be executed in an environment that provides an efficient checkpointing mechanism, if the re-execution of processes is to be avoided. Checkpointing requires sophisticated and costly schemes to guarantee consistency and allow MPI processes rollback their execution to their last recorded state [4]. However, the penalty suffered to re-execute failed short-lived processes may be sufficiently small enough to dispense with a checkpoint-based scheme. Since all processes are already in execution under the 1PProc model, any change in the environment which affects performance (e.g. processor failure, external workload) will require the application to adjust through process migration (i.e. the process must be stopped, its context transferred and the process restored on the new resource).

With the increasing demands for grid resources, their efficient utilization is fundamental. Programmers wish to explore the benefits of grid computing without having to be concerned its dynamic behavior. Programming in an architecturally independent style not only simplifies the programming effort but make the grid more accessible to less experienced programmers. However, in order to achieve high performance, it is imperative to employ a management system capable of dealing with the characteristics of grids accordingly and thus adapting MPI application transparently. The execution model adopted by such management systems directly influences their efficiency.

4. EasyGrid AMS

The EasyGrid middleware is an Application Management System (AMS) for any MPI implementation with dynamic process creation. The EasyGrid AMS is automatically embedded into the MPI parallel application without modifications to the user's original code at compilation time. It transforms applications into autonomic versions, capable of adapting their execution in accordance with changes in the grid environment [13, 14]. These applications have high deployability since the EasyGrid AMS is not dependent on other grid system middleware, needing only the Globus Toolkit and the standard LAM MPI library [10] to be installed on grid resources.

Each MPI application has its own EasyGrid AMS (tuned to the specific needs of the application), which is a three level hierarchical management system composed of: a single *Global Manager* (GM), at the top level that supervises the sites in the grid where the application is running; at each site, a *Site Manager* (SM) is responsible for the allocation of the application processes to the resources at the site; and finally, the *Host Manager* (HM), one for each resource, takes on the responsibility for the scheduling, creation and execution of application processes allocated to that respective host. Management and application processes may execute on the same processor, competing for CPU time. However, the intrusion of the management processes is minimal because they behave like event-driven daemons.

Each management process is based on a subsumption architecture and is composed of four layers: application monitoring, process management, dynamic scheduling and fault tolerance. Each layer has a distinct function but can modify the behavior of a lower one. The dynamic creation of MPI processes and the routing of messages between them are performed by the *process management* layer. The proactive *dynamic scheduling* and *fault tolerance* layers utilize status information provided periodically by the *monitoring* layer, to decide if, and when, it is necessary to activate a re-scheduling mechanism and to detect and treat process and resource failures.

Although the AMS follows the 1PTask execution model, and unlike the standard execution of MPI programs, the AMS does not create all application processes at once. Their creation is carried out dynamically and is carefully orchestrated according the AMS's scheduling policy. Since this is transparent to the user's MPI program, the AMS must also manage all communications between processes (e.g. it is possible for a process to send a message to another that has yet to be created). To minimize overheads, the EasyGrid AMS only re-allocates those processes which have yet to be created, processes in execution are never migrated.

The EasyGrid AMS employs a hybrid scheduling approach which combines the benefits of static and dynamic scheduling [13]. The static scheduler can use more sophisticated heuristics, while the dynamic scheduling subsystem, executing concurrently with the application, can make its scheduling decisions based on more precise runtime information. This subsystem is distributed in the dynamic schedulers associated with each of the management processes in the three level AMS hierarchy. This architecture allows each application to have its own scheduling policies, and that different policies be used at different levels of the hierarchy or within the same level. This approach allows the application to match its requisites with the characteristics of the available grid environment [13].

The EasyGrid AMS's management of processes permits the recovery from faults without the need to interrupt the execution of the application [14]. The AMS uses distinct inter-communicators between each pair of processes (for both management and application processes) so that faults can be easily identified and isolated. The AMS implements both *retry* and *alternative resource* techniques together with a *message log* to provide a means to recover from both process and processor failures [14]. In the case of failures, processes are re-executed since no checkpoints are carried out.

5. Experimental Analysis

5.1. Case Study Applications

Although the EasyGrid AMS can be used in conjunction with a variety of classes of MPI parallel applications, this paper investigates the benefits of executing a MPI parallel implementation for CPU-bound master-worker applications under the 1PProc and 1PWork execution models. This class of application was chosen for three reasons; this is a commonly adopted paradigm for implementing data parallel programs [6, 15]; a version of this strategy which dynamically adjusts the workload given to each processor is often used in dynamic heterogeneous environments; finally it has a communication pattern that stresses the EasyGrid AMS management hierarchy.

The master-worker application analyzed in this work, denoted as *MWork*, is a synthetic one where the amount of work executed by each MPI process can be controlled in order to evaluate the performance in relation to the application's granularity. The minimum amount of work performed by each worker is a *workload unit*.

In traditional MPI master-worker implementations, where one MPI process per processor is created statically [6], one master process distributes *workload units* among the remaining worker processes. Here we consider three different implementations: (1) static, where the master process distributes all of the *workload units* evenly among the worker processes, assuming they are executing on homogeneous resources; (2) static enhanced, where the master process distributes the *workload units* among the worker processes proportionally (determined apriori) to the computational power of their respective resources; and (3) on demand, where the master process distributes *workload units* to worker processes as they become idle. These MPI implementations are referred to as: *MWork* S-MPI, *MWork* SE-MPI and *MWork* OD-MPI, respectively. These *MWork* MPI versions are not executed under the EasyGrid AMS and the total amount of *workload units* that the master should distribute is predefined.

In the *MWork* implementation managed by the EasyGrid AMS and denoted by *MWork* AMS, each worker process executes only one workload unit. Although the total number of worker processes is predefined (equivalent to the workload of the application) and their allocation is specified *a priori* by the AMS static scheduler, each MPI worker process is created dynamically by the AMS on the resource considered most appropriate by the global and site managers and at a time determined by that resource's host manager. The master communicates with each worker indirectly via the AMS management processes.

In addition, a real world SPMD parallel application which computes macroscopic thermal dispersion in a porous media (the *Thermions* application) [16] is also analyzed. Given that a porous media is composed by solid and fluid elements, the thermal dispersion is evaluated by the movement of a large number of hypothetical particles, called *Thermions*, from a fixed release point, through the media. The position, energy, a random component, and the thermal properties of the solids or the flux velocity of the fluid determine the distance traveled within a period of time, by each individual thermion. The computational cost to calculate the path for each thermion varies and cannot be precisely determined *a priori*. For this application, the *workload unit* corresponds to the determination of the path of a single thermion. Two versions of the *Thermions* were evaluated: an on demand MPI implementation, called as *Thermions* OD-MPI; and one executed under the EasyGrid AMS, denoted as *Thermions* AMS.

5.2. The Grid Environment

The experiments were carried out on a geographically distributed, four site grid environment interconnected by gigabyte and fast Ethernet switches. All the available processors run Linux Fedora Core 2, Globus Toolkit 2.4 and LAM/MPI 7.0.6. Sites 1, 2 and 3 are composed of 28 Pentium 4 2.6 GHz processors and 3 Pentium 4 3.2 GHz processors (2 in Site 2 and 1 in Site 3), where Site 1 and 3 contains 8 processors each and Site 2, 15 processors. Site 4 is composed of 22 Pentium II 400 MHz processors.

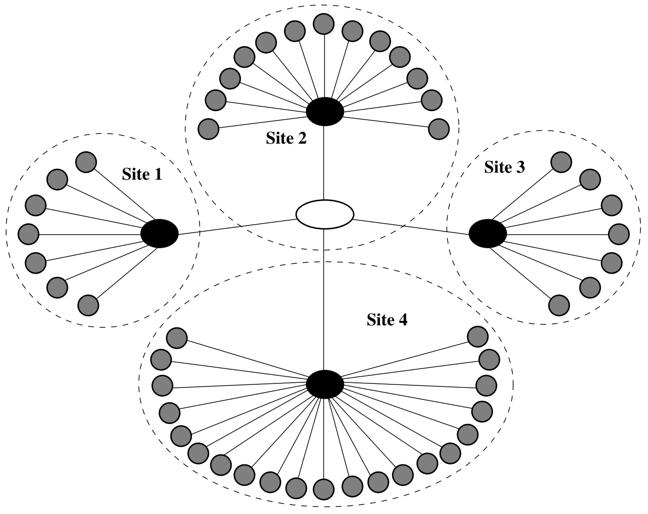

Figure 1. The EasyGrid AMS configuration.

Of the given configuration, 53 resources were available for execution, but only 52 execute *worker* processes. One of the resources located at Site 2 was chosen to execute the *master*. The EasyGrid AMS management structure for this application is presented in Figure 1. The *white* ellipse is chosen to execute only the GM and *master* processes, while each one of the remaining resources executes a HM process that manages the creation and execution of each worker process allocated to their respective host machine. Additionally, each resource shown in *black* executes a SM to manage the execution at that site. Thus, the resources in *black* have an extra overhead produced by the execution of two management processes (SM and HM).

5.3. Heterogeneous Resources

The purpose of the first set of experiments is to analyze the impact of resource heterogeneity on the performance of the four MPI implementations. Three execution environments are compared: Scenario (1) represents a dedicated and quasi-homogeneous cluster, with only resources from Site 2 being used; Scenario (2) represents a heterogeneous but static computational grid, composed of resources from Sites 1, 2 and 3; and Scenario (3): a heterogeneous and dynamic grid, again composed of resources from Sites 1, 2 and 3. As the resources of these sites have almost the same clock speed, heterogeneity was introduced by running external CPU-bound programs concurrently with the *MWork* applications. The resources of Site 2 provide 100% of their computational power to the application, while Sites 1 and 3 only provide 25% and 50%, respectively. To increase heterogeneity inside the sites, two machines from each site were chosen to run extra CPU-bound programs, and so only deliver half of the computational power offered by the remaining machines of that respective site. In Scenario (3) these extra CPU-bound programs migrate to different machines during the execution of the *MWork* applications, to represent dynamic grid behavior.

Table 1 presents the average execution times of three executions of each *MWork* implementation for a total of 500 uniform workload units. Three separate instances were also run for different sizes of workload unit equivalent to 5, 10 and 20 seconds on an idle P4 2.6 GHz machine.

Table 1. Comparison of the executions times obtained by the *MWork* MPI and *MWork* AMS (in seconds)

cluster scenario	*wload unit*	*MWork MPI* S	*MWork MPI* SE	*MWork MPI* OD	*MWork* AMS
(1)	5s	180.36	180.36	180.30	181.97
	10s	359.56	359.56	359.01	362.67
	20s	718.54	718.54	717.06	723.81

grid scenario	*wload unit*	*MWork MPI* S	*MWork MPI* SE	*MWork MPI* OD	*MWork* AMS
(2)	5s	643.78	140.39	157.99	141.79
	10s	1312.32	279.98	316.37	282.56
	20s	2632.91	559.56	634.80	563.88

grid scenario	*wload unit*	*MWork MPI* S	*MWork MPI* SE	*MWork MPI* OD	*MWork* AMS
(3)	5s	492.03	210.08	157.85	142.66
	10s	986.22	420.66	317.13	282.97
	20s	1943.10	838.16	633.54	573.44

The results obtained for Scenario (1) shows that even in an homogeneous and dedicated environment, that is completely favorable to MPI 1PProc execution model, the

MWork AMS achieves very good execution times considering the overheads to manage the execution of each worker process, one at a time on each processor, as well as monitor for failures. The difference was only 1% at most from the best values obtained by *MWork* MPI.

In Scenario (2), the best execution times were achieved by *MWork* SE-MPI and *MWork* AMS. In this case, as the environment is static and not shared, the estimates given to *MWork* SE-MPI allow the algorithm to optimally divide the workload before execution and thus obtain good performance. Note however that this information must be provided by the programmer or some other middleware capable of verifying the computational power of available resources and the master must be coded to balance the workload distribution. All this extra work does not lead to an efficient execution when the environment is dynamic, as shown in Scenario (3). In this scenario, common in grid environments, the best results were achieved by *MWork* AMS thanks to its ability to efficiently and proactively redistribute the workload accordingly. Here *MWork* OD-MPI performs better than *MWork* SE-MPI since the former is able to adjust to variations in the computational power of resources.

The benefits of the EasyGrid AMS management are not only to provide efficient and robust execution in grid environments, but also to make the MPI programmer's task easier. As the *MWork* OD-MPI, shows the best performance among the MPI implementations for grid platforms, the following experiments only compare the *on demand* MPI implementation with the AMS one on the full *four site* grid environment. In each experiment, extra CPU-bound programs were executed on four different machines (1 in Site 1, 1 in Site 3 and 2 in Site 4).

The results of a second set of experiments, seen in Table 2, show the average execution times obtained by running the *MWork* applications with 1000 non-uniform workload units. In this case, 50% of the workload units required 20 seconds of computing time, 25% 10 seconds, and the remaining 25% required 5 seconds.

Table 2. Comparison of the executions times achieved by the *MWork* with non-uniform workload units (in seconds)

Random *MWork* OD-MPI	*Sorted MWork* OD-MPI	*MWork* AMS
584.45	538.43	438.41

Two OD-MPI implementations are considered. The first does not have knowledge of the workload unit distribution. In this case, five different random orders were specified and each executed three times so that the average execution time appears in column Random *MWork* OD-MPI. In the second version, denoted in Table 2 as Sorted *MWork* OD-MPI, workload units were delivered by the master in non-increasing order of their size as specified by the programmer. In the case of the *MWork* AMS, both the static and dynamic schedulers sort the tasks in non-increasing order of their computation cost. The results in Table 2 shows that the average execution times achieved by both OD-MPI versions are worse than those obtained by *MWork* AMS.

It is important to clarify that the grid environment used in the previous experiment was dedicated, stable and with resources of various fixed computational powers. The EasyGrid AMS deals very nicely with the heterogeneity of the system automatically, since its schedulers have the ability to adjust when and where a chosen process should be executed. This is achieved efficiently by monitoring computational power being delivered to the application by the available resources, and predicting the *completion times* of processes waiting to be executed. The same cannot be said of *MWork-OD* MPI, since it employs a greedy approach - delivering the next *workload unit* to the worker that finishes first independent of the resource on which it is executing - minimizing the workload's start time rather than its finish time. Smarter implementations of traditional MPI applications are possible, but the intricacies of the grid environment should not be the responsibility of the grid programmer.

Table 3. Comparison between OD-MPI and AMS-based implementations of the *Thermions* application (in seconds)

N	*Thermions* OD-MPI	*Thermions* AMS
1000	136.08	130.73
3000	377.18	373.20
5000	619.88	615.84
10000	1355.56	1302.56

The following experiment investigates the scalability of the AMS approach with a large-scale, real world application. The results, reported in Table 3, were obtained for executions of the *Thermions* application with different numbers, N, of thermions. The computational cost of each thermion is difficult to predict *a priori* and varies between 4 and 4.5 seconds on an idle 2.6 GHz machine. The *Thermions* AMS must manage the execution of more than N processes (including the management processes themselves). The applications were executed on the four site grid with dedicated resources. The results show that the 1PWork execution model is competitive in this stable environment.

5.4. Grid Dynamism *versus* Different Policies

To evaluate the performance in dynamic environments, the following experiments were carried out considering the same grid environment but with different resource usage policies. For these tests, Sites 2 and 3 were dedicated to the experiment, while Sites 1 and 4 were shared with other applications. The resource usage policy of Sites 1 and 4 only permit resources to accept grid jobs when they are idle.

The experiments were semi-controlled, initially a CPU-bound program was launched on each machine of Sites 1 and 4, so that only resources in Sites 2 and 3 were available to the *Thermions* applications. Furthermore, on average 10 machines from Site 4 were being used by local users. The *Thermions* OD-MPI could only execute on Sites 2 and 3 because its processes are created at startup and only on the resources available. At a certain point during the execution of each *Thermions* instance, all of the CPU-bound programs that were initially launched in the Sites 1 and 4 complete. The *Thermions* AMS management processes are capable of detecting that all the resources of Site 1 and some from Site 4 have become idle, and so redistribute some of the worker processes to these resources.

Table 4 presents the average of the execution times obtained by the *Thermions* applications with 1000 workload units. The *Thermions* OD-MPI results are shown in the first column, while the results achieved by the execution of *Thermions* AMS, under four different scenarios, are presented in the remaining columns. Each scenario indicates the approximate time that the shared resources were made available to the *Thermions* processes, counting from the beginning of its execution: (1) just after the initialization of the application; (2) after 50 seconds; (3) after 100 seconds; (4) and never.

Table 4. Comparison of the executions times achieved by the *Thermions* in shared environments (in seconds)

Thermions	*Thermions* AMS			
OD-MPI	(1)	(2)	(3)	(4)
197.87	139.78	152.57	170.22	199.32

The execution time for *Thermions* OD-MPI was better than the *Thermions* AMS only in scenario (4), where the grid available (resource in Sites 2 and 3 only) is ideal for the standard MPI model: stable, dedicated and homogeneous. As before, the 1% difference is due to AMS overheads to initialize the management structure and implement the execution model. On the other hand, the results obtained by the *Thermions* AMS in scenarios 1, 2 and 3 show that the middleware is able to deal well with the grid's dynamic and shared behavior and adapt to take advantage of different resource usage policies. Note that scenario 1 gives a larger execution time than the corresponding experiment in Table 3. This is due to the fact the AMS must detect and be sure that each resource has really become idle.

5.5. Grid Processes Failures

Grids systems are more prone to failures than traditional high performance environments like supercomputers and clusters. The purpose of the next experiment is to evaluate the performance of the *MWork* AMS when multiple process failures occur. This analysis highlights the benefits of creating workers dynamically and scheduling them carefully. In this experiment, when a process fails, its execution will be re-started from the beginning, since no checkpoint mechanisms are being considered.

The following execution times on the four site grid were obtained running the *MWork* AMS and *MWork* OD-MPI with 1000 *workload units* of 5 seconds each. The results achieved with no process failures were 164.00 and 202.65 seconds, respectively. Three failure scenarios (S) were then evaluated: 1% of the application processes failed (10 processes); 3% of the application processes failed (30 processes); and 5% of the application processes failed (50 processes). To simulate processes failures, a script killed the respective number of processes in five different resources distributed across Site 1, 2 and 3. Also, the total number of process failures was evenly distributed among these 5 resources. Two different situations were considered with regard to the moment at which the processes started to be killed: 50 seconds after application startup; and 100 seconds after. The processes on each of the chosen resources were killed at intervals of 6 seconds. Thus, for example, if a total of 50 processes fail (10 per processor), the application would suffer process failures during 60 seconds approximately on five resources.

Figure 2 presents the performance of each of the three failure scenarios. Two important characteristics of the fault tolerance mechanisms embedded in the *MWork* AMS can be highlighted. One, the penalty of re-executing these processes is very low, proportional to the quantity of failures and to the average workload lost (2.5s). Although an overhead is incurred on the resources where the failures occurred, the extra amount of time spent with process re-execution was efficiently dealt with by the dynamic scheduler. Another conclusion is that, the results show that the performance of the application does not depend on the moment of failure, so long as the dynamic scheduler has time to balance the workload evenly. Even in an extreme situation (5% of failures occurring after 100 seconds), when the last process failure occurred almost at the end of the execution, the increase in the execution time was less than 1.1%.

Figure 2 also presents the theoretical performance of the *MWork* OD-MPI application given that since processes typically share the same MPI communicator, should one process fail the whole application fails and therefore requires to be re-executed. Application failure could be overcome if the programmer uses distinct communicators, however, the resources on which processes fail can only be used after failure if the master process is able to recreate worker processes dynamically.

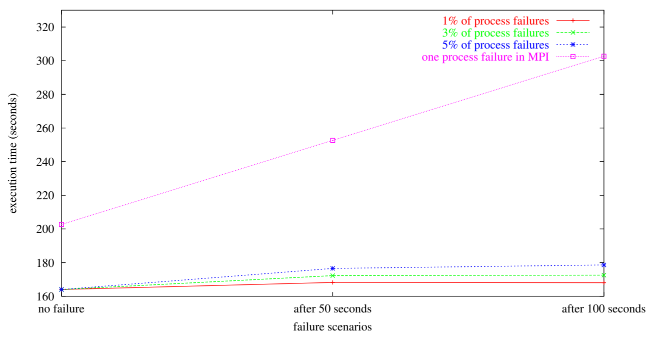

Figure 2. Application performance with failures.

6. Concluding Remarks

Traditional implementations of MPI applications by the majority of programmers (typically scientists) are based on an execution model that is efficient in stable and dedicated homogeneous environments such as clusters or supercomputers. Since grid platforms have quite the opposite characteristics, this work proposes that parallel MPI applications adopt an alternative execution model - a model which allows the programmer to develop their applications in an architectural independent manner. Allowing programmers to focus their time and energy solely on their application problem (maximizing the parallelism within it), and ignoring the complexities of the target architecture, will greatly facilitate the design of better algorithms and new, larger applications that require grid computing.

Of course, the gridification burden has now been passed to a more knowledgeable grid middleware designer. Few MPI management systems exist to deal with the intricacies of grid environments. One such system is the EasyGrid AMS, an application-level middleware, based on standard LAM/MPI, capable of managing the execution of MPI applications in grids or clusters more efficiently than traditional implementations. These improvements are derived ultimately from the fact that the application is executed by the AMS according to *one process per task* execution model. Versions of the EasyGrid AMS for other classes of applications are being developed and optimizations to reduce the overheads further are being investigated.

References

[1] C. E. R. Alves, E. N. Caceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proc. 14th Symposium on Parallel Algorithms and Architectures (ACM-SPAA)*, pages 275–281. ACM Press, 2002.

[2] C. Boeres and V. E. F. Rebello. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience*, 16(5):425–432, April 2004.

[3] R. Buyya and S. Venugopal. The gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *Proc. of the 1st IEEE Int. Workshop on Grid Economics and Business Models (GECON 2004)*, pages 19–66, Seoul, Korea, April 2004. IEEE Computer Society Press.

[4] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[6] I. Foster. *Designing and Programming Parallel Programs*. Addison-Wesley, 1995.

[7] C. Huang, O. Lawlor, and L. V. Kale. Adaptive MPI. In *Proc. 16th Int. Workshop on Languages and Compilers for Parallel Computing*, pages 306–322. Springer, 2003.

[8] E. Huedo, R. S. Montero, and I. M. Llorente. The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience*, 6(3):1–8, September 2005.

[9] L. V. Kale and S. Krishnan. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In A. Paepcke, editor, *Proc. Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108. ACM Press, September 1993.

[10] LAM/MPI Parallel Computing. `http://www.lam-mpi.org/`, Last access 25/04/2006.

[11] Message Passing Forum. MPI: A Message Passing Interface. Technical report, University of Tennessee, 1995.

[12] MPICH-G2. `http://www3.niu.edu/mpi/`, Last access 27/09/2006.

[13] A. P. Nascimento, A. C. Sena, C. Boeres, and V. E. F. Rebello. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience*, Published online Nov. 2006.

[14] A. P. Nascimento, A. C. Sena, J. A. da Silva, D. Q. C. Vianna, C. Boeres, and V. Rebello. Managing the execution of large scale MPI applications on computational grids. In *Proc. 17th Int. Symp. on Computer Architecture and High Performance Computing (SBACPAD 2005)*, pages 69–76, Brazil, October 2005. IEEE Computer Society Press.

[15] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1996.

[16] H. Souto, O. da Silveira Filho, C. Moyne, and S. Didierjean. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics*, 21(2):513–544, 2002.

[17] S. Vadhiyar and J. Dongarra. Self-adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, February 2005.

[18] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[19] N. Woo, H. S. Jung, H. Y. Yeom, T. Park, and H. Park. MPICH-GF: Transparent checkpointing and rollback recovery for grid-enabled MPI processes. *IEICE Transactions on Information and Systems*, E87-D(7):1820–1828, July 2004.