Proc. 6th International Conference on Parallel Processing and Applied Mathematics
R. Wyrzykowski, J. Dongarra, N. Mayer and J. Wasniewski (Eds.), Poznan, Poland,
September 2005, LNCS, vol. 3911, Springer Verlag, Berlin, 2006, pp. 258-271

An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments

Kaoutar El Maghraoui, Boleslaw K. Szymanski, and Carlos Varela

Rensselaer Polytechnic Institute, Troy, NY 12180, USA,
elmagk@cs.rpi.edu,
http://www.cs.rpi.edu/

Abstract. With the proliferation of large scale dynamic execution environments such as grids, the need for providing efficient and scalable application adaptation strategies for long running parallel and distributed applications has emerged. Message passing interfaces have been initially designed with a traditional machine model in mind which assumes homogeneous and static environments. It is inevitable that long running message passing applications will require support for dynamic reconfiguration to maintain high performance under varying load conditions. In this paper we describe a framework that provides iterative MPI applications with reconfiguration capabilities. Our approach is based on integrating MPI applications with a middleware that supports process migration and large scale distributed application reconfiguration. We present our architecture for reconfiguring MPI applications, and verify our design with a heat diffusion application in a dynamic setting.

1 Introduction

A wide variety of computational environments are increasingly available to host the execution of distributed and parallel applications. Examples include large scale supercomputers, shared or dedicated clusters, grid environments, and metacomputing environments. Performance variability in such environments is the rule and not the exception. For application developers, this variability poses new challenges that go far beyond those of parallelism and scalability. The issue here is not only how to optimize large applications to run on a given set of distributed resources, but also how to maintain the desired performance anytime there is a change in the pool or characteristics of the resources or in the application's demands during its lifetime. In conventional distributed and parallel systems, achieving high performance was a matter of application-level scheduling or application-specific tunings. Such techniques relied on the following assumptions: 1) the application's performance model is known, 2) the number of resources is static, and 3) the characteristics of the resources are known. An obvious solution is dynamic application reconfiguration; i.e., adjusting the allocation of resources as the demand on the system and its availability varies.

While this new generation of computational environments presents multiple resource management challenges, it also provides an abundant pool of resources that is appealing to large scale and computationally demanding distributed applications. Examples are parallel computational science and engineering applications that arise in diverse disciplines such as astrophysics, fluid dynamics, materials science, biomechanics,

or nuclear physics. These applications often involve simulating multi-scale problems and exhibit an insatiable need for computational resources. Many of these applications have been implemented with the Message Passing Interface (MPI) [1]. MPI is a widely used standard to develop parallel applications that harness several processors. However, the issues of scalability, adaptability and load balancing still remain a challenge. To maintain a good performance level, MPI applications need to be able to scale up to accommodate new resources or shrink to accommodate leaving or slow resources. Most existing MPI implementations assume a static network environment. MPI implementations that support the MPI-2 Standard [2, 3] provide some support for dynamic process management by allowing running processes to spawn new processes and communicate with them. However, developers still need to handle explicitly issues such as resource discovery, resource allocation, scheduling, profiling, and load balancing. Additional middleware support is therefore needed to relieve application developers from non-functional concerns while allowing high performance.

The Internet Operating System (IOS) [4, 5] is a distributed middleware framework that provides support for dynamic reconfiguration of large-scale distributed applications through opportunistic load balancing capabilities, resource-level profiling and application-level profiling. IOS has a modular nature that allows developers to create easily various reconfiguration policies. One key ingredient to application reconfiguration is the support for process migration. Applications should support process mobility to be able to benefit from IOS reconfiguration policies.

We target in this work the broad class of iterative applications. A large number of scientific and engineering applications exhibit an iterative nature. Examples include partial differential equation solvers, particle simulations, and circuit simulations [6]. We have chosen to experiment initially with the class of iterative applications for two reasons: this class is important in the scientific and engineering communities, and 2) It exhibits predictable profiling and reconfiguration points that could easily be automated through static software analysis or code-level annotations. To allow such applications to benefit from the reconfiguration capabilities of IOS middleware, we have developed a user-level library on top of MPI that allows process migration [7]. Our strategy achieves portability across different implementations of the the MPI standard. MPI/IOS is a system that integrates IOS middleware strategies with existing MPI applications. MPI/IOS adopts a semi-transparent checkpointing mechanism, where the user needs only to specify the data structures that must be saved and restored to allow process migration. This approach does not require extensive code modifications. Legacy MPI applications can benefit from load balancing features by inserting just a small number of calls to a simple application programming interface. In previous work [7], we described in detail the IOS architecture and evaluated our migration scheme. We take this work further in this paper by demonstrating the capability of IOS to adapt the class of iterative MPI applications to changing load conditions.

The remainder of the paper is organized as follows. Section 2 presents related work. In Section 3, we give an overview of the IOS middleware. Section 4 presents the MPI/IOS architecture with details on how MPI has been extended to support reconfiguration with IOS. Section 5 discusses the adopted reconfiguration policies. In

Section 6, we present performance evaluation. We conclude with discussion and future work in Section 7.

2 Related Work

There are a number of conditions that can introduce computational load imbalances during the lifetime of an application: 1) the application may have irregular or unpredictable workloads from, e.g., adaptive refinement, 2) the execution environment may be shared among multiple users and applications, and/or 3) the execution environment may be heterogeneous, providing a wide range of processor speeds, network bandwidth and latencies, and memory capacity. Dynamic load balancing (DLB) is necessary to achieve a good parallel performance when such imbalances occur. Most DLB research has targeted the application level (e.g., [8, 9]), where the application itself continuously measures and detects load imbalances and tries to correct them by redistributing the data, or changing the granularity of the problem through domain repartitioning. Although such approaches have proved beneficial, they suffer from several limitations. First they are not transparent to application programmers. They require complex programming and are domain specific. Second, they require applications to be amenable to data partitioning, and therefore will not be applicable in areas that require rigid data partitioning. Lastly, when these applications are run on a dynamic grid, application-level techniques which have been applied successfully to heterogeneous clusters [8, 10] may fall short in coping with the high fluctuations in resource availability and usage. Our research targets middleware-level DLB which allows a separation of concerns: load balancing and resource management are transparently dealt with by the middleware, while application programmers deal with higher level domain specific issues.

Several recent efforts have focused on enhancing MPI run-time systems to adapt applications to dynamic environments. Adaptive MPI (AMPI) [11, 12] is an implementation of MPI on top of light-weight threads that balances the load transparently based on a parallel object-oriented language with object migration support. Load balancing in AMPI is done through migrating user-level threads that MPI processes are executed on. This approach limits the portability of process migration across different architectures since it relies on thread migration. Process swapping [13] is an enhancement to MPI that uses over-allocation of resources and improves performance of MPI applications by allowing them to execute on the best performing nodes. MPI process swapping has been also used for the class of iterative applications. Our approach is different in the sense that we do not need to over-allocate resources initially. Such a strategy, though potentially very useful, may be impractical in large-scale dynamic environments such as grids where resources join and leave and where an initial over-allocation may not be possible. We allow new nodes that become available to join the computational grid to improve the performance of running applications during their execution.

Other efforts have focused on process checkpointing and restart as a mechanism to allow applications to adapt to changing environments. Examples include CoCheck [14], starFish [15], MPICH-V [16], and the SRS library [17]. CoCheck, starFish, and MPICH-V support checkpointing for fault-tolerance, while we provide this feature to allow process migration and hence load balancing. Our framework could be integrated with the

sophisticated checkpointing techniques used in these projects to be able to support also non-iterative applications. SRS supports checkpointing to allow application stop and restart. Our work differs in the sense that we support migration at a finer granularity. Application-transparent process checkpointing is not a trivial task, could be very expensive, and is architecture-dependent as it requires saving the entire process state. Semi-transparent checkpointing provides a simpler solution and a more portable approach. It has been been proved useful for the important class of iterative applications [13, 17]. API calls are inserted in the MPI program that informs the middleware of the important data structures to save. This is an attractive solution that can benefit a wide range of applications and does not incur significant overhead since only relevant state is saved. The instrumentation of applications could be easily automated since iterative application have a common structure.

Several projects have actively investigated the issue of application adaptivity in grid environments. Examples include GrADS [18], AppLeS [19], Cactus [20], and GridWay [21]. We share several performance and scheduling ideas with these projects. Most of the strategies they have adopted rely on the application 's stop and restart mechanism; i.e., the entire application is stopped, checkpointed, migrated, and restarted in another hardware configuration. Although this strategy can result in improved performance in some scenarios, a more effective adaptivity could be achieved if migration is supported at a finer granularity. We address reconfigurability at the process-level of the application to increase flexibility.

3 Overview of the IOS Framework

The goal of IOS middleware is to provide effective decentralized middleware-triggered dynamic reconfiguration strategies that enable application adaptation to the constantly changing behavior of large scale shared networks. To distributed application developers who often lack the time and expertise to handle complex performance tunings, IOS is a promising approach that combines both ease of use and high performance.

Applications wishing to interact with IOS need to have a flexible structure that synergizes easily with the dynamic nature of shared networks. They should exhibit a large degree of processing and/or data parallelism for efficient use of the system and scalability to a large number of resources. We assume that every application consists of distributed and migratable entities. In the case of MPI application, such entities refer to MPI processes.

IOS reconfiguration mechanisms allow 1) analyzing profiled application communication patterns, 2) capturing the dynamics of the underlying physical resources, 3) and utilizing the profiled information to reconfigure application entities by changing their mappings to physical resources through migration. A key characteristic of IOS is that it adopts a decentralized strategy that avoids the use of any global knowledge to allow scalable reconfiguration.

The IOS architecture consists of distributed middleware agents that are capable of interconnecting themselves in various virtual topologies. We support both hierarchical and peer-to-peer (P2P) topologies. The first is more suitable in grid environments that

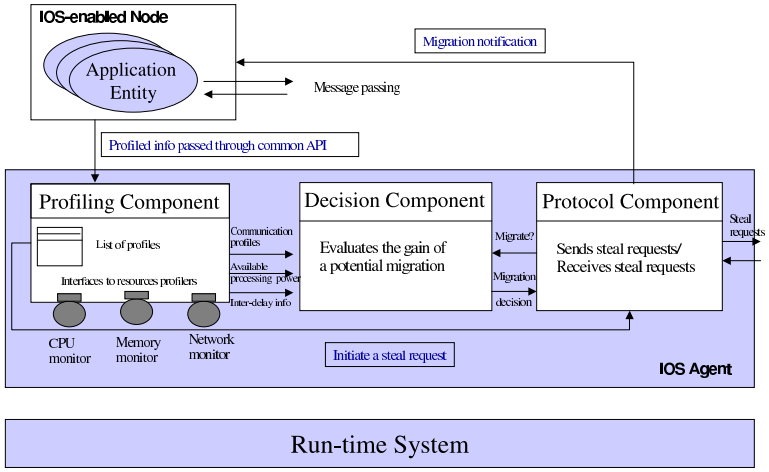

Fig. 1. Interactions between a reconfigurable application and the local IOS agents.

usually consist of a hierarchy of homogeneous clusters while the second is more suitable for Internet environments that usually lack a well defined structure.

Figure 1 shows the architecture of an IOS agent and how it interacts with the application entities that are hosted in a given node. Every IOS agent consists of a profiling component, a decision component, and a protocol component:

- Dynamic Profiling Component
  Resource-level and application-level profiling is used to gather dynamic performance profiles about physical resources and application entities. Profiling generates performance profiles that are used in the reconfiguration decisions. Every application entity profiles its processing, communication, data accesses and memory usage. Every resource has also a profiling monitor that monitors periodically its utilization. Information about the resource's available CPU power, memory, and disk storage are measured and recorded periodically. The IOS architecture defines well-defined interfaces for profiling monitors which allow using several profiling technologies as long as they implement the appropriate interfaces. Examples include the Network Weather Service (NWS) [22] and MDS [23].
- Protocol Component
  The protocol component is responsible for inter-agent communication and virtual topology creation. The middleware agents form a virtual network. When new nodes join the network or existing nodes become idle, their corresponding protocol component contact peers randomly to steal work [24]. This strategy aids in advertising new or existing resources as they become available. Every work stealing request carries with it performance-related information about the node that originated the request.
- Reconfiguration Module
  Upon receiving a work stealing request, the reconfiguration module tries to evaluate

whether there are any potential candidate entities that will benefit from migration to the originator of the request. The decision is done by examining the performance of the application entities in the local node and the predicted performance in the remote node. If a gain is predicted, the local protocol component notifies the selected entities to migrate.

Applications communicate with the IOS middleware through clearly defined interfaces that permit the exchange of profiled information and reconfiguration requests. Figure 2 shows the profiling interface that allows applications to notify the middleware about all communication exchanges. The profiling component maintains a list of all application entities hosted in its local run-time system. For every application entity, the list maintains information about all other entities that have been exchanging messages with it and their frequency of communication. In other words, this list represents a weighted communication subgraph of every application entity where the nodes represent application entities, the edges represent communication links, and the weights represent communication rates. Every entity has a unique name (UAN) associated with it and a universal locator (UAL) that keeps track of the current location where the entity is currently hosted. A UAN stands for Universal Actor Name while a UAL stands for Universal Actor Locator. We adopt here the naming conventions of the SALSA [25] language. SALSA is a language for developing actor oriented application. It is a dialect of Java with high-level constructs for universal naming, remote message sending, and coordination. IOS has been prototyped using both SALSA and Java.

```
//The following methods notify the profiling agent of entities
//entering and exiting the local run-time system due
//to migration or initial entity startup.

public void addProfile(UAN uan);
public void removeProfile(UAN uan);
Public void migrateProfile(UAN uan, UAL target)

//The profiling agent updates its entity profiles based
//on message sending with these methods

public void msgSend(UAN uan, UAL targetUAL, UAL sourceUAL,
                    int msg_size);

//The profiling agent updates its entity profiles based
//on message reception with this method

public void msgReceive(UAN uan, UAL targetUAL, UAL sourceUAL,
                       int msg_size);

//The following methods notify the profiling agent of the start
//of a message being processed and the end of a message being processed,
//with a UAN or UAL to identify the sending entity

public void beginProcessing(UAN uan, UAL targetUAL, UAL sourceUAL,
                            int msg_size);
public void endProcessing(UAN uan, UAL targetUAL, UAL sourceUAL,
                          int msg_size);
```

Fig. 2. IOS Profiling API

4 Reconfiguring MPI applications with IOS

4.1 Process Migration Support

In MPI, any communication between processes needs to be done as part of a *communicator*. An MPI communicator is an opaque object with a number of attributes, together with simple functions that govern its creation, use and destruction. An *intracommunicator* delineates a communication domain which can be used for point-to-point communications as well as collective communication among the members of the domain. On the other hand, an *intercommunicator* allows communication between processes belonging to disjoint intracommunicators.

We achieve MPI process migration by rearranging MPI communicators. Migration is performed by a collaboration of all the participating MPI processes. It has to be done at a point where there are no pending communications. Process migration requires careful update of any communicator that involves the migrating process. The class of iterative application have natural barrier points. When necessary, we perform all reconfiguration at the beginning of each iteration. A migration request forces all running MPI processes to enter a reconfiguration phase where they all cooperate to update their shared communicators. The migrating process spawns a new process in the target location and sends it its local checkpointed data.

Process migration and checkpointing support have been implemented as part of a user-level library. This approach allows portability across several vendor MPI implementations that support the MPI-2 process spawning feature since the library is implemented entirely in the user space and does not require any infrastructural changes. The library is called PCM (Process Checkpointing and Migration).

4.2 Profiling MPI Application

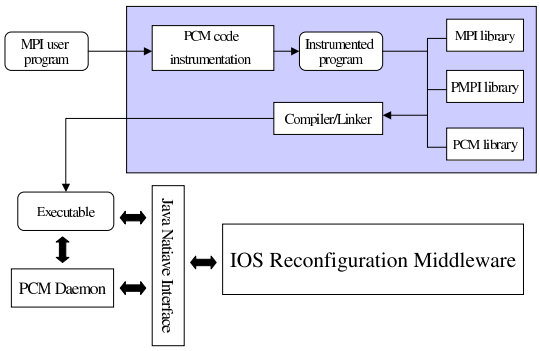

Fig. 3. Library and executable structure of an MPI/IOS application

MPI processes need to send periodically their communication patterns to their corresponding IOS profiling agents. To achieve this, we have built a profiling library that is based on the MPI profiling interface (PMPI). The MPI specification provides a general mechanism for intercepting calls to MPI functions. This allows the development of portable performance analyzers and other tools without access to the MPI implementation source code. The only requirement is that every MPI function be callable by an alternate name (`PMPI_Xxxx` instead of the usual `MPI_Xxxx`.). The built profiling library intercepts all communication methods of MPI and sends any communication event to the profiling agent.

All profiled MPI routines call their corresponding `PMPI_Xxxx` and, if necessary, PCM routines. Figure 3 shows the library structure of the MPI/IOS programs. The instrumented code is linked with the profiling library PMPI, the PCM library, and a vendor MPI implementation's library. The generated executable passes all profiled information to the IOS run-time system through Java Native Interface (JNI) and also communicates with a local PCM Daemon (PCMD) that is started in every node. The PCMD is responsible for storing local checkpoints and passing reconfiguration decisions across a socket API from the IOS agent to the MPI processes. For more details about the PCMD, the reader is referred to [7].

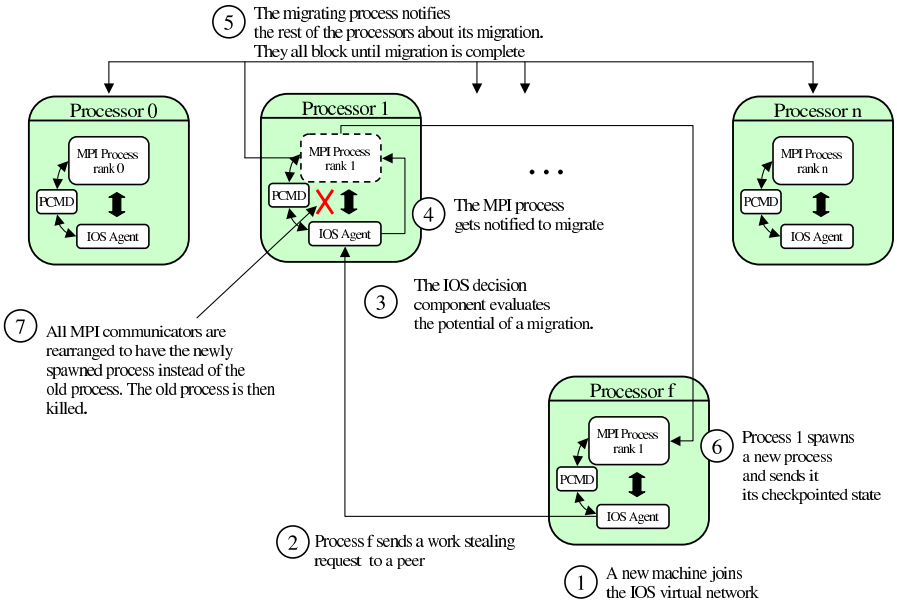

Fig. 4. A reconfiguration scenario of an MPI/IOS application

4.3 A Simple Scenario for Adaptation

MPI applications interact with each other, with the checkpointing and migration services provided by the PCM library, and with the profiling and reconfiguration services provided by IOS agents. Walking through the simple scenario of an application adaptation that is shown in Figure 4 further explains these interactions. The example illustrates the execution of a parallel MPI application that is composed of n processes running on n different processors.

1. An available processor joins the IOS virtual network.
2. The new processor starts requesting work from its peers.
3. Processor 1 receives the work stealing request. The decision component in its local IOS agent predicts that there will be a gain migrating process 1 to the remote processor f.
4. MPI process 1 gets notified of a migration event.
5. At the beginning of the next iteration, the migrating process broadcasts a message to the rest of the processors so that they enter a reconfiguration phase.
6. The migrating process checkpoints its local state, and spawns an image of itself in the remote processor f. The local PCMD takes care of transferring the checkpointed state to the newly created process and notifying the rest of the processes.
7. As a part of completing the migration process, the PCM library takes care of rearranging the `MPI_COM_WORLD` communicator by removing the old process and including the new one. The newly created process gets assigned rank 1.

5 Reconfiguration Policies

The reconfiguration policies use the application's characteristics and the underlying resources' characteristics to evaluate whether there will be any performance gain through migration. Resources such as storage, CPU processing power, network bandwidth, and network latencies are ranked according to their importance to the running application. For instance, if the application is computationally intensive, more weight is given to the CPU processing power. If the application is communication intensive and the messages exchanged have large sizes, more weight is given to the network bandwidth. Whereas, if it is communication intensive with small exchanged message sizes, the network latency is very important.

Let P be the set of m processes running on a local node n.

$$P = \{p_0, p_1, ..., p_m\}$$

Let R be a set of resources available in the local node n.

$$R = \{r_0, r_1, ..., r_l\}$$

Let ω be the weight assigned to a given resource r_i based on its importance to the performance of the set P.

$$0 \leq \omega(r_i, P) \leq 1 \; and \; \sum_{i=0}^{l} \omega(r_i, P) = 1$$

Let $a(r, f)$ be the amount of resource r available at foreign node f, $u(r, l, A)$ be the amount of resource r used by the processes P at local node l, $M(P, l, f)$ be the estimated cost of migration of the set P from l to f, and $L(P)$ be the average life expectancy of the set of processes P. The predicted increase in overall performance Γ gained by migrating P from l to f, where $\Gamma \leq 1$ is:

$$\Delta_{r,l,f,P} = \frac{a(r, f) - u(r, l, P)}{a(r, f) + u(r, l, P)} \tag{1}$$

$$\Gamma = (\sum_{r} \omega(r, P) * \Delta_{r,l,f,P}) - (\frac{M(P, l, f)}{(10 + log(L(P))}) \tag{2}$$

When a node l receives a work stealing request from a node f, a process or a group of processes will be migrated if the evaulation of the gain Γ is positive. The larger this value is, the more beneficial the migration is.

It is important to factor in Equation 2 the remaining life expectancy of the group of processes P. The intuition behind this is that processes who are expected to have a short remaining life are not migrated because the cost of the reconfiguration might exceed the benefit over their remaining life expectancy. For the case of iterative applications, we estimate the life expectancy by looking at the number of remaining iterations multiplied by the average time each iteration takes in the current node.

6 Performance Evaluation

6.1 Application Case Study

We have used a fluid dynamic problem that solves heat diffusion in a solid for testing purposes. This applications is representative of the large class of highly synchronized iterative mesh-based applications. The application has been implemented using C and MPI and has been instrumented with PCM library calls. We have used a simplified version of this problem to evaluate our reconfiguration strategies. A two-dimensional mesh of cells is used to represent the problem data space. The mesh initially contains the initial values of the mesh with the boundary values. The cells are uniformly distributed among the parallel processors. At the beginning, a master process takes care of distributing the data among processors. For each iteration, the value of each cell is calculated with the values of its neighbor cells. Each cell needs to maintain a current version of the values of its neighboring cells. To achieve this, processors exchange values of the neighboring cells, also referred to as ghost cells. To sum up, every iteration consists of doing computation and exchange of ghost cells from the neighboring processors. Figure 5 shows the structure of the parallel decomposition of the heat diffusion problem.

6.2 Adaptation Experiments

We have used the iterative 2-dimensional heat diffusion application to evaluate the reconfiguration capabilities of the MPI/IOS framework. The original MPI code was instrumented with PCM calls to enable checkpointing and migration. For the experimental

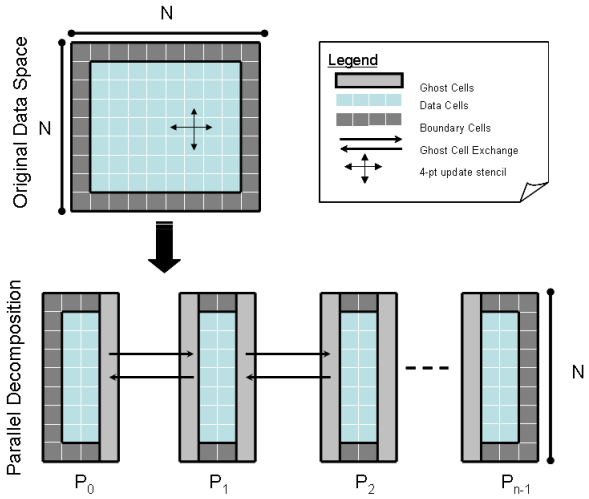

Fig. 5. Parallel decomposition of the 2D heat diffusion problem.

testbed we used a 4-dual node cluster of SUN Blade 1000 machines. Each node has a processing speed of 750M cycles per second and 2 GB of memory. For comparative purposes, we used MPICH2 [26], a free implementation of the MPI-2 standard. We emulated a shared and dynamic environment with varying load conditions by introducing artificial load in some of the cluster nodes and varying it periodically.

We conducted two experiments using the heat application using the MPI/IOS framework and MPICH2 under similar load conditions. For both experiments, we started running the application, then we kept increasing the load in some of the cluster nodes and watched how the application's performance was affected in each case.

The first experiment was conducted with MPI/IOS. Figure 6 shows the performance of the application. We started running the application with 8 processes on 4 processors. the remaining 4 processors joined the virtual IOS network gradually. We started increasing gradually the load on one of the cluster nodes (two processors) participating in the computation. One node joined the virtual IOS network around iteration 1000 and started sending work stealing requests. This caused one process to migrate to the new machine and to reduce the load on its original hosting node. We notice an increase in the application throughput. The load in the slow machine increased even further around iteration 2500. Around iteration 3000, a fourth node joined the virtual network and started sending work stealing packets. At this point, two processes migrated to this machine. This caused the slow processors to be eliminated from the computation. The application ended up using a total of the 6 best available processors, which caused a substantial increase in its performance. The total execution time of the application was 645.67s.

Figure 7 shows the performance of the application using MPICH2. We emulated the same load conditions as in the first experiment. With no ability to adapt, the application was stuck with the first hardware configuration and experienced a constant slowdown in

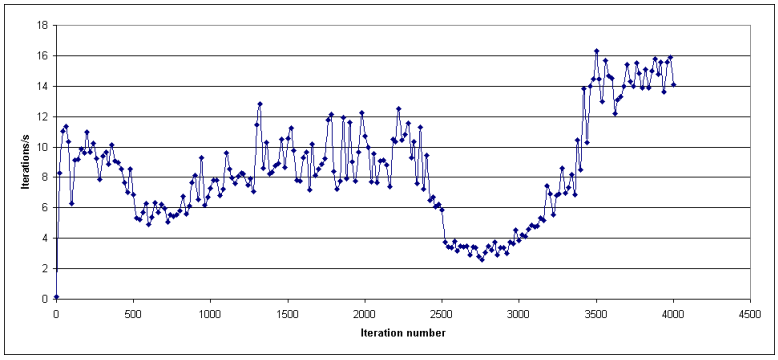

Fig. 6. Performance of the the two-dimensional heat simulation application using the reconfiguration mechanism of MPI/IOS. The experiment was conducted on a 4 dual-processor SUN blade 1000 cluster.

its performance. The highly synchronized nature of this application causes it to run as fast as the slowest processor. The application took 1173.79s to finish, about an 81.8% decrease in performance compared to the adaptive execution.

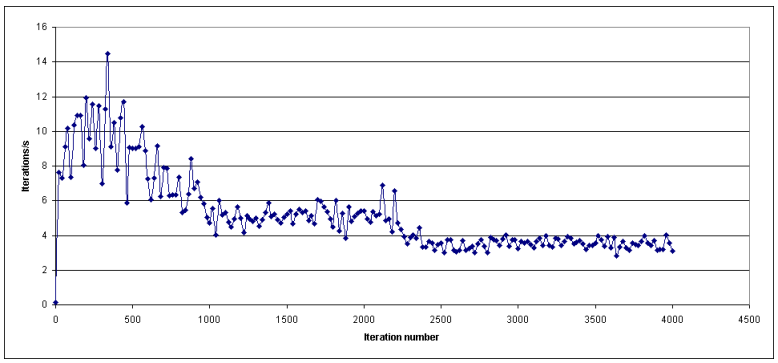

Fig. 7. Performance of the the two-dimensional heat simulation application using MPICH2. The experiment was conducted on a 4 dual-processor SUN blade 1000 cluster.

7 Discussion and Future Work

We presented in this paper an architecture that enhances the performance of iterative applications under dynamically changing conditions. We implemented a user-level library that adds checkpointing and process migration features to existing iterative MPI applications. We also integrated our library with a middleware for reconfigurable computing. The experimental evaluation has demonstrated the importance of augmenting

long-running MPI application with reconfiguration capabilities to achieve high performance.

This work opens up several interesting future directions. One direction is the evaluation of different reconfiguration policies and how effective they are in dynamic execution environments. The sensitivity of these policies to application's characteristics needs also to be investigated. Another important direction is extending our framework beyond the class of iterative applications. This will require developing more sophisticated checkpointing and migration policies. We plan also to evaluate our framework on larger networks with interesting and realistic load characteristics and network latencies.

8 Acknowledgments

The authors would like to acknowledge the members of the Worldwide Computing Laboratory at Rensselaer Polytechnic Institute. In particular, our special thanks go to Travis Desell for his contributions to the IOS middleware. This work has been supported by the following grants: IBM SUR Award 2003, IBM SUR Award 2004, NSF CAREER Award No. CNS-0448407, and NSF-INT 0334667.

References

1. Message Passing Interface Forum: MPI: A message-passing interface standard. The International Journal of Supercomputer Applications and High Performance Computing **8**(3/4) (1994) 159–416
2. Gropp, W., Lusk, E.: Dynamic process management in an MPI setting. In: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, IEEE Computer Society (1995) 530
3. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1996)
4. Desell, T., Maghraoui, K.E., Varela, C.: Load balancing of autonomous actors over dynamic networks. In: Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track, Hawaii (2004)
5. Maghraoui, K.E., Desell, T., Varela, C.: Network sensitive reconfiguration of distributed applications. Technical Report CS-05-03, Department of Computer Science, Rensselaer Polytechnic Institute (2005)
6. Fox, G.C., Williams, R.D., Messina, P.C. In: Parallel Computing Works. Morgan Kaufmann Publishers, San Fransisco, CA (1994) Available at http://www.npac.syr.edu/pcw/.
7. Maghraoui, K.E., Desell, T., Szymanski, B.K., Teresco, J.D., Varela, C.A.: Towards a middleware framework for dynamically reconfigurable scientific computing. In Grandinetti, L., ed.: Grid Computing and New Frontiers of High Performance Processing. Elsevier (2005) to appear.
8. Elsasser, R., Monien, B., Preis, R.: Diffusive load balancing schemes on heterogeneous networks. In: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press (2000) 30–38
9. Flaherty, J.E., Loy, R.M., Özturan, C., Shephard, M.S., Szymanski, B.K., Teresco, J.D., Ziantz, L.H.: Parallel structures and dynamic load balancing for adaptive finite element computation. Applied Numerical Mathematics **26** (1998) 241–263

10. Teresco, J.D., Faik, J., Flaherty, J.E.: Resource-aware scientific computation on a heterogeneous cluster. Technical Report CS-04-10, Williams College Department of Computer Science (2005) To appear, *Computing in Science & Engineering*.
11. Bhandarkar, M.A., Kale;, L.V., de Sturler, E., Hoeflinger, J.: Adaptive load balancing for MPI programs. In: Proceedings of the International Conference on Computational Science-Part II, Springer-Verlag (2001) 108–117
12. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas (2003)
13. Sievert, O., Casanova, H.: A simple MPI process swapping architecture for iterative applications. International Journal of High Performance Computing Applications **18**(3) (2004) 341–352
14. Stellner, G.: Cocheck: Checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium, IEEE Computer Society (1996) 526–531
15. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society (1999) 31
16. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: toward a scalable fault tolerant mpi for volatile nodes. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press (2002) 1–18
17. Vadhiyar, S.S., Dongarra, J.J.: SRS - a framework for developing malleable and migratable parallel applications for distributed systems. In: Parallel Processing Letters. Volume 13. (2003) 291–312
18. Vadhiyar, S., Dongarra: Self adaptivity in grid computing. Concurrency and Computation: Practice and Experience **17**(2-4) (2005) 235–257
19. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive Computing on the Grid Using AppLeS. IEEE Trans. Parallel Distrib. Syst. **14**(4) (2003) 369–382
20. Allen, G., Dramlitsch, T., Foster, I., Karonis, N.T., Ripeanu, M., Seidel, E., Toonen, B.: Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), New York, NY, USA, ACM Press (2001) 52–52
21. Huedo, E., Montero, R.S., Llorente, I.M.: A framework for adaptive execution in grids. Softw. Pract. Exper. **34**(7) (2004) 631–651
22. Wolski, R.: Dynamically forecasting network performance using the network weather service. Cluster Computing **1**(1) (1998) 119–132
23. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: Proceedings of the 10th IEEE Symposium On High Performance Distributed Computing. (2001)
24. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94), Santa Fe, New Mexico (1994) 356–368
25. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings **36**(12) (2001) 20–34 http://www.cs.rpi.edu/ cvarela/oopsla2001.pdf.
26. Argone National Laboratory: (MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2)