A Middleware Framework for Dynamically Reconfigurable MPI Applications

Kaoutar ElMaghraoui, Carlos A. Varela, Boleslaw K. Szymanski, and Joseph E. Flaherty
Department of Computer Science
Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590, USA
{*elmagk, cvarela, szymansk, flaherje*}*@cs.rpi.edu*

James D. Teresco
Department of Computer Science
Williams College, 47 Lab Campus Drive, Williamstown, MA 01267, USA
terescoj@cs.williams.edu

Abstract

Computational grids are characterized by their dynamic, non-dedicated, and heterogeneous nature. Novel application-level and middleware-level techniques are needed to allow applications to reconfigure themselves and adapt automatically to their underlying execution environments to be able to benefit from computational grids' resources. In this paper, we introduce a new software framework that enhances the Message Passing Interface (MPI) performance through process checkpointing, migration, and an adaptive middleware for load balancing. Fields as diverse as fluid dynamics, material science, biomechanics, and ecology make use of parallel adaptive computation where target architectures have traditionally been supercomputers and tightly coupled clusters. This framework is a first step in allowing these computations to use computational grids efficiently. Preliminary results demonstrate that application reconfiguration through middleware-triggered process migration achieved performance improvement in the range of 33% to 79%.

1 Introduction

Computational grids [16] have become very attractive platforms for high performance distributed applications due to their high availability, scalability, and computational power. However, nodes in grid environments, such as individual processors or symmetric multiprocessors (SMPs), are not necessarily dedicated to a single parallel or distributed application. They experience constantly changing loads and communication demands. Achieving the desired high performance, requires augmenting applications with appropriate support for reconfiguration and adaptability to the dynamic nature of computational grids.

Scientific and engineering distributed and parallel applications have a complex nature. They are computationally demanding, involve solving or simulating multi-scale problems with dynamic behavior, and often require sophisticated adaptive methods and dynamic load balancing techniques to achieve high performance. In particular, engineering applications that arise extensively in diverse disciplines such as fluid dynamics, material science, biomechanics, and ecology require a continuous adaptation of meshes and numerical methods to achieve specified levels of solution accuracy [7]. These challenges have been addressed by complex application-level load balancing algorithms and adaptive refinement techniques [9] . However, most existing solutions assume either a static number of cooperating processes or dedicated resources. Running such applications on computational grids introduces additional challenges that cannot be addressed sufficiently by application-level load balancing.

MPI [20] has been widely adopted as the de-facto standard to implement parallel and distributed applications that harness several processors. However, the issues of scalability, adaptability and load balancing still remain a challenge. Most existing MPI implementations assume a static network environment. MPI implementations that support the MPI-2 standard [17, 21] provide partial support for dynamic process management, but still require complex application development from end-users: process management needs to be handled explicitly at the application level, which requires the developer to deal with issues such as resource discovery and allocation, scheduling, load balancing, etc. Additional middleware-support is therefore needed to relieve application developers from non-functional concerns such as load balancing through application reconfiguration.

In this paper, we introduce MPI/IOS, a middleware in-

frastructure that permits the automatic reconfiguration of MPI applications in a dynamic setting. The Internet Operating System (IOS) [8, 10] is a distributed middleware framework that provides opportunistic load balancing capabilities through resource-level profiling and application-level profiling. MPI/IOS adopts a semi-transparent checkpointing mechanism, where the user needs only specify the data structures to save and restore to facilitate process migration. This approach does not require extensive code modifications and allows legacy MPI applications to benefit from load balancing features by just inserting a small number of API calls. In shared environments where many applications are running, having application-level resource management is not enough to efficiently balance the load of the entire system. A middleware layer is the natural location where to place OS-like resource management of several applications running simultaneously.

We believe that providing simple APIs and delegating most of the load distribution and balancing to middleware will allow smooth and easy migration of MPI applications from static and dedicated clusters to highly dynamic computational grids. Our framework is more beneficial for long running applications involving large numbers of machines, where the probability of load fluctuations is high. In such situations, it will be helpful for the running application to have means by which to evaluate continuously its performance, discover new resources, and be able to migrate whole or parts of the application to better nodes. We target initially highly synchronized iterative applications that have the unfortunate property of running as slow as the slowest process. Eliminating the slowest processor from the computation results in many cases, in an overall improved performance.

The remainder of the paper is organized as follows. Section 2 presents related work. In Section 3, we describe the approach of MPI/IOS. Section 4 details the architecture of the framework. In Section 5, we present experimental results. We conclude with discussion and future work in Section 6.

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., [9, 12, 15]), 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 the more dynamic grid, application-level techniques which have been applied successfully to heterogeneous clusters [12, 14] 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 middleware-level technologies for the emerging computational grids. Adaptive MPI (AMPI) [4, 18] 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 [22] 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. Our approach is different in that we do not need to over-allocate resources initially. Such a strategy, though potentially very useful, may be impractical in grid environments 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 [23], starFish [1], and the SRS library [24]. Both CoCheck and starFish support checkpointing for fault-tolerance, while we provide this feature to allow process migration and hence load balancing. SRS supports this feature to allow application stop and restart. Our work differs in the sense that we support migration at a finer granularity. Process checkpointing is a non-functional concern that is needed to allow dynamic reconfiguration. To be able to migrate MPI processes to better performing nodes, processes need to save their state, migrate, and restart from where they left off. Application-transparent process checkpointing is not a triv-

ial task and can be very expensive, as it requires saving the entire process state. Semi-transparent checkpointing provides a simple solution that has been proved useful for iterative applications [22, 24]. 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.

3 An Approach for Dynamically Reconfiguring MPI Applications

Traditional MPI programs are designed with dedicated resources in mind. Developers need to know initially what resources are available and how to assign them to MPI processes. To permit a smooth migration of existing MPI applications to dynamic grid environments, MPI runtime environments should be augmented with middleware tools that free application developers from concerns about what resources are available and when to use them. Simply acquiring the resources is not enough to achieve peak MPI performance. Effective scheduling and load balancing decisions need to be performed continuously during the lifetime of a parallel application. This requires the ability to profile application behavior, monitor the underlying resources, and perform appropriate load balancing of MPI processes through process migration.

Process migration is a key requirement to enable malleable applications. We describe in what follows how we achieve MPI process migration. We then introduce our middleware-level support.

3.1 MPI Process Migration

MPI processes periodically get notified by the middleware of migration or reconfiguration requests. When a process receives a migration notification, it initiates checkpointing of its local data in the next synchronization point. Checkpointing is achieved through library calls that are inserted by the programmer in specific places in the application code. We currently support iterative applications since their iterative structure exhibits natural locations (at the beginning of each iteration) to place polling, checkpointing and resumption calls. When the process is first started, it checks whether it is a fresh process or it has been migrated. In the second case, it proceeds to data and process interconnectivity restoration.

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. While an *intercommunicator* allows communication between processes belonging to disjoint intracommunicators. MPI process migration requires careful update of any communicator that involves the migrating process. 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. Figure 1 describes the steps involved in managing the MPI communicators for a sample process migration. In the original communicator, (Communicator 1), P7 has received a migration request. P7 cooperates with the processes of communicator 1 to spawn the new process, P0 in communicator 2, which will replace it. The intercommunicator that results from this spawning is merged into one global communicator. Later, the migrating process is removed from the old communicator and the new process is assigned rank 7. The new process restores the checkpointed data from its local daemon and regains the same state of the migrating process. All processes then get a handle to the new communicator and the application resumes its normal execution.

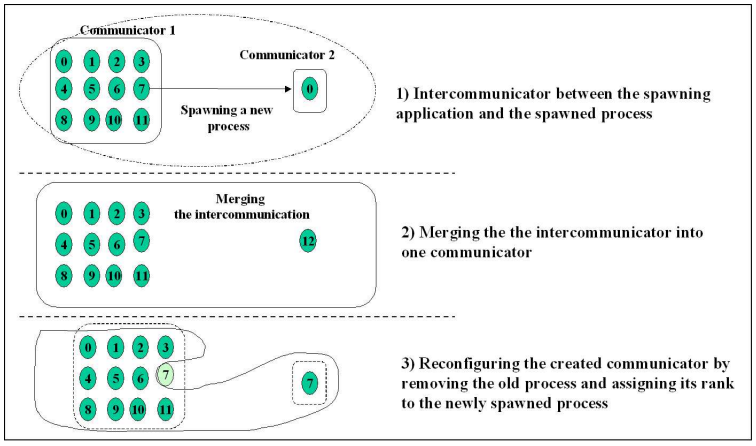

Figure 1. Steps involved in communicator handling to achieve MPI process migration.

3.2 Middleware-triggered Reconfiguration

Although MPI processes are augmented with the ability to migrate, middleware support is still needed to guide the application as to when it is appropriate to migrate processes and where to migrate them. IOS middleware analyzes both the underlying physical network resources and the application communication patterns to decide how applications should be reconfigured to accomplish load bal-

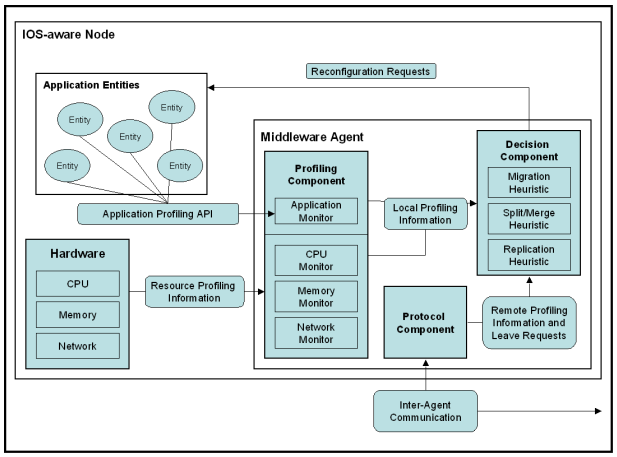

Figure 2. Architecture of a node in the Internet Operating System middleware (IOS). An agent collects profiling information and makes decisions on how to reconfigure the application based on its decisions, protocol, and profiling information.

ancing through process migration and other non-functional concerns such as fault tolerance through process replication. Resource profiling and reconfiguration decisions are embodied into middleware agents whose behavior can be modularly modified to implement different resource management models. Figure 2 shows the architecture of an IOS agent and how it interacts with applications. Every agent has a profiling component that gathers both application and resource profiled information, a decision component that predicts based on the profiled information when and where to migrate application entities, and a protocol component that allows inter-agent communication. Application entities refer to application components. In the case of MPI applications, they refer to MPI processes.

The middleware agents form a virtual network. When new nodes join the network or existing nodes become idle, their corresponding agents contact peers to steal work [5]. In previous work [8], we have shown that considering the application topology in the load balancing decision procedures dramatically improves throughput over purely random work stealing. IOS supports two load-balancing protocols: 1) application topology sensitive load balancing and 2) network topology sensitive load balancing [8, 10].

Applications communicate with the IOS middleware through clearly defined interfaces that permit the exchange of profiled information and reconfiguration requests. Applications need to support migration to react to IOS reconfiguration requests.

4 MPI/IOS Runtime Architecture

MPI/IOS is implemented as a set of middleware services that interact with running applications through an MPI wrapper. The MPI wrapper contains a Process Checkpointing and Migration (PCM) library [11]. The MPI/IOS runtime architecture consists of the following components (see Figure 3): 1) the PCM-enabled MPI applications, 2) the wrapped MPI that includes the PCM API, the PCM library, and wrappers for all MPI native calls, 3) the MPI library, and 4) the IOS runtime components.

4.1 Process Checkpointing and Migration API

PCM is a user-level process checkpointing and migration library that acts on top of native MPI implementations and hides several of the issues involved in handling MPI communicators and updating them when new nodes join or leave the computation. This work does not alter existing MPI implementations and hence, allows MPI applications to continue to benefit from the various implementations and optimizations while being able to adapt to changing loads when triggered by IOS middleware load balancing agents.

MPI/IOS improves performance by allowing running processes to migrate to the processors with the best performance and collocating frequently communicating processes within small network latencies. The MPI-1 standard does not allow dynamic addition and removal of processes from MPI communicators. MPI-2 supports this feature; however existing applications need extensive modification to benefit from dynamic process management. In addition, application developers need to explicitly handle load balancing issues or interact with existing schedulers. The PCM runtime system utilizes MPI-2 dynamic features, however it hides how and when reconfiguration is done. We provide a semi-transparent solution to MPI applications in the sense that developers need to include only a few calls to the PCM API to guide the underlying middleware in performing process migration.

Existing MPI applications interact with the PCM library and the native MPI implementation through a wrapper as shown in Figure 3. The wrapper MPI functions are provided to perform MPI-level profiling of process communication patterns. This profiled information is sent periodically to the IOS middleware agent through the PCM runtime daemon.

4.2 The PCM Library

Figure 4 shows an MPI/IOS computational node running MPI processes. A PCM daemon (PCMD) interacts with the IOS middleware and MPI applications. A PCMD is started in every node that actively participates in an application. A PCM dispatcher is used to start PCMDs in various nodes

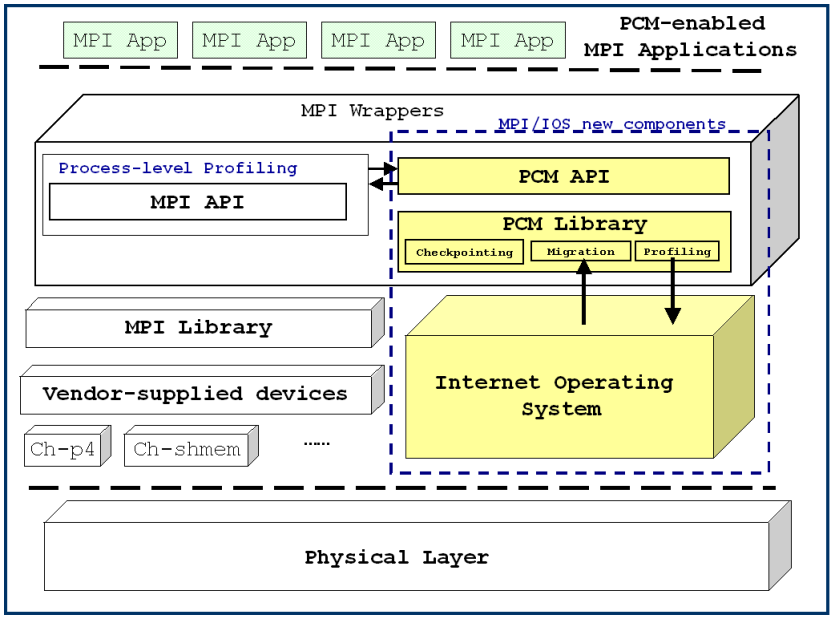

Figure 3. The layered design of MPI/IOS which includes the MPI wrapper, the PCM runtime layer, and the IOS runtime layer.

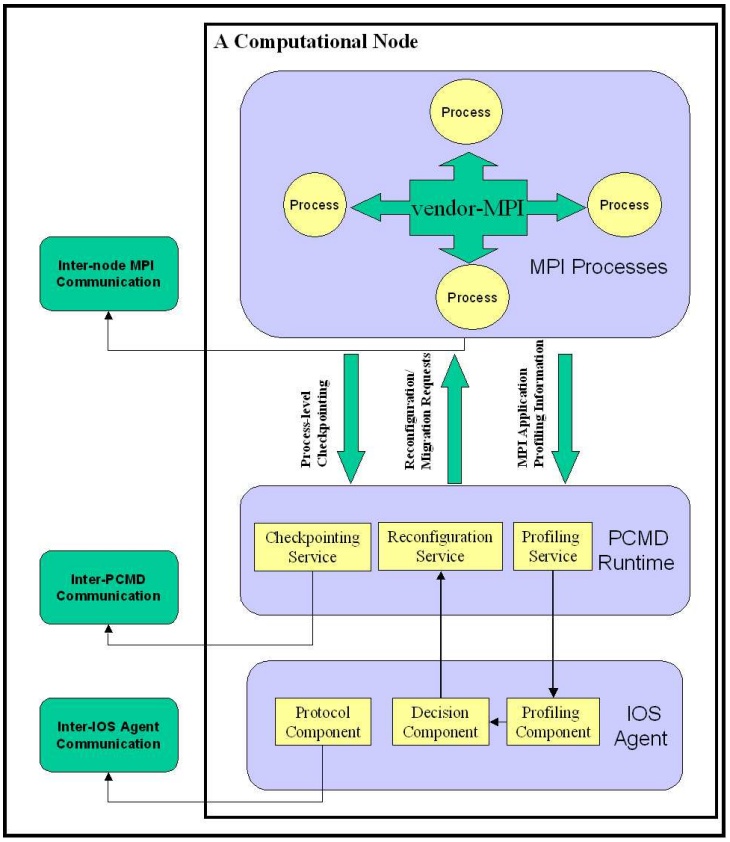

Figure 4. Architecture of a node running MPI/IOS enabled applications.

and used to discover existing ones. The application initially registers all MPI processes with their local daemons. The port number of a daemon is passed as an argument to `mpiexec` or read from a configuration file that resides in the same host.

Every PCMD has a corresponding IOS agent. There can be more than one MPI process in each node. The daemon consists of various services used to achieve process communication profiling, checkpointing and migration. The MPI wrapper calls record information pertaining to how many messages have been sent and received and their source and target process ranks. The profiled communication information is passed to the IOS profiling component. IOS agents keep monitoring their underlying resources and exchanging information about their respective loads.

When a node's available resources fall below a predefined threshold or a new idle node joins the computation, a *work steal* packet is propagated among the actively running nodes. The IOS agent of a node responds to work stealing requests if it becomes overloaded and its decision component decides according to the resource management model which process(es) need(s) to be migrated. Otherwise, it forwards the request to an IOS agent in its set of peers. The decision component then notifies the reconfiguration service in the PCMD, which then sends a migration request to the desired process(es). At this point, all active PCMDs in the system are notified about the event of a reconfiguration. This causes all processes to cooperate in the next iteration until migration is completed and application communicators have been properly updated. Although this mechanism imposes some synchronization delay, it ensures that no messages are being exchanged while process migration is taking place and avoids incorrect behaviors of MPI communicators.

5 Experimental Results

We have used an MPI program that computes a two-dimensional heat distribution matrix to evaluate the performance of process migration. This application models iterative parallel applications that are highly synchronized and therefore require frequent communication between the boundaries of the MPI processes. The original MPI code was manually instrumented by inserting PCM API calls to enable PCM checkpointing. It took 10 lines of PCM library calls to instrument this application, which consists originally of 350 lines of code.

The experimental test-bed consists of a multi-user cluster that consists of a heterogeneous collection of Sun computers running Solaris. We used a cluster of 20 nodes that consist of 4 dual-processor SUN Blade 1000 machines with 750 MHz per processor and 2 GB of memory, and 16 single-processor SUN Ultra 10 machines with 400MHz and 256 MB of memory. We used MPICH2 [2], a freely available implementation of the MPI-2 standard.

The goal of the first experiment was to determine the overhead incurred by the PCM API. The heat distribution program was executed using both MPICH2 and MPI/IOS with several numbers of nodes. We run both tests under a controlled load environment to make sure that the machine load is somehow balanced and no migration will be triggered by the middleware. Both implementations demonstrated similar performance. Figure 5 shows that the overhead of the PCM library is negligible.

The second experiment aims at evaluating the impact of process migration. The cluster of 4 dual-processor nodes was used. Figures 6 and 7 show the breakdown of the iterations execution time of the heat application using MPICH2 and MPI/IOS respectively. The load of the participating nodes was controlled to provide a similar execution environment for both runs. The application was allowed to run for a few minutes, after which the load of one of the nodes was artificially increased substantially. In Figure 6, the overall execution time of the application iterations increased. The highly synchronized nature of this application forces all the processes to become as slow as the one assigned to the slowest processor. The application took 203.97 seconds to finish. Figure 7 shows the behavior of the same application under the same load conditions using MPI/IOS. At iteration 260, a new node joined the computation. This resulted in migration of an MPI process from the overloaded node to the available new node. The figure shows how migration corrected the load imbalance. The application took 115.27 seconds to finish in this case, which is almost a 43% improvement over the non-adaptive MPICH2 run.

In a third experiment, we evaluated the adaptation of the heat application to changing loads. Figure 8 shows the behavior of the application's throughput during its lifetime.The total number of iterations per second gives a good estimate of how good the application is performing for the class of highly synchronized applications. We run the heat program using the 4 dual-processor cluster and increased the load in one of the participating nodes. MPI/IOS helped the application to adapt by migrating the process from the slow node to one of the cooperating nodes. The application was using only 3 nodes after migration; however, its overall throughput improved substantially. The application execution time improved with 33% compared to MPICH2 under the same load conditions. In Figure 9, we evaluated the impact of migration when a new node joins the computation. In this experiment, we used 3 fast machines and a slow machine. We increased the load of the slow machine while the application was running. The throughput of the application increased dramatically when the slow process migrated to a fast machine that joined the IOS network. The performance of the program improved with 79% compared with

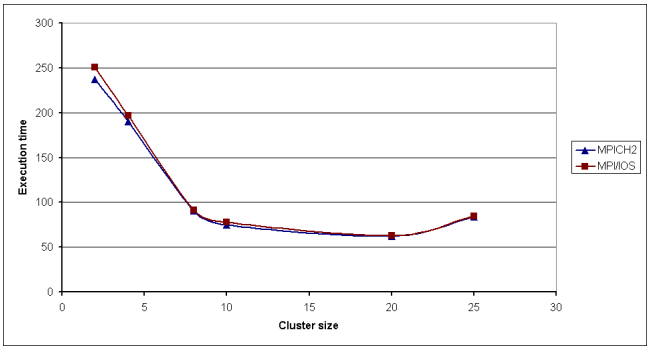

Figure 5. Overhead of the PCM library: Execution time of the heat application using different numbers of nodes with and without the PCM layer.

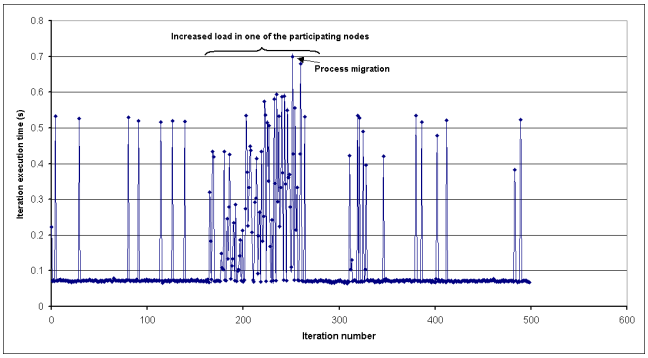

Figure 7. Breakdown of execution time of 2D heat application iterations on a 4 node cluster using MPI/IOS prototype.

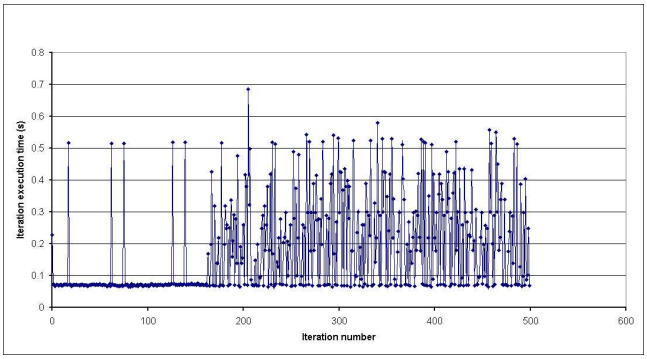

Figure 6. Breakdown of execution time of 2D heat application iterations on a 4 node cluster using MPICH2.

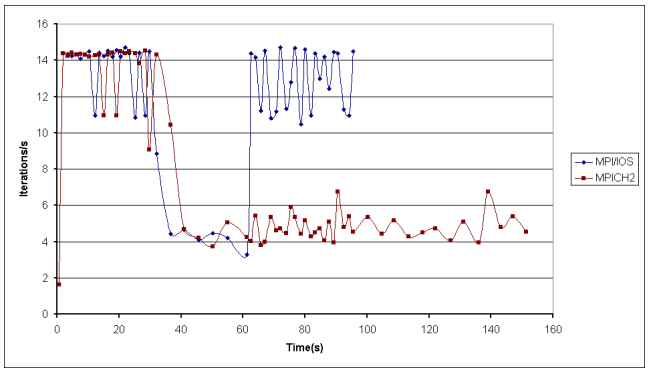

Figure 8. Measured throughput of the 2D heat application using MPICH2 and MPI/IOS. The applications adapted to the load change by migrating the affected process to one of the participating nodes in the case of MPI/IOS.

MPICH2.

6 Discussion and Future Work

This paper introduced several enhancements to MPI to allow for application reconfiguration and middleware-triggered dynamic load balancing. MPI/IOS improves MPI runtime systems with a library that allows process-level checkpointing and migration. This library is integrated with an adaptive middleware that triggers dynamic reconfiguration based on profiled resource usage and availability. The PCM library has been initially introduced in previous work [11]. We have made major redesign and improvements over the previous work, where the PCM architecture was centralized and supported only application-level migration. The new results show major improvements in scalability and performance. Our approach is portable and suitable for grid environments with no need to modify existing MPI implementations. Application developers need only insert a small number of API calls in MPI applications.

Our preliminary version of MPI/IOS has shown that process migration and middleware support are necessary to improve application performance over dynamic networks. MPI/IOS is a first step in improving MPI runtime environments with the support of dynamic reconfiguration. Our implementation of MPI process migration can be used on top of any implementation that supports the MPI-2 standard. It could also be easily integrated with grid-enabled implementations such as MPICH-G2 [19] once they become MPI-2 compliant. Our load balancing middleware could be combined with several advanced checkpointing techniques (e.g., [3, 6, 13, 23]) to provide a better integrated software support for MPI application reconfiguration.

MPI/IOS is still a work in progress. Future work includes: 1) using the MPI profiling interface to dis-

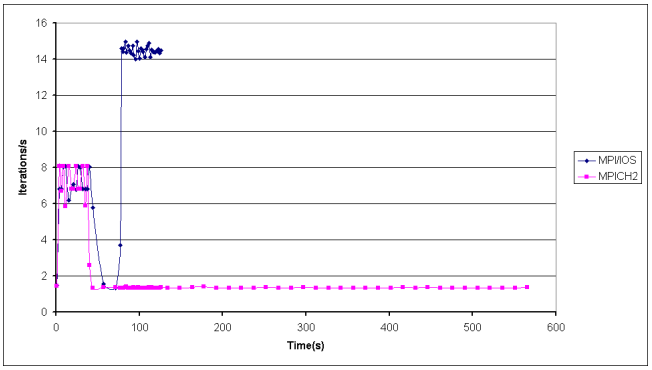

Figure 9. Measured throughput of the 2D heat application using MPICH2 and MPI/IOS. The applications adapted to the load change by migrating the affected process to a fast machine that joined the computation in the case of MPI/IOS.

cover communication patterns in order to provide a better mapping between application topologies and environment topologies, 2) evaluating different resource management models and load balancing decision procedures, 3) extending our approach to support non-iterative applications, 4) changing the granularity of reconfiguration units through middleware-triggered splitting and merging of executing processes, and 5) targeting more complex applications.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31. IEEE Computer Society, 1999.

[2] Argone National Laboratory. MPICH2,`http://www-unix.mcs.anl.gov/mpi/mpich2`.

[3] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte. MPI/FTTM: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 26. IEEE Computer Society, 2001.

[4] M. A. Bhandarkar, L. V. Kale;, E. de Sturler, and J. Hoeflinger. Adaptive load balancing for MPI programs. In *Proceedings of the International Conference on Computational Science-Part II*, pages 108–117. Springer-Verlag, 2001.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[6] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[7] K. Clark, J. E. Flaherty, and M. S. Shephard. *Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations*, 14, 1994.

[8] T. Desell, K. ElMaghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, Hawaii, January 2004.

[9] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. Technical Report Technical Report CS-04-02, Williams College Department of Computer Science, 2004. To appear, *Appl. Numer. Math.*

[10] K. ElMaghraoui, T. Desell, and C. Varela. Network sensitive reconfiguration of distributed applications. In *Submitted to the 25th international conference on distributed computing systems*, 2005.

[11] K. ElMaghraoui, J. E. Flaherty, B. K. Szymanski, J. D. Teresco, and C. Varela. Adaptive computation over dynamic and heterogeneous networks. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, *Proc. Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003)*, volume 3019 of *Lecture Notes in Computer Science*, pages 1083–1090, Czestochowa, 2004. Springer Verlag.

[12] R. Elsasser, B. Monien, and R. Preis. Diffusive load balancing schemes on heterogeneous networks. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 30–38. ACM Press, 2000.

[13] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353. Springer-Verlag, 2000.

[14] J. Faik, L. G. Gervasio, J. E. Flaherty, J. Chang, J. D. Teresco, E. G. Boman, and K. D. Devine. A model for resource-aware load balancing on heterogeneous clusters. Technical Report CS-04-03, Williams College Department of Computer Science, 2004. Presented at Cluster '04.

[15] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1998.

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15, 2001.

[17] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 530. IEEE Computer Society, 1995.

[18] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[19] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: a grid-enabled implementation of the Message Passing Interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[20] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall/Winter 1994.

[21] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1996.

[22] O. Sievert and H. Casanova. A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341–352, 2004.

[23] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

[24] S. S. Vadhiyar and J. J. Dongarra. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. In *Parallel Processing Letters*, volume 13, pages 291–312, June 2003.