Grid Computing: New Frontiers of High Performance Computing, Elsevier, 2005

Towards a Middleware Framework for Dynamically Reconfigurable Scientific Computing

Kaoutar El Maghraoui[a], Travis Desell[a], Boleslaw K. Szymanski[a], James D. Teresco[b], and Carlos A. Varela[a]

[a]Department of Computer Science, Rensselaer Polytechnic Institute,
110 8th Street, Troy, NY 12180-3590, USA

[b]Department of Computer Science, Williams College,
47 Lab Campus Drive, Williamstown, MA 01267, USA

Computational grids are appealing platforms for the execution of large scale applications among the scientific and engineering communities. However, designing new applications and deploying existing ones with the capability of exploiting this potential still remains a challenge. 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. In this paper, we introduce a new software framework that enhances the performance of Message Passing Interface (MPI) applications through an adaptive middleware for load balancing that includes process checkpointing and migration. Fields as diverse as fluid dynamics, materials science, biomechanics, and ecology make use of parallel adaptive computation. 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.

1. Introduction

Computational grids [1] have become very attractive platforms for high performance distributed applications due to their high availability, scalability, and computational power. However, nodes in grid environments (e.g., uniprocessors, symmetric multiprocessors (SMPs), or clusters) are not necessarily dedicated to a single parallel or distributed application. They experience constantly changing processing 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. Since they span a wider range of geographical locations and involve large numbers of computational nodes, the potential for failures and load fluctuations increases significantly.

Computationally-demanding scientific and engineering applications that arise in diverse disciplines such as fluid dynamics, materials science, and biomechanics often involve solving or simulating multi-scale problems with dynamic behavior. Solution procedures use

sophisticated adaptive methods underlying data structures (e.g., meshes) and numerical methods to achieve specified levels of solution accuracy [2]. This adaptivity, when used for parallel solution procedures, introduces load imbalance which can be corrected using application-level dynamic load balancing techniques [3]. These applications generally deal with huge amounts of data and require extensive computational resources, but they usually assume a fixed number of cooperating processes running in a dedicated and mostly homogeneous computing environment. Running such applications on computational grids, with their dynamic, heterogeneous, and non-dedicated resources, makes it difficult for application-level load balancing alone to take full advantage of available resources and to maintain high performance. Application-level load-balancing approaches have a limited view of the external world where the application is competing for resources with several other applications. Middleware is a more appropriate location where to place resource management and load balancing capabilities since it has a more global view about the execution environment, and can benefit a large number of applications.

MPI [4] has been widely adopted as the de-facto standard to implement single-program multiple-data (SPMD) parallel applications. Extensive development effort has produced many very large software systems using MPI for parallelization. It's wide availability has enabled portability of applications among a variety of parallel computing environments. 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 [5,6] 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 for application reconfiguration is therefore needed to relieve application developers from such concerns. Augmenting MPI applications with automated process migration capabilities is a necessary step to enable dynamic reconfiguration through load balancing of MPI processes among geographically distributed nodes. We initially address dynamic reconfiguration through process migration for the class of iterative applications since a large number of legacy MPI applications have this property.

The purpose of this paper is two-fold: first we demonstrate how we achieve process migration in applications that follow the MPI programming model. Our strategy doesn't require modifying existing MPI implementations. Second we introduce the design of a middleware infrastructure that enhances existing MPI applications with automatic reconfiguration in a dynamic setting. The Internet Operating System (IOS) [7,8] is a distributed middleware framework that provides opportunistic load balancing capabilities through resource-level profiling and application-level profiling. 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 shared environments where many applications are running, having application-level resource management is not enough to balance the load of the

entire system efficiently. A middleware layer is the natural place to manage the resources of several distributed applications running simultaneously.

Providing simple application programmer interfaces (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 its performance continuously, discover new resources, and be able to migrate some or all of the application's cooperating processes 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 or migrating its work to a faster processor can, in many cases, lead to a significant overall performance improvement.

The remainder of the paper is organized as follows. Section 2 presents application motivating scenarios for reconfigurable execution. Section 3 discusses the requirements and benefits of dynamically reconfigurable parallel and distributed applications. In Section 4, we describe our methodology for enabling reconfiguration of distributed and parallel applications using the MPI programming model. We then present IOS resource model and load balancing strategies in Section 5. Section 6 details the architecture of the MPI/IOS framework and presents experimental results. Section 7 presents related work. We conclude with discussion and future work in Section 8.

2. Dynamically Changing Computations

In a grid environments, one of the following scenarios might happen:

1. The application can predicate initially its resource requirements and the allocated resources' utilization and availability do not change drastically over time.

2. The application has a dynamic nature. The initial resource requirements of the application are hard to predicts or the application's problem size might grow or shrink in time.

3. The execution environment is dynamic. This is usually the case of dynamic grid environments where resource are shared. Resources experience varying loads and availability.

In the first case, a good initial resource allocation might suffice to provide the desired performance throughout the lifetime of the application. However in the second and third cases, adaptive execution is needed to either cope with the dynamic nature of the application or to adapt to the dynamic nature of the execution environment. The last two scenarios are expected to be the rule and not the exception in dynamic grids. Therefore, providing the necessary support for reconfiguration is indispensable. The rest of this section describes the characteristics of mesh-based adaptive scientific computation, a motivating application scenario for reconfigurable applications.

Adaptive scientific computation is dynamic by nature. A typical simulation begins with a small initial mesh, and adaptive refinement produces finer meshes (i.e., meshes with larger number of elements) in regions where interesting solution features are present, and does this as part of the simulation process. As these solution features arise and dissipate or move through the domain, some of the refined mesh may be coarsened. Thus, the locations in the domain where the finer mesh is needed as well as the total mesh size change throughout the computation.

When using the traditional MPI model, the computing resources are allocated initially and simulations begin with a partitioning of the (small) initial mesh and dynamic load balancing procedures are applied periodically to redistribute the mesh among the cooperating processes on the allocated processors. The computing resources assigned to the problem remain the same throughout, but the workload is redistributed among them. Even when the problem size is small and the solution would be more efficient on fewer processors (because of reduced communication volume), all of the allocated processors are used, in part because it is difficult to add or remove processes using the MPI model.

In the more dynamic environments we target and that the middleware described herein is intended to support, resources may be added to or removed from the computation as the simulation proceeds. This sort of reconfiguration may be in response to the changing computational needs of the simulations or to the changing availability of resources. Parts of the computation that can be executed more efficiently on fewer processors may do so, leaving other resources available for other purposes. Additional resources can be requested and the computation can be reconfigured to take advantage of those resources when the computation grows sufficiently large.

3. Reconfigurable Distributed and Parallel Applications

Grid computing thrives to provide mechanisms and tools to allow decentralized collaboration of geographically distributed resources across various organizations in the Internet. One of its main goals is to maximize the use of underutilized resources and hence offer efficient and low cost paradigms for efficiently executing distributed applications. Grid environments are highly dynamic, shared, and heterogeneous. Running distributed or parallel applications on such platforms is not a trivial task. Applications need to be able to adapt to the various dynamics of the underlying resources through dynamic reconfiguration mechanisms. Dynamic reconfiguration implies the ability to modify the application's structure and/or modify the mapping between physical resources and application's components while the application continues to operate without any disruption of service.

Grid applications have several needs:

- Availability: The ability of the application to be resilient to failures.
- Scalability: The ability of the application to use new resources while the computation is taking place.
- High Performance: The ability to adapt to load fluctuations, which results in high performing applications.

Several resource management requirements need to be addressed to satisfy applications' demands in grid environments such as: 1) resource allocation, 2) reallocation or reconfiguration of these resources, and 3) resource profiling to provide an optimal reconfiguration. All of these resource management issues are beyond the scope of applications and should be embodied in smart middleware that is capable of harnessing available resources and allocating them properly to running applications.

3.1. Resource Allocation and Reallocation

Resource discovery and allocation have been some of the paramount issues in the grid community. Several applications may be competing of resources. Some sort of admission control needs to be established to ensure sufficient resources exist before admitting any new resource requests. The initial allocation of resources may not be the final configuration to achieve the desired performance. The dynamic nature of grids necessitates a constant evaluation of the application needs, of the resource availability, and an efficient reallocation strategy.

Being able to change the mapping of applications' components to physical resources requires having the ability to migrate whole or parts of the application's processes or data at runtime. Process migration requires being able to save the current state of the running application, ensuring no loss of messages while migration is in progress, and then restoring the state. Data migration, on the other hand, requires programming support from developers. Migration is an expensive procedure and should be performed only if it will yield gains in the overall performance of the running application. Process or data migration capabilities allow also load balancing.

3.2. Resource Profiling

Understanding the behavior of the application's topology helps to provide a good partitioning and an optimal placement of its components over decentralized heterogeneous resources. One of the important characteristics of distributed/parallel applications are the communication patterns between their several components. Applications can range from highly synchronized where components communicate frequently to massively parallel. An optimal mapping of application's components to physical resources must try to maximize the utilization of resources (CPU, memory, storage, etc.) while minimizing communication delays across links.

Changing effectively the configuration of running applications entails understanding the computational and communication requirements of their various components. One approach is to infer this information statically from performance models supplied by the users. The problem with this approach is that it may not very accurate. Additionally several applications do not render themselves nicely to mathematical models due to their complexity. One way to address this issue is by learning the topology of the application at runtime through application-level profiling. This approach requires modifying existing applications to include profiling and might incur some overhead.

4. Dynamically Reconfigurable 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-triggered reconfiguration.

4.1. Application Support: 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. Iterative applications exhibit 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.

4.2. 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

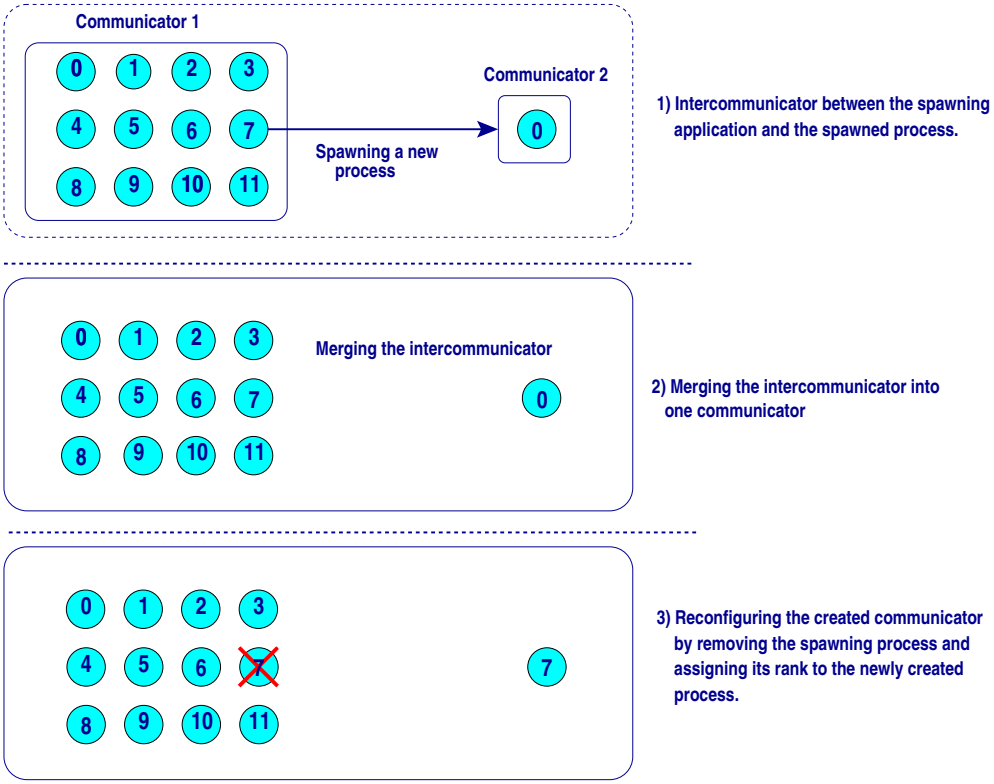

Figure 1. Steps involved in communicator handling to achieve MPI process migration.

should be reconfigured to accomplish load balancing 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 [9]. We have shown that considering the application topology in the load balancing decision procedures dramatically improves throughput over purely random work stealing [7]. IOS supports two load-balancing protocols: 1) application topology sensitive load balancing and 2) network topology sensitive load balancing [7,8]. More details about IOS virtual network topologies and its load balancing strategies are presented in Section 5.

Applications communicate with the IOS middleware through clearly defined interfaces that permit the exchange of profiled information and reconfiguration requests. Applica-

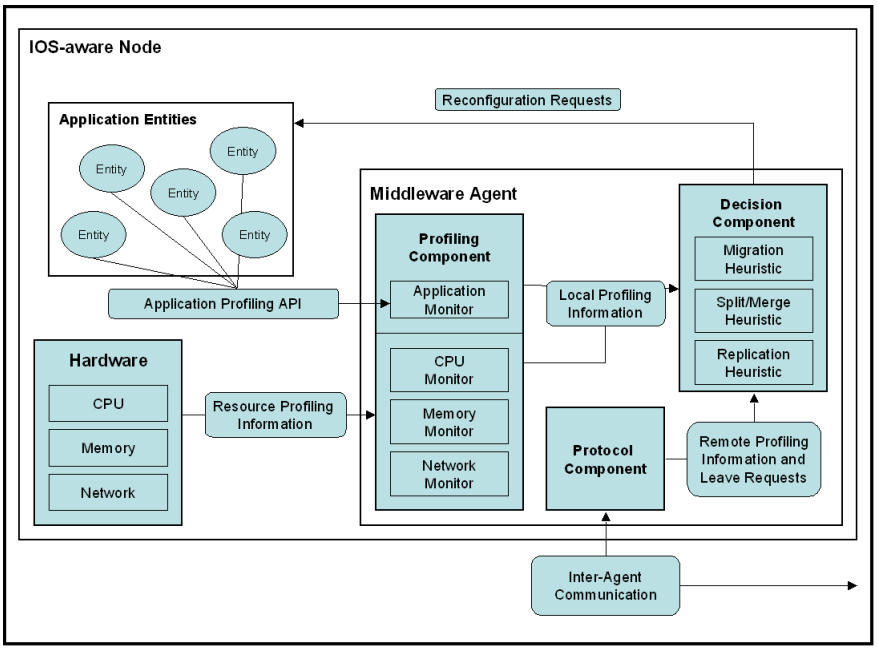

Figure 2. Architecture of a node in the Internet Operating System middleware (IOS). An agent collects profiling information and makes decisions on how to reconfigure the application based on its decisions, protocol, and profiling components.

tions need to support migration to react to IOS reconfiguration requests.

5. Middleware-level Load Balancing Policies

The IOS architecture has a decentralized architecture to ensure robustness, scalability and efficiency. Each IOS-ready node is equipped with a *middleware agent*. Agents organize themselves in various virtual network topologies to sense the underlying physical environment and trigger accordingly applications' reconfiguration. Decision components are embodied in each agent to evaluate the surrounding environment and decide based on a resource sensitive model (RSM) how to balance the resource consumption of application's entities in the physical layer. RSM provides a normalized measure of the improvement in resource availability an entity would receive by migrating between nodes (see Figure 3 for details). The RSM uses the profiled information about applications' entities to decide which ones are the most beneficial to migrate. In what follows we describe the different virtual topologies of the IOS agents and the various load balancing policies implemented as part of the the IOS framework.

5.1. Virtual Network Topologies

We are considering *network sensitive* virtual network topologies which adjust themselves according to the underlying network topologies and conditions. We present two types of representative topologies: a *peer-to-peer* (p2p) topology and a *cluster-to-cluster*

Notation	Explanation
A	A group of application entities.
$\mathcal{A}_{r,f}$	The amount of available resource r at node f.
$\mathcal{U}_{r,l,A}$	The amount of resource r used by A at node l.
R	The set of all resources to be considered by the resource sensitive model.
w_r	A weight for a given resource r, where $\sum w_r = 1$
$\mathcal{C}_{l,f,A}$	The cost of migrating the set of application entities A from l to f
$\mathcal{E}_A$	The average life expectancy of the set of application entities A, where $0 \leq (\frac{\mathcal{C}_{l,f,A}}{(10+log(\mathcal{E}_A))}) \leq 1$
$\Delta_{r,l,f,A}$	The overall improvement in performance the application would receive in terms of resource r by migrating the set of entities A from node l to node f, where $\Delta_{r,l,f,A}$ is normalized between -1 and 1. $\Delta_{r,l,f,A} = \frac{\mathcal{A}_{r,f} - \mathcal{U}_{r,l,A}}{\mathcal{A}_{r,f} + \mathcal{U}_{r,l,A}}$
$gain(l, f, A)$	A normalized measure of the overall improvement gained by migrating a set of entities A from local node l to foreign node f. $gain(l, f, A) = (\sum_r w_r * \Delta_{r,l,f,A}) - (\frac{\mathcal{C}_{l,f,A}}{(10+log(\mathcal{E}_A))})$

Figure 3. The resource sensitive model (RSM) used by the IOS decision component to determine which entities to migrate between nodes.

(c2c) topology. The p2p topology consists of several heterogeneous nodes inter-connected in a peer-to-peer fashion while the c2c topology imposes more structure on the virtual network by grouping homogeneous nodes with low inter-network latencies into clusters.

A Network Sensitive Peer-to-Peer Topology (NSp2p)

Agents initially connect to the IOS virtual network either through other known agents or through a *peer server*. Peer servers act as registries for agent discovery. Upon contacting a peer server, an agent registers itself and receives a list of other agents (peers) in the virtual network. Peer servers simply aid in discovering peers in a virtual network and are not a single point of failure. They operate similarly to gnutella-hosts in Gnutella peer-to-peer networks [10]. After an agent has connected to the virtual network, it can discover new peers as information gets passed across peers. Agents can also dynamically leave the virtual network. Previous work discusses dynamic addition and removal of nodes in the IOS middleware [7].

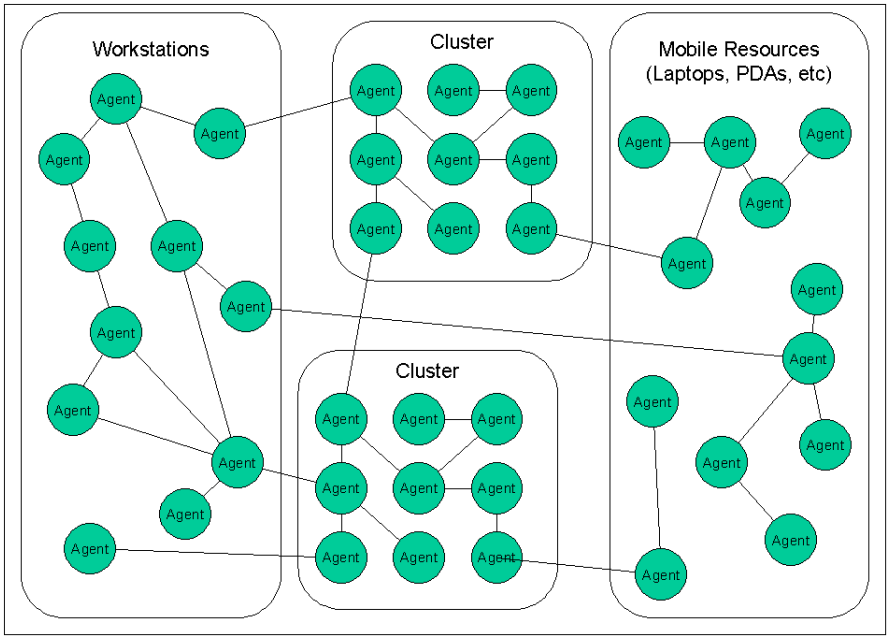

Figure 4. The peer-to-peer virtual network topology. Middleware agents represent heterogeneous nodes, and communicates with groups or peer agents. Information is propagated through the virtual network via these communication links.

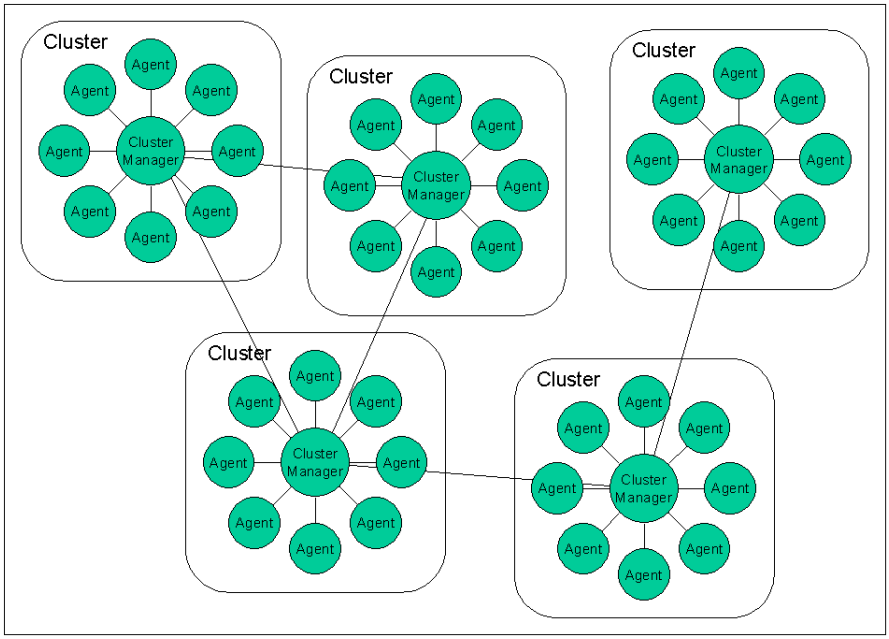

Figure 5. The cluster-to-cluster virtual network topology. Homogeneous agents elect a cluster manager to perform intra and inter cluster load balancing. Clusters are dynamically created and readjusted as agents join and leave the virtual network.

A Network Sensitive Cluster-to-Cluster Topology (NSc2c)

In NSc2c, agents are organized into groups of virtual clusters (VCs), as shown in Figure 5. Each VC elects one agent to act as the cluster manager. VCs may reconfigure themselves as necessary by splitting or merging depending on the overall performance of the running applications. Cluster managers view each other as peers and organize themselves as a NSp2p virtual network topology.

5.2. Autonomous Load Balancing Strategies

IOS adopts two load balancing strategies depending on the kind of virtual topology used by the middleware agents.

Peer-to-peer Load Balancing

Peer-to-peer load balancing is based on a simple but effective work stealing algorithm described by [9]. Agents configuring themselves in the NSp2p topology keep a list of peers and arrange these peers into four groups based on communication latency [11]: 1) local (0 to 10 ms), 2) regional (11 to 100 ms), 3) national (101 to 250 ms), and 4) global (251 ms and higher).

Agents on nodes which are *lightly loaded* (have more resources available than are currently being utilized) will periodically send reconfiguration request packets (RRPs) containing locally profiled information to a random peer in the local group. The decision component will then decide if it is beneficial to migrate entities to the source of the RRP according to the RSM. If it decides not to migrate any entities, the RRP is propagated to a local peer of the current agent. This progresses until the RRP's time to live has elapsed, or the desired entities have been migrated. If no migration happens, the source of the RRP will send another RRP to a regional peer, and if no migration occurs again, an RRP is sent nationally, then globally. As reconfiguration is only triggered by lightly loaded nodes, no overhead is incurred when the network is fully loaded, and thus this approach is stable [12].

Cluster-to-cluster Load Balancing

The cluster-to-cluster strategy attempts to utilize central coordination within VCs in order to obtain an overall picture of the applications' communication patterns and resource consumption as well as the physical network of the VC. A cluster manager acts as the central coordinator for a VC and utilizes this relatively global information to provide both intra- and inter-VC reconfiguration.

Every cluster manager sends periodic profiling requests to the agents in its respective VC. Every agent responds with information from its profiling component about the local entities and their resource consumption. The cluster manager uses this information to determine which entities should be migrated from the node with the least available resources to the node with the most available resources. Let n_1 and n_2 be the number of entities running on two nodes, and $r_{i,j}$ be the availability of resource i on node j with a resource weight w_i. The intra-cluster load balancing continuously attempts to achieve the relative equality of application's entities on nodes according to their relative resource availability: $\frac{n_1}{n_2} = \frac{\sum w_i r_{i,1}}{\sum w_i r_{i,2}}$.

For inter-cluster load balancing, NSc2c uses the same strategy as peer-to-peer load balancing, except that each cluster manager is seen as a peer in the network. The cluster managers decision component compares the heaviest loaded node to the lightest loaded node at the source of the RRP to determine which entities to migrate.

Migration Granularity

Our resource model supports both single migration and group migration of application entities. In single migration, the model is applied to determine an estimation of the gain that would be achieved from migrating an entity from one theater to another. If the gain will be achieved by migrating a group of entities, single migration attempts to migration one entity at a time while group migration strategy will migrate a group of entities simultaneously. One advantage of group migration is that it helps to speed up the load balancing. However it might cause trashing behavior if the load of the nodes fluctuates very frequently.

5.3. Experimental Evaluation

We have evaluated IOS different load balancing strategies using benchmarks that represent various degrees of computation to communication ratios. The benchmarks have been developed using Java and SALSA [13], a dialect of Java with high level programming abstractions for universal naming, asynchronous message passing, and coordination strategies. Both hypercube and tree application topologies represent applications that have a high communication to computation ratio. While the sparse and tree application topologies represent applications with a low communication to computation ratio.

The experiments were evaluated using two different physical environments to model Internet-like networks and Grid-like networks. The first physical network consists of 20 machines running Solaris and Windows operating systems with different processing power and different latencies to model the heterogeneity of Internet computing environments. The second physical network consists of 5 clusters with different inter-cluster network latencies. Each cluster consists of 5 homogeneous SUN Solaris machines. Machines in different clusters have different processing power.

Figures 6 and 7 show that the p2p topology performs better in Internet-like environments that lack structure for highly synchronized parallel and distributed applications, while the c2c topology is more suitable for grid-like environments that have a rather hierarchical structure.

For a more thorough evaluation of IOS load balancing strategies, readers are referred to [8].

6. MPI Process Migration and Integration with IOS Middleware

MPI/IOS is implemented as a set of middleware services that interact with running applications through an MPI wrapper. The MPI wrapper uses a Process Checkpointing and Migration (PCM) library [14]. The MPI/IOS runtime architecture consists of the following components (see Figure 8): 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.

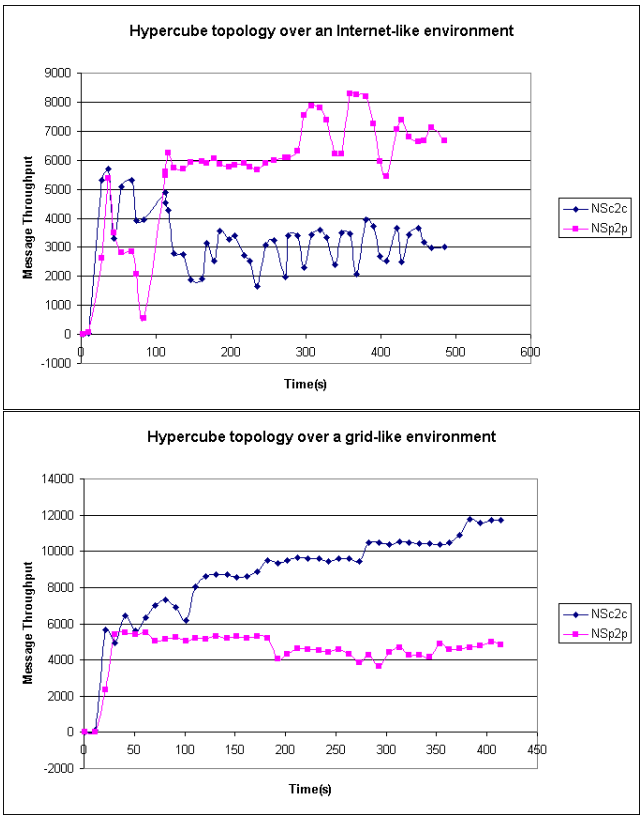

Figure 6. Message throughput for the hypercube application topology on Internet- and Grid-like environments.

6.1. 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

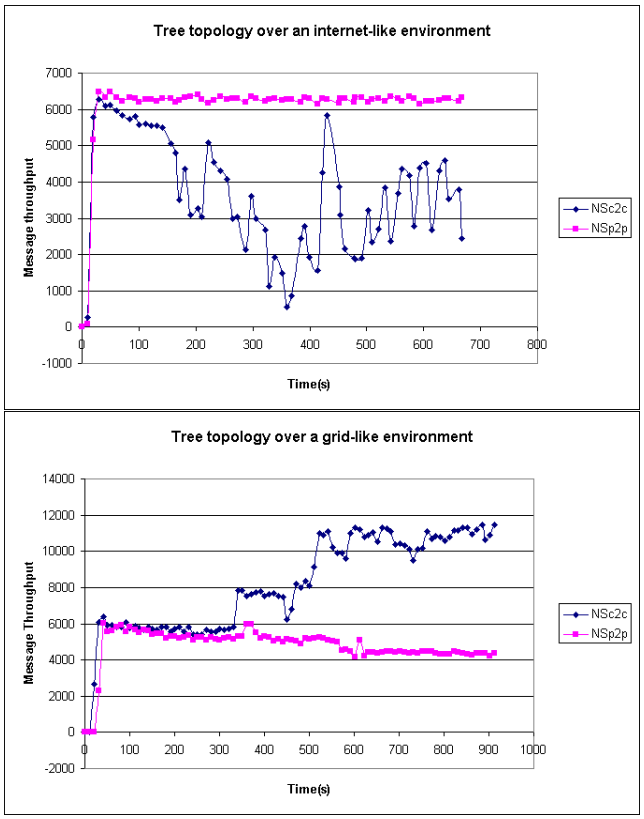

Figure 7. Message throughput for the tree application topology on Internet- and Grid-like environments.

process migration. Figures 10 and 11 show a skeleton of an MPI program and its modified version with the PCM calls to interface with IOS middleware.

Existing MPI applications interact with the PCM library and the native MPI implementation through a wrapper as shown in Figure 8. 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.

6.2. The PCM Library

Figure 9 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 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

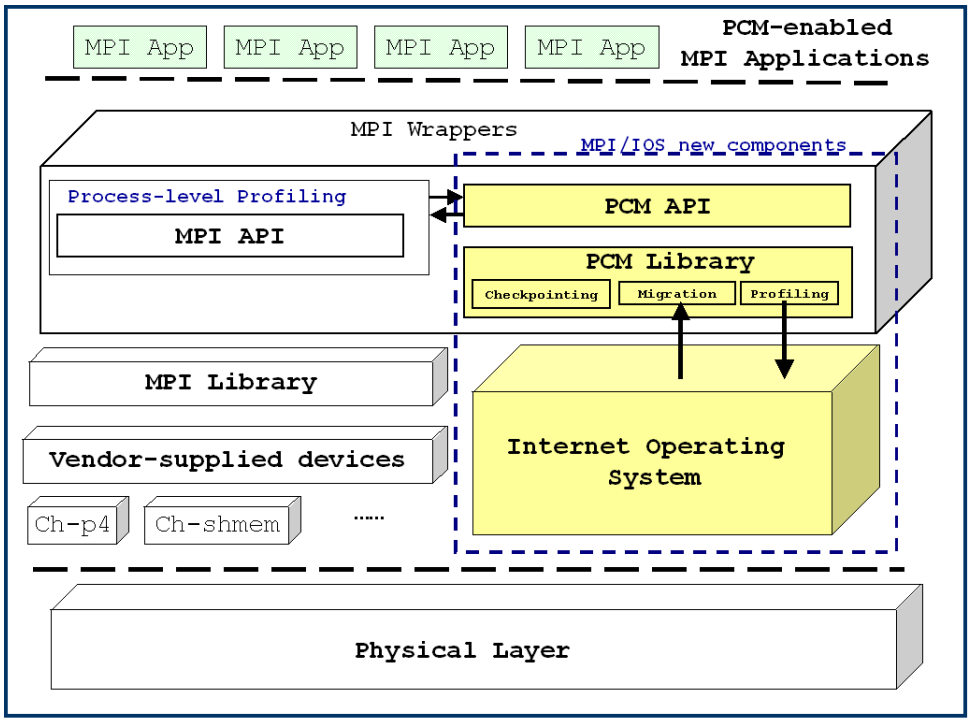

Figure 8. The layered design of MPI/IOS which includes the MPI wrapper, the PCM runtime layer, and the IOS runtime layer.

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.

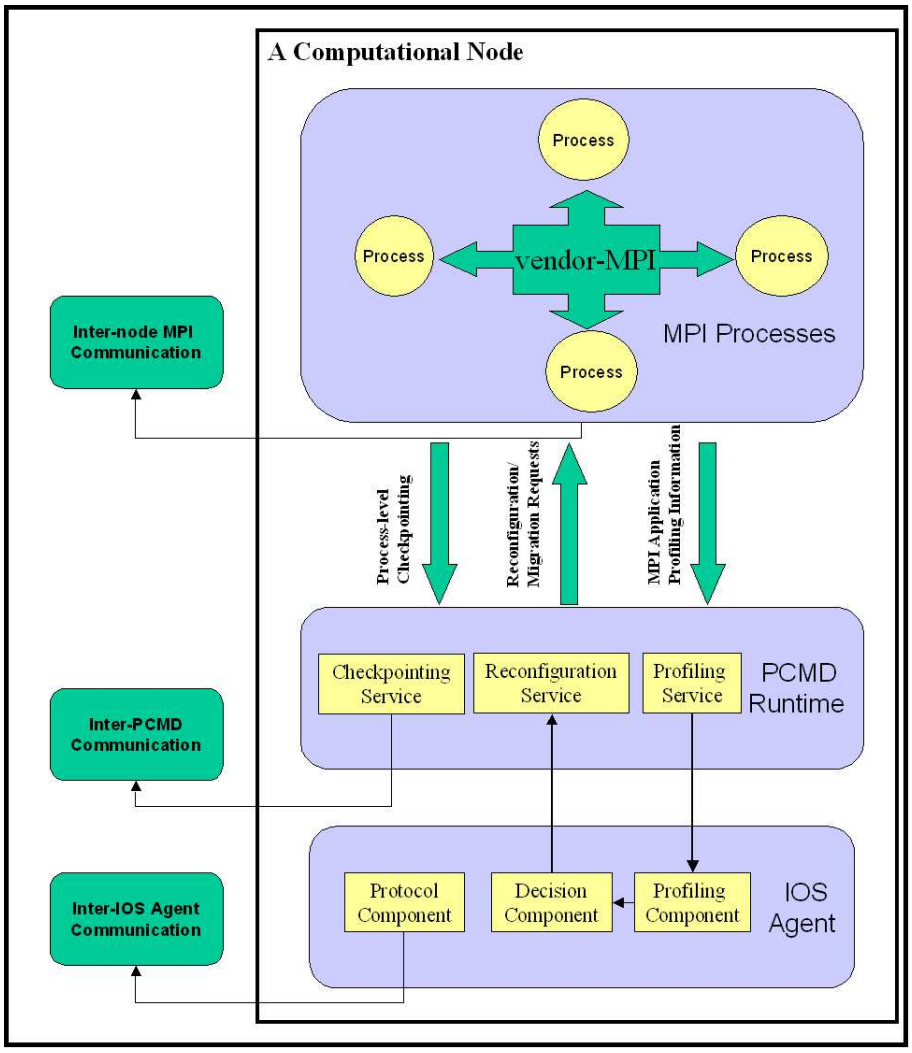

Figure 9. Architecture of a node running MPI/IOS enabled applications.

6.3. 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

```
#include <mpi.h>
...

int main(int argc, char **argv) {
   //Declarations
   ....

   MPI_Init( &argc, &argv );


   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &totalProcessors );

   current_iteration = 0;

   //Determine the number of columns for each processor.
   dataWidth = (WIDTH-2) / totalProcessors;

   //Initialize and Distribute data among processors
   ...

   for(iterations=current_iteration; iterations<TOTAL_ITERATIONS; iterations++){

      // Data Computation.
      ...

      //Exchange of computed data with neighboring processes.
      // MPI_Send() || MPI_Recv()
      ...
   }

   // Data Collection
   ...
   MPI_Barrier( MPI_COMM_WORLD );

   MPI_Finalize();
   return 0;
}
```

Figure 10. Skeleton of the original MPI code of a heat diffusion problem.

MB of memory. We used MPICH2 [15], a freely available implementation of the MPI-2 standard. Most of the experiments conducted try to demonstrate the usefulness of process migration when the allocated resources' load varies during the lifetime of the running application.

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 12 shows that the overhead of the PCM library is less than 5% for several sizes of the cluster.

The second experiment aims at evaluating the impact of process migration. The cluster of 4 dual-processor nodes was used. Figures 13 and 14 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 the same 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 13, the overall execution time of the application's iterations increased. The highly synchronized

```
#include "pcm.h"
...

MPI_Comm PCM_COMM_WORLD;

int main(int argc, char **argv) {
   //Declarations
   ....
   int spawnrank=-1, current_iteration;
   PCM_Status pcm_status;
   MPI_Init( &argc, &argv );
   PCM_COMM_WORLD = MPI_COMM_WORLD;
   PCM_Init(PCM_COMM_WORLD);

   MPI_Comm_rank( PCM_COMM_WORLD, &rank );
   MPI_Comm_size( PCM_COMM_WORLD, &totalProcessors );

   spawnrank = PCM_Process_Status();

   if(spawnrank <= 0){
      current_iteration = 0;

      //Determine the number of columns for each processor.
      dataWidth = (WIDTH-2) / totalProcessors;

      //Initialize and Distribute data among processors
      ...
   }
   else{
      PCM_Load(spawnrank, "iterator",&current_iteration);
      PCM_Load(spawnrank, "datawidth", &dataWidth);
      prevData = (double *)calloc( (dataWidth+2)*WIDTH,sizeof(double) );
      PCM_Load(spawnrank, "myArray",prevData);
   }

   for(iterations=current_iteration; iterations<TOTAL_ITERATIONS; iterations++){
      pcm_status = PCM_Status(PCM_COMM_WORLD);
      if(pcm_status == PCM_MIGRATE){
         PCM_Store(rank, "iterator", &iterations, PCM_INT, 1);
         PCM_Store(rank, "datawidth", &dataWidth, PCM_INT, 1);
         PCM_Store(rank,"myArray", prevData, PCM_DOUBLE, (dataWidth+2)*WIDTH);

         PCM_COMM_WORLD = PCM_Reconfigure(PCM_COMM_WORLD,"mpiheat");

      }
      else if(pcm_status == PCM_RECONFIGURE)
      {
         PCM_COMM_WORLD =  PCM_Reconfigure(PCM_COMM_WORLD,"mpiheat");
         MPI_Comm_rank(PCM_COMM_WORLD, &rank);
      }

      // Data Computation.
      ...

      //Exchange of computed data with neighboring processes.
      // MPI_Send() || MPI_Recv()
      ...
   }

   // Data Collection
   ...
   MPI_Barrier( PCM_COMM_WORLD );

   PCM_Finalize(PCM_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Figure 11. Skeleton of the instrumented MPI code of a heat diffusion problem with PCM calls.

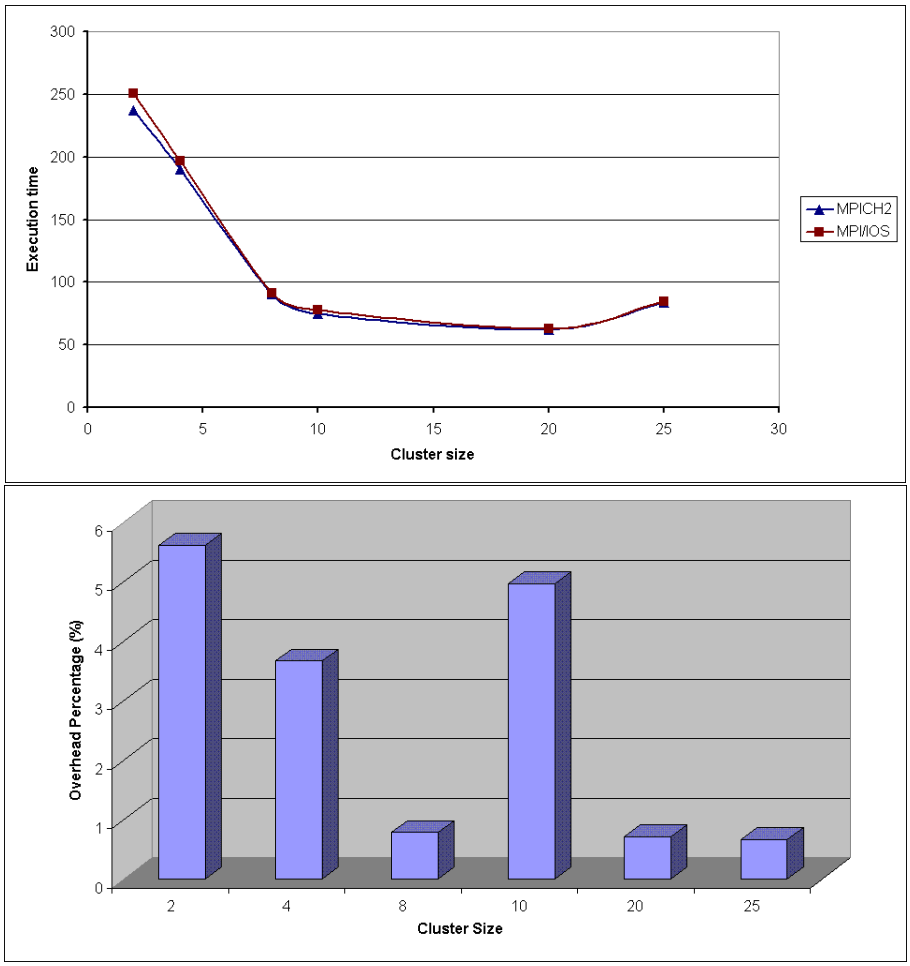

Figure 12. Overhead of the PCM library: Execution time of the heat application using different numbers of nodes with and without the PCM layer.

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 14 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. Figure 14 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 15 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

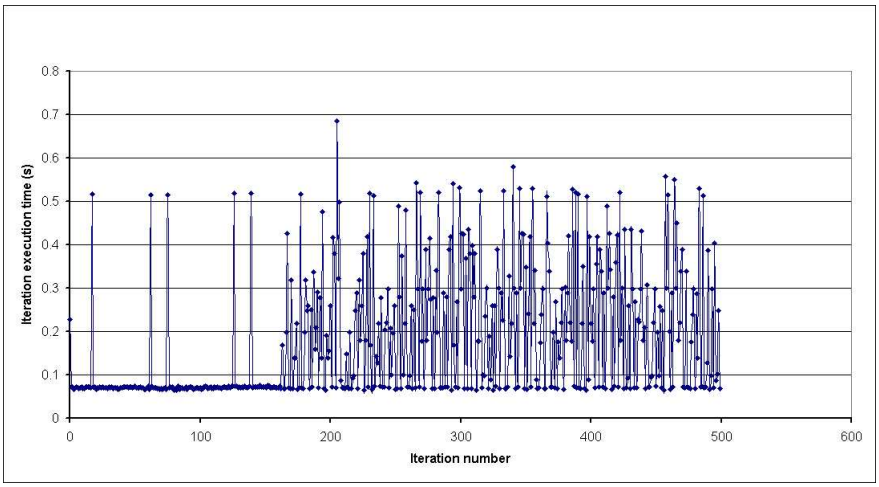

Figure 13. Breakdown of execution time of two-dimensional heat application iterations on a 4-node cluster using MPICH2.

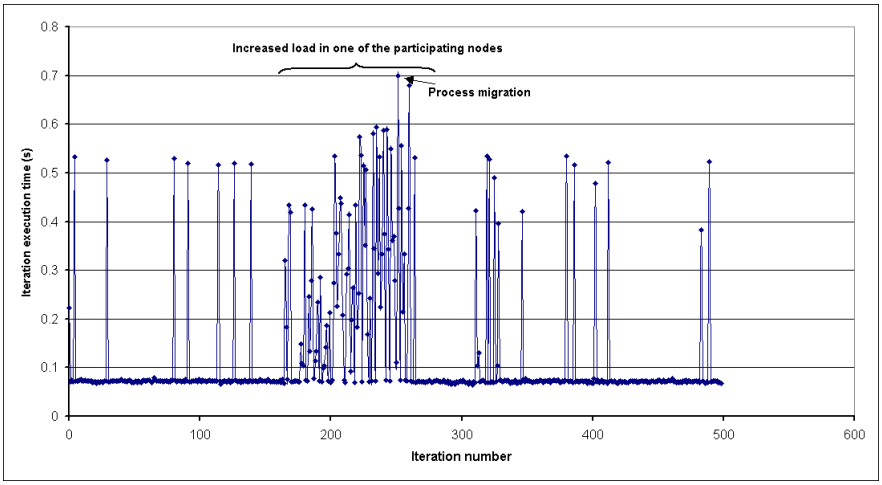

Figure 14. Breakdown of execution time of two-dimensional heat application iterations on a 4-node cluster using MPI/IOS prototype.

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 16, 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

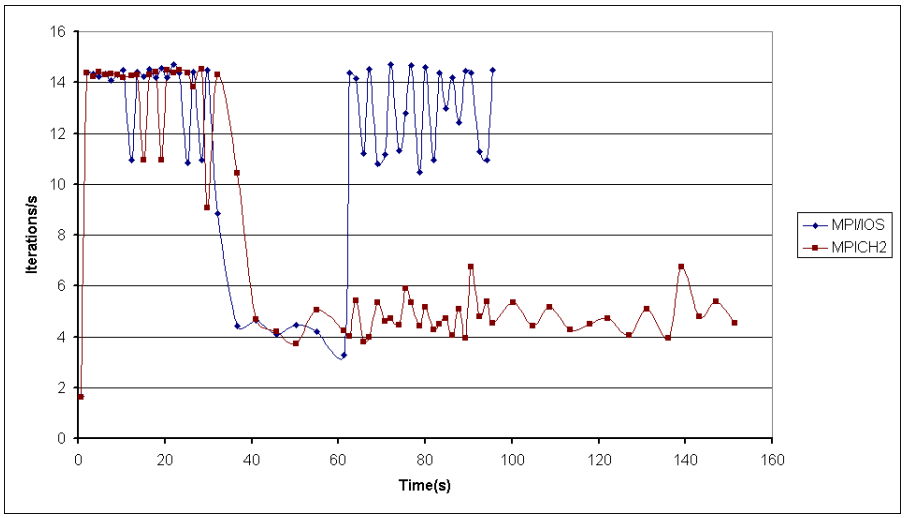

Figure 15. Measured throughput of the two-dimensional heat application using MPICH2 and MPI/IOS. The applications adapted to the load change by migrating the affected process to one of the participating nodes in the case of MPI/IOS.

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 MPICH2.

To evaluate the cost of reconfiguration, we varied the problem data size and measured the overhead of reconfiguration in each case. In the conducted experiments, we started the application on a local cluster. We then introduced artificial load in one of the participating machines. One execution was allowed to reconfigure by migrating the suffering process to an available node that belongs to a different cluster, while the second execution was not allowed to reconfigure itself. The experiments in Figure 17 show that in the studied cases, reconfiguration overhead was negligible. In all cases, it accounted for less than 1% of the total execution time. The application studied is not data-intensive. We also used an experimental testbed that consisted of 2 clusters that belong to the same institution. So the network latencies were not significant. The reconfiguration overhead is expected to increase with larger latencies and larger data sizes. However, reconfiguration will still be beneficial in the case of large-scale long-running applications. Figure 18 shows the breakdown of the reconfiguration cost. It consists of checkpointing, loading checkpoints, and re-arranging the communicators in the case of MPI-based applications.

7. 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

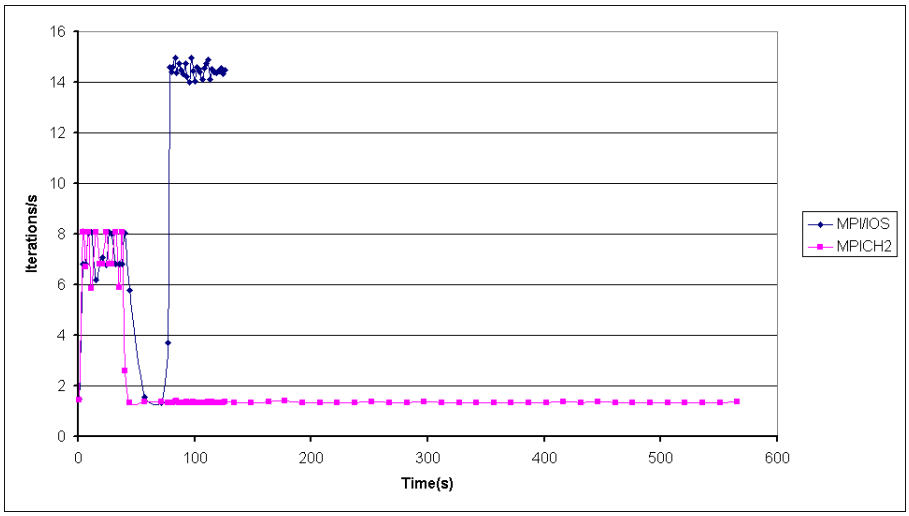

Figure 16. Measured throughput of the two-dimensional heat application using MPICH2 and MPI/IOS. The applications adapted to the load change by migrating the affected process to a fast machine that joined the computation in the case of MPI/IOS.

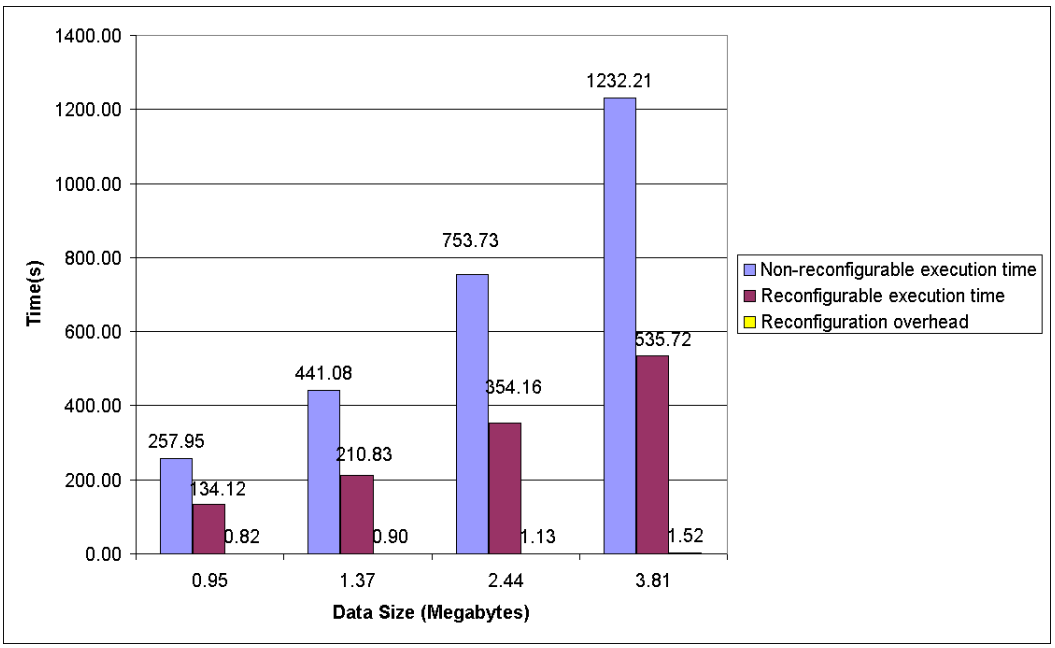

Figure 17. Execution time for a reconfigurable and non-reconfigurable execution scenarios for different problem data sizes. The graph shows also the reconfiguration overhead for each problem size.

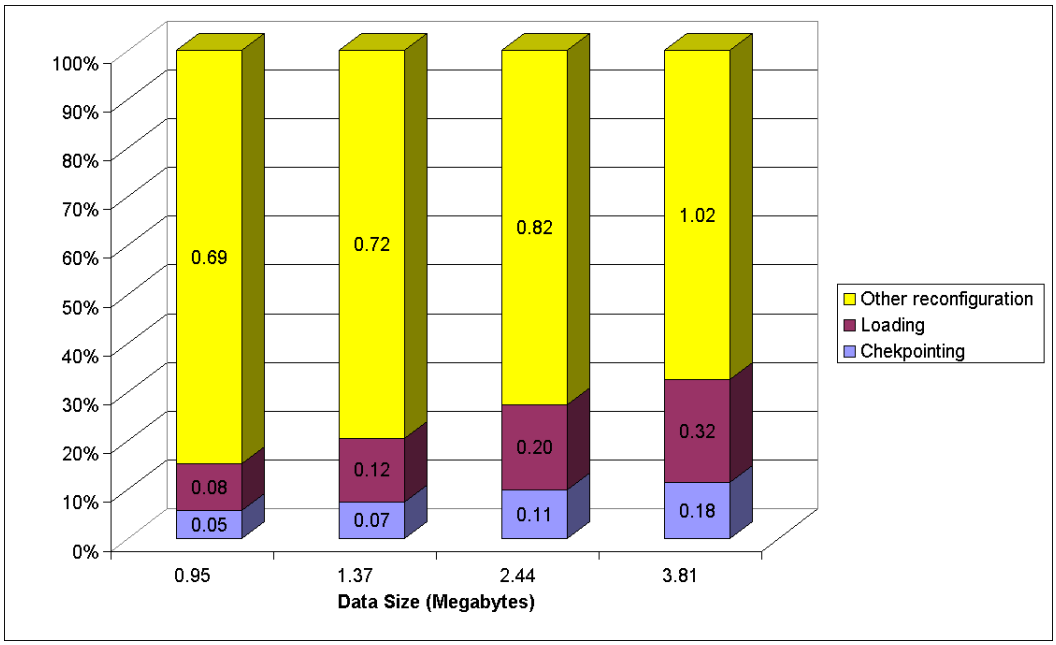

Figure 18. Breakdown of the reconfiguration overhead for the experiment of Figure 17.

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., [3,16,17]), 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 [16,18] 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) [19,20] 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 [21] 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 [22], starFish [23], 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 trivial 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 [21,24]. 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.

8. Discussion and Future Work

This paper introduced several enhancements to MPI to allow for application reconfiguration through 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 [14]. 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 [25] once they become MPI-2 compliant. Our load balancing middleware could be combined with several advanced checkpointing techniques (e.g., [22,26–28]) 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 discover 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.

9. 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 Joseph Chabarek, and WeiJen Wang for their careful readings and comments. The machines used in our experiments have been partially supported by the IBM SUR 2003 Award. Any errors or omissions remain our own. This work has been partially supported by NSF CAREER Award No. CNS-0448407.

REFERENCES

1. I. T. Foster, The anatomy of the grid: Enabling scalable virtual organizations, in: Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, Springer-Verlag, 2001, pp. 1–4.
2. K. Clark, J. E. Flaherty, M. S. Shephard, Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations 14.
3. J. D. Teresco, K. D. Devine, J. E. Flaherty, Numerical Solution of Partial Differential Equations on Parallel Computers, Springer-Verlag, 2005, Ch. Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations.
4. 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.
5. W. Gropp, E. Lusk, Dynamic process management in an MPI setting, in: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, IEEE Computer Society, 1995.
6. Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface (1996).
   URL `citeseer.ist.psu.edu/396449.html`
7. T. Desell, K. E. Maghraoui, C. Varela, Load balancing of autonomous actors over dynamic networks, in: Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track, Hawaii, 2004.
8. K. E. Maghraoui, T. Desell, C. Varela, Network sensitive reconfiguration of distributed applications, Tech. Rep. CS-05-03, Rensselaer Polytechnic Institute (2005).
9. R. D. Blumofe, C. E. Leiserson, 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, pp. 356–368.

10. Clip2.com, The Gnutella protocol specification v0.4 (2000).
URL http://www9.limewire.com/developer/gnutella_protocol_0.%4.pdf
11. T. T. Kwan, D. A. Reed, Performance of an infrastructure for worldwide parallel computing, in: 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico, 1999, p. 379.
12. N. G. Shivratri, P. Kreuger, M. Ginghal, Load distributing for locally distributed systems, IEEE Computer 25 (92) 33–34.
13. C. Varela, G. Agha, 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.
14. K. E. Maghraoui, J. E. Flaherty, B. K. Szymanski, J. D. Teresco, C. Varela, Adaptive computation over dynamic and heterogeneous networks, in: R. Wyrzykowski, J. Dongarra, M. Paprzycki, J. Wasniewski (Eds.), Proc. Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003), Vol. 3019 of Lecture Notes in Computer Science, Springer Verlag, Czestochowa, 2004, pp. 1083–1090.
15. Argone National Laboratory, MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2.
16. R. Elsasser, B. Monien, R. Preis, Diffusive load balancing schemes on heterogeneous networks, in: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, ACM Press, 2000, pp. 30–38.
17. J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, L. H. Ziantz, Parallel structures and dynamic load balancing for adaptive finite element computation, Applied Numerical Mathematics 26 (1998) 241–263.
18. J. D. Teresco, J. Faik, J. E. Flaherty, Resource-aware scientific computation on a heterogeneous cluster, Computing in Science & Engineering 7 (2) (2005) 40–50.
19. M. A. Bhandarkar, L. V. Kaleé;, E. de Sturler, J. Hoeflinger, Adaptive load balancing for MPI programs, in: Proceedings of the International Conference on Computational Science-Part II, Springer-Verlag, 2001, pp. 108–117.
20. C. Huang, O. Lawlor, L. V. Kaleé, Adaptive MPI, in: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, 2003.
21. O. Sievert, H. Casanova, A simple MPI process swapping architecture for iterative applications, International Journal of High Performance Computing Applications 18 (3) (2004) 341–352.
22. G. Stellner, Cocheck: Checkpointing and process migration for MPI, in: Proceedings of the 10th International Parallel Processing Symposium, IEEE Computer Society, 1996, pp. 526–531.
23. A. Agbaria, 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, IEEE Computer Society, 1999, p. 31.
24. S. S. Vadhiyar, J. J. Dongarra, SRS - a framework for developing malleable and migratable parallel applications for distributed systems, in: Parallel Processing Letters, Vol. 13, 2003, pp. 291–312.
25. N. T. Karonis, B. Toonen, I. Foster, MPICH-G2: a grid-enabled implementation of the Message Passing Interface, J. Parallel Distrib. Comput. 63 (5) (2003) 551–563.

26. R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, 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, IEEE Computer Society, 2001, p. 26.
27. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, A. Selikhov, 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, pp. 1–18.
28. G. E. Fagg, 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, Springer-Verlag, 2000, pp. 346–353.