Towards an Internet Operating System: Middleware for Adaptive Distributed Computing

Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela

Department of Computer Science
Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590, USA
{elmagk,deselt,szymansk,cvarela}@cs.rpi.edu

Abstract

Large-scale, dynamic, and heterogeneous networks of computational resources (a.k.a. grids) promise to provide high performance and scalability to computationally intensive applications. These environments also introduce the need for complex resource management strategies. We propose decentralized middleware-triggered dynamic reconfiguration strategies to enable application adaptation to the constantly changing behavior of large scale shared computational grids. As a proof of concept, we present a framework for dynamically reconfigurable distributed applications. The Internet Operating System (IOS) is a middleware infrastructure which aims at relieving application developers from dealing with non-functional concerns while seeking to optimize application performance and global resource utilization. IOS consists of distributed middleware agents that are capable of interconnecting themselves in various virtual peer-to-peer topologies. IOS middleware agents: 1) profile application communication patterns, 2) evaluate the dynamics of the underlying physical resources, and 3) reconfigure application components by changing their mappings to physical resources through migration and by changing their granularity through a split and merge mechanism. A key characteristic of IOS is that it adopts a decentralized strategy that avoids the use of any global knowledge to enable scalable reconfiguration. The IOS middleware is programming model-independent: we have implemented an actor programming model interface for SALSA programs and also a process programming model interface for MPI programs. Experimental results show that adaptive middleware can be an effective approach for reconfiguring distributed applications with various ratios of communication to computation in order to improve their performance, and more effectively utilize grid environment resources.

1 Introduction

A wide variety of computational environments are increasingly available to host the execution of distributed and parallel applications. Examples include large scale supercomputers, shared or dedicated clusters, grid environments, and metacomputing environments. Performance variability in such environments is the rule and not the exception. This poses new challenges for application developers that go far beyond those of parallelism and scalability. The issue here is not only how to optimize large applications to run on a given set of distributed resources, but also how to maintain the desired performance when the pool of resources might change anytime, the characteristics of the resources are variable, and the application demands change during its lifetime. In conventional distributed and parallel systems, achieving
high performance was a matter of application-level scheduling or application-specific tunings. Such tech-
niques relied on the following assumptions: 1) having a good knowledge of the application’s performance
model, 2) having a static number of resources, and 3) having a knowledge of the characteristics of these
resources.

Previous approaches have used application-level, or user defined reconfiguration mechanisms. Exam-
ples include adaptive parallel applications [45, 22, 27, 26, 48]. Other approaches have relied on new
programming models and abstractions that promote composability, migration, and decentralized coordi-
nation [58, 10, 30, 7, 53]. This work advocates a separation of concerns between the applications and the
middleware. The middleware is responsible for addressing issues such as when and how to reconfigure
applications. While applications need to be amenable to reconfiguration by providing tools that allow
migration which implies moving computational units across physical resources, and/or malleability which
implies splitting and merging computational units to change the application’s granularity.

We present a modular framework that supports application adaptation to both changing computational
needs of applications and changing conditions of the execution environment. Reconfiguration is supported
at the entity-level of applications to allow for flexible adaptation and a large range of distributed appli-
cations. The adopted approach tries to minimize the use of a detailed application performance model for
the purpose of performance predictions. Resource-level profiling is used to learn on the fly the behavior
of the applications and to predict based on the profiled information how the application is expected to
perform with new joining resources or leaving resources. Another key issue in large-scale network systems
is the difficulty of having global and accurate knowledge about the behavior of the available resources.
Designing reconfiguration policies that rely on partial and inaccurate knowledge becomes inevitable to
achieve fast and scalable decisions.

The framework described herein consists of a middleware layer that is used to analyze both the under-
lying physical network resources and the application communication patterns to decide how applications
should be reconfigured to maintain good performance. Resource profiling and reconfiguration decisions
are embodied into middleware agents. The middleware agents form a virtual network. Every agent mon-
itors one or more application entities and coordinates with peer agents to learn about available resource
characteristics and application communication patterns. Two virtual topologies for agents’ coordination
are have been investigated: a peer-to-peer and a cluster-to-cluster virtual topology.

The remainder of the paper is organized as follows. In Section 2, we discuss key design concepts
of building middleware that is capable of reconfiguring and hence adapting applications to dynamic
environments. Section 3 outlines some of the key issues that arise at the middleware level and the
application level. Sections 4 present IOS middleware as a prototype that has been designed and developed
with these design concepts and issues in mind. Performance evaluation is discussed in Section 5. A survey
of related work is given in Section 6. The paper concludes with a discussion and future work in Section 7.

2 Middleware-driven Application Reconfiguration

Computational grids are rapidly emerging as the execution environments of wide area distributed
computing. These environments are inherently heterogeneous, shared, and dynamic. The efficient de-
ployment of applications on such environments demands complex resource management. The sheer size
and dynamic behavior of grid environments introduces the need of dynamic reconfiguration mechanisms.
Dynamic reconfiguration implies the ability to modify the application’s granularity and/or modify the
mapping between physical resources and application’s entities while the application continues to operate
without any disruption of service. Applications should be able to scale up to exploit more resources
as they become available or shrink down as some resources leave or experience failures. Dealing with
reconfiguration issues are beyond the scope of application developers. It is therefore imperative to adopt
a middleware-driven approach to achieve efficient deployment of distributed applications in a dynamic setting in a portable and transparent manner. One way of achieving this vision is embodying various middleware strategies and services within autonomous agents that co-exist with applications.

A middleware geared toward application’s reconfigurability should address the following issues:

- **Architectural Modularity**: Middleware agents should provide mechanisms that allow profiling meta-information about application’s behavior and resource characteristics, propagating this meta-information, and deciding when and how to reconfigure applications. It is hard if not impossible to adopt one single strategy that is applicable to different types of execution environments and a large spectrum of applications. For example, it should be possible for the middleware to be able to tune the amount of profiling done depending on the load of the underlying resources. The more accurate the profiling is, the more informed the reconfiguration decisions are. A modular architecture is therefore critical to have extensible and pluggable reconfiguration mechanisms. Several profiling, coordination, and reconfiguration decisions strategies can be used depending on the class of applications and the type of the underlying execution environment.

- **Generic Interfaces**: To allow a middleware to handle various programming models and technologies, common and generic interfaces need to be designed. The functional details of how to accomplish reconfigurability such as migration, replication, or repartition of computation are different among various programming models. However, the strategies embodied in the middleware to achieve application’s adaptation such as deciding how to disseminate resource characteristics and how to use meta-level information about applications and resources to effectively reconfigure applications should be similar and reusable across various programming paradigms. Generalizing and standardizing the interactions between applications and middleware leads to a more generic and portable approach. An obvious advantage of this approach is the ability to use new programming models and to maintain legacy code with existing models.

- **Decentralized and scalable strategies**: Future grids will consist of thousands of globally distributed computational nodes. To address the demands of such large-scale environments with higher degrees of node failure and global latencies, decentralized management becomes a necessity. The sheer size of these computational environments renders using decisions based on global knowledge infeasible and unpractical since the overhead of obtaining global information may overwhelm the benefits of using it. This forces relying on making decisions based on local information. There is an inherent trade-off between the overhead of information dissemination and the maintenance of high quality of information.

- **Management of dynamicity**: Large execution environments are expected to be more dynamic. Resource availability will be constantly changing. Resources might join anytime during the lifetime of an application, for example, when users voluntarily engage their machines as part of a computation, or when an over-loaded machine become suddenly available upon the completion of some tasks. Resources might also leave at anytime, for example, when a user withdraws his/her machine from a computation or because of a failure. For maximum efficiency, middleware-drive reconfiguration should help applications to adjust their resource allocation as the availability of the underlying physical resources change.
3 Virtual Network Topologies and Models of Reconfiguration

3.1 Middleware Issues

Computation grids may consist of different topologies of computational nodes. In some cases nodes may be highly heterogeneous and geographically distributed, resembling nodes on the Internet, while in other cases, certain nodes may be tightly clustered, and these clusters may be geographically distributed with high bandwidth connections, resembling hierarchical computational grids. Therefore the type of decentralized coordination will play an important role in the efficiency of the middleware. Depending on which scenarios, a purely peer-to-peer coordination might be more effective than a hierarchical coordination. In other scenarios a hybrid approach will be more effective.

3.2 Application Issues

Functional issues of reconfiguration such as how to migrate application entities or how to change the granularity of entities are highly dependent on the programming model or the programming language used by the given application. It is imperative to have tools that augment the application with these capabilities. When such tools are available, the middleware can readily use them to automate the process of application adaptation in dynamic environments. The middleware can shield application developers from dealing with complex reconfiguration issues such as when and where to migrate application’s entities and which entities need to be reconfigured.

There are several ways to application reconfigurability at the level of application: 1)migration of computation, 2)splitting and merging of computation, and 3)Replication.

As there are several ways to accomplish application’s reconfiguration depending on the nature of applications’ topology, their degrees of parallelism, their communication or computational patterns, the middleware needs to be well informed about such characteristics to come up with the right reconfiguration strategy. This requires dynamically profiling both the behavior of the application and the behavior of the physical resources. Dynamic profiling is crucial to having a generic middleware that can be benefit applications with various behavior, as opposed to requiring knowledge of the semantics of the application prior to reconfiguration.

4 Generic Modular Middleware Architecture

4.1 Generic Interfacing

To motivate time and effort in developing middleware, it should be applicable to as wide an audience as possible. A programming model independent interface along with generically implementable reconfiguration tools will not limit a middleware to any particular application or language, and will justify the time and effort in developing intelligent reconfiguration methods due to the fact that these will be beneficial to a large group of applications.

For middleware-driven reconfiguration, this requires two-way communication. Profiling information must be sent from the application or it’s run-time environment to the middleware. This latter must communicate reconfiguration decisions to the application. Section discusses the use of the generic IOS profiling API in two different programming models (SALSA and MPI), and how the reconfiguration tool migration was implemented.
4.2 Modular Middleware

The tasks of a decentralized middleware for application reconfiguration can be divided into three sections:

- **Profiling**: The middleware must be able to profile the application and its execution environment in order to make informed decisions about reconfiguration.

- **Communication**: Distributed middleware agents will need to communicate profiled information and other decision making and management information to enable intelligent reconfiguration.

- **Decision Making**: Given local and remote profiling information, middleware agents must be able to decide what the needs of the application are and reconfigure it accordingly.

By dividing the needs of the middleware into these three areas, it becomes possible to create a modular middleware which can pick and choose profiling, communication and decision making strategies. This becomes especially important as grids support a wider range of architectures and computational nodes, and run different types of applications. It is an open research question as how to appropriately reconfigure applications on the grid, so having a middleware which can pick and choose strategies for reconfiguration becomes highly desirable. Likewise, the ability to pick and choose the types of profiling are highly important for efficiency concerns.

Section 4.3 describes the modular IOS architecture, which separates these concerns into different pluggable components and Section 4.5 discusses various implementations of these components which have been used.

4.3 The IOS Architecture

The IOS architecture consists of distributed middleware agents that are capable of interconnecting themselves in various communication topologies. Figure 1 shows the architecture of an IOS agent and how it interacts with the application entities that are hosted in a given node. Every IOS agent consists of three pluggable components that allow for testing of different types of autonomous reconfiguration, and also allow users to develop their own fine-tuned components for particular applications. These are the profiling component, decision component, and protocol component:
The following methods notify the profiling agent of entities entering and exiting the local run-time system due to migration or initial entity startup.

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

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

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

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

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

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

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

### Figure 2. IOS Profiling API

- **Dynamic Profiling Component:** Resource-level and application-level profiling is used to gather dynamic performance profiles about physical resources and application entities (such as processes, agents, active objects, or actors). Application entities communicate processing, communication, data access and memory usage profiles to the middleware via a profiling API, shown in Figure 2. Profiling monitors periodically measure and collect information about resources such as available CPU power, memory, and disk storage. The IOS architecture defines well-defined interfaces for profiling monitors which allow the use of various profiling technologies as long as they implement the appropriate interfaces. Examples include the Network Weather Service (NWS) [61] and MDS [20].

- **Protocol Component:** The protocol component is responsible for inter-agent communication. The protocol components arrange the middleware agents into a virtual network, using various communication strategies such as peer-to-peer or hierarchical communication. The agents can connect themselves into various virtual agent topologies.

- **Decision Component:** The protocol components are used to distribute profiled information among the different middleware agents. Using local and remote profiling information, the decision component decides how reconfiguration should be accomplished. Different reconfiguration strategies such as random stealing (RS), application topology sensitive random stealing (ARS) and network topology sensitive random stealing (NRS) have been implemented. For more details, the reader is referred to [23].

4.4 **Generic Interface Implementations**

To further develop the generic IOS profiling API, two different programming models, SALSA and MPI have implemented this API to enable middleware triggered reconfiguration. The following sections discuss the implementation details for both programming models. Section 5.4 show that both implementations enable application behavior profiling with minimal overhead.
4.4.1 SALSA/IOS Implementation

SALSA [59] is a language for developing actor-oriented applications. SALSA is the acronym for Simple Actor Language System and Architecture. SALSA is pre-processed to Java. Therefore it inherits all the object-oriented features of Java such as inheritance, encapsulation, and polymorphism. SALSA provides programming abstractions to implement actor primitives notably creation and asynchronous communication. It has been designed with additional extensions to facilitate programming reconfigurable distributed applications: universal naming, migration support, and coordination primitives.

Actors are inherently concurrent and distributed objects that communicate with each other via asynchronous message passing. An actor is an object because it encapsulates a state and a set of methods. It is autonomous because it is controlled by a single thread of execution. When an actor receives a message, one of three primitive operations might happen: 1) changing its internal state through invoking one of its internal methods, 2) sending a message to another actor, or 3) creating a new actor.

The anatomy of actors simplifies concurrency, synchronization, coordination, namespace management, memory management, scheduling, and mobility in distributed systems. The actor model has been successfully used in several applications. To name a few: enterprise integration [54], real-time programming [46], fault-tolerance [1], and distributed artificial intelligence [24].

SALSA and the actor model are extremely good candidates for use in developing autonomously reconfigurable applications, as the profiling and migration required are done transparently to the application developer. In order to have a SALSA application use IOS, the application developer need only have their actors extend an AutonomousActor class.

Migrating SALSA Actors  Actors in SALSA can be migrated transparently to the application developer, as references to other actors are location independent, and the language runtime determines if communication is remote or local. The anatomy of an actor simplifies migration since every actor encapsulates cleanly its state and since it is based on asynchronous message passing. Message forwarding mechanisms are used to ensure that no messages in transit are lost while an actor is migrating to a new location.

Profiling SALSA Actors  To enable profiling of SALSA actors, an actor can extend the AutonomousActor class. AutonomousActors automatically profile communication, memory and CPU usage. This information is directly communicated to the IOS profiling component through the IOS profiling API (see Figure 2).

4.4.2 MPI/IOS Implementation

MPI [41] is a widely used standard to develop parallel applications that harness several processors. However, the issues of scalability, adaptability and load balancing still remain a challenge. To maintain a good performance level, MPI applications need to be able to scale up to accommodate new resources or shrink to accommodate leaving or slow resources. Most existing MPI implementations assume a static network environment. MPI implementations that support the MPI-2 Standard [32, 42] provide some support for dynamic process management by allowing running processes to spawn new processes and communicate with them. However, developers still need to handle explicitly issues such as resource discovery, resource allocation, scheduling, profiling, and load balancing. Additional middleware support is therefore needed to relieve application developers from non-functional concerns while allowing high performance. In what follows we describe mechanisms adopted to achieve migration of MPI processes at the user-level and how iterative MPI applications have been integrated with IOS.
Migrating MPI Processes  In MPI, any communication between processes needs to be done as part of a communicator. An MPI communicator is an opaque object with a number of attributes, together with simple functions that govern its creation, use and destruction. An intracommunicator delineates a communication domain which can be used for point-to-point communications as well as collective communication among the members of the domain. On the other hand, an intercommunicator allows communication between processes belonging to disjoint intracommunicators.

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

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

Profiling MPI Applications  To accomplish application behavior profiling of MPI processes, their communication patterns must be periodically sent to their corresponding IOS profiling agents. This is achieved through a profiling library based on the MPI profiling interface (PMLI). The MPI specification provides a general mechanism for intercepting calls to MPI functions. This allows the development of portable performance analyzers and other tools without access to the MPI implementation source code. The only requirement is that every MPI function be callable by an alternate name (PMLI_{Xxxx} instead of the usual MPI_{Xxxx}). This profiling library intercepts all communication methods of MPI using a local PCM Daemon (PCMD) that sends periodically sends communication summaries to the local IOS profiling component. The PCMD is required to accomplish the communication between MPI (which is written in C) and IOS (which is written in Java). For more details about the PCMD, the reader is referred to [39].
4.5 IOS Module Implementations

The modularity of IOS allows for different implementations of the protocol, decision and profiling components. The following sections discuss a selection of component implementations. Section 5 discusses the evaluation of these components.

4.5.1 Virtual Network Implementations

As discussed in Section 4.3, the IOS middleware consists of distributed middleware agents, each with three pluggable components. Different protocol components have been implemented to organize the middleware agents into different decentralized virtual networks, which can adjust themselves according to the network topology of the execution environment. Two types of representative virtual networks are presented: peer-to-peer (p2p) and cluster-to-cluster (c2c). The p2p virtual network provides strictly peer-to-peer communication between the middleware agents, represented in Figure 4. The c2c virtual network provides a hierarchical arrangement of agents, by electing middleware agents to act as manager for groups of agents. This virtual network topology is represented in Figure 5. Both virtual networks are sensitive to the fluctuations of the network, and will dynamically change themselves in response to changes in the network.

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 can operate in conjunction and are only used for discovering peers in a virtual network and are not a centralized point of failure. They operate similarly to gnutella-hosts in Gnutella peer-to-peer networks [19]. 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. Each agent will keep a list of peers at various distances (determined by latency), and
will request reconfiguration with low latency peers before high latency peers. The arrangement of peers and information propagation strategies are discussed in greater detail in [23].

A Network Sensitive Cluster-to-Cluster Topology (NSc2c) In NSc2c, agents are organized into groups of virtual clusters (VCs) which have collectively low latency communication times. 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 in the same manner as the NSp2p virtual network topology.

4.5.2 Reconfiguration Decision Strategies

Current implementations of decision strategies involve a decision function which can be applied to profiled information about the application at two separate nodes, resulting in a listing of the benefit of migration for all actors or processes between those nodes. For hierarchical and clustered virtual networks, this decision function can applied between multiple nodes and the results compared to determine the most beneficial migrations that should be done. Two decision functions that have been implemented are random stealing (RS) and application-sensitive random stealing (ARS), for further details on these and other decision making strategies the reader is referred to [23, 40].

Random Work Stealing (RS) Random work stealing used by IOS is based on a strategy discussed in [11]. Nodes with light computation and communication loads will attempt to steal work from heavily loaded nodes. This reconfiguration strategy is quite simple and has low overhead as only the load of communication and computation at the nodes must be profiled. It also does not require information about the application other than what actors or processes are migratable. It also is stable, in that a heavily loaded system will not attempt to perform reconfiguration, resulting in further overhead.

Application-sensitive Random Stealing (ARS) Application-sensitive random stealing builds upon RS in that it takes into account the behavior of the application. Work is request as in RS, however instead of migrating any actor or process from a heavily loaded node to a lightly loaded node, it attempts to only perform migrations which will colocate synchronized actors and processes. Results in section 5 show that for many applications ARS provides significant benefit over RS.

5 Profiling and Reconfiguration Performance Results

This section discusses some key experimental results pertaining to the IOS middleware. We show the effect of using knowledge about the application’s topology in the reconfiguration decisions and the effect of certain virtual topologies on the quality of reconfiguration. We also evaluate the effect of reconfiguration using both SALSA and MPI.

5.1 Behavior-Based Reconfiguration Results

It has been shown that using application behavior information to accomplish autonomous reconfiguration can be highly effective. Additionally, in both programming models used, the overhead of IOS is very low and in some cases negligible.
Figure 6. Performance of the massively parallel unconnected benchmark using different reconfiguration strategies.

Figure 7. Performance of the massively parallel sparse benchmark using different reconfiguration strategies.

Figure 8. Performance of the highly synchronized tree benchmark using different reconfiguration strategies.

Figure 9. Performance of the highly synchronized hypercube benchmark using different reconfiguration strategies.

5.1.1 Autonomous Reconfiguration using SALSA

Benchmarks were developed using SALSA that model various communication topologies with an emphasis on the computation-to-communication ratio of the applications. Four different application topologies are represented, each pertaining to different levels of inter-actor synchronization, represented by different connectivity:

- **The unconnected topology.** A representative for massively parallel applications where actors continuously perform computations without exchanging any messages.

- **The sparse topology.** A model of applications that have a moderate level of synchronization but have a higher communication to computation ratio than massively parallel applications.

- **The tree topology.** Actors are inter-connected in a tree structure to model a higher degree of synchronization, which results in a high communication to computation ratio.

- **the hypercube topology.** This benchmark provides the highest amount of synchronization, modeling a very high communication to computation ratio.
Figures 6, 7, 8, and 9 show that while application behavior information is not necessary in massively parallel applications, reconfiguring applications based on ARS, a strategy which takes application behavior into consideration (such as the application’s communication topology), results in largely improved performance on applications with varying levels of synchronization, over strategies that do not, such as round robin and RS.

### 5.1.2 Autonomous Reconfiguration using MPI

An implementation of the 2-dimensional heat diffusion application in MPI was used to evaluate the reconfiguration capabilities of the MPI using IOS. The original MPI code was instrumented with PCM calls to enable checkpointing and migration. For the experimental testbed we used a 4-dual node cluster of SUN Blade 1000 machines. Each node has a processing speed of 750M cycles per second and 2 GB of memory. For comparative purposes, we used MPICH2 [9], a free implementation of the MPI-2 standard. We emulated a shared and dynamic environment with varying load conditions by introducing artificial load in some of the cluster nodes and varying it periodically.

We conducted two experiments using the heat application using MPI/IOS and MPICH2 under similar load conditions. For both experiments, after running the application, artificial increases to the load on different cluster nodes was applied.

The first experiment was conducted with MPI/IOS. Figure 10 shows the performance of the application. The application was initialized with 8 processes on 4 processors. The remaining 4 processors joined the virtual IOS network gradually. Artificial load was gradually applied to one of the cluster nodes (which consisted two processors) participating in the computation. One node joined the virtual IOS network around iteration 1000, causing IOS to migrate one process to the new machine and to reduce the load on its original hosting node, increasing the application throughput. More artificial load was applied to the same node around iteration 2500. Around iteration 3000, a fourth node joined the virtual network. At this point, IOS migrated two processes to this machine, eliminating the loaded node from the computation. The application ended up using a total of the 6 best available processors, causing a substantial increase in its performance. The total execution time of the application was 645.67s.

![Figure 10. Performance of the two-dimensional heat simulation application using the reconfiguration mechanism of MPI/IOS. The experiment was conducted on a 4 dual-processor SUN blade 1000 cluster.](image1)

![Figure 11. Performance of the two-dimensional heat simulation application using MPICH2. The experiment was conducted on a 4 dual-processor SUN blade 1000 cluster.](image2)
Figure 12. The tree benchmark on a dynamic environment using an application behavior sensitive reconfiguration strategy (ARS) and a non-behavior sensitive strategy (RS). A processor was added every 30 seconds, then the application was allowed to stabilize, and then a processor was removed every 30 seconds.

5.2 Throughput and Scalability Improvements

Migration, as described in Sections 4.4.1 and 4.4.2, has been used to allow applications utilize dynamic environments, by spreading the application to new nodes as they become available, and retracting the application from nodes that become unavailable. Figures 13 and 12 shows two different applications, one tightly coupled and the other massively parallel, executing on a dynamic environment using two different reconfiguration strategies. In both tests, the applications were run on a single node, and every 30 seconds an additional node was made available. After allowing the application to execute on 8 nodes for three minutes, a node was made unavailable every thirty seconds until only one node was remaining. These results show that the middleware can effectively reconfigure the application to use the available resources as they change over time. While the massively parallel application was able to utilize the available resources using both an application topology sensitive reconfiguration strategy (ARS) and a a strategy based solely on resource availability (RS), the tightly coupled application only showed improvement using ARS.

5.3 Virtual Network Evaluation

We have evaluated different arrangements of middleware agents using NSp2p and NSc2c distribute profiled information for use in different reconfiguration strategies (see [23] for a detailed descriptions of these reconfiguration strategies). Benchmarks that represent various degrees of computation to communication ratios (see Section 5.1.1) were used as a means to evaluate these virtual networks.

Two different physical environments were used to model Internet-like networks and Grid-like networks. The first physical network consisted 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 consisted 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 14 and 15 show not only that decentralized middleware management can accomplish intelligent application reconfiguration, but also that the virtual network used plays a role in the effectiveness of the reconfiguration. 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 [40].

### 5.4 Overhead Evaluation

The overhead of using IOS with SALSA was evaluated using two applications, a massively parallel Astronomy application and a tightly coupled 2-dimensional heat diffusion application, similar to the hypercube benchmark. Figures 16 and 17 show the time to accomplish an iteration in the application using SALSA and SALSA with IOS. In both cases the overhead was minimal, around 0.5% for the astronomy application and 2% for the 2-dimensional heat diffusion application.

Figure 11 shows the performance of the application using MPICH2. The same load conditions as the first experiment were applied. With no ability to adapt, the application used the initial configuration throughout the course of the experiment and experienced a constant degradation in its performance. This degradation was significant as the highly synchronized nature of this application causes it to run
Figure 16. Overhead of using SALSA/IOS on a massively parallel astronomy data modeling application with different parallelization.

Figure 17. Overhead of using SALSA/IOS on a tightly synchronized 2-dimensional heat diffusion application with different parallelization.

Figure 18. Execution of the 2-d heat diffusion application using different levels of parallelism, with and without the PCM library.

Figure 19. Overhead of using PCM with the 2-d heat diffusion application.

as fast as the slowest processor. The application took 1173.79s to finish, about an 81.8% decrease in performance compared to the adaptive execution.

The heat application was executed with different numbers of nodes, using both MPI/IOS and MPICH2. Figure 5.4 shows that the overhead of using the PCM library for application behavior profiling is less than 5% for several sizes of the cluster.

6 Related Work

Several research groups are trying to achieve distributed computing on a large scale. Wisconsin’s Condor project studies high throughput computing by “hunting” for idle workstation cycles [38]. Berkeley’s NOW project effectively distributes computation on a building-wide scale [6], and Berkeley’s Millennium project exploits a hierarchical cluster structure to provide distributed computing on a campus-wide scale [15].

The Globus project seeks to enable the construction of larger computational grids [28]. Virginia’s Legion meta-system integrates research in object-oriented parallel processing, distributed computing,
and security to form a programmable world-wide virtual computer [31]. Caltech’s Infospheres project envisions a world-wide pool of millions of objects (or agents) much like the pool of documents on the World-Wide Web today [16]. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage, to wide area applications [57]. UIUC’s 2K is an integrated operating system architecture addressing the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of component-based distributed applications [35].

Research on process and data migration for grid applications includes [56, 55, 49]. A key idea behind these approaches is to detect service degradations and react to such events by dynamically reconfiguring application processes to effectively use available computational resources. Research on data and process replication includes [18, 17, 51, 47, 44, 60, 3, 43]. Most of these approaches only consider immutable data replication to avoid having to devise and implement expensive replica consistency protocols. An adaptive middleware layer is needed, capable of migrating and replicating data and processes proactively based on the dynamically changing availability of resources on the grid [52, 23, 2, 36, 37].

While adaptive process and data migration and replication can have a large impact on the performance of grid computing applications, they both assume a reasonable underlying model of resource usage and expected future performance and availability of grid resources. Two mechanisms to predict performance based on profiling resource usage are the Network Weather Service (NWS) [62] and the Globus Meta Discovery Service (MDS) [21]. Recent research has devised and evaluated different mechanisms for resource management in dynamic heterogeneous grid environments—e.g., see [63, 29, 8, 33, 21, 34, 4, 5].

An important consideration when adapting applications to dynamic grid environments through proactive data and process migration and replication is that the failure semantics of applications changes considerably. Research on fault detection and recovery through checkpointing and replication includes [25, 50, 14, 13, 12]. Notice that an application checkpointing mechanism is necessary for adaptive application migration and can readily be used for fault tolerance as well. More fine-grained process-level rather than application-level checkpointing and migration requires logging messages in transit to properly restore a distributed application state upon failure [25].

Message passing models such as MPI and PVM are good for the programmer to reason about locality (and thus performance) of parallel programs. On the other hand, directly adding malleability to this model adds significant programming complexity because all the burden of keeping track of memberships and the mapping of data to nodes is on the programmer. Phoenix parallel programming model [53] improves this situation by introducing communication via abstract (virtual) node names.

There is a large body of research into computational grids and grid-based middleware, this section only attempts to discuss a selection of this work.

7 Discussion and Future Work

This work argues that middleware can provide effective application reconfiguration using different adaptation tools, based on profiling of application behavior and the execution environment, thus making it generically applicable to different applications and programming models. Additionally, such types of middleware and reconfiguration tools can enable applications to efficiently utilize dynamically changing environments, which could lead to more efficient utilization of grid resources. Finally, for future grid environments, decentralized management is possible and worthy of further research.

Acknowledgment

The authors would like to acknowledge the members of the Worldwide Computing Laboratory at Rensselaer Polytechnic Institute. This work has been supported by the following grants: IBM SUR
Award 2003, IBM SUR Award 2004, NSF CAREER Award No. CNS-0448407, and NSF-INT 0334667.

References


