AUTONOMIC GRID COMPUTING USING MALLEABILITY AND MIGRATION: AN ACTOR-ORIENTED SOFTWARE FRAMEWORK

By

Travis Desell

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Approved:

Carlos A. Varela
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2007
(For Graduation May 2007)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As distributed computing environments increase in size by adding more participants, their computing architectures become more heterogeneous and dynamically available, while the chance of hardware failure increases. In order to efficiently develop applications for these environments, new programming methods and services must be made accessible which enable dynamic reconfiguration and fault tolerance.

As part of an effort to provide these services and methods, this thesis examines two reconfiguration methods that allow applications to dynamically respond to changes in their environment. Migration allows for the computational components of an application to move between processors, while malleability enables a more general reshaping of an application by adding or removing computational components and redistributing data. Migration and malleability have been implemented in the SALSA programming language. Migration is provided transparently without any effort required from a developer, while malleability is accomplished through a simple API. The Internet Operating System (IOS), a modular middleware, is used to autonomously reconfigure applications which implement these reconfiguration methods.

Three different autonomous reconfiguration strategies using migration were implemented in IOS, each with different levels of profiling which can provide more informed reconfiguration. A minimum amount of profiling and a simple reconfiguration strategy is shown to be best suited for simple massively parallel applications, while more complex profiling provides significant advantages for applications with different communication topologies. Additionally, a strategy using fuzzy logic is shown to be able to optimally reconfigure applications with a more complex communication topology, without any centralized knowledge. Malleability is also examined using a hierarchical reconfiguration strategy. Malleability is shown to be an order of magnitude faster than an already existing reconfiguration strategy, application stop-restart, and for iterative applications it can provide improved performance over migration alone.

The results from this thesis provide strong evidence that applications can be dynamically reconfigured efficiently for significant improvement in performance on dynamic environments. Additionally that it may be possible for certain types of applications to be optimally reconfigured using strategies that do not require centralized knowledge. This work is part of an effort to enable applications to run on an upcoming campus-wide RPI grid.

CHAPTER 1
Introduction

The increasing size and complexity of large scale execution environments and distributed applications is driving a need for middleware which can alleviate the challenges of developing such systems. These systems offer many distributed resources, such as computational power, communication links, memory and storage. As they increase in size, they are becoming increasingly dynamic, in terms of processor availablility, and heterogeneous, in terms of communication and architecture, while at the same time demanding better performance.

This thesis presents a distributed middleware, IOS – the Internet Operating System, for large scale computing, which enables autonomous reconfiguration of distributed applications. The middleware uses a modular architecture with plug-in services that can profile applications and their environment to make informed decisions about how to perform reconfiguration. The SALSA programming language has been extended to transparently use the IOS middleware to perform autonomous reconfiguration via migration of computational components, and with an API for autonomous reconfiguration via malleability to dynamically change an application's data distribution and computational component granularity. This provides a significant reduction in the amount of work required by an application developer by removing the need to develop approaches to deal with the dynamic nature of large scale environments and dynamic performance tuning.

This chapter first discusses some current large scale computing environments in Section 1.1, then uses these as examples to highlight current large scale computing challenges in Section 1.2. The methods used for autonomous reconfiguration in response to some of these challenges, migration and malleability, are presented in Section 1.3. The contributions of this thesis are summarized in Section 1.4. The chapter concludes with an overview of the thesis in Section 1.5.

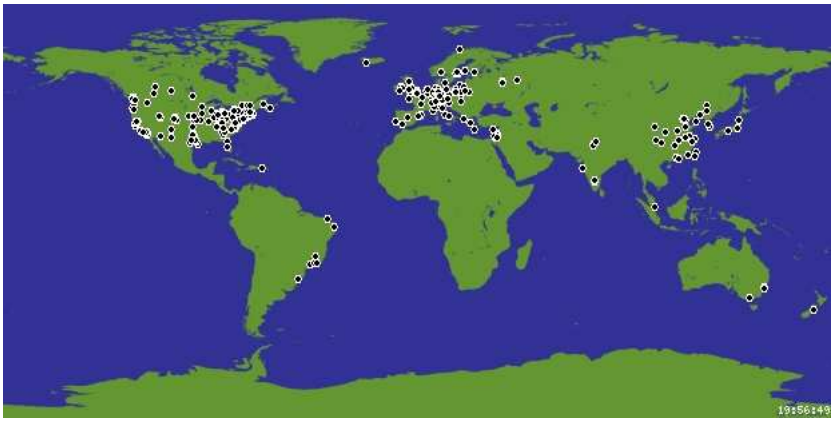

Figure 1.1: Geographic distribution of PlanetLab participants.

1.1 Large-Scale Computing Environments

Large scale computing environments are becoming increasingly popular. There are open collaborative environments in which participants can share computational resources for research into services and middleware for these environments, such as PlanetLab [14]. Others additionally allow home computing resources for massively parallel distribution of applications, as in BOINC [8]. Many scientific grids [1, 2], are globally distributed with different research institutions participating with large clusters. Even institution wide computing systems are reaching increasingly larger scales, such as the Rensselaer Grid, which is in development. The remainder of this section describes these large scale computing environments in detail to present the complexity and scale of these systems.

PlanetLab is a large scale computing infrastructure for developing distributed technologies on a world-wide scale (see Figure 1.1[1]). Participants install a distributed operating system on machines which participate in PlanetLab and run various distributed services being tested on PlanetLab. By volunteering machines to participate in PlanetLab, users gain access to developing their own services which they can test on the PlanetLab network. As of January 2007, PlanetLab consists of 735 machines hosted at over 360 sites in 25 different countries.

BOINC is an open-source infrastructure for large scale volunteer computing.

[1]Image from http://www.planet-lab.org/

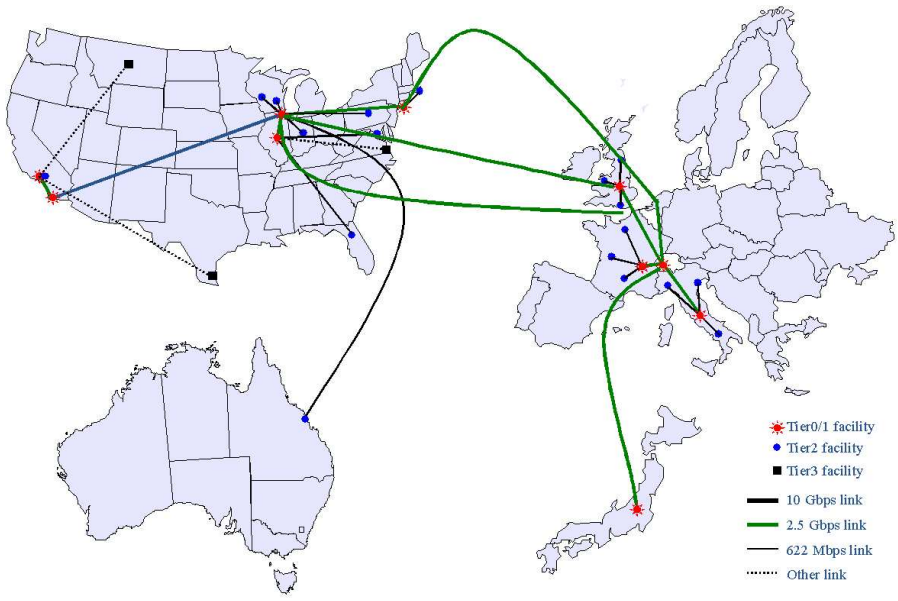

Figure 1.2: Geographic distribution of iVDGL participants.

Applications such as SETI@Home [7] and Folding@Home [35] are implemented using the BOINC APIs and the BOINC computing infrastructure allows users to volunteer their personal computers, as well as researchers to volunteer clusters of processors to participate in massively parallel computation. BOINC provides services for fault tolerance and distributed clients which allow users to specifiy usage rules. Applications using BOINC can scale into the millions of processors, which have highly hetereogeneous architectures, computational ability, available memory and communication speeds.

The International Virtual Data Grid Laboratory (iVDGL), is used for developing and executing Petabye-scale virtual data applications in disciplines such as high-energy and nuclear physics, gravitational wave research, astronomy, astrophysics, earth observations, and bioinformatics. Participating sites are globally distributed, in Australia, Europe, Japan and the United States 1.2, and connected by high bandwidth communication links (see Figure 1.2[2]).

The LHC Computing Project (LCG) is being developed as a data storage and analaysis infrastructure for the Large Hadron Collider (LHC) being built at

[2]Image from http://www.ivdgl.org/projinfo/index.html

Figure 1.3: Geographic distribution of LCG@CERN participants.

Cern. It currently consists of over 32,000 processors at 177 sites in Europe and the United States, with over 13,000 Petabytes of storage (see Figure 1.3[3]). It operates hierarchically, with the most powerful sites being in the first tier, which are first to recieve new data, which then gets propagated down to the lowest tiers.

This work is in part motivated to provide services to enable use of the campus wide grid in development at Rensselaer Polytechnic Institute. On campus collaboration includes over 1,000 processors in clusters from the Computer Science, Nanotechnology, Bioscience departments, as well as other multipurpose clusters. In addition, an in development off campus super computing facility with 32,000 IBM Blue Gene processors will add a highly significant contribution.

1.2 Large-Scale Computing Challenges

Large scale environments, like those discussed, introduce many challenges to developing efficient distributed computing systems. Challenges mainly fall into three

[3]Image from http://goc02.grid-support.ac.uk/googlemaps/sam.html

different categories: scalability, heterogeneity and uncertainty. Due to the large size of such environments, applications and middleware must be able to scale to hundreds or even thousands of processors, which may be geographically distributed. This need for scalability requires research into new distributed methods of computation, such as peer-to-peer or hierarchical strategies. Heterogeneity becomes an issue because not only can participating processors have different architectures, which can require different executable binary files, network connections between processors can have drastically different latencies and bandwidth.

Uncertainty factors into many areas of large scale computing. These systems often use scheduling mechanisms which schedule the actual execution of an application for some point in the future, in this case the execution environment may be unknown at load time, which may prevent specialized tuning for the applications target execution environment. In many cases, architecture specific tuning can provide significant performance improvements. The resource utilization of application components can also change over the course of its execution. Additionally, during the course of an applications execution, the resources available to the application may change or become unavailable, due to competition with other applications or system users removing certain resources. In these cases, the application may need to be stopped or reconfigured, or otherwise suffer failure or large performance degradation. It is also not possible to determine when failures will occur over communication links or at processing nodes.

These factors are compounded when applications are iterative, requiring some or all computational components to synchronize after performing some computation. In these cases, a single computational component performing poorly can severely degrade the performance of the entire application. Additionally, a single failure of a computational component can cause the failure of the entire application.

In light of these challenges, applications for these large scale systems must be scalable, fault tolerant, and dynamically reconfigurable. Reconfiguration strategies and middleware must be scalable and fault tolerant as well, while performing efficiently with low overhead. Advances in these areas will allow future large scale systems to be easily programmable while still allowing applications to harness the

full potential that large scale computing environments provide.

1.3 Migration and Malleability

Previous approaches for dynamic reconfiguration have involved *fine grained migration*, which moves application entities such as actors, agents or processes to utilize unused cycles [29, 31] or *coarse grained migration* which uses checkpointing to restart applications with a different set of resources to utilize newly available resources or to stop using badly performing resources [11, 44]. Coarse grained migration can prove highly expensive when resources change availability frequently, as in the case of web computing [7, 35] or shared clusters with multiple users. Additionally, coarse grained migration can prove inefficient for grid and cluster environments, because data typically has to be saved at a single point, then redistributed from this single point, which can cause a performance bottleneck. Fine grained migration enables different entities within applications to be reconfigured concurrently in response to less drastic changes in their environment. Additionally, concurrent fine-grained reconfiguration is less intrusive, as the rest of the application can continue to execute while individual entities are reconfigured. However various checkpointing and consistency methods are required to allow such reconfiguration.

Approaches using fine-grained migration allow reconfiguration by moving around entities of fixed size and data. However, such fine grained reconfiguration is limited by the granularity of the applications entities, and cannot provide reconfiguration in regards to heterogeneity of memory hierarchies and data distribution. Different load balancing approaches [13, 28] allow for dynamic redistribution of data, however they cannot change task granularity or do not allow inter-task communication.

Because fine-grained migration strategies only allow for migration of entities of fixed size, data imbalances can occur if the granularity is too coarse. An iterative application is used to illustrate this limitation, a distributed maximum likelihood computation used for astronomical model validation (for more details see Section 2.1.3.1). This application is run on a dynamic cluster consisting of five processors. Figure 1.4 shows the performance of different approaches to dynamic

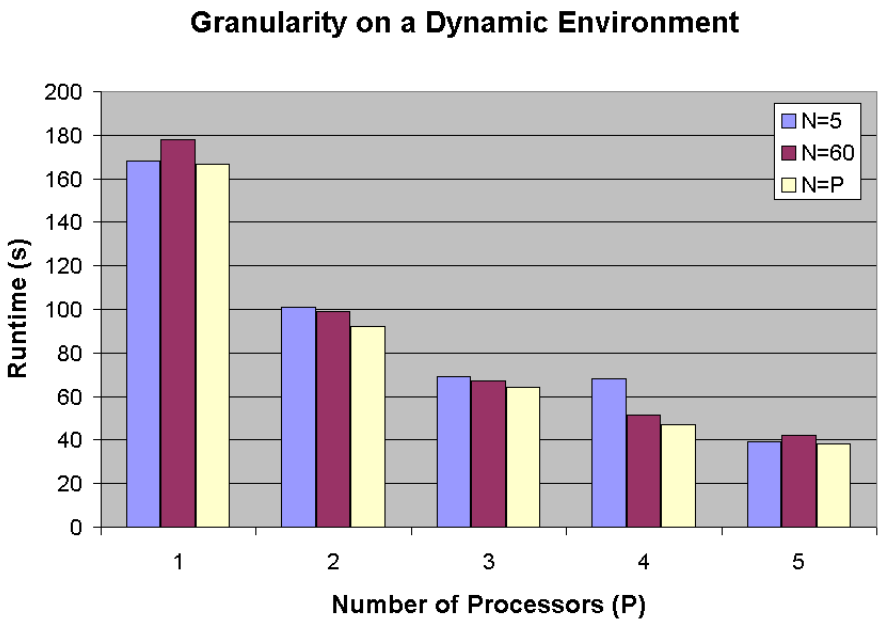

Figure 1.4: Different possibilities for data distribution on a dynamic environment with N entities and P processors. With N = 5, data imbalances occur, degrading performance. N = 60 allows even data distribution, but suffers from increased context switching. N = 5 and N = 60 can be accomplished using fine-grained migration, while N = P requires malleability.

distribution. Three options are shown, where N is the number of entities and P is the number of processors. Each entity has an equal amount of data. $N = 5$ maximizes granularity, reducing the overhead of context switching. However this approach cannot evenly distribute data over each processor, for example, with 4 processors, one processor must have two entities. $N = 60$ uses a smaller granularity which can evenly distribute data in any configuration of processors, however it suffers from additional overhead from context switching. It is also not scalable as the number of components required for this scheme increases exponentially as processors are added. Additionally, in many cases, the availability of resources is unknown at the applications startup so an effective number of components cannot be statically determined. For optimal performance in this example, N should be equal to the number of processors P, however this cannot be accomplished by migration alone.

Using the correct granularity can provide significant benefits to applications

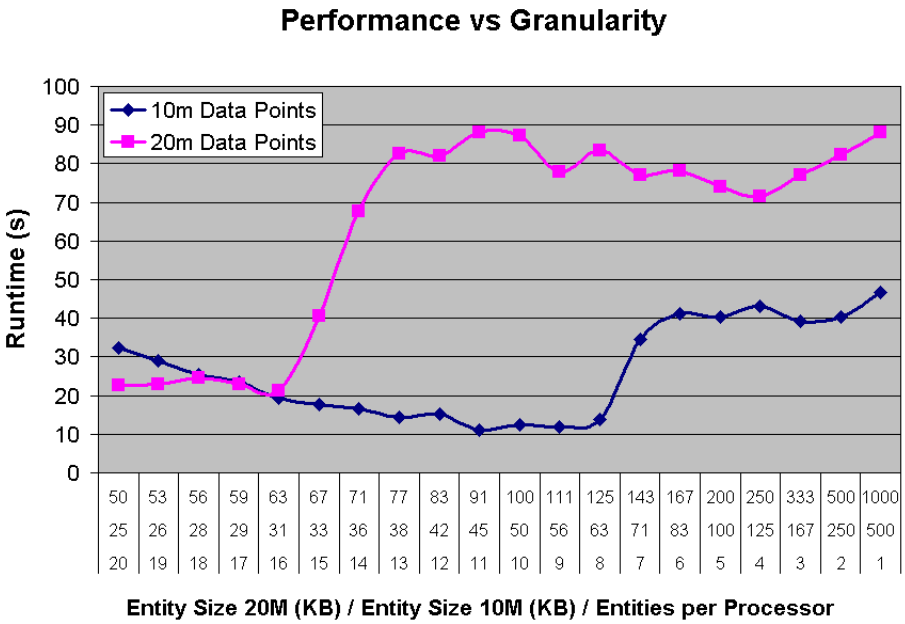

Figure 1.5: A heat diffusion application (see Section 2.1.3.2), run with different data sizes and granularities on 20 processors. Decreasing granularity allows the entities to execute entirely within a 64KB L1 cache, greatly increasing performance. This figure also illustrates how overly decreasing granularity can degrade performance by incurring additional overhead.

by enabling entity computation and data usage to execute in lower levels of memory hierarchy. For example, in Figure 1.5, the runtime for a scientific application modeling heat diffusion over a solid (for a more detailed description see Section 2.1.3.2) is plotted as entity granularity changes. The application was run with 10 million and 20 million data points over a wide range of granularities. All sampled runs performed 50 iterations and the same amount of work per data point. Experiments were run on five quad 2.27GHz Opteron processors with 64KB L1 cache. As the data per entity decreases, computation can be done within L1 cache, instead of within L2 cache, with over a 10x performance increase for 20 million data points and 320 entities, and an over 5x performance increase for 10 million data points and 160 entities, compared to a typical distribution of one or two entities per processor.

In order to effectively utilize these large scale, dynamic and heterogeneous environments, applications need to be able to dynamically change their granularity

and data distribution. Dynamic data redistribution and granularity modification, or *malleability*, presents significant design challenges over fine-grained migration. Distributed applications have a wide range of behaviors, data structures and communication topologies. In order to change the granularity of entities in a distributed application, entities must be added or removed, which requires redistribution of data to new entities, or away from entities being removed. Data redistribution must also be aware of the available resources where data is being moved to or from, and must also be done to keep the load balanced with respect to processing power. Communication topologies must be modified when entities are added or removed, and entity behavior may change. Addressing these issues becomes even more important when applications are iterative, or require all entities to synchronize after finishing some work before the application can continue. With an iterative application, each iteration runs as slow as the slowest entity, so proper data distribution efficient communication topologies are required for good performance. Additionally, language constructs and libraries need to be provided to allow malleability to be accessible to developers, because data redistribution and communication topology modification cannot be entirely automated.

1.4 Contributions

This thesis describes how the Internet Operating System (IOS) [32], was extended to provide autonomous reconfiguration for applications developed in the SALSA programming language [46]. IOS was extended with policies for two different types of reconfiguration, migration and malleability. For many applications, migration is sufficient to enable improved performance on large scale environments. Additionally, profiling, decision making and migration can be done in a completely transparent manner in SALSA, minimizing the work required to use such environments. However, for some applications, such as iterative applications, malleability can provide additional benefit due to reduced context switching overhead and improved load balancing. Malleability, unlike migration, requires the developer to implement a simple API to facilitate the malleability process.

This thesis presents and evaluates the different autonomous reconfiguration

policies implemented in the IOS middleware framework. Three different decentralized strategies for autonomous migration have been studied, each with different levels of transparency and required profiling information: (i), Simple Random Actor Stealing (SRAS), (ii) Application Topology Sensitive Random Actor Stealing (ATS-RAS), and (iii), Fuzzy Logic Controlled Random Actor Stealing (FLC-RAS). Results show that with minimal profiling information, SRAS is sufficient to reconfigure massively parallel applications. At the cost of more profiling overhead, for applications with more complex communication topologies, ATS-RAS outperforms SRAS due to taken into account the communication topology of the application. Lastly, it is shown that if a user wishes to take the time to tune a fuzzy logic controller, FLC-RAS can optimally reconfigure an application in a peer-to-peer manner without any centralized or global knowledge. A hierarchical policy for autonomous malleability is presented, as well as different algorithms for performing malleability on actors that implement a simple API. The algorithms shown enable malleability for two representative application types with different communication topologies and data distributions. Autonomous reconfiguration using malleability is shown to be significantly more efficient than application stop-restart, and can provide improved performance for iterative applications over migration alone.

1.5 Overview

The remainder of this thesis proceeds as follows. The approach taken in developing the IOS middleware, as well as its modular framework and implementation details are discussed in Chapter 2. The details of enabling migration and malleability in the SALSA programming language are discussed in Section 2.1. Different peer-to-peer policies used by IOS for autonomous reconfiguration using migration and malleability are discussed in Section 2.2. The performance of reconfiguration and the overhead and benefits of the autonomous reconfiguration polices are evaluated in Chapter 3. This work is compared to other related work in Chapter 4. This thesis concludes with a final discussion of this research and proposed future work is given in Chapter 5.

CHAPTER 2
The Internet Operating System (IOS)

The Internet Operating System (IOS) [32] is a modular middleware that enables autonomous reconfiguration of distributed applications. IOS is responsible for the profiling and reconfiguration of applications, allowing an application's computational entities to be transparently reconfigured at runtime. IOS interacts with applications through a generic profiling and reconfiguration interface, and can therefore be used with different applications and programming paradigms, such as SALSA [19] and MPI [30]. Applications implement an interface through which IOS can dynamically reconfigure the application and gather profiling information. IOS uses a peer-to-peer network of agents with pluggable communication, profiling and decision making modules, allowing it to scale to large environments and providing a mechanism to evaluate different methods for reconfiguration.

An IOS agent is present on every node in the distributed environment. Each agent is modular, consisting of three plug-in modules: *i*) a *profiling module* that gathers information about applications' communication topologies and resource utilization, as well as the resources locally available, *ii*) a *protocol module* to allow for inter-agent communication, allowing the IOS agents to arrange themselves with different virtual network topologies, such as hierarchical or purely peer-to-peer topologies [32], and *iii*) a *decision module* which determines when to perform reconfiguration and how reconfiguration can be done.

The modules interact with each other and applications through well-defined interfaces, making it possible to easily develop new modules, and to combine them in different ways to test different types of application reconfiguration. The decision module interacts with applications through asynchronous message passing. For this work, the decision module autonomously creates directors for malleability and sends *migrate* messages to entities based on its decision making strategy.

IOS has been designed to address the needs of the large scale, dynamic and heterogeneous nature of emerging environments, as well as the various needs of the

applications which run on those environments. To effectively utilize these environments, the middleware must be able to run on heterogeneous compuational nodes and utilize heterogeneous profiling information. Due the the large scale and dynamic nature of these environments, the middleware cannot have centralized coordination because this approach is not scalable, and leads to a single point of failure.

The issue of heterogeneity was resolved by implementing IOS in Java and the SALSA Programming Language[46], which pre-compiles into Java and can run on Java Virtual Machines. This allows IOS to run on heterogeneous architectures with Java installed. IOS is made interactable with applications and third party profiling services via a socket based communication protocol. It is possible to provide built-in support for transparent or semi-transparent use of this interface with IOS with different programming models, such as has been done with SALSA [19] and MPI [31]. Otherwise, this socket based interface can be implemented by applications themselves. It is also possible for IOS to interact with third party profiling tools such as the Network Weather Service [48] and the Globus Meta Discovery Service [17], by implementing simple wrappers that interface with the profiling module.

Different protocol modules can connect the IOS agents with different communication topologies. This allows for centralized, hierarchical or pure peer-to-peer (p2p) strategies for information dispersal. Different peer-to-peer communication and reconfiguration strategies allow IOS to scale to a very large number of execution nodes. Additionally, they can be used to arrange IOS agents in a network-aware manner for additional improvement[32].

Additionally, different applications require different reconfiguration strategies, which may require different profiling information. Since increasing the amount of profiling done decreases performance by incurring additional overhead, it is not efficient to profile all possible information. Plug-in profiling modules allow the decision and protocol components to use only what profiling services are needed, improving performance.

Section 2.1 discusses the implementation of autonomous actors which enable transparent migration, and malleable actors which enable malleability. The process of making two representative applications malleable is discussed in detail. Follow-

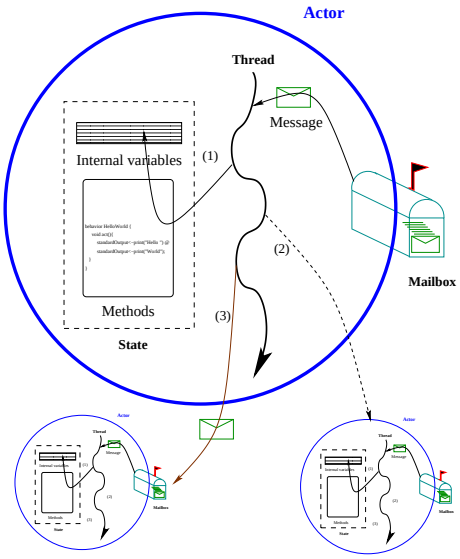

Figure 2.1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to peer actors (image from [45]).

ing this, implementations of different protocol, decision and profiling modules for autonomous reconfiguration of applications using IOS is detailed in Section 2.2.

2.1 Migration and Malleability Implementations in SALSA

Both migration and malleability have been implemented in the SALSA programming language [46]. SALSA is a distributed programming language based on the actor [4] model of computation. Actors, see Figure 2.1, encapsulate state and process into a single entity. Each actor has a unique name, which can be used as a reference by other possibly distributed actors. Communication between actors is purely asynchronous. The actor model assumes guaranteed message delivery and fair scheduling of computation. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: alter its state, create new actors, or send messages to peer actors. Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [26] and facilitates mobility [5].

SALSA programs may be executed in heterogeneous distributed environments.

SALSA code is compiled into Java source and then byte-code. This allows SALSA programs to employ components of the Java class library. It also enables SALSA developers to view a heterogeneous network of physical machines as a homogeneous network of virtual machines. For this work, each actor is seen as a reconfigurable component running on a *theater* which represents its local runtime environment. SALSA actors can migrate transparently without any modification of code, enabling easy autonomous reconfiguration through migration. Section2.1.1 describe how actors in SALSA have been modified to transparently provide profiling information for the IOS profiling module. Following this, Section 2.1.2 describes how actors have been extended to allow middleware driven autonomous malleability and Section 2.1.3 provides details about implementing these extensions in two sample applications.

2.1.1 Autonomous Actors

SALSA Actors can extend the `AutonomousActor` behavior, which tranparently provides profiling and interaction with the IOS middleware. Autonomous actors can profile their cpu usage, data usage, number of messages processed, where messages are sent to, and where messages are received from. All SALSA actors implement a `migrate` method, which allows them to be relocated arbitrarily at runtime. Migration is accomplished with the following protocol:

- Every theater has a *SystemActor* which handles local services and aids in migration.
- When an actor receives a `migrate` message, it sends a message to the `SystemActor` at its migration destination that causes the `SystemActor` to create a new actor locally. This new actor has the same unique name as the actor to be simulated, and upon creation it updates all references from the actor to be migrated to this new actor. While reference updating is taking place, both the old and new actor continue to receive messages in transit, but do not process them.
- When all references are updated, the old actor will refuse to receive any new

Return Type	Method Name	Parameters
long	getDataSize	()
Object	getData	(long size, Options options)
void	receiveData	(Object data, Options options)
void	redirectReference	(String refName, Reference newTarget)
void	handleMalleabilityMessage	(Message message)

Table 2.1: `MalleableActor API`

messages, instead redirecting them to the new actor. It then transfers its state and all messages in it's mailbox to the new actor. After the transfer, the old actor is removed.

- After receiving the state and messages from the old actor, the new actor will update its state, add these messages to its mailbox and being to process messages.

- All messages that were in transit to the old actor, upon not finding it at its former location, will do a new name to location lookup and then be sent to the new actor.

Because message passing is asynchronous, changing the order in which messages are received and processed will not effect the applications. This protocol also prevents the loss or duplication of messages, thus allowing actors to be migrated during runtime without effecting semantics of the application. When the IOS middleware decides to reconfigure a SALSA application via migration, it simply sends a migrate message to the actors to be migrated, which then use the above protocol.

2.1.2 Malleable Actors

Malleability has been implemented in SALSA [46] by providing extensible *directors* that control malleability on actors that implement the `MalleableActor` behavior (see Table 2.1). A user must implement methods which allow the directors to redirect references to accommodate newly created and removed actors, and redistribute data between malleable actors participating in reconfiguration. The

provided directors use a generic protocol to perform malleability operations. The method is language independent and can be applied to other distributed programming methods such as MPI [33] or agents. The reconfiguration method consists of three basic parts:

1. locking all participants in the reconfiguration.
2. redistributing data to newly created participants or from removed participants.
3. redirecting the references between them.

Locking malleable actors is required to prevent data inconsistencies that could arise during the redistribution of data and references. If the malleable actors can continue to process other messages while performing a reconfiguration, responses to messages and data could be sent to the wrong actors, resulting in incorrect computation. All implementations of malleability presented in this paper use the same locking strategy. Actors specify when they are at a stable state and can be locked by the library through invoking the `setMalleable` and `setUnmalleable` methods. Malleable actors are locked by the directors using the following protocol:

1. The director broadcasts an initiate malleability message to all malleable actors involved in the reconfiguration.
2. If any malleable actors broadcast to are involved in another reconfiguration, or have set to be unmalleable through `setUnmalleable`, they respond to the director with a `isUnmalleable` message. Otherwise, the actor responds to the director with a `isMalleable` message. After this, the actors responding with a `isMalleable` message will enqueue all messages received that are not related to the initiated reconfiguration, and only process them after the reconfiguration has completed.
3. If any malleable actor responds with a `isUnmalleable` message or a timeout elapses without all malleable actors responding with a `isMalleable` message, the director broadcasts a `cancelMalleability` message to all participating actors, who will process all messages they enqueued and continue to process messages as normal.

4. If all malleable actors respond with a `isMalleable` message, the director will then redirect messages and redistribute data between the participating actors. Section 2.1.3 describes implementations of data redistribution and reference redirection for sample applications.

5. After all redirection and redistribution is finished, the director will broadcast a `malleabilityFinished` message to all participating actors.

6. When an actor receives a `malleabilityFinished` message, it will process all messages that it queued while doing the reconfiguration via `handleMalleabilityMessage(Message m)`, then continue to process messages as normal.

The remainder of this section provides example implementations of directors and malleable actors for different applications, with an emphasis on algorithms that can efficiently redistribute data and redirect references. It concludes with a discussion of the complexity of modifying the applications to use the malleability libraries.

2.1.3 Example Applications

Directors have been created for two sample SALSA applications which have been modified to utilize the malleable actor API. Both applications are iterative. During each iteration they perform some computation and then exchange data. The solution is returned when the problem converges or a certain number of iterations have elapsed. The first application is an astronomical application [38] based on data derived from the SLOAN digital sky survey [40]. It uses linear regression techniques to fit models to the observed shape of the galaxy. This application can be categorized as a loosely coupled farmer/worker application. The second application is a fluid-dynamics application that models heat transfer in a solid. It is iterative and tightly coupled, having a relatively high communication to computation ratio.

Two types of data are defined for the purpose of this work: *spatially dependent data* that is dependent on the behavior of its containing entity and *spatially independent data* that can be distributed irrespective of entity behavior. For example,

the astronomy code has spatially independent data. It performs the summation of an integral over all the stars in its data set, so the data can be distributed over any number of workers in any way. This significantly simplifies data redistribution. On the other hand, each worker in the heat application has spatially dependent data because each data point corresponds to the heat of a point inside the solid that is being modeled. This means that worker actors are arranged as a doubly linked list, and each worker has a representative data slice with its neighbor having the adjacent data slices. This makes data redistribution more difficult than in sptially-independent data, because the data needs to be redistributed such that it preserves the adjacency of data points.

2.1.3.1 Astronomical Data Modeling

The astronomy code follows the typical farmer-worker style of computation. For each iteration, the farmer generates a model and the workers determine the accuracy of this model, calculated by using the model on each workers' individual set of star positions. The farmer then combines these results to determine the overall accuracy, determines modifications for the model and the process repeats. Each iteration involves testing multiple models to determine which model parameters to change, using large data sets (hundreds of thousands or millions of stars), resulting in a low communication to computation ratio, allowing for massive distribution.

In the malleability algorithm for data redistribution, `getDataSize` returns the total number of data points at a worker, `getData` removes and returns the number of data points specified, and `receiveData` appends the data points passed as an argument to the worker. Algorithm 1 is used to redistribute data in the astronomy application or any application with spatially-independent data. The middleware determines the workers that will participate in the malleability, `workers`, creating new workers if needed, and the `desiredWorkerData` for each of these workers (for more details, see Section 2.2). The algorithm divides `workers` into `workersToShrink`, specifying workers that will be sending data, and `workersToExpand`, specifying workers that will receive data. It then sends data from `workersToShrink` to `workersToMerge`. Message passing, denoted by the $\leftarrow$ operator, can be done asyn-

Algorithm 1: Spatially Independant Data Redistribution

```
Data: MalleableActor[] workers, long[] desiredWorkerData
Result: workers will have an amount of data specified by
        desiredWorkerData
long[] workerData = new long[workers.length]
for i = 1 ... workers.length do
    workerData[i] = workers[i]←getDataSize()
MalleableActor[] wte = {}, wts = {}
long[] edd = {}, ecd = {}, sdd = {}, scd = {}
/* split workers into the workers to shrink (wts) and the
   workers to expand (wts) and create arrays with their
   current (ecd, scd) and desired (edd, sdd) data size      */
for i = 1 ... workers.length do
    if workerData[i] < desiredWorkerData[i] then
        wte.append(workers[i])
        edd.append(desiredWorkerData[i])
        ecd.append(workerData[i])
    else if workerData[i] < desiredWorkerData[i] then
        wts.append(workers[i])
        sdd.append(desiredWorkerData[i])
        scd.append(workerData[i])
/* send data from workers to shrink to workers to expand    */
while currentShrink ≤ wts.length and currentExpand ≤ wte.length do
    dts = scd[currentShrink] - sdd[currentShrink]
    dtr = edd[currentExpand] - ecd[currentExpand]
    toSend = min(dts, dtr)
    wte←receiveData(wts←getData(toSend))
    scd[currentShrink] -= toSend
    dts -= toSend
    dtr -= toSend
    if dts == 0 then
        currentShrink++
    if dtr == 0 then
        currentExpand++
```

chronously and messages can be processed in parallel.

In the case of a split, new actors will be created by the director and appended to *workers*, and the middleware will specify the desired data for each and append this to `desiredWorkerData`. In the case of a merge, the actors that will be removed have their desired data set to 0, and after reference redirection, data distribution and processing all messages with `handleMalleabilityMessage`, they are garbage collected. The middleware ensures that the data sizes are correct and enough data is available for the algorithm.

If n is the number of `workersToShrink`, and m is the number of `workersToExpand`, this algorithm will produce $O(n+m)$ `getData` and `receiveData` messages, because after each transfer of data, either the next workerToShrink will be sending data, or the next workerToExpand will be receiving data, and the algorithm ends when the last workerToShrink sends the last amount of data to the last workerToMerge. This algorithm requires at most $\lceil \frac{m}{n} \rceil$ sequential `getData` messages and $\lceil \frac{n}{m} \rceil$ sequential `receiveData` messages to be processed by each worker. Each worker can perform it's sequential `getData` and `receiveData` messages in parallel with the other workers and all data can be transferred asynchronously and in parallel, as the order of message reception and processing will not change the result.

Reference redirection in the astronomy application is trivial. The farmer only needs to be informed of the new workers on a split, and of the removed workers on a merge.

Malleability provides a significant improvement over application stop-restart for reconfiguration of the astronomy application. To stop and restart the application, the star data needs to be read from a file and redistributed to each node from the farmer. Apart from the relative slow speed of file I/O, the single point of data distribution acts as a bottleneck to the performance of this approach, as opposed to split and merge which concurrently redistribute that is already in memory.

2.1.3.2 Simulation of Heat Diffusion

The heat application's components communicate as a doubly linked list of workers. Workers wait for incoming messages from both their neighbors and use this

data to perform the computation and modify their own data, then send the results back to the neighbors. The communication to computation ratio is significantly higher than the astronomy code, and the data is spatially dependent making data redistribution more complicated.

To redistribute data in the heat application, the data must be redistributed without violating the semantics of the application. Each worker has a set of columns containing temperatures for a slice of the object the calculation is being done on. Redistributing data must involve shifting columns of data between neighboring workers, to preserve the adjacency of data points. For this application, because data can only be shifted to the left or right neighbors of a worker, malleability operations are done with groups of adjacent workers. Section 2.2.2.3 describes how the middleware obtains groups of adjacent workers.

Algorithm 2: Spatially Dependant Data Redistribution

```
Data: MalleableActor[] workers, long[] desiredWorkerData
Result: workers will have an amount of data specified by
        desiredWorkerData
/* obtain the amount of data at each worker                    */
long[] workerData = new long[workers.length]
for i = 1 ... workers.length do
    workerData[i] = workers[i]←getDataSize()
current = 1
while current ≤ workers.length do
    if workerData[current] < desiredWorkerData[current] then
        Shift-Left(current, workers, desiredWorkerData, workerData)
        /* see algorithm 3                                     */
    else if woorkerData[current] > desiredWorkerData[current] then
        Shift-Right(current, workers, desiredWorkerData, workerData)
        /* see algorithm 4                                     */
    else
        current++
```

The `getData`, `receiveData` and `getDataSize` methods are implemented differently for the heat application. The `getDataSize` method returns the number of data columns that a worker has, `getData` removes and returns data columns from the workers data, while `receiveData` adds the columns to the workers data. Options

Algorithm 3: Shift-Left

```
Data: int current, MalleableActor[] workers, long[] desiredWorkerData,
      long[] workerData
Result: data is shifted left to the current worker until it has the desired
        amount of data
/* the current worker needs more data so shift data left to
   this worker                                                  */
dataToSend = desiredWorkerData[current]-workerData[current]
target = current+1
while dataToSend > 0 do
    while workerData[target] = 0 do
        target++
    toSend = min(dataToSend, workerData[currentTarget])
    data = workers[target]←getData(toSend, left);
    workers[current]←receiveData(data, right); dataToSend -= toSend
    workerData[target] -= toSend
    workerData[current] += toSend
```

to these messages can be set to `left` or `right`, which determines if the columns are placed or removed from: the front of the columns, left, or at the end of the columns, right. These methods allow the director to shift data left and right over the group of actors.

Algorithm 2 is used by the director to redistribute data in the heat application. The arguments to the algorithm are the same as in the astronomy split algorithm and message passing is denoted by the ← operator. The algorithm iterates through the workers, shifting data left and right as required to achieve the `desiredWorkerData`. The middleware will create new workers and/or specify workers to be removed (by setting their desired data to 0) and determines the data that each worker should have. It then creates a director which uses this method to redistribute data. To preserve data consistency, unlike in the astronomy algorithms, the `getData` and `receiveData` messages sent to each actor must be processed in the order they were sent to that actor, however, these messages and the data transfer can still be done concurrently.

At most O(n) `getData` and `receiveData` messages will be sent by this algorithm, where n is the number of workers. This is because shifting data rightward or

Algorithm 4: Shift-Right

```
Data: int current, MalleableActor[] workers, long[] desiredWorkerData,
      long[] workerData
Result: data is shifted right from the current worker, until it has the
        desired amount of data
/* the current worker has too much data so shift data right
   from this worker                                              */
dataToSend = workerData[current]-desiredWorkerData[current]
/* find the rightmost worker that data needs to be shifted to
   */
target = current
totalDesired = desiredWorkerData[target]
totalCurrent = workerData[target]
while totalDesired < totalCurrent do
    target++
    totalDesired += desiredWorkerData[currentTarget]
    totalCurrent += workerData[currentTarget]
/* find the initial amount of data to send to the target    */
if totalCurrent > totalDesired then
    dataToSend = (totalDesired-desiredWorkerData[target])-
    (totalCurrent-workerData[target])
else
    dataToSend = (desiredWorkerData[target]-workerData[target])
/* shift data right, towards the target                     */
source = target-1
while target > current do
    while dataToSend > 0 do
        toSend = min(workerData[source], dataToSend)
        data = workers[source]←getData(toSend, right)
        workers[target]←receiveData(data, left)
        dataToSend -= toSend
        workerData[target] += toSend
        workerData[source] -= toSend
        if workerData[source] == 0 then
            source–
    target–; if source = target then
        source–;
    dataToSend = desiredWorkerData[target]-workerData[target];
```

	Heat	Astronomy
Unmodified	1462	1163
Malleable	1588	1363
Malleable including Director	1874	1520

Figure 2.2: Lines of code in the unmodified (non-malleable) code, and malleable code with and without the lines of code of the split and merge directors. Director code can be reused by the same type of applications.

leftward as in the algorithm involves at most $2m$ `getData` and `receiveData` messages, where m is the subset the data is being shifted over and data is only shifted over a subset once. At most $\lceil \frac{\max_{\forall i,\ s.t.\ d_i - ds_i > 0}(d_i - ds_i)}{\min_{\forall i,\ s.t.\ ds_i - d_i > 0}(ds_i - d_i)} \rceil$ `receiveData` messages and at most $\lceil \frac{\max_{\forall i,\ s.t.\ ds_i - d_i > 0}(ds_i - d_i)}{\min_{\forall i,\ s.t.\ d_i - ds_i > 0}(d_i - ds_i)} \rceil$ `getData` messages must be processed by each actor, where d_i is the amount of data at worker i and ds_i is the desired data for worker i.

Reference redirection in the heat application works the same as adding and removing nodes using a doubly linked list. After data redistribution, workers that have no remaining data are removed from the list of workers. After this, the director updates the left and right references of each worker with the `redirectReference` method and the removed workers can be garbage collected. All these messages can be sent concurrently and processed in parallel.

Using malleability with the heat application provides another benefit over application stop-restart than with the astronomy application. In astronomy application the data points (which are star coordinates) do not change over the execution of the application, however each iteration modifies the data points in the heat application during the simulation of heat diffusion. Because this is the case, each worker needs to report its data back to the farmer which has to combine the data of all the workers and store the data to a file, and then redistribute it to the newly restarted application. The provided algorithm allows the data to be redistributed concurrently using data already in memory for a large improvement in reconfiguration time.

2.1.3.3 Programming Complexity

The programming complexity of extending an application to implement the `MalleableActor` API, in terms of lines of code is shown in Figure 2.2. Appendix A shows the implementation of the `Malleable Actor` API for the heat application. Code added to implement the API is in bold. The astronomy application took more lines of code to implement malleability, due to a more complex data representation. The *getData* and *receiveData* methods make up the majority of the additional lines of code, due to the complexity of manipulating the data structures. Director code can be reused by the same type of applications, significantly simplifying the malleability programming process.

2.2 Reconfiguration Policies

IOS has been used to implement different reconfiguration policies, due to the fact that as applications become more complex, they require more complicated profiling and reconfiguration to achieve performance gains from autonomous reconfiguration. This section discusses the use of IOS to implement three different strategies for autonomous migration, each with different benefits and drawbacks. The general protocol module used by all the autonomous reconfiguration strategies is described in Section 2.2.1. Different implementations of profiling and decision modules of varying complexity are described. The least complex, simple random actor stealing, is described in Section 2.2.1.1. Two more involved strategies, a heuristic based strategy in Section 2.2.1.2 and a fuzzy logic based strategy in Section 2.2.1.3. Following this, the implementation of a hierarchical strategy for autonomous malleability in IOS is discussed in Section 2.2.2.

2.2.1 The Random Actor Stealing (RAS) Protocol

All the autonomous migration strategies presented in this section use the same Random Actor Stealing (RAS) protocol module for IOS, which drives the reconfiguration process and arranges the IOS nodes into a virtual network. RAS is a autonomous reconfiguration protocol loosely based on MIT's Cilk [13], and peer to peer systems such as Gnutella [16]. Each protocol module contains a list of neigh-

bors, other IOS nodes that are running on machines participating to execute the application. An IOS node can connect to other IOS nodes via *peer servers*, decentralized bootstrapping servers which contain lists of other IOS nodes, or by directly adding known IOS nodes to their neighbor list (through a command line argument on startup). Given a protocol module with a list of neighbors, the following protocol is used to perform autonomous reconfiguration:

- The RAS protocol module will attempt to perform reconfiguration when the local CPU usage drops below a certain threshold percentage and there is more than one autonomous actor present locally. When this happens, the protocol module will grab a snapshot of the local profiling information from the profiling module, and create a random steal request message with a time to live value. A neighbor is chosen at random and this steal request message will be sent to the protocol module at that IOS node. The RAS protocol module will not send another steal request message until it has recieved a response that the time ot live of the currently active message has been reached without finding a successful candidate for reconfiguration, or another RAS protocol module responds that it wishes to perform reconfiguration.

- Upon receiving a steal request message, the RAS protocol module will send the profiling information from the steal request to the decision module and use it to determine if reconfiguration should occur with the initiator of the steal request message. If the decision module specifies reconfiguration should occur, then the decision module will send reconfiguration messages to the application that will perform the desired reconfiguration. In this case the protocol module responds to the initiator of the steal request message that reconfiguration will be performed. If the decision module specifies that no reconfiguration should be done, if the time to live value is zero, the protocol module responds to the initiator that it's steal request was unsuccessful, otherwise the time to live value is decremented and the steal request is forwarded to a neighbor of the current protocol module.

This protocol provides allows the IOS network to form a decentralized network that can efficiently provide autonomous reconfiguration with low overhead (see Chapter 3). This protocol is also *stable*, in that it doesn't incur additional overhead when the network is heavily loaded, as no steal requests are sent out.

2.2.1.1 Simple Random Actor Stealing (S-RAS)

Simple random actor stealing (SRAS) requires a minimal amount of profiling information, and a minimal amount of computational power to determine how autonomous reconfiguration should be done. The profiling module at each IOS node only calculates the available computational power at that node. The available computational power is used by the protocol module to determine when steal requests are generated and by the decision module to determine when autonomous reconfiguration should be done. When a steal request packet is recieved, if there are multiple actors present locally, and all the computational resources are in use (the available computational power is approximately 0%), one of local the actors is chosen at random and migrated to source of the steal request.

2.2.1.2 Application Topology Sensitive RAS (ATS-RAS)

Application topology aware random stealing (ATS-RAS) provides a layer of complexity on top of SRAS. The profiling module keeps track of what theaters actors are sending messages to. This allows the decision module to determine reconfiguration that will co-locate *tightly coupled* actors, actors which communicate frequently, within the same theater and locate *loosely coupled* actors, actors which communicate infrequently or not at all, in separate theaters. This attempts to provide a mapping of the application topology to physical network in a decentralized fashion without global knowledge. The decision module computes an estimation of the number of messages an actor would process per second in the remote theater, and compares it to the number of messages the actor processes per second in the local theater. The decision module chooses the actor with the best estimated increase in processed messages per second (if one exists), and will migrate this actor to the source of the random steal packet.

Formally, the ATS-RAS decision function can be defined as follows. Given an

actor A, local theater L and remote theater R, the gain of moving actor A from theater L to theater R, denoted as $G(L, R, A)$ is:

$$G(L, R, A) = M(L, A) - M(R, A) \tag{2.1}$$

where $M(t, a)$ is an estimation of the number of messages actor a will process per second at theater t. $M(L, A)$ can be calculated as follows:

$$M(L, A) = \frac{P(A)}{T(A)} \tag{2.2}$$

where $P(A)$ is the number of messages the actor has processed since the last time its profiling information was reset, and $T(A)$ was the time since this last reset. The IOS profiling module periodically resets profiling information it is up to date with the current conditions of the application and its environment. $M(R, A)$, an estimation of the number of messages the actor would process at R, can be calculated by:

$$M(R, A) = \frac{F(A)}{P(A)} + \frac{T(A)}{F(A) * (S(A, L) - S(A, R))} \tag{2.3}$$

where $F(A)$ is the average number of FLOPS actor A takes to process a message and $S(A, t)$ is the number of messages actor A has sent to theater t since the last time its profiling information was reset.

The ARS decision function provides a measure of how fast an actor will process messages in theater that is the source of the steal packet. If the actor will process messages faster remote, it is migrated to the source of the steal packet. The results section 3 shows that ARS results in slightly more overhead than RS, but is much more accurate in load balancing applications that involve inter-actor communication, providing significant performance increases.

2.2.1.3 Fuzzy Logic Controlled RAS (FLC-RAS)

Fuzzy logic controlled random actor stealing (FLC-RAS) uses a decision module which generates for each actor a measure of the gain in communication and a measure of the gain in computation which would result from moving that local actor

Notation	Explanation
$p_l(A)$	Percentage of the CPU used by actor A at the local theater l
$p_f(A)$	Percentage of the CPU available at the remote theater f
$c_l(A)$	Communication (in bytes) done by actor A to all other actors at the local theater l
$c_f(A)$	Communication (in bytes) done to all actors at remote theater f
$g_p(A,l,f)$	The benefit in computation gained by migrating actor A from local theater l to remote theater f. $g_p(A,l,f) = p_f(A) - p_l(A)$
$g_c(A,l,f)$	The benefit in communication gained by migrating actor A from local theater l to remote theater f. $g_c(A,l,f) = c_l(A) - c_f(A)$

Table 2.2: Gain Measurement Equations

to the source of the reconfiguration request message. A fuzzy membership function is then applied to both these values, and used within a set of rules to generate an approximate value of how desirable that reconfiguration is. The decision module will then inform the protocol agent of the most desirable reconfiguration over a certain threshold (if one exists), or otherwise inform the protocol agent that no reconfiguration should be done.

The measures of the gain from actor migration are defined in Table 2.2. Both the communication and computation gain are centered at 0 for no difference, with a negative value being a loss, and a positive value being a gain. However, these values are not normalized and will have different scales for different applications, depending on the amount of communication and computation being done.

Membership functions are then applied to each of these measures, one membership function for the measure being good, and another for the measure being bad. These membership functions are similar to trapezoidal membership functions, except each is open ended. Figure 2.3 gives a graphical representation of the good and bad membership functions, and Table 2.3 provides their mathematical representation. Both membership functions rely on two cutoff values, one determining when the measure is definitely bad, and another determining when the measure is definitely good. Currently, these values are tuned by the application designer.

The fuzzy values generated from the membership functions above are:

Notation	Explanation
g_r	The gain measurement for resource r. For the scope of this work, r can be either communication or computation.
$t_{r,bad}$	The cutoff for the measurement of resource r being definitely bad.
$t_{r,good}$	The cutoff for the measurement of resource r being definitely good.
$good(g_r)$	The fuzzy value designating the "goodness" of this measurement of resource r. if $g_r < t_{r,bad}$ then $good(g_r) = 0$ if $t_{r,bad} < g_r < t_{r,good}$ then $good(g_r) = \frac{g_r - t_{r,bad}}{t_{r,good} - t_r, bad}$ if $t_{r,good} < g_r$ then $good(g_r) = 1$
$bad(g_r)$	The fuzzy value designating the badness of resource r. $badg_r = 1 - good(g_r)$

Table 2.3: Good and Bad Membership Function Definitions

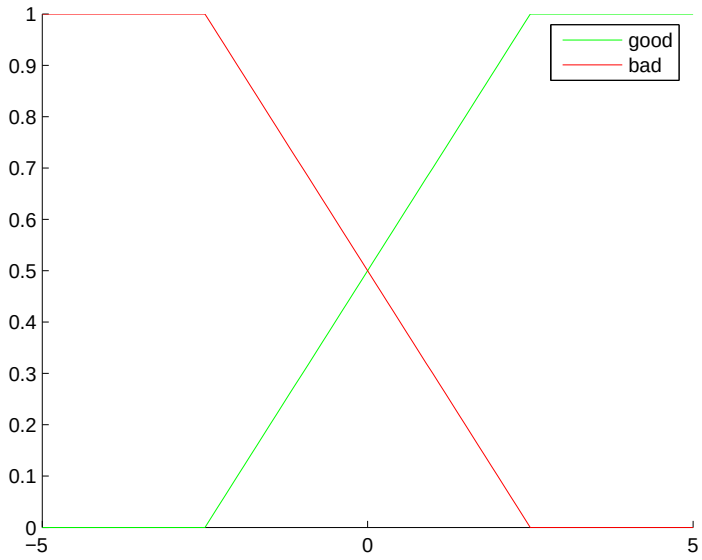

Figure 2.3: Graphical representations of the good and bad membership functions, with the bad cutoff at -2.5 and the good cutoff at 2.5.

1. communication will improve (c_{good})
2. communication will be impaired (c_{bad})
3. computation will improve (p_{good})
4. computation will be impaired (p_{bad})

These use a Sugeno style method of combination, with the following rule set, using the and (or minimum) fuzzy operator:

- if c_{bad} and p_{bad} then 0
- if c_{good} and p_{bad} then $\frac{c_{good}+p_{good}}{2}$
- if c_{bad} and p_{good} then $\frac{c_{good}+p_{good}}{2}$
- if c_{good} and p_{good} then 1

Whichever rule satisfies the conditional the most is used to determine the measure of desirability to move this actor to the source of the reconfiguration message. It should be noted that applying these rules returns a value between 0 and 1, where 0 is definitely do not move, and 1 is definitely move. The threshold used to determine the minimum desirability required for migration is 0.5. It is possible to weight the return values for the middle two rules, however this is done by modifying the cutoff values used in the membership functions, instead of introducing two new values to be tuned.

Tuning the cutoff values of the membership function described is essential to obtaining good reconfiguration results. Increasing the cutoff values would weight the value of that resource (computation or communication), because better gain measures would still be considered bad. Appropriate cutoffs for communication and computation gains needed to be found, because if communication is weighted too heavily, all the actors will converge to a single processor, minimizing all remote communication. Likewise, if computation was weighted too heavily, the actors will disperse evenly amongst all the processors, but there would be little attention paid to co-locating communicating actors, so a large of overhead will be incurred from the

large amount of remote communication. Additionally, the distance between the good and bad cutoffs needs to be modified, because the communication and computation values needed to be weighted correctly to achieve a stable result. If the cutoffs are incorrect, near optimal configuration solutions are reached, but actors would still migrate between similar theaters, resulting in an unstable configuration. Actor migration can be an expensive operation, so it is possible for unstable configurations to lead to performance degradation.

2.2.2 Hierarchical Grid-Aware Malleability

The IOS middleware was used to implement a decision module that autonomously creates directors to reconfigure applications using malleability, in response to changes in the environment. Developers specify how to make entities malleable through the `MalleableActor` API and can use previously defined directors or create new ones which specify how to redistribute data and redirect references. This allows the middleware to determine when to perform malleability and with what actors. When the availability of resources in an execution environment changes, the application needs to be reconfigured to ensure good performance. However, reconfiguration cannot be oversensitive to resource availability, which can result in the application constantly being reconfigured, degrading performance. In general, resource availability can change when processors become available or unavailable, or due to applications competing for resources on the same processors.

This section continues with a description of the policy used to determine what and when to reconfigure in Sections 2.2.2.1 and 2.2.2.2. Finally, restrictions the middleware places on malleability to prevent performance degradation, as well as the role application behavior can play in determining which entities can participate as a group in malleability and how the middleware can autonomously overcome this are discussed in Section 2.2.2.3.

2.2.2.1 A Hierarchical Malleability Policy

Malleability, as implemented, can be performed by any number of actors. The severity of change in resource availability can be reflected by the change in granularity. Just as the grid is arranged hierarchically with groups of processors in different

computing nodes making clusters, and groups of clusters forming grids, malleability can be done hierarchically - at the node level to accomplish memory hierarchy and processor optimization, at the cluster level for intra-cluster changes in processor availability, and at the grid level for changes in cluster availability. Grid-level reconfiguration can be done at different scales as well, on different subsets of clusters or the entire grid depending on what is feasible for the application. In all cases of reconfiguration, the same mechanics are used, just at different scales, which allows for a simple but effective strategy for application reconfiguration. This section describes the general problem to determine actor granularity for an application, then describes how a hierarchical strategy can be used to solve this problem efficiently on a dynamic environment.

IOS agents manage data redistribution for processors at their local node. The protocol module arranges IOS agents hierarchically [32]. Groups of IOS agents elect a cluster manager to perform data redistribution among themselves and groups of cluster managers will elect grid managers to perform data redistribution among the clusters represented by the cluster managers. If required, multiple grid managers can elect a super-grid manager, and so on. A manager views its subordinate IOS agents as collective individuals, and redistributes data between them if the average iteration time of the slowest agent is some threshold slower than the IOS agent with the fastest average iteration time. A manager will also perform reconfiguration if one of it's subordinates and the group of agents managed by that subordinate become unavailable, such as in the case of a cluster becoming unavailable. If a manager needs to become unavailable, but not the group of agents it manages, that group of agents will select another manager which will perform reconfiguration among that group in response to the old manager becoming unavailable. If the manager determines it will perform reconfiguration, it first polls its subordinate agents to calculate the total amount of data it is responsible for, d_t, and the available processing power, p_{a_i} of each of it's subordinate agents a_i. p_{a_i} is determined by the sum over all a_i's managed processors, mp_{a_i}, of the available percentage of cpu, $mp_{a_i,a}$, multiplied by that processors clock rate, $mp_{a_i,ghz}$:

$$p_{a_i} = \sum_{\forall mp_{a_i}} mp_{a_i,ghz} * mp_{a_i,a} \tag{2.4}$$

A user defined iteration operations estimation function, $f(d)$, estimates the number of operations performed by an iteration given an amount of data d. This can reflect the fact that the application scales linearly, polynomially or exponentially with respect to data size. To determine the amount of data that should be at each of its subordinate agents, d_{a_i}, given the normalized collective processing power of an agent p'_{a_i}, the manager solves the following system of equations:

$$p'_{a_i} = \frac{p_{a_i}}{\sum_{i=1}^{n} p_{a_i}} \tag{2.5}$$

$$\frac{f(d_{a_1})}{p'_{a_1}} = \frac{f(d_{a_2})}{p'_{a_2}} = ... = \frac{f(d_{a_n})}{p'_{a_n}} \tag{2.6}$$

Subject to restrictions:

$$\sum_{i=1}^{n} d_{a_i} = d_t \tag{2.7}$$

$$d_{a_1}, d_{a_2}, ..., d_{a_n} > 0 \tag{2.8}$$

The manager will then request the desired actor size for each actor managed by its subordinate agents given the amount of data that the manager will have after the reconfiguration, d_{a_i}. If the subordinate agent is also a manager, it solves the above system of equations for its subordinates and requests the desired actor size for the actors at each of its subordinate agents. At the lowest level of the hierarchy, the size of individual actors at each processor is determined. With d_{p_i}, the data that should be at processor p_i known, the desired granularity for entities on that processor, g_{p_i}, can be calculated. M is a user-defined variable that specifies how efficiently the application can utilize the memory hierarchy. M can be set to 1 for applications that have highly complex data patterns which do not gain any benefit from memory caching mechanisms. For applications which can gain some benefit from caching mechanisms, M can be set to reflect how many times larger than the cache actor data can be and still effectively use that cache. For applications that

have optimized data access that always uses the fastest level of memory hierarchy, M can be set to infinity. Given c_{j,p_i}, the memory available in the level j cache of processor p_i:

$$g_{p_i} = \begin{Bmatrix} \min_{\forall c_{j,p_i}} \ c_{j,p_i} * M, \ s.t. \ \frac{d_{p_i}}{c_{j,p_i} * M} < max_e \\ \frac{d_{p_i}}{max_e}, \ otherwise \end{Bmatrix} \tag{2.9}$$

This determines the granularity an actor will need to run in the fastest level of memory hierarchy, without creating a number of entities that would surpass a user specified limit on entities per processor max_e. Given d_{p_i} and g_{p_i}, the number of entities at that processor, e_{p_i}, can be determined as follows:

$$e_{p_i} = \max(\frac{d_{p_i}}{g_{p_i}}, 1) \tag{2.10}$$

Solving for the data size of actors propagates down the hierarchy, and then they gather the desired actor data size for all actors involved in the reconfiguration, and respond back up the hierarchy to the initiator of the reconfiguration. The manager that initiated the reconfiguration then creates a director to perform the reconfiguration between all involved actors. If required due to a large number of actors being involved in the reconfiguration, multiple directors with different subsets of the group can be used.

This strategy allows the lowest level of the hierarchy to handle the most common changes in resource availability. Fluctuations in performance at individual processors are handled locally by that node's IOS Agent if it is a multi-processor node, or by that processors cluster manager otherwise. This allows reconfiguration to be done more efficiently than if it was done on an application wide scale. Additionally, these small scale fluctuations can be identified and responded to faster because they are done locally. Larger fluctuations in performance, such as an entire cluster or large number of processors at a cluster becoming available or unavailable can be handled by large scale reconfiguration, done at the grid level. Section 3.4 discusses the performance of node, cluster and grid level reconfiguration.

2.2.2.2 Determining Data Redistribution

For most iteration operations estimation functions, a solution to the system of equations described by Equations 2.6, 2.7 and 2.8 can be solved algebraically. This allows the data size for each subordinate agent (or processor) to be determined in linear time. When the number of operations scales linearly with respect to data size, such as in the applications discussed in this paper, the linear iteration time estimator can be defined by an estimation of iteration start up operations, $b \geq 0$, and a scaling factor, $a \geq 0$:

$$f(d) = a * d + b \tag{2.11}$$

Then all d_{a_i} can be solved for by setting all $\frac{f(d_{a_i})}{p'_{a_i}}$ equal to some variable y, which can then be determined.

$$\frac{a * d_{a_1} + b}{p'_{a_1}} = \frac{a * d_{a_2} + b}{p'_{a_2}} = ... = \frac{a * d_{a_n} + b}{p'_{a_n}} = y,\ d_{a_i} = \frac{y - \frac{b}{p'_{a_i}}}{\frac{a}{p'_{a_i}}} \tag{2.12}$$

$$d_t = \sum_{i=1}^{n} d_{a_i} = \sum_{i=1}^{n} \frac{y - \frac{b}{p'_{a_i}}}{\frac{a}{p'_{a_i}}} \tag{2.13}$$

$$y = \frac{a * d_t + n * b}{\sum_{i=1}^{n} p'_{a_i}},\ \sum_{i=1}^{n} p'_{a_i} = 1,\ y = a * d_t + n * b \tag{2.14}$$

Thus the data for each agent a_i can be determined:

$$d_{a_i} = p'_{a_i} * (d_t + \frac{n * b}{a}) - \frac{b}{a} \tag{2.15}$$

Table 2.4 gives a simple example of using these functions on four different processors. Assuming no start-up cost ($b = 0$), and a linear scaling coefficient of 2 ($a = 2$), this table demonstrates the amount of data that each processor would be assigned (assuming a total amount of 10GB data).

	Sun (0.8GHz)	AIX (1.7GHz)	Opteron (2.2GHz)
Number of Actors	3	5	8
$pa_i{}'$	0.17	0.36	0.47
Data Assignment (d_{a_i})	1.7GB	3.6GB	4.7GB
Data per Actor	0.57GB	0.72GB	0.59GB

Table 2.4: An example of using the malleability data distribution functions. For 10GB data, assuming no start-up cost ($b = 0$) and a linear scaling coefficient of 2 ($a = 2$), this table provides the amount of data per processor and per actor for the given three machines.

2.2.2.3 Restrictions on Malleability

In certain situations, application performance may degrade by performing malleability. A split may degrade application performance if granularity becomes too small, causing the additional communication and context switching overhead from entities to become more expensive than the computational advantage of using additional resources. Another user-specified variable, *minimum actor data*, specifies the minimum amount of data an actor should have. If malleability will reduce the amount of data at actors below that threshold (assuming that actor is not being removed), the malleability will not occur. Another instance where a split can degrade application performance, in the case of an iterative application, is when a new cluster with a smaller memory hierarchy joins the computation. The application can split and the data size of the actors at the new cluster will fit into a slower cache than on the other clusters. This will degrade the performance of the entire application, because each iteration has to wait for the slow cluster to finish computation. To determine if this is the case, in applications where memory hierarchy plays a role in determining performance, after determining the size of the data to be sent to the new cluster, that cluster manager determines if the entities will use a comparable level of memory hierarchy to the other clusters. If not, splitting to that cluster is disabled. Similarly, a merge could increase the data at the remaining actors, resulting in increased use of slower memory. However, in this case the merge is still performed, because the resources need to be freed, but processor-level malleability will re-split at processors that would suffer from performance degradation, if possible.

For some applications, such as the heat application, the groups of actors that can perform malleability depend on the behavior of the application. For example, in the heat application, malleability can only be done with groups of adjacent entities, to ensure that the data in the new set of entities is made of adjacent slices. Application-topology sensitive migration [19], which co-locates communicating and tightly coupled entities, is leveraged to autonomously collocate adjacent entities on clusters and processors. Before malleability is performed, the director determines if the reconfiguration is possible. If the group is malleable (in the case of the Heat application, all entities are adjacent), the reconfiguration proceeds. If the group is not malleable, the middleware waits for a period of time for migration to occur to collocate the entities, then tries again. For other applications with behavioral restrictions on malleability, the process is the same: an IOS decision agent is used that autonomously co-locates entities with migration so malleability can occur.

CHAPTER 3
Results

This chapter evaluates the different autonomous reconfiguration strategies presented in the thesis. IOS was evaluated using different sample applications and benchmarks on various environments. Section 3.1 describes the different environments used in the evaluations. The overhead incurred by the different profiling and reconfiguration strategies is examined in Section 3.2. The benefits of using the autonomous reconfiguration strategies presented for migration with different types of applications are examined in Section 3.3. Reconfiguration using application stop-restart is compared to malleability in Section 3.4. Lastly, this chapter concludes with Section 3.5 providing a comparison of autonomous migration to autonomous malleability for the iterative heat and astronomy applications discussed in Section 2.1.

3.1 Test Environments

Four different clusters were used to evaluate the performance and overhead of migration and malleability. The first, the AIX cluster, consists of four quad-processor single-core Power-PC processors running at 1.7GHz, with 64KB L1 cache, 6MB L2 cache and 128MB L3 cache. Each machine has 8GB RAM, for a total of 32GB ram in the entire cluster. The second cluster, the single-core Opteron cluster, consists of twelve quad-processor, single-core Opterons running at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. Each machine has 16GB RAM, for a total of 192GB RAM. The third cluster, the dual-core Opteron cluster, consists of four quad-processor, dual-core opterons running at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. Each machine has 32GB RAM, for a total of 128GB RAM. The last cluster, the SUN Solaris cluster, consists of four dual-processor, single-core SUN Solaris processors running at 800MHz, with 64KB L1 cache and 8MB L2 cache. Each machine has 2GB RAM, for a total of 8GB RAM.

The single- and dual-core Opteron clusters are connected by 10GB/sec bandwidth, 7usec latency Infiniband, and 1GB/sec bandwidth, 100μsec latency Ethernet.

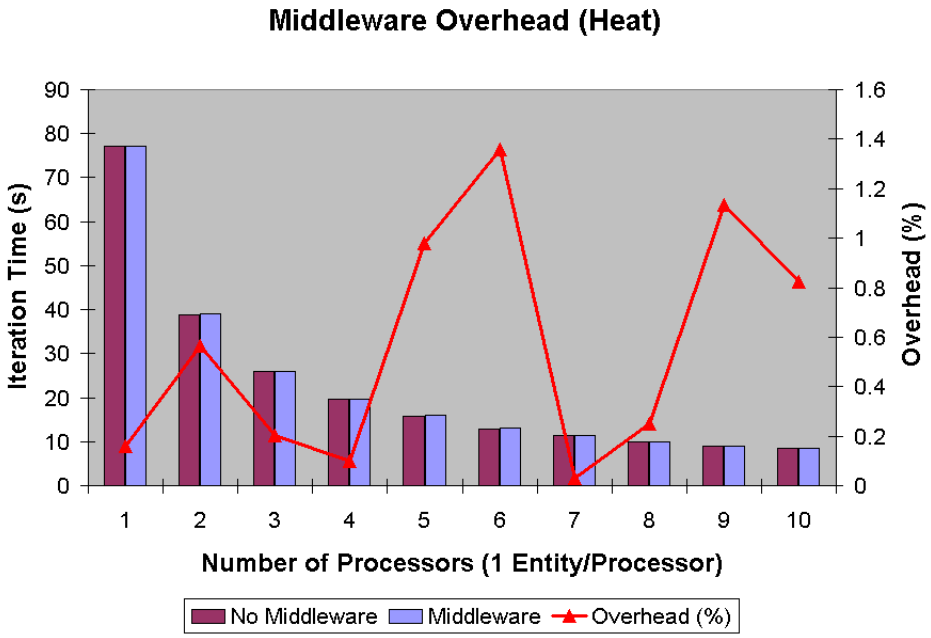

Figure 3.1: Overhead of using autonomous middleware and the `MalleableActor` interface for the heat and astronomy applications. The application was tested with the same configurations with different amounts of parallelism, with and without the middleware and `MalleableActor` interface.

Intra-cluster communication on the AIX and Sunclusters is with 1GB/sec bandwidth, 100μsec latency Ethernet, and they are connected to the each other and the Opteron clusters over the RPI campus wide area network.

3.2 Overhead

To evaluate the overhead of using the IOS middleware, the heat and astronomy applications were run with and without middleware and profiling services. The applications were run on the same environment (the AIX cluster) with the same configurations, however autonomous reconfiguration by the middleware was disabled. Figures 3.1 and 3.2 shows the overhead of the middleware services. The average overhead over all tests for the applications was between 1-2% (i.e., the application was only 1-2% slower using the middleware).

Figure 3.3 demonstrates the low overhead of implementing the MalleableActor

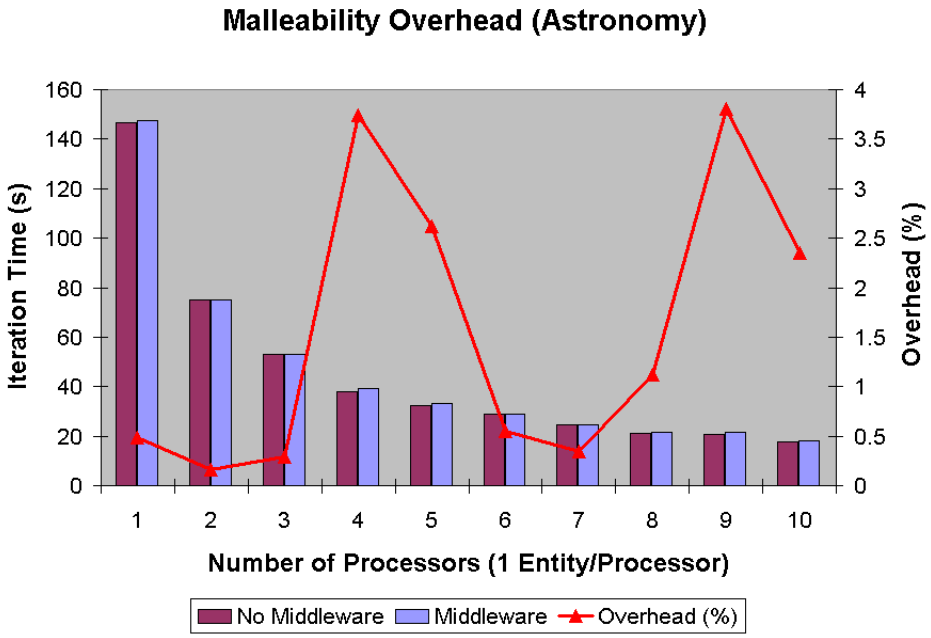

Figure 3.2: Overhead of using autonomous middleware and the `MalleableActor` interface for the heat and astronomy applications. The application was tested with the same configurations with different amounts of parallelism, with and without the middleware and `MalleableActor` interface.

API. The heat application was run on the four clusters with and without implemeting the MalleableActor API, with 500KB and 10MB of data. For larger data sizes and the astronomy application, overhead was negligible. For each environment, the heat application was run with one malleable actor per processor, so 8 actors on the SUN cluster, 16 on the AIX cluster, 32 on the 4x2 Opteron cluster, and 48 on the 4x1 Opteron cluster. For 500KB the overhead ranged from 2% to 7.5%, while for 10MB the overhead was below 1%. For 500KB of data, the 4x2 Opteron cluster had the highest overhead because it performed the computation and communication the fastest as the application could not scale to the 4x1 Opteron cluster with that amount of data. These results show that for typical grid applications, which have much larger data sizes than those evaluated, the overhead of using malleability is negligible.

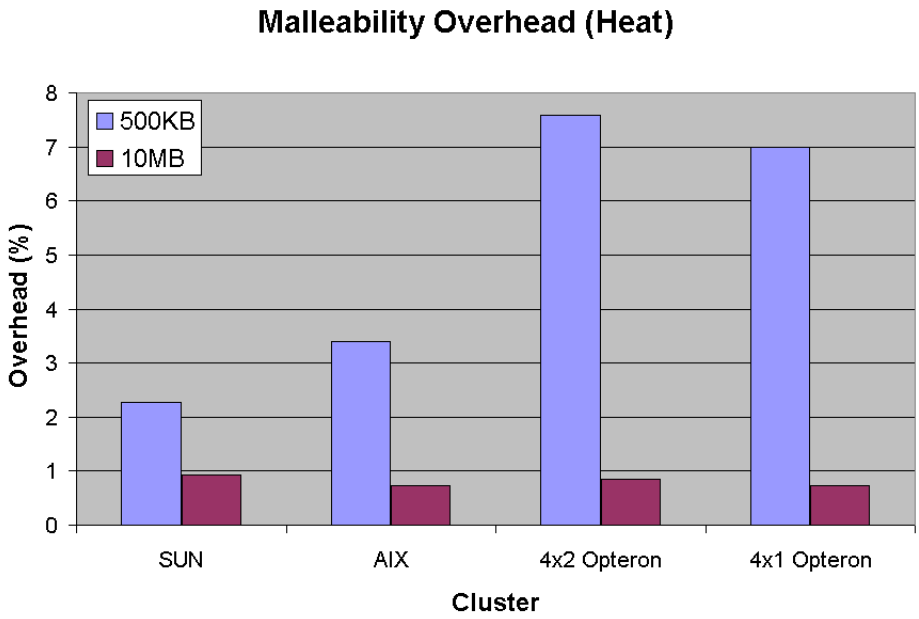

Figure 3.3: `MalleableActor` **overhead for the heat application on the Sun, AIX, 4x1 Opteron and 4x2 Opteron cluster environments, with one actor per processor for 8, 16, 32 and 48 malleable actors respectively. The heat application was run with 500KB and 10MB data, demonstrating the effect of communication and computation speed on overhead. For the Astronomy application and larger data sizes, overhead was negligible.**

3.3 Migration

Autonomous migration using S-RAS and ATS-RAS was evaluated using bechmarks with representative communication topologies. Each benchmark consists of actors communicating in different topologies, after an actor receives communication from its neighbors it performs computation before sending messages back to its neighbors. Three communicating benchmarks were tested, with actors arranged in sparse (communicating neighbors form a randomly generated graph), tree, and hypercube topologies. A fourth benchmark, an unconnected topology, was also used to simulate massively parallel applications. These four benchmarks were evaluated on the Solaris cluster and compared to a typical static round robin distribution. Throughput, or number of messages processed with respect to time, was measured to compare performance. Each message processed performed the same

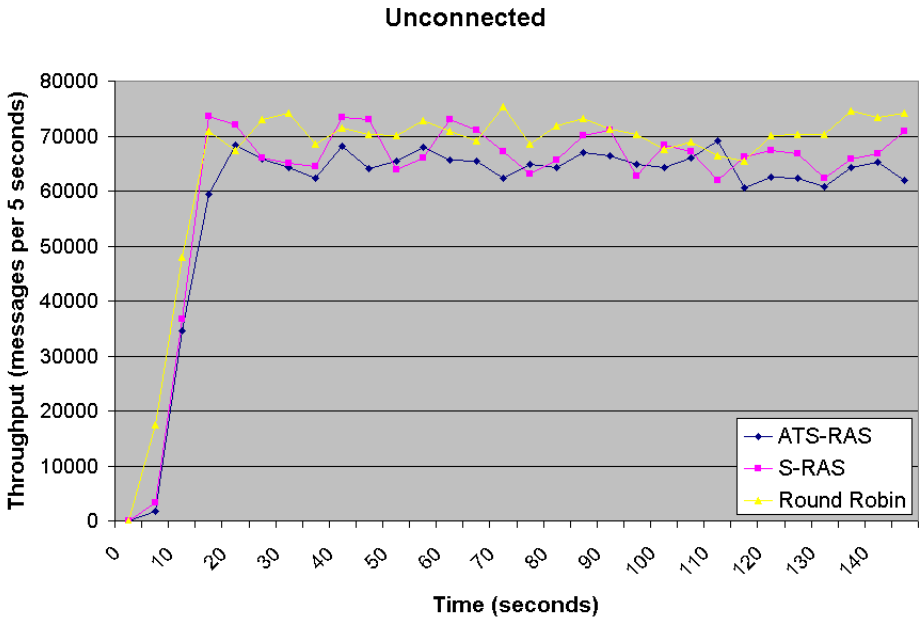

Figure 3.4: S-RAS and ATS-RAS compared to a round robin distribution of actors for the unconnected benchmark. S-RAS performs the best due to the low profiling overhead as the unconnected topology does not suffer performance degradations from communication overhead.

amount of computation, so improved throughput is improved application performance. Throughput was calculated every five seconds to minimize overhead from data collection.

Figure 3.4 shows the improvement in throughput as the application is reconfigured by the IOS middleware using S-RAS and ATS-RAS. For the S-RAS and ATS-RAS experiments, the application was loaded onto a single initial processor and the middleware was allowed to reconfigure it. Reconfiguration with both RAS and ATS-RAS was quick, distributing actors over nodes and reaching a stable result where no migration occured within 20 seconds (fluctuations in throughput were due to other executing system processes and communication time irregularity). S-RAS reconfigured the application to perform within 4% of the throughput of round robin (which is optimal for this benchmark) and ATS-RAS within 7%. This test shows that the additional profiling overhead of ATS-RAS is makes S-RAS a better choice

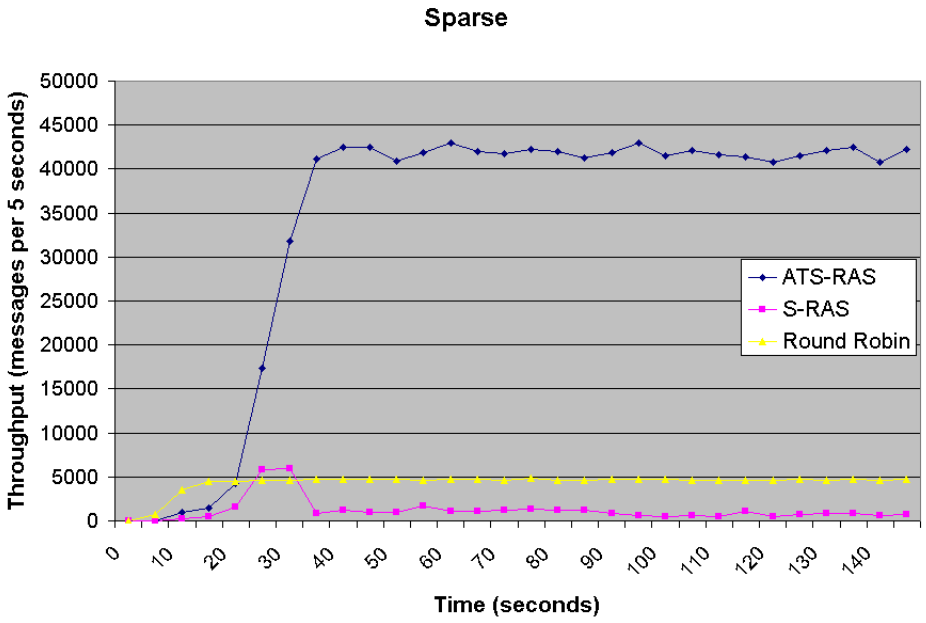

Figure 3.5: S-RAS and ATS-RAS compared to a round robin distribution of actors for the sparse benchmark. ATS-RAS performs the best due to profiling communication patterns and reconfiguring to minimize communication overhead.

for reconfiguration of massively parallel applications.

The sparse topology benchmark was tested in the same fashion as the unconnected benchmark. All actors were loaded onto the same initial processor and the IOS middleware reconfigured the application with S-RAS or ATS-RAS. Figure 3.5 gives a comparison of these results a round robin distribution of actors. Stabilization occured quickly, within 40 seconds. While S-RAS was unable to account for the communication topology of the application and actually degraded performance, ATS-RAS was able to provide a significant performance increase by colocating actors which communicated frequently.

The tree and hypercube benchmarks were also tested against round robin distribution (see Figures 3.6 and 3.7). As with the sparse benchmark, S-RAS was unable to account for the communication topology of these applications and decreased performance by reconfiguring the application based on computational load alone, which resulted in excessive communication because S-RAS did not stabilize.

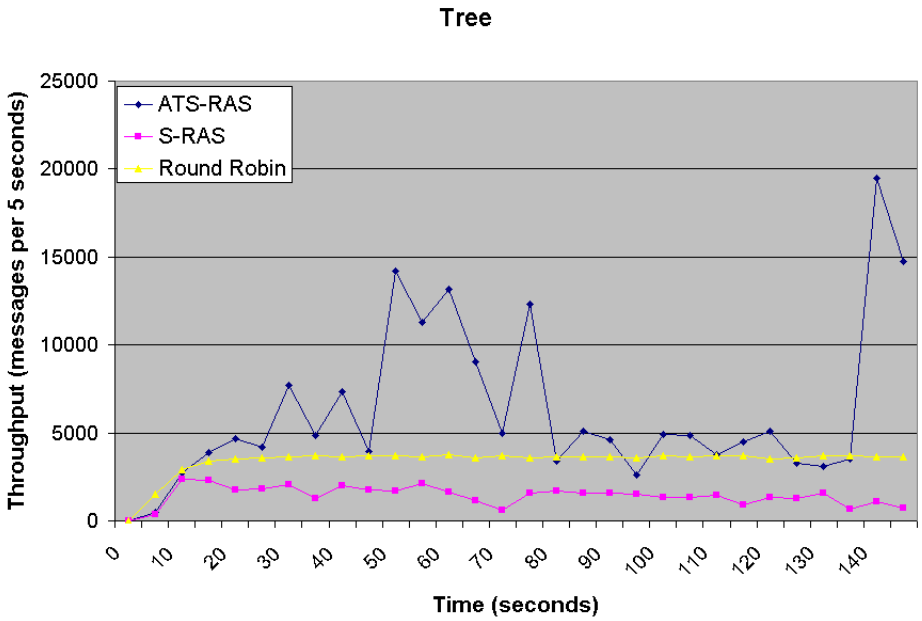

Figure 3.6: S-RAS and ATS-RAS compared to a round robin distribution of actors for the tree benchmark. ATS-RAS was able to improve performance, while S-RAS was not. Neither were able to obtain a stable configuration.

ATS-RAS was able to improve performance in both the hybercube and tree benchmarks compared to round robin, however it was unable to attain a stable result for the tree benchmark due to the complexity of the communication topology. Even though the result for the tree benchmark was unstable, it still performed better than round robin. These results demonstrate the ability of ATS-RAS to account for communication topology and determine configurations that perform well for the application without using distributed knowledge. S-RAS provides a good alternative for applications that are not communication bound, with minimal overhead from profiling and decision making.

FLC-RAS was tested against ATS-RAS on the AIX cluster using the heat application. Two different tests were run (see Figures 3.8 and 3.9). In the first, the heat application was loaded entirely onto a single processor, then using either ATS-RAS or FLC-RAS, IOS migrated the actors of the heat application to improve performance. The heat application was also run with the known optimal configura-

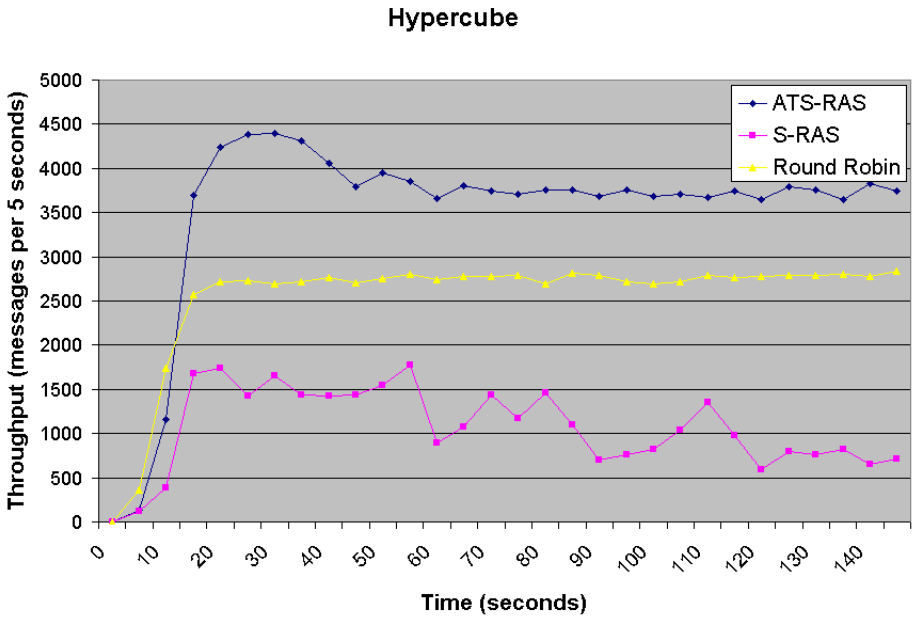

Figure 3.7: S-RAS and ATS-RAS compared to a round robin distribution of actors for the hypercube benchmark. ATS-RAS was able to improve performance while S-RAS was not. ATS-RAS was able to obtain a stable configuration while S-RAS was not.

tion. While ATS-RAS stabilized more quickly (performing no additional migration), after only 35 iterations, the configuration was 1.4 times slower than the optimal configuration. FLC-RAS stablized after 75 iterations, however the configuration was less than 1.02 times slower than the optimal configuration, performing well within the overhead measured in Section 3.2.

In the second test, the actors in the heat application were distributed randomly across all the processors in the AIX cluster. Again, IOS using ATS-RAS or FLC-RAS was used to autonomously reconfigure the application. These were both compared against the initial random configuration, and the known optimal configuration. Again, ATS-RAS stabilized more quickly, after 20 iterations, however it was 1.87 times slower than the optimal configuration. Compared to the initial random configuration which was 2.11 times slower than optimal, this was not a great improvement in performance. FLC-RAS stablized slower, after 45 iterations, however it was less than 1.01 times sloewr than optimal, again performing well within the

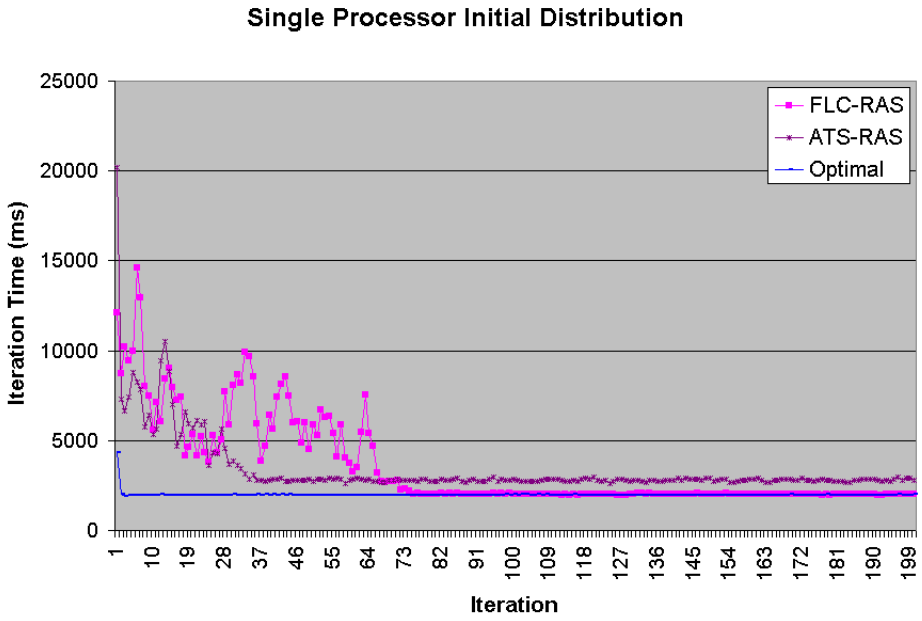

Figure 3.8: FLC-RAS and ATS-RAS compared to the optimal configuration. The application was loaded onto a single processor and then the middleware performed reconfiguration until a stable point was reached.

measured overhead.

In both tested cases, FLC-RAS was able to attain the optimal configuration with minimal overhead. Stablization was slower than ATS-RAS, because ATS-RAS tries to monotonically improve performance, so it reaches local optimums. FLC-RAS is able to reconfigure itself in a fashion that does not have local minima, always reaching the optimal configuration.

3.4 Malleability

Because malleability is performed hierarchically, reconfiguration time was evaluated for cluster and grid-level malleability. Malleability reconfiguration time is compared to application stop-restart. In all the evaluations, the reconfiguration results in the same number of actors and the same data distribution. Cluster-level reconfiguration was done with the applications loading 10MB data and grid-level reconfiguration with 100MB, 200MB and 300MB of data. It is important to note

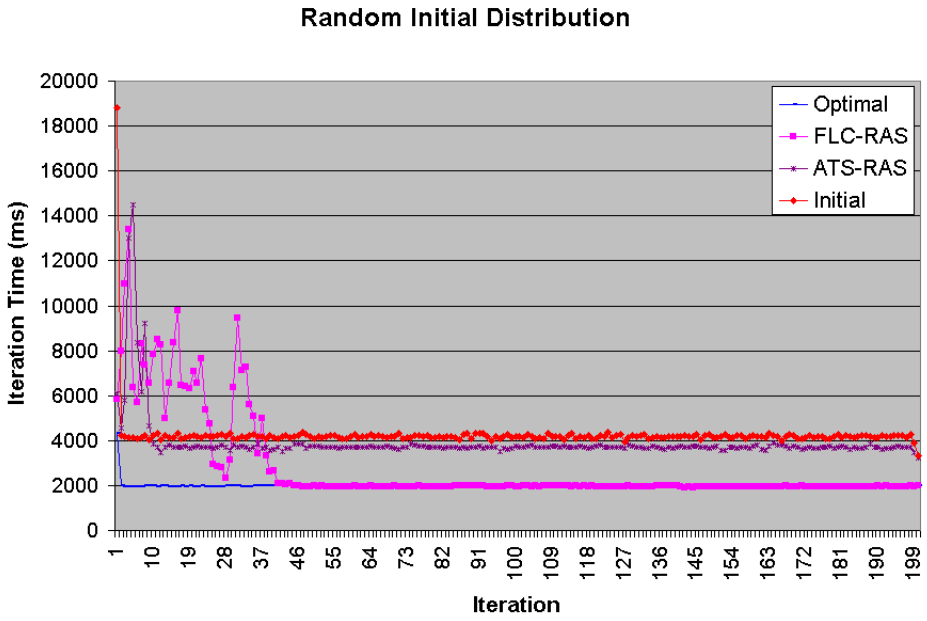

Figure 3.9: FLC-RAS and ATS-RAS compared to the optimal configuration. The application's actors were loaded onto random, then the middleware performed reconfiguration until a stable point was reached. The performance of the initial random configuration is also shown.

that stop-restart must be done over the entire application, so while malleability can reconfigure an individual node or cluster, stop-restart must stop and restart the entire application to accomplish the same reconfiguration. For an application distributed over a grid, processor-level and cluster-level reconfiguration is not possible for stop-restart without incurring grid-level reconfiguration times.

Figure 3.1 compares the reconfiguration time using stop-restart and malleability to the runtime and iteration times of the astronomy and heat applications. The values given show the range of reconfiguration times using the clusters in the described test environment. Typical runtime and iteration time show the optimal values for the application if the given environment is static with all resources available. Both applications were run for a typical amount of iterations. The astronomy application was run 200 iterations, and the heat application was run for 1000 iterations. Iteration time for the astronomy application can vary drastically based on

	Heat		Astronomy	
	300MB	**10MB**	**300MB**	**10MB**
Avg. Runtime	50-200m	1.5-50m	40+h	2-40h
Avg. Iteration	3-10s	0.1-3s	10+m	30s-10m
Grid-Level				
Stop-Restart	1.5-3m	n/a	30-50m	n/a
Malleability	2-15s	n/a	20-50s	n/a
Cluster-Level				
Stop-Restart	n/a	4-35s	n/a	15-30s
Malleability	.5-5s	.2-2s	5-20s	1-4s
Node-Level				
Malleability	0.01-0.2s	0.01-0.2s	0.05-0.3s	0.05-0.3s

Table 3.1: Reconfiguration time compared to application iteration time and runtime. The applications were run on the all three clusters described in Section 1.1 with 300MB data, and reconfiguration time was measured. The applications were also run on each cluster individually with 10MB data, and reconfiguration time was measured. The values given show typical iteration and runtime for these environments given various input parameters.

input parameters which effect the complexity of the computation, as shown in Figure 3.1. Both astronomy and heat were run on all three clusters with 300MB data, and grid-level, cluster-level and processor-level reconfiguration times were measured. Reconfiguration time was also measured for running the applications with 10MB data on the different clusters in the test environment. Differences in reconfiguration time reflect the different network and processing speeds in the test environment. The results show that malleability is significantly faster than stop-restart in all cases, and can also do finer grained reconfiguration very efficiently, in much less than the time of an iteration.

Grid-level reconfiguration time was measured by making clusters available and unavailable to the heat and astronomy applications (see Figures 3.11 and 3.10). The heat and astronomy applications were executed on the initial cluster with different amounts of data, then another cluster was added. The time to reconfigure using stop-restart and split was measured. Then a cluster was removed, and the time to reconfigure using stop-restart and merge was measured. The AIX cluster was added

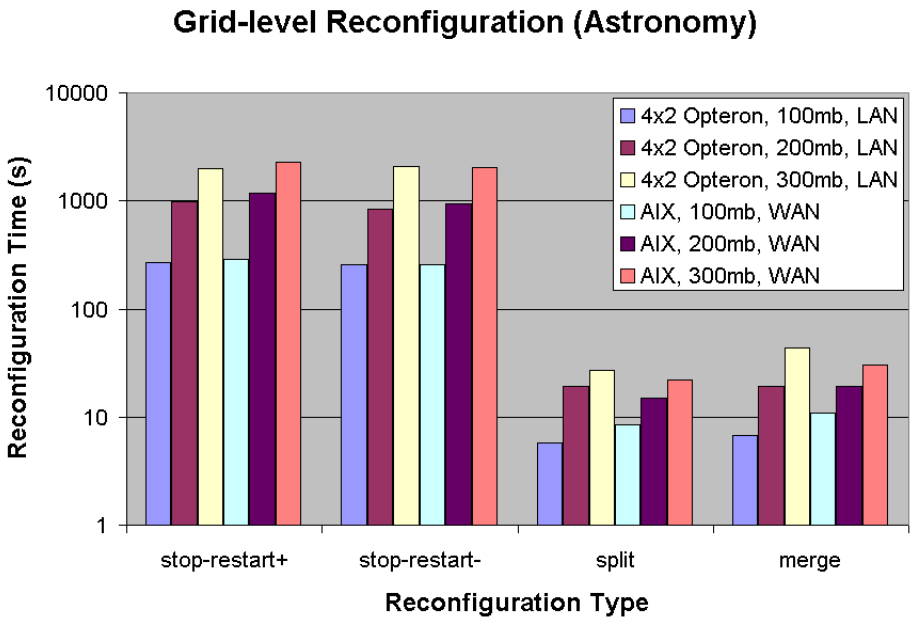

Figure 3.10: Performance of grid-level reconfiguration using the heat and astronomy applications with different data sizes over local and wide area networks. For reconfiguration over a WAN, the AIX cluster was added and removed from 4x1 opteron cluster, while the 4x2 opteron was added and removed for reconfiguration over a LAN. Stop-restart+ shows reconfiguration using stop restart when a cluster was made available, and stop-restart- measures the reconfiguration time for removing the cluster. Split shows the time to reconfigure the application using split when a cluster was added, and merge shows the time taken to when a cluster was removed.

and removed from the 4x1 Opteron cluster to measure reconfiguration time over a WAN, and the 4x2 Opteron cluster was added and removed for the other set to measure reconfiguration time over a LAN.

For the astronomy application, reconfiguration time was comparable over both the LAN and WAN, due to the fact that very little synchronization needed to be done by the reconfiguration algorithms and the middleware sent less data to the AIX cluster than the 4x2 Opteron cluster. Due to the differences in speed of the clusters, the middleware transferred 20% of the total data to the AIX cluster, and 40% of the total data to the 4x2 Opteron cluster. Malleability is shown to be more scalable

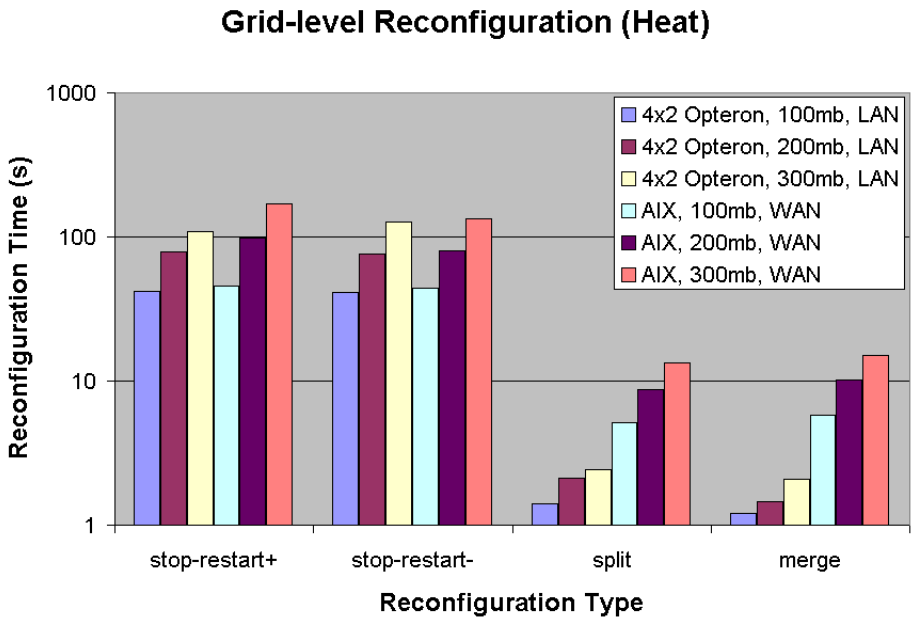

Figure 3.11: Performance of grid-level reconfiguration using the heat and astronomy applications with different data sizes over local and wide area networks. For reconfiguration over a WAN, the AIX cluster was added and removed from 4x1 opteron cluster, while the 4x2 opteron was added and removed for reconfiguration over a LAN. Stop-restart+ shows reconfiguration using stop restart when a cluster was made available, and stop-restart- measures the reconfiguration time for removing the cluster. Split shows the time to reconfigure the application using split when a cluster was added, and merge shows the time taken to when a cluster was removed.

than application stop-restart as data size increased. Over the LAN, splitting went from being 46x to 73x faster as data size increased from 100MB to 300MB, while merging improved 38x to 57x. Malleability is shown to be even more scalable over a WAN, as split improved reconfiguration time 34x to 103x and merge improved 23x to 67x as data size increased from 100MB to 300MB.

The heat application also gained significant reconfiguration time improvements using split and merge. Over a LAN, split improved from 30x to 45x faster, and merge improved from 34x to 61x faster than stop-restart as data size increased from 100MB to 300MB. Due to the additional synchronization required by the spatially-

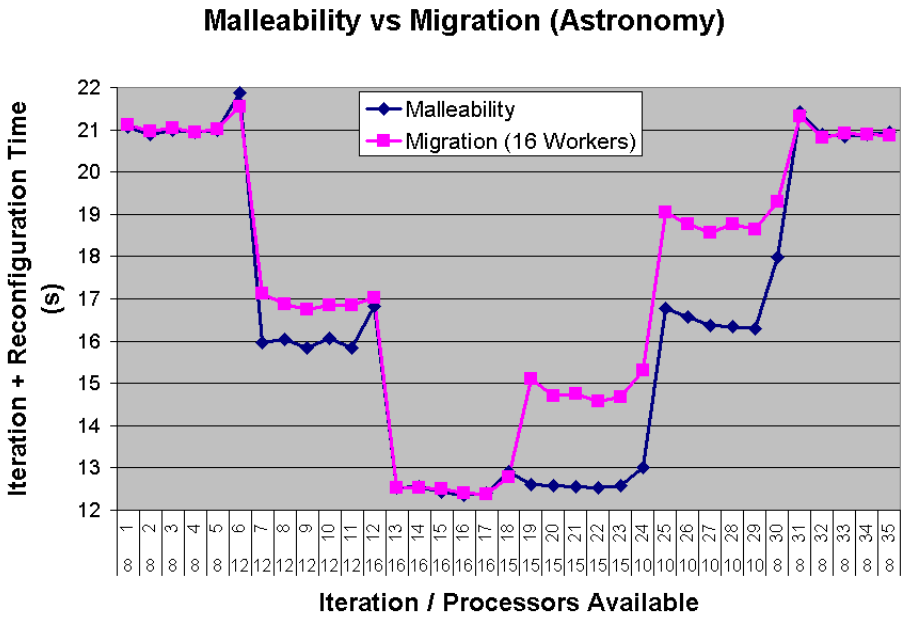

Figure 3.12: Autonomous reconfiguration using malleability and migration compared to autonomous reconfiguration only using migration. Every 6 iterations, the environment changed, from 8 to 12 to 16 to 15 to 10 to 8 processors. The last 8 processors removed were the initial 8 processors. A non-reconfigurable application would not scale beyond 8 processors, nor be able to move to the new processors when the initial ones were removed.

dependant data reconfiguration algorithms, split and merge did not improve as significantly over the WAN. Reconfiguration time using split increased 9x to 13x and increased 7x to 9x using merge, as data increased from 100MB to 300MB.

The astronomy application was able to gain more from split and merge because of the difficulty of accessing and reconfiguring its more complex data representation. The concurrency provided by split and merge allowed this to be divided over multiple processors, resulting in the much greater improvement in reconfiguration time. However, for both the heat and astronomy applications, these results show that using malleability is a highly scalable and efficient reconfiguration method.

3.5 Migration vs Malleability

The benefit of malleability compared to migration alone is demonstrated by executing the astronomy application on a dynamic environment. This test was run on the AIX cluster. In one trial only migration is used, while the other used malleability. Figure 3.12 shows the iteration times for the application as the environment changes dynamically. After 5 iterations, the environment changes. Typically, the application reconfigures itself in one or two iterations, and then the environment stays stable for another 5 iterations. For both tests, the dynamic environment changed from 8 to 12 to 16 to 15 to 10 and then back to 8 processors. The 8 processors removed were the initial 8 processors. Iteration times include the time spent on reconfiguration, resulting in slower performance for iterations when the application was reconfigured due to a change in the environment. Autonomous reconfiguration using malleability was able to find the most efficient granularity and data distribution, resulting in improved performance when the application was running on 12, 15 and 10 processors. Performance was the same for 8 and 16 processors for both migration and malleability, as migration was able to evenly distribute the workers in both environments. However, for the 12 processor configuration the malleable components were 6% faster, and for the 15 and 10 processor configurations, malleable components were 15% and 13% faster respectively. Overall, the astronomy application using autonomous malleability and migration was 5% faster than only using autonomous migration. Given the fact that for half of the experiment the environment allowed autonomous migration to evenly distribute workers, this increase in performance is considerable. For more dynamic environments with a less easily distributed initial granularity, malleability can provide even greater performance improvements.

The configurations available to a fixed number of migrating entities was compared to using malleability for the heat application. Figure 3.13 compares the performance obtained by using malleability to migration using 200 and 400 fixed equal size entities using 200MB data, malleability used a granularity of one entity per procesor. Different numbers of actors were used for migration to illustrate the tradeoff of more entities allowing better load distribution, but increasing overhead. Malleability resulted in a 40% to 80% performance increase over 200 migrating entities,

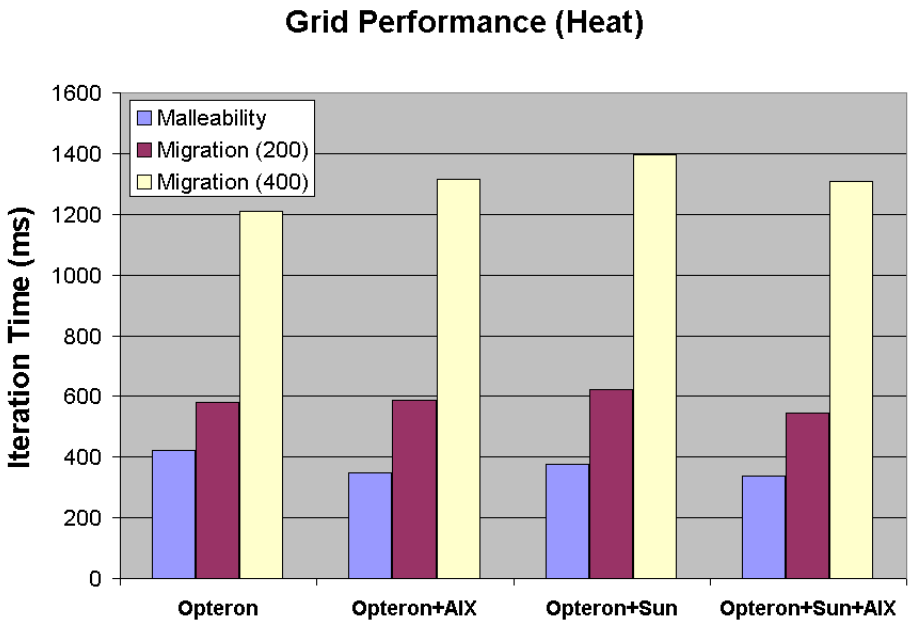

Figure 3.13: Performance of the heat application on dynamic environments using malleability and migration was tested. Initally, for migration, 200 or 400 equal sized actors were loaded on both Opteron clusters and the IOS middleware redistributed the actors. For malleability, 80 actors were loaded and the data was redistributed. The Opteron column shows the average iteration time for these configurations. Following this, clusters were added to the computation. Opteron+AIX, Opteron+Sun, and Opteron+Sun+AIX show the performance after IOS migrated actors or performed malleability, when the AIX, SUN or both the AIX and Sun clusters were added, respectively.

and a 180% to 400% increase over 400 migrating entities. Additionally, malleability was able to improve performance when the slower Sun and AIX clusters were added to the computation, while migration was not. This was because actors on those processors were assigned significantly less data than the even distribution used by migration strategies because of the additional overhead of WAN communication and their slower processing speed, and malleability only required one actor per processor resulting in less overhead. These results show that malleability can be much more effective than migration in performing load balancing for dynamic and highly

hetereogeneous networks.

CHAPTER 4
Related Work

A large body of research addresses the problem of enabling efficient utilization of large scale computing systems. Different middleware technologies provide scheduling for multiple applications or dynamic reconfiguration to balance the load of individual applications. Also, different grid architectures and toolkits are providing new platforms to enable development of grid computing systems. In many cases, the research in these areas overlaps, as certain schedulers also provide load balancing for their applications and different dynamic reconfiguration strategies can be applied over multiple running applications.

Abu-Ghazaleh and Lewis describe the concept of *self-organizing grids* [3], which eliminate the administrative bottlenecks limiting the expansion of current grids to truly massive scales. The administrative bottlenecks described that would require automation are:

- automatic discovery of the network structure,
- automatic monitoring of grid resources for scheduling and load balancing,
- automated application deployment and scheduling, and
- fault tolerance strategies to protect against resource failures.

In addition to the four described in that work, as a fifth we suggest security features to prevent against intentional or unintentional malicious use.

Most research for grid environments involves providing toolkits, middleware services or other tools which enable grid autonomy [47]. SALSA enables distribution on heterogeneous environments by leveraging the Java Virtual Machine. By providing reconfiguration options to application developers, either transparently via migration or with some user input via malleability, the SALSA programming language has been extended to facilitate the easy development of highly reconfigurable grid-scale applications. The work on IOS utilizes these reconfiguration tools, which

can be implemented in any language, to allow for autonomous reconfiguration in response to dynamic changes in the applications environment. This approach effectively seperates the concerns of application development from many of the challenges of large scale deployment.

4.1 Large-Scale Distributed Computing

The Globus Toolkit [22] provides a set of tools and discovery services for applications running on grids. These tools enable users to develop and deploy applications and obtain information about resource availability. While not providing an execution environment, Globus does allow grid users to schedule and deploy applications on a heterogeneous environment by providing the required information to determine what type of binaries to run and when to run them.

In contrast to Globus, which provides a set of tools to enable grid computing, Legion provides a virtual operating system which operates over a distributed set of host computing nodes [34, 23]. Legion supports multiple programming languages and uses a single unified object model and programming-level abstractions to hide the complexity of the underlying grid from the user and simplify the development process. Legion also provides a web-based user interface for job scheduling, monitoring and visualization.

Other approaches, such as BOINC [8] provide execution environments for geographically wide-spread volunteer computing hosts. Processors for these environments are generally slower and more heterogeneous, but still can be effectively harnessed for massively parallel and computationally intensive problems. In general, BOINC allows applications to utilize otherwise unused CPU cycles by running as a screen saver or by letting users volunteer a percentage of their CPU.

Parashar et al have used the concept of autonomic computing to build a grid computing framework [37] called *Automate*. Autonomic computing is based off of biological systems, in that it sees applications as having viability conditions, and enables them to autonomously respond to environmental changes to keep different parameters of the application within these viability conditions. Automate has its own programming model [36], where applications are made up of autonomic elements,

which separate function, control and allow for dynamic insertion and management of rules which come in the form of `if-then` expressions.

WaveGrid [49] provides for more efficient utilization of volunteer resources by providing a *self-organized timezone-aware overlay network*. WaveGrid profiles host resources and time zone, scheduling jobs on hosts in night time zones, which in volunteer computing are typically the most available. By scheduling jobs on hosts in night time zones with the most available resources, WaveGrid can increase turnaround time.

PlanetLab [14] also provides an execution environment for geographically distributed volunteer computing hosts, however these nodes are usually used as a testbed for development of networking and low overhead middleware services for these types of environments, and users typically have strict limits on the amount of resource usage they are allowed.

Overcoming firewalls, private networks, and other communication structures which can limit access to valid users and applications is also a great concern in grid computing. PlanetLab requires volunteered machines not to be behind a firewall, and BOINC communication is done over HTTP which is not firewalled. However, for many grid users firewalls are in place and this restriction needs to be overcome. Virtual Networks [43, 18] can overcome these limitations by abstracting away lower level communication or providing services which automatically route or tunnel communication.

SALSA's run-time system provides a distributed execution environment based on Java, which allows for execution on and communication between heterogeneous hosts. Additionally, the actor model [4], transparently provides asynchronous distributed communication through message passing and migration. Malleability allows for even greater levels of dynamic reconfiguration by allowing generic reshaping of applications. This work is novel in that it provides a unified actor-based programming language, middleware for autonomous reconfiguration, and distributed execution environment. The actor model facilitates concurrent and deadlock resistent applications, in addition to language- and library-based support for reconfiguration through transparent migration, and semi-transparent malleability.

4.2 Dynamic Reconfiguration

Dynamic reconfiguration in grid environments includes the GrADS project [11] which includes SRS [44], a library which allows stop and restart of applications in grid environments using the Globus Toolkit [22] based on dynamic performance evaluation.

Adaptive MPI (AMPI) [12, 24] 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 [39] 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.

Phoenix [41] is a programming model which allows for a dynamic environment by creating extra initial processes and using a virtual name space and process migration to load balance and scale applications. 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. This approach also does not scale to large scale environments, and choosing an incorrect number of initial processes will also limit scalability.

Active harmony [15] provides a different approach to performance tuning by using repeated offline short but representative runs of applications to drive maximum likelihood evaluation using the simplex algorithm to determine optimal configuration parameters. Additionally, it has been shown to be effective by comparing the results of this performance tuning to systematic sampling of possible configurations.

Another hierarchical optimization framework is proposed by Kandasamy et al [25], which performs load balancing and also enables applications to perform under QoS constraints such as energy usage. A mathematical model is used to forecast application resource consumption and performance and is used to reconfigure applications in response to dynamic resource usage and availability. However, tasks are assumed to be parallel and non-communicating.

Prophesy [27, 42] stores application data over multiple executions in a database for use in modeling and dynamically load balancing distributed applications. It distributedly uses global and local load balancing phases for structured adaptive mesh refinement, which are iterative applications similar to the heat application discussed in this thesis. The global load balancing phase redistributes data between groups of homogeneous processors, and the local load balancing phase redistributes data within these groups. A heuristic approach is used to determine the amount of data for each processor. However, this also does not allow for dynamically joining and leaving resources and there is only a single level of hierarchy, which may not be scalable.

Zoltan [20] is a framework for dynamically load balancing distributed applications. Users implement functions which enable Zoltan to perform data partitioning and load balancing. Different types of load balancing and data migration are provided with different levels of complexity, such as geometric bisection, space-filling curves and graph partitioning. Similar to split/merge, Zoltan requires applications to implement a specific API to enable data redistribution. In contrast to this work, which provides transparent process and data migration, and only requires implementing an API for data redistribution, Zoltan also does not allow for dynamically joining and leaving resources.

The Dynamic Resource Utilization Model (DRUM) [21], has been developed to address load balancing and mesh partitioning on heterogeneous and non-dedicated clusters. It provides a set of tools to discover and gather information about the execution environment, such as the connection topology and static resource availability. A system of distributed agents, similar to IOS, is used to determine this information. DRUM also allows for interaction with Zoltan. DRUM uses a tree structure to represent a distributed environment , where leaves are single-processor or shared-memory multiprocessing nodes, and non-leaf nodes represent routers or switches. Initially, to build this tree structure, a user generated XML file is required. Following this, the tree structure can be updated with profiled information which is then used to perform mesh refinement. DRUM does not allow for dynamic addition or removal of nodes.

Additionally, work has been done with the analysis of distributed load balancing techniques, e.g., [10]. The reconfiguration strategies presented in this thesis could benefit from such analysis by possibly determining ways to improve both the time to reach well performing configurations as well as the performance of those configurations.

4.3 Scheduling

The distinction between scheduling and dynamic reconfiguration is increasingly becoming blurred. Middleware which provides dynamic reconfiguration such as SRS [44] and GrADS [11] in a sense also schedules applications to grid resources, however such schedules are more dynamic in that the placement of applications can change. Research in application scheduling has expanded towards such dynamic schedules, allowing for replacement and rescheduling of applications for performance improvement [6, 9]. This section provides a brief overview of some of the research being done in dynamic scheduling.

A-Steal [6] dynamically schedules non-communicating parallel jobs across heterogeneous distributed environments. A-Steal uses an extension to random stealing to perform the scheduling of jobs across the distributed hosts. Where basic random stealing will have processors with no jobs stealing work from nearby processors with jobs, A-Steal first tries to steal unscheduled jobs before stealing jobs from nearby processors. A-Steal has also been shown to scale linearly with respect to the number of jobs, and to perform using at most 20% of the distributed systems cycles, even with adversarial job schedulers.

Beaumont et al [9] evaluates methods for scheduling non-communicating parallel jobs across heterogeneous distributed environments. Various approaches such as linear programming, first-come first-serve, coarse-grain bandwidth-centric, parallel bandwidth-centric and data-centric scheduling are evaluated through simulation.

CHAPTER 5
Discussion

This thesis has presented two methods for reconfiguring distributed applications, migration and malleability. Three different policies for autonomous migration were discussed and evaluated on different computational environments, as well as a hierarchical policy for autonomous malleability. Four representative benchmarks and two scientific applications were used to evaluate these methods and policies. Conclusions drawn from this evaluation are discussed in Section 5.1. This chapter ends with avenues for future work in Section 5.2.

5.1 Conclusions

This work has presented and analyzed the performance of three different reconfiguration strategies for autonomous migration: simple random actor stealing (S-RAS), actor topology sensitive random actor stealing (ATS-RAS) and fuzzy logic controlled random actor stealing (FLC-RAS). Results using four different benchmarks applications with different communication topologies – unconnected, sparse, tree and hypercube – demonstrate that an extremely simple reconfiguration strategy (S-RAS) is sufficient for reconfiguration of applications similar to the unconnected benchmark, such as massively parallel applications. These results also show that when an application's computational components communicate with each other, the additional overhead incurred by more complicated strategies is outweighed by the performance improvements those strategies (ATS-RAS and FLC-RAS) provide. These results also show that with some parameter tuning by an application developer, FLC-RAS can determine the optimal configuration of an application without centralized or global knowledge. ATS-RAS tends to improve application performance monotonically and can become stuck at local optima, however it converges to a stable configuration faster than FLC-RAS. Given this behavior, ATS-RAS may provide better overall performance than FLC-RAS if resource utilization and availability fluctuate frequently.

Malleability has been enabled through an API and built in directors which manage the malleability process. These directors are extensible and can be developed as part of a library for enabling malleability for different types of applications. With minimal programming effort, SALSA applications can be extended with autonomous malleability using these directors in conjunction with IOS, which dynamically changes data distribution and actor granularity to optimize performance. For iterative applications, malleability is shown to provide a significant advantage over migration alone, because it allows for optimal data distribution over any number of actors. Migration, on the other hand, is restricted to a fixed data distribution and number of actors, which can result in performance degrading data imbalances. Additionally, malleability is shown to provide large improvements in reconfiguration time, compared to application stop-restart.

5.2 Future Work

This work provides an initial step towards a fully fledged decentralized middleware for autonomous reconfiguration of applications running on large scale environments, however much work still needs to be done to reach that goal. Autonomous reconfiguration was tested on a limited number of benchmarks and applications, on relatively small, but representative environments. Using larger and globally distributed environments, as well as more applications running at larger scales, will allow for additional research into the scalability of IOS, and for testing new distributed reconfiguration protocols and strategies. Testing different virtual network arrangements of IOS agents along side reconfiguration strategies will aid in the development of an autonomous reconfiguration library, in which application developers can plug-in what reconfiguration strategies and virtual networks work best for their application. Additionally, it could be possible for middleware to autonomously choose which strategies to use, in effect reconfiguring itself.

The IOS middleware was only used to reconfigure single applications. For future work, the reconfiguration strategies presented, as well as newly developed strategies can be used to reconfigure multiple applications running using the IOS middleware at the same time. This will allow for advanced application schedul-

ing and reconfiguration and would be integral to using IOS on a large scale grid with multiple users. This would provide multiple applications the ability to share resources which could optimize resource usage and shorten scheduling time.

Different types of reconfiguration and autonomous reconfiguration methods, such as malleability, are still in development. By developing malleability strategies for more types of applications, it may be possible to develop language semantics and behavior that will allow for generic and transparent malleability. Providing a library of directors to provide malleability to applications in SALSA is also work in progress. Additionally, middleware support for fault tolerance will allow for IOS to enable environment-aware fault tolerance for applications.

The FLC-RAS strategy can provide autonomous reconfiguration via migration that will always converge to the optimal result, without requiring centralized or global knowledge. Unfortunately, this strategy requires parameter tuning by the application developer. Future work for development of the FLC-RAS strategy involves using investigating the stability of these parameters and developing fuzzy strategies which do not require user input, such as using learning methods to automate the parameter tuning process, which would make this strategy far more accessible.

Another area of future research involves using IOS to autonomically replicate application data and processing components for fault tolerance. Autonomic fault tolerance could dynamically change the amount of replication depending on the reliability of applications' run-time environments, which could improve performance by only replicating the parts of the applications executing on unreliable computing nodes.

IOS has also been tested for use in other programming models, such as MPI [32]. Developing malleability and migration for MPI and other programming models for use with IOS will allow for IOS to work with multiple applications developed in different programming languages for more widespread use and accessibility.

This thesis presents initial work towards developing IOS as a middleware for adaptation of applications running on large-scale environments. Results show that using middleware for autonomous reconfiguration can provide significant benefits

to applications with little or no additional work from the developer. This work opens many new avenues for research into large scale environments, distributed applications and middleware.

LITERATURE CITED

[1] The international virtual data grid laboratory. http://www.ivdgl.org.

[2] The large hadron collider computing project. http://www.cern.ch/lcg.

[3] Nael Abu-Ghazaleh and Michael J. Lewis. Short paper: Toward self organizing grids. *HPDC-15: The 15th IEEE International Symposium on High Performance Distributed Computing (Hot Topics Session)*, Paris, France, June 2006.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[5] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[6] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 19, Washington, DC, USA, 2006. IEEE Computer Society.

[7] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[8] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203. IEEE Computer Society, 2005.

[9] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2006*. IEEE Computer Society Press, 2006.

[10] Petra Berenbrink, Tom Friedetzky, and Zengjian Hu. A new analytical method for parallel, diffusion-type load balancing. In *International Parallel and Distributed PRocessing Symposium (IPDPS)*, April 2006.

[11] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for

high-level grid application development. *International Journal of High-Performance Computing Applications*, 15(4):327–344, 2002.

[12] Milind A. Bhandarkar, Laxmikant V. Kale;, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *Proceedings of the International Conference on Computational Science-Part II*, pages 108–117. Springer-Verlag, 2001.

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

[14] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[15] I-Hsin Chung and Jeffrey K. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. In *High Performance Distributed Computing (HPDC)*, pages 45–56. IEEE Computer Society Press, June 2006.

[16] Clip2.com. The gnutella protocol specification v0.4, 2000. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[17] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[18] Alexandre Denis, Olivier Aumage, Rutger Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.

[19] T. Desell, K. El Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, pages 1–10, January 2004.

[20] Karen Devine, Erik Boman, Robert Heapby, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engg.*, 4(2):90–97, 2002.

[21] Jamal Faik. *A Model for Resource-Aware Load Balancing on Heterogeneous and Non-Dedicated Clusters*. PhD thesis, Rensselaer Polytechnic Institute, Troy, 2005.

[22] I. Foster and C. Kesselman. The Globus Project: A Status Report. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 4–18. IEEE Computer Society, March 1998.

[23] A. S. Grimshaw, M. A. Humphrey, and A. Natrajan. A philosophical and technical comparison of legion and globus. *IBM J. Res. Dev.*, 48(2):233–254, 2004.

[24] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, pages 306–322, College Station, Texas, October 2003.

[25] Nagarajan Kandasamy, Sherif Abdelwahed, and Mohit Khandekar. A hierarchical optimization framework for autonomic performance management of distributed computing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 9, Washington, DC, USA, 2006. IEEE Computer Society.

[26] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, page 39, 1995.

[27] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. Dynamic load balancing of samr applications on distributed systems. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 979–993, New York, NY, USA, 2001. ACM Press.

[28] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. Dynamic load balancing of SAMR applications on distributed systems. *Scientific Programming*, 10(4):319–328, 2002.

[29] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[30] K. El Maghraoui, J. Flaherty, B. Szymanski, J. Teresco, and C. Varela. Adaptive computation over dynamic and heterogeneous networks. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, *Proc. of the Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM'2003)*, number 3019 in LNCS, pages 1083–1090, Czestochowa, Poland, September 2003.

[31] Kaoutar El Maghraoui, Travis Desell, Boleslaw K. Szymanski, James D. Teresco, and Carlos A. Varela. Towards a middleware framework for dynamically reconfigurable scientific computing. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, pages 275–301. Elsevier, 2005.

[32] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. The internet operating system: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 10(4):467–480, 2006.

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

[34] A. Natrajan, A. Nguyen-Tuong, M. Humphrey, and A. Grimshaw. The legion grid portal. *Grid Computing Environments Special Issue, Concurrency and Computation: Practice and Experience*, 14(13-14):1365–1394, 2002.

[35] Vijay Pande et al. Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2002. Peter Kollman Memorial Issue.

[36] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Cluster Computing*, 9(2):161–174, 2006.

[37] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pages 257–269, 2004.

[38] Jonathan Purnell, Malik Magdon-Ismail, and Heidi Newberg. A probabilistic approach to finding geometric objects in spatial datasets of the Milky Way. In *Proceedings of the 15th International Symposium on Methodoligies for Intelligent Systems (ISMIS 2005)*, pages 475–484, Saratoga Springs, NY, USA, May 2005. Springer.

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

[40] A. Szalay and J. Gray. The world-wide telescope. *Science*, 293:2037, 2001.

[41] Kenjiro Taura, Kenji Kaneda, and Toshio Endo. Phoenix: a Parallel Programming Model for Accommodating Dynamically Joininig/Leaving Resources. In *Proc. of PPoPP*, pages 216–229. ACM, 2003.

[42] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[43] Mauricio Tsugawa and Jose A. B. Fortes. A virtual network (vine) architecture for grid computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 10pp, April 2006.

[44] Sathish S. Vadhiyar and Jack Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[45] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, 2001. http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.

[46] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[47] Carlos A. Varela, Paolo Ciancarini, and Kenjiro Taura. Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. *Scientific Programming Journal*, 13(4):255–263, December 2005. Guest Editorial.

[48] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Comput. Syst.*, 15(5-6):757–768, October 1999.

[49] Dayi Zhou and Virginia Lo. Wavegrid: A scalable fast-turnaround heterogeneous peer-based desktop grid system. In *International Parallel and Distributed PRocessing Symposium (IPDPS)*, page 10pp, April 2006.

APPENDIX A
Malleable Heat Code

The following chapter provides an example of the heat application implemented in the SALSA programming language. It has been modified to implement the `MalleableActor` API, which enables autonomic malleability using the IOS middleware. Changes required to implement malleability are given in bold.

```
module heat;
import ios.salsa.language.MalleableActor;

behavior HeatWorker extends MalleableActor {
  int iterations;
  Data data;

  HeatFarmer farmer;
  HeatWorker left, right;

  public HeatWorker(int iterations, HeatWorker farmer) {
    this.iterations = iterations;
    this.farmer = farmer;
  }

  void connectRight(HeatWorker w) { right = w; }
  void connectLeft(HeatWorker w) { left = w; }

  void startWork() {
    if (right != null) right←leftRow(data.getRightRow());
    if (left != null) left←rightRow(data.getLeftRow());
  }
```

```
boolean leftSet = false, rightSet = false;
double[] queuedLeftRow, queuedRightRow;

void leftRow(double[] row) {
  if (leftSet == true) {
    queuedLeftRow = row;
  } else {
    data.setLeftRow(row);
    leftSet = true;
    if (rightSet || right == null) doWork();
  }
}

void rightRow(double[] row) {
  if (rightSet == true) {
    queuedRightRow = row;
  } else {
    data.setRightRow(row);
    rightSet = true;
    if (leftSet || left == null) doWork();
  }
}

void doWork() {
  if (!finished) {
    /* perform work here */
    ...
    if (right != null) right←leftRow(data.getRightRow());
    if (left != null) left←rightRow(data.getLeftRow());
    rightSet = false;
    leftSet = false;
```

```
    iterations--;
    if (iterations == 0) {
      finished = true;
      farmer←reportResults(data);
    }

    if (queuedRightRow != null) {
      data.setRightRow(queuedRightRow);
      rightSet = true;
      queuedRightRow = null;
    }
    if (queuedLeftRow != null) {
      data.setLeftRow(queuedLeftRow);
      leftSet = true;
      queuedLeftRow = null;
    }
    if ((leftSet || left == null) && (rightSet || right == null)) {
      doWork();
    }
  }
}
```

/* The following methods have been added to enable malleability. */
void redirectReference(String name, ActorReference reference) {
** if (name.equals("left") left = (HeatWorker)reference;**
** else right = (HeatWorker)reference;**
}

long getDataSize() { return data.size(); }
Object getData(long size, String options) {

```
  long rows = size / (data.size()/data.rows());
  if (options.equals("left")) return data.getLeftRows(rows);
  else return data.getRightRows(rows);
}

void receiveData(Object received, String options) {
  if (options.equals("left")) data = ((Data)received).append(data);
  else data = data.append((Data)received);
}

void handleMalleabilityMessage(Message message) {
  if (data.size() == 0) {
    /* this worker has been removed */
    if (message.getName().equals("leftRow")) {
      right←message;
    } else if (message.getName().equals("rightRow")) {
      left←message;
    }
  } else {
    /* data has been moved around, discard message and get */
    /* new left and right rows */
    if (message.getName().equals("leftRow")) {
      left←requestRightRow(self);
    } else if (message.getName().equals("rightRow")) {
      right←requestLeftRow(self);
    } else if (message.getName().equals("doWork")) {
      doWork();
    }
  }
}
```

```
    void requestRightRow(HeatWorker worker) {
      worker←leftRow(data.getRightRow());
    }
    void requestLeftRow(HeatWorker worker) {
      worker←rightRow(data.getLeftRow());
    }
}
```