Malleable Components for Scalable High Performance Computing

Travis Desell, Kaoutar El Maghraoui, Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
Email: [deselt, elmagk, cvarela]@cs.rpi.edu

***Abstract*— This paper describes a modular decentralized middleware framework for dynamic application reconfiguration in large scale heterogeneous environments. Component malleability is presented as a dynamic reconfiguration strategy. Dynamic component granularity enables improved load balancing by component migration, and most importantly, it enables applications to scale up to arbitrarily large numbers of processing nodes in a relatively transparent way. Preliminary experimental results show that without significant overhead, malleable components can improve applications performance and distributed resource utilization.**

I. Introduction

As high performance computing (HPC) environments scale to hundreds of parallel-processors and even millions of workstations, the demands on an application developer are becoming increasingly challenging and unmanageable. Most current HPC environments assume dedicated resources and a reservation based model for scheduling. Unfortunately in such a model, users can either overschedule resources, resulting in wasted resources and time, or underschedule resources which can lead to lost work. However, these issues can be overcome with a model that assumes dynamic and shared resources. In order for applications to be able to appropriately utilize the available resources in a dynamic shared HPC environment, new models for high performance computing are required, along with middleware and application level support. This work is in part motivated by the Rensselaer Grid, an institute-wide HPC environment, with dynamic and shared resources.

To appropriately utilize a dynamic and shared computing environment, application components must have the ability to be reconfigured at run-time. In a complex large-scale HPC environment, it is neither desirable nor plausible for an application developer to have to design and implement profiling strategies to examine the dynamic network and what other (possibly unknown) applications are competing for resources, and to design reconfiguration strategies based on this information.

Previous approaches for dynamic reconfiguration have involved migrating components (such as actors, agents or processes) to utilize unused cycles [1], [2] or using checkpointing to restart applications with a different set of resources to utilize newly available resources or to remove badly performing resources [3], [4]. Checkpointing and application restart can prove highly expensive, especially when resources change availability frequently, as in the case of web computing [5], [6] or shared clusters with multiple users.

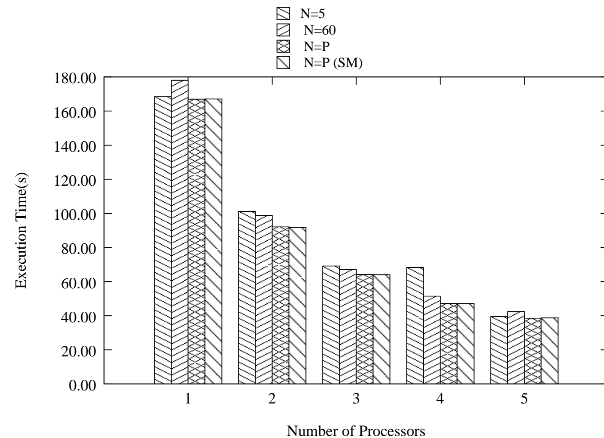

Fig. 1. Average Iteration time for an application running on one to five processors with different component configurations. N is the number of components and P is the number of processors. N=P (SM) uses malleable components, while N=P shows the optimal configuration (with no dynamic reconfiguration and middleware overhead). N=60 and N=5 show the best configuration possible using migration with a fixed number of components.

Component migration allows applications to be reconfigured with finer granularity (application restart can be seen as application migration). If required, components can be checkpointed easier and more concurrently than an entire application. Additionally, concurrent reconfiguration is less intrusive. However, component migration is limited by the application component granularity.

To illustrate this limitation, we use an iterative application (a distributed maximum likelihood computation used for astronomical model validation, for more details see Section V-A). This application is run on a dynamic cluster consisting of five processors. In order to use all the processors, at least one component per processor is required. When a processor becomes unavailable, the component on that processor can migrate to a different processor. With five components, regardless of how migration is done, there will be imbalance of work on the processors, so each iteration needs to wait for the pair of components running slower because they share a processor (in this example, 5 components running on 4 processors was

45% slower than 4 components running on 4 processors, with otherwise identical parameters). One alternative to fix this load imbalance is to increase the number of components to enable a more even distribution of components no matter how many processors are available (in this example, 60 components were used). Unfortunately, the increased amount of components causes the application to run slower, approximately 7.6% for this example. Additionally, this approach is not scalable, as the number of components required for this scheme is the least common multiple of different combinations of processor availability. In many cases, the availability of resources is unknown at the applications startup so an effective number of components cannot be determined. Figure 1 shows these two approaches compared to an ideal distribution of work, one component per processor. In this example, if a fixed number of components is used, averaged over all configurations, five components is 13.2% slower , and sixty components results is 7.6% slower.

To solve the problem of appropriately using resources in the face of a dynamic execution environment where the available resources may not be known, *malleable* components are introduced. Instead of having a static number of components, malleable components can *split*, creating more components, and *merge*, reducing the number of components, redistributing data based on these operations. With malleable components, the application's granularity (or number of components) can be changed dynamically. Applications define *how* components split and merge, while the middleware determines *when* based on resource availability information. As the dynamic environment of an application changes, in response, the granularity and data distribution of that application can be changed to most efficiently utilize its environment. For this example, using split and merge operations reconfigures the application to be able to operate within 0.2% of the speed of the optimal configuration, regardless of processor availability.

This work argues that malleable components will provide applications with the ability to more effectively scale and efficiently use resources in HPC environments, especially in the case of dynamic and/or shared resources. The responsibility of the application developer to implement split and merge can be minimized through an API in conjunction with generic split and merge algorithms. Split and merge are shown to be scalable and efficient operations. Middleware is capable of autonomously splitting, merging and migrating components, improving the performance of applications on dynamic HPC environments with minimal overhead (less than 2% on the applications tested).

This paper continues as follows. Section II discusses different possibilities for the implementation of malleable components in HPC environments, and the potential for autonomous malleability. An approach for developing HPC middleware for autonomous reconfiguration is presented in Section III. An implementation of malleable components and a middleware for autonomous reconfiguration is given in Section IV. Section V presents representative applications and how they were used to evaluate the split and merge operations and autonomous malleability. The paper concludes with a brief description of related work in Section VI and a discussion of the results and avenues for future work in Section VII.

II. Component Malleability

This section discusses the various considerations taken in evaluating malleable components. Due to geographical, administrative and scalability concerns, it is not possible to assume shared memory or synchronous communication in HPC environments. Thus, for the considerations of this discussion, components are assumed to encapsulate data and communicate asynchronously. Distributed memory and asynchronous communication alleviate some of the concerns in implementing split and merge. Distributed memory is an easier model to preserve data consistency, as opposed to replicated shared memory. Likewise, asynchronous communication eliminates the concerns of preserving invocation stacks used in synchronous communication. This section discusses different possibilities for component malleability, with respect to concurrency and complexity, and how malleability can be used as a reconfiguration tool.

A. Component Malleability Strategies

Splitting or merging will involve one or more components to participate in reconfiguring themselves, where the communication topology of those components will be modified for the addition or removal of new components, and the data will be redistributed between these components. This reconfiguration may need to be done atomically to preserve application semantics and data consistency.

The simplest possibility for split and merge, mimicking cell division in biological organisms, is to limit the semantics of a split operation to having one component split into two components, and two components merge into one component. The major benefit of this approach is that it is the easiest to implement. It is also a highly concurrent approach, as all components could conceivably split at the same time, doubling the granularity of the application. Merging is also concurrent in that each component could pair up with another and merge concurrently, reducing the granularity of the application by half.

Another possibility is to allow a single component to split into multiple components, and multiple components to merge into a single component. While slightly more difficult to implement than the previous approach, it encompasses the previous approach and allows the same level of concurrency. However, the use of this type of split and merge semantics seems rather limited, in that it is best suited for expanding a small number of components to a larger number to utilize more resources, or releasing a large number of resources. For smaller changes in resources it is possible for this and the previous approach to succumb to the same problems as migration. Mainly, these approaches are prone to data imbalances.

The most versatile approach would be to allow any number of components, N, to split or merge into any other number of components, M. This would solve problems of data

imbalances for minor changes in resources, and still allow for modification of granularity to handle major changes in resource availability. However, due to the number of components involved, this approach is not as concurrent as the previous approaches (although, it does not necessarily require parallelization). The most serious drawback is the fact that the implementation of such an approach could be very difficult due to its complexity. Additionally, this kind of split and merge operations could prove to be very expensive, similar to application start and restart.

For this work, we define the split operation as taking any number of components N and generating $N+1$ components. Merge takes N components and results in $N-1$ components. This provides a middle ground where the split and merge operations are not as expensive or complex as the N to M approach, and more appropriate for a dynamic grid environment, where large changes in resource availability would not be overly common (which is more suited for a one to N and N to one approach).

B. Malleability as a Reconfiguration Tool

Rather than considering malleability as an alternative to migration, it can be used as an additional tool in the repertoire of an autonomous reconfiguration middleware. Malleability can modify application granularity allowing for more appropriate reconfiguration via migration. Splitting and merging can be used on their own to allow for greater parallelization of the application to more appropriately utilize resources in the case of computing nodes with multiple processors. Applications with malleable components will be able to scale as far as data limitations and resource availability allow. With a well defined interface, a middleware can autonomously reconfigure the granularity of the application through splitting and merging of components to improve application performance.

III. Modular Middleware for Autonomous Reconfiguration

The challenges of a HPC environment require a separation of concerns, alleviating application developers from many of the difficulties involved. An application developer should not need to determine at runtime what other applications are running in a shared environment, nor their resource usage. It is more practical to have a middleware layer to gather profiling information about applications and the HPC environment.

A generic middleware allows a wider adoption of different classes of applications as well as various execution environments. The middleware can perform dynamic reconfiguration based on profiled information of applications and environments with significant benefit [7]. Figure 2 represents the separation advocated in this paper. At the application or language level, a developer specifies *how* reconfiguration should be done. Profiling, through an API or implemented at the language level, provides the middleware with information about application status and resource usage. Having profiling and reconfiguration tools conform to a specific API allows the middleware to receive profiling information and reconfigure

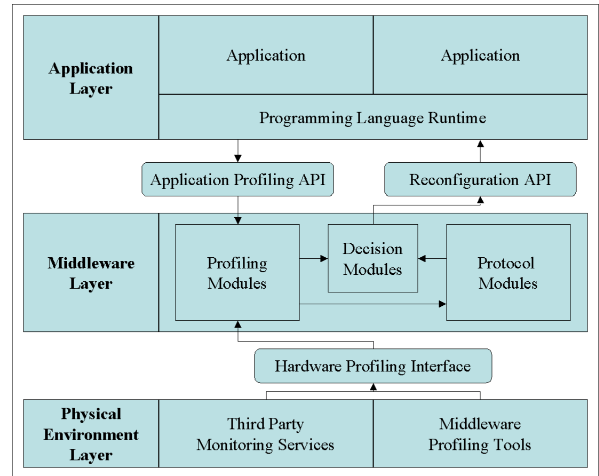

Fig. 2. Approach for autonomous middleware in HPC environments. The middleware receives profiling information about the applications and environment through profiling interfaces. Profiling information is distributed in the middleware using various protocols. Decisions are taken based on local and remote profiling information, both of the application and hardware, and the application is reconfigured through a reconfiguration interface.

applications in an application and programming language independent manner. Likewise, middleware can profile resource availability and interact with third party clients, such as the Network Weather Service (NWS) [8] or the Globus Meta Discovery Service (MDS) [9], allowing more informed reconfiguration decisions based on both the environment and executing applications.

To achieve scalability and to accommodate the dynamicity of HPC environments, middleware should be decentralized and allow for the dynamic addition and removal of resources. Additionally, the middleware should not be restricted to a single type of profiling, coordination or method of reconfiguration. Different decision strategies will require different types of profiled information. The communication topologies used by the decentralized middleware also play a role in the performance of the middleware, as certain topologies are more suited to specific HPC environments [2] (e.g., environments that are more homogeneous, like clusters, or more heterogeneous, like the Internet). As such, a modular approach with different pluggable decision, communication and profiling modules will allow a HPC middleware to most appropriately meet the requirements of its applications and environment.

IV. Implementation

To implement component malleability, previous work using the SALSA programming language [10] and the Internet Operating System (IOS) [2] is leveraged. SALSA is a distributed programming language and IOS is a the decentralized middleware with plug-in decision, profiling and communication modules. Sections IV-A.1 and IV-A.2 provide background information about the SALSA programming language and IOS middleware, respectively. Language extensions to SALSA for malleability are discussed in Section IV-B.1 and their use in

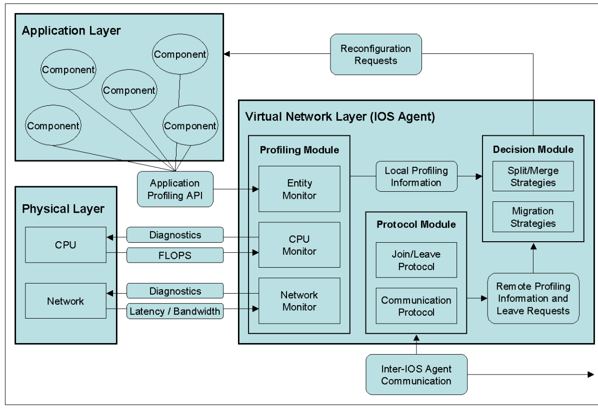

Fig. 3. IOS Agents consist of three modules: a decision module, profiling module and protocol module. The profiling module gathers information about the components executing locally, as well as the underlying hardware. The protocol module gathers information from other agents. The decision module takes local and remote information and uses it to decide how the application should be reconfigured.

generic split and merge algorithms is presented in Section IV-B.2. Finally, the strategy used for deciding when to split and merge is described in Section IV-B.3.

A. Software Framework

1) The SALSA Programming Language: SALSA applications consist of *actors*. The Actor [11] model of computation is based around the concept of encapsulating state and process into a single entity. Each actor has a unique name, which can be used as a reference by other 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 [12] and facilitates mobility [13].

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. SALSA actors can migrate transparently without any modification of code, enabling easy autonomous reconfiguration through migration.

2) The IOS Middleware: The Internet Operating System (IOS) [7] is a modular middleware for autonomous reconfiguration of distributed applications. IOS shifts the concerns of HPC environment profiling and reconfiguration decisions from the applications to the middleware, allowing application components to be reconfigured autonomously at run-time in a transparent manner. IOS is generic and can therefore be used with different applications and programming paradigms. An application developer is only required to 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 algorithms for reconfiguration. Figure 3 shows the architecture of an IOS agent. An IOS agent is present on every node in the HPC environment. Each agent is modular, consisting of three plug-in modules:

- A **Profiling Module** that gathers information about the components' communication topology and resource utilization, the physical network topology, and the resources available locally.
- 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.
- A **Decision Module** which determines when to perform reconfiguration, and how reconfiguration can be done. The decision module interacts with applications through asynchronous message passing. For this work, the decision module autonomously sends *migrate*, *split* and *merge* messages to the application based on its decision making strategy.

The modules interact with each other and the 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. Section IV-B.3 describes the decision modules used to evaluate component malleability.

B. Extensions for Malleability

1) Language Extensions: Split and merge operations require data redistribution and modification of the application's communication topology. These features are not required by component migration. Migration and profiling have been supported at the SALSA language-level and are transparent to the programmer. However, split and merge features require collaboration from programmers: e.g., a SALSA developer needs to extend the `MalleableActor` class shown in Table I and implement the required abstract methods. Optional abstract methods allow for automated data redistribution and reference redirection. Figure 4 shows the use of this API in the massively distributed Twin Primes application written in SALSA.

2) Split and Merge Algorithms: The split and merge algorithms performed by malleable actors proceed as follows. The split or merge of an actor is initiated when the middleware sends a *split(Reference[] dataSources)* or *merge(Reference[] dataSources)* message to an actor. The *dataSources* are all other actors that will collaborate in the split or merge.

First, the algorithm gets a lock on all data sources and owners of references needing modification during the split or merge. Locks are obtained if the actors are not participating in another split or merge operation, and the *canSplitOrMerge()*

TABLE I

MALLEABLEACTOR CLASS

Return Type	Method Name	Parameters
void	registerAsDataSource	(boolean requiresLock)
void	removeFromDataSources	()
void	registerSplitReference	(String fieldName, String updateType, Reference owner, boolean requiresLock)
void	removeSplitReference	(String fieldName, Reference owner)
void	registerMergeReference	(String fieldName, String updateType, Reference owner, boolean requiresLock)
void	removeMergeReference	(String fieldName, Reference owner)
abstract	boolean canSplitOrMerge	()
abstract	Reference createNewActor	()
abstract	double getDataSize	()
abstract	Object getSplitData	(Reference[] dataSources, ...)
abstract	void receiveSplitData	(Object[] data)
abstract	Object[] getMergeData	(Reference[] dataTargets, ...)
abstract	void receiveMergeData	(Object data)
abstract	void updateReferenceOnSplit	(String referenceName, String updateType, Reference newActor)
abstract void	updateReferenceOnMerge	(String referenceName, String updateType, Reference mergedActor)
abstract void	handleMergeMessage	(Message message)

```
behavior TPFarmer {
   NumberRangeGenerator nrg;
   void act(String[] args) {
      int numberWorkers = Integer.parseInt(args[0]);
      TPWorkers[] workers = new TPWorker[numberWorkers];
      nrg = new NumberRangeGenerator(args[1], args[2]);
      for (int i = 0; i < numberWorkers; i++)
         workers[i] = new TPWorker(this);
   }
   void requestWork(TPWorker source) {
      if (nrg.hasMoreSegments())
         source<-findPrimes(nrg.getNextSegment());
   }
   void receivePrimes(TPWorker source, DataSegment primes) {
      primes.saveToDisk();
   }
}
behavior TPWorker extends MalleableActor {
   TPFarmer farmer;
   TPWorker(TPFarmer farmer) {
      this.farmer = farmer;
      farmer<-requestWork(this);
   }
   void findPrimes(DataSegment primes) {
      primes.findPrimes();
      farmer<-receivePrimes(primes) @ farmer<-requestWork(this);
   }
   boolean canSplitOrMerge() { return true; }
   Reference createNewActor() { return new TPWorker(farmer); }
   void handleMergeMessage(Message message) {
      if (message.getName().equals("findPrimes")) this.process(message);
   }
}
```

Fig. 4. Example of the Twin Primes application using malleable components. Workers repeatedly request intervals of integers, find the primes in those intervals and return the results to the farmer. Changes for malleability are in bold.

method returns true. Locking may be required to prevent data inconsistencies, as it allows the split or merge operation to be performed as an atomic action. After a participant becomes locked, it will only process messages related to split or merge operations. All other messages are enqueued to be processed when the split or merge completes. Additionally, in the case of a merge, the actor that is to be removed by the merge operation will process the messages remaining in this queue with the *handleMergeMessage(Message message)* method, allowing the application to redirect these messages to appropriate actors, after which the merged actor will be garbage collected.

When all participants are locked, in the case of a split, the actor that initiated the split will call the *createNewActor()* method to create a new actor. Following this, all the data sources and the source actor will receive a *getSplitData(Reference[] dataSources, ...)* message, where the *dataSources* argument are all the other data sources for the split (including the source actor). This method should return the data which is then received by the newly created actor with the *receiveSplitData(Object data)* message. In the case of a merge, the actor that initiated the merge will divide its data with the *getMergeData(Reference[] dataTargets, ...)* and send the slices of the data to the corresponding *dataTargets*, who receive this with the *receiveMergeData(Object data)* method. Both the *getSplitData* and *getMergeData* methods have optional arguments by which the middleware can pass additional information, such as resource availability at the locations of target actors, which can allow for more asymmetrical splitting and merging for heterogeneous systems. This data is optional, and dependant on the decision module being used.

When the *receiveSplitData* or *receiveMergeData* message completes being processed, the references specified by the *registerSplitReference* or *registerMergeReference* methods can be updated. The source actor will send *updateReferenceOnSplit(String fieldName, String updateType)* or *updateReferenceOnMerge(String fieldName, String updateType)* messages to all owners of references to be modified, with the name of the reference to be modified (*fieldName*), and a specifier of how to update that reference (*updateType*). After all the references have been updated, the source actor and all participants can release their locks, and continue the computation.

3) Malleability Decision Module: In a dynamic environment, when resources become available or unavailable, modifying the application granularity will enable a more accurate mapping of the application components to the available resources. This new mapping can be achieved through migration of components. For different types of applications, different granularities can result in faster execution due to process and thread scheduling. For this work, a decision module is used that changes application granularity when resource availability changes (e.g., a new processor becomes available, or an old

processor gets removed), attempting to keep a granularity that optimizes usage of processing availability on each processor.

V. Results

This section describes the tests used to evaluate split and merge. First, in Section V-A, two representative applications for HPC are presented. Following this, Section V-B evaluates split and merge using these applications with respect to scalability, performance and overhead.

A. Sample Applications

Two sample SALSA applications have been modified to utilize the API described in Section IV-B. Both applications have an iterative nature. 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 [14] based on data derived from the SLOAN digital sky survey [15]. 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 a tightly coupled iterative application. Iterative applications are particularly difficult to reconfigure, due to the fact that the application only proceeds as fast as its slowest component. As such, the results in Section V-B show the benefits for more difficult cases of reconfiguration. For massively parallel and non-iterative applications, even greater performance benefits are possible, with less overhead.

We define two types of data for the purpose of this work: *location-dependent data* that is dependent on the behavior of its containing component and *location-independent data* that can be distributed irrespective of component behavior. For example, the astronomy code performs the summation of an integral over all the stars in it's 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 location dependant data because a worker modifies its data based on the data received from its adjacent neighbors. In this case, if another worker is added, the data needs to be distributed such that each worker has a slice of the data with the correct neighbors.

1) Astronomy: The astronomy code follows the typical farmer/worker style of computation. For each iteration, the farmer generates a model and requests the accuracy of this model from the workers, which calculate the accuracy based on a set of star positions. The farmer then combines these results, modifies the model and repeats. Each iteration involves multiple integrals on large sets of stars (in the thousands or millions), resulting in a low communication to computation ratio, allowing for massive distribution.

Because the astronomy application has location-independent data, the split process can be nearly entirely automated by the given API. For split and merge operations, participating data sources do not need to be locked. In the case of a split, all participating workers provide a set of stars to the newly created worker. For a merge, the merged actor splits its data up and sends each slice to a participating worker. As the data is location-independent, redistributed data slices can be of arbitrary size, and sent to any other worker in the computation.

2) Heat: The heat application's components communicate as a doubly linked list of workers. Workers wait for incoming messages from both their neighbors, use this data to perform the computation and modify their own data, then send the results to back to the neighbors. The communication to computation ratio is significantly higher than the astronomy code which makes the application more difficult to distribute, and the data is location-dependant making the behavior of split and merge more complicated.

To redistribute data in the heat application, the data must be redistributed in a fashion that does not violate 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. For each iteration, a worker will receive the rightmost column from its left neighbor, and the leftmost column from its right neighbor. It then uses these values to recalculate the temperature in its own columns and send its leftmost column to its left neighbor, and its rightmost column to its right neighbor. Redistributing data involves shifting columns left and right between participants, to make room for a new actor or to accept data from the merged actor. For this application, while data can only be shifted to the left or right neighbors of a worker, a group of adjacent workers can coordinate to communally redistribute work.

B. Evaluating Malleability

To evaluate autonomous malleability, the overhead of using autonomous middleware and the `MalleableActor` implementation are shown with the heat and astronomy applications in Section V-B.1. In Section V-B.2, the performance of the split and merge operations are tested by comparing them to similar reconfiguration using migration. The scalability of split and merge operations is also tested by showing the performance compared to migration for various numbers of participants. Lastly, in Section V-B.3, the applications were run on a dynamic environment, using component migration with and without component malleability, to demonstrate the benefit of autonomously malleable components.

1) Profiling Overhead: To evaluate the overhead of autonomous malleability, the heat and astronomy applications were run with and without middleware and profiling services. The applications were run on the same environments with the same configurations, however autonomous reconfiguration by the middleware was disabled. Figure 5 shows the overhead of the middleware and `MalleableActors` with the heat application. The average overhead over all tests for the application was 1% (i.e., the application was only 1

2) Malleability Performance: To determine the usability and efficiency of the split and merge operations, they were compared to migration. In Figure 6, the x-axis is the initial number of processors for the application. For the test, an

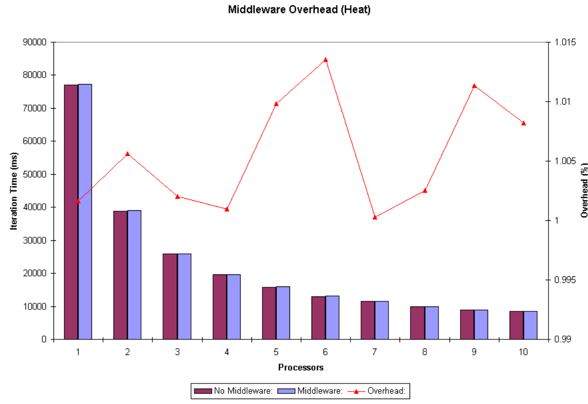

Fig. 5. Overhead of using autonomous middleware and the malleable actor interface with the heat application. The application was tested with the same configurations with different amounts of parallelization, with and without the middleware and malleable actor interface.

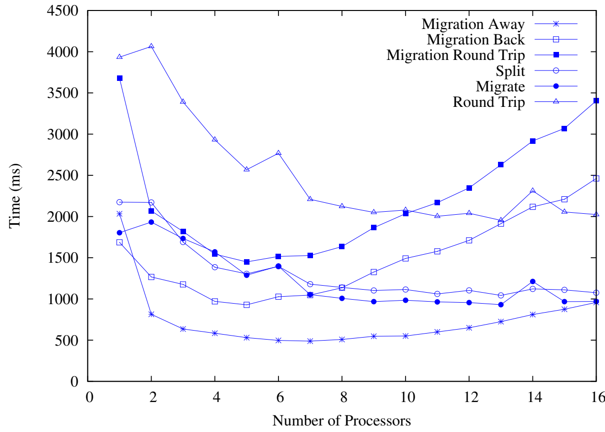

Fig. 6. Migration compared to split and merge reconfiguration times for various numbers of participants using the astronomy application.

additional processor was added and removed (requiring a split and merge, or migrations). To ensure equal data was being transferred, enough initial actors were created for the tests using migration. For example, with 5 initial processors, 30 actors were used, so that data transferred to and from the 6th processor was the same for both split, merge and migration. In this case, 6 actors are migrated to and from the new processor, while with split and merge, 6 initial actors participate in the split and 7 participate in the merge. For all tests, the amount of data transferred was the same. This test also shows the scalability of split and merge, as the number of participants increases. The times to migrate away from the initial processors (migration out) and back to the initial configuration (migration in), as well as the time for the split and merge operations. Total reconfiguration time for migration and split and merge are also shown. Reconfiguration time was averaged over 100 reconfigurations.

Figure 6 shows that while the split and merge operations are more complex than migration, in the worst case being

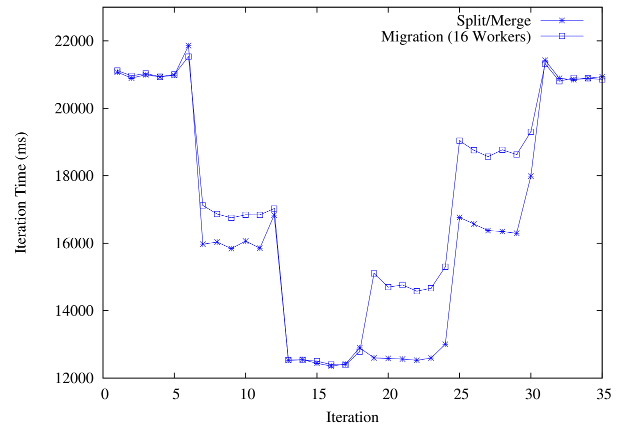

Fig. 7. Autonomous reconfiguration using malleability and migration compared to autonomous reconfiguration only using migration. Every 6 iterations, the environment changed, from 8 to 12 to 16 to 15 to 10 to 8 processors. The last 8 processors removed were the initial 8 processors. While split and merge are more expensive reconfiguration operations, the malleable components outperformed solely migratable components by 5%. 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.

twice as slow as similar migration, it is more scalable for similar reconfiguration scenarios. With reconfiguration from more than 10 initial processors, split and merge performed faster than similar migration.

3) Malleability on a Dynamic Environment: The benefit of malleability is demonstrated by executing the astronomy application on a dynamic environment using autonomous reconfiguration. In one case, only migration is used, but in the other split and merge, in addition to migration, was used. Figure 7 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. 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 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 environments were easy for autonomous migration to 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.

VI. Related Work

There is a wide range of literature for dynamic reconfiguration on HPC environments. As far as the authors know, this work is novel in that it is the first presentation of a generic framework for autonomous reconfiguration using dynamic component granularity. Selected work for dynamic reconfiguration in HPC environments includes the GrADS project [3], [4], a middleware which allows stop and restart of applications in grid environments using the Globus Toolkit [16] based on dynamic performance evaluation. Phoenix [17] 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. The selected work is far from comprehensive. For a more in depth discussion of middleware and reconfiguration for HPC environments, the reader is referred to [18].

VII. Discussion and Future Work

In this paper, malleable components are proposed as a reconfiguration strategy for autonomous (middleware-driven) application adaptation to dynamic and shared HPC environments. Language level extensions were provided for the SALSA programming language, to allow application developers to easily describe the parts of split and merge operations which could not be done transparently. These extensions are used by generic algorithms to split and merge the components. An approach for distributing data asymmetrically in a split and merge was given to improve efficiency and performance.

The IOS middleware was modified to use component malleability in addition to component migration for autonomous application reconfiguration. On sample applications, overhead of using this middleware was shown to be under 1.5%. Splitting and merging components was shown to be a scalable operation. In the worst case, splitting and merging components was twice as slow as equivalent migration, however in some cases it was shown to be faster. Autonomous reconfiguration using only migration was compared to autonomous reconfiguration using migration and malleability. On a dynamic environment, malleability was shown to provide a 5% improvement in performance, with potential for larger performance gains. The sample applications used in this evaluation were both iterative, and as such are particularly challenging applications for autonomous reconfiguration. These were chosen as opposed to massively parallel applications which could easily gain improved performance, as motivation for usefulness of this type of reconfiguration.

This work is a motivation for the use of component malleability, and it only briefly describes possibilities for autonomous malleability. There are also many different ways migration could be used in conjunction with malleability. Future work involves providing increased middleware support for autonomous malleability and language level support for other programming paradigms. MPI has already been extended to allow autonomous migration using IOS [2] and extensions to allow malleability in MPI applications are in progress. Determining the appropriate granularity for an application based on dynamically profiled information is still an open question. Using IOS to develop different decision modules will help gain insights into how to appropriately use migration, malleability and other types of application reconfiguration.

Acknowledgements

We would like to acknowledge the National Science Foundation (NSF CAREER Award No. CNS-0448407) and IBM (SUR Awards 2003 and 2004) for partial support for this research.

References

[1] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988, pp. 104–111.

[2] K. E. Maghraoui, T. Desell, B. K. Szymanski, J. D. Teresco, and C. A. Varela, "Towards a middleware framework for dynamically reconfigurable scientific computing," in *Grid Computing and New Frontiers of High Performance Processing*, L. Grandinetti, Ed. Elsevier, 2005.

[3] 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*, vol. 15, no. 4, pp. 327–344, 2002.

[4] S. S. Vadhiyar and J. Dongarra, "Srs: A framework for developing malleable and migratable parallel applications for distributed systems." *Parallel Processing Letters*, vol. 13, no. 2, pp. 291–312, 2003.

[5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[6] V. Pande *et al.*, "Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing," *Biopolymers*, 2002, peter Kollman Memorial Issue.

[7] T. Desell, K. E. 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*, January 2004, pp. 1–10.

[8] R. Wolski, N. T. Spring, and J. Hayes, "The Network Weather Service: A distributed resource performance forecasting service for metacomputing," *Future Generation Comput. Syst.*, vol. 15, no. 5-6, pp. 757–768, October 1999.

[9] 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.

[10] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, vol. 36, no. 12, pp. 20–34, Dec. 2001, http://www.cs.rpi.edu/˜cvarela/oopsla2001.pdf.

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

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

[13] G. Agha and N. Jamali, "Concurrent programming for distributed artificial intelligence," in *Multiagent Systems: A Modern Approach to DAI.*, G. Weiss, Ed. MIT Press, 1999, ch. 12.

[14] J. Purnell, M. Magdon-Ismail, and H. 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)*, Saratoga Springs, NY, USA, May 2005.

[15] A. Szalay and J. Gray, "The world-wide telescope," *Science*, vol. 293, p. 2037, 2001.

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

[17] K. Taura, K. Kaneda, and T. Endo, "Phoenix: a Parallel Programming Model for Accommodating Dynamically Joininig/Leaving Resources," in *Proc. of PPoPP*. ACM, 2003, pp. 216–229.

[18] C. A. Varela, P. Ciancarini, and K. Taura, “Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments,” *Scientific Programming Journal*, vol. 13, no. 4, pp. 255–263, December 2005, guest Editorial.