Cluster Comput
DOI 10.1007/s10586-007-0032-9

Malleable applications for scalable high performance computing

Travis Desell · Kaoutar El Maghraoui · Carlos A. Varela

Abstract Iterative applications are known to run as slow as their slowest computational component. This paper introduces *malleability*, a new dynamic reconfiguration strategy to overcome this limitation. Malleability is the ability to dynamically change the data size and number of computational entities in an application. Malleability can be used by middleware to autonomously reconfigure an application in response to dynamic changes in resource availability in an architecture-aware manner, allowing applications to optimize the use of multiple processors and diverse memory hierarchies in heterogeneous environments.

The modular Internet Operating System (IOS) was extended to reconfigure applications autonomously using malleability. Two different iterative applications were made malleable. The first is used in astronomical modeling, and representative of maximum-likelihood applications was made malleable in the SALSA programming language. The second models the diffusion of heat over a two dimensional object, and is representative of applications such as partial differential equations and some types of distributed simulations. Versions of the heat application were made malleable both in SALSA and MPI. Algorithms for concurrent data redistribution are given for each type of application. Results show that using malleability for reconfiguration is 10 to 100 times faster on the tested environments. The algorithms are also shown to be highly scalable with respect to the quantity of data involved. While previous work has shown the utility of dynamically reconfigurable applications using only computational component migration, malleability is shown to provide up to a 15% speedup over component migration alone on a dynamic cluster environment.

This work is part of an ongoing research effort to enable applications to be highly reconfigurable and autonomously modifiable by middleware in order to efficiently utilize distributed environments. Grid computing environments are becoming increasingly heterogeneous and dynamic, placing new demands on applications' adaptive behavior. This work shows that malleability is a key aspect in enabling effective dynamic reconfiguration of iterative applications in these environments.

Keywords High performance computing · Malleability · Dynamic reconfiguration · MPI · SALSA · Actors

T. Desell (✉) · K.E. Maghraoui · C.A. Varela
Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590, USA
e-mail: deselt@cs.rpi.edu

K.E. Maghraoui
e-mail: elmagk@cs.rpi.edu

C.A. Varela
e-mail: cvarela@cs.rpi.edu

1 Introduction

As high performance computing (HPC) environments scale to hundred or thousands of parallel-processors, the demands on an application developer are becoming increasingly challenging and unmanageable. In such large-scale heterogeneous execution environments, traditional application or middleware models that assume dedicated resources or fixed resource allocation strategies fail to provide the desired high performance that applications expect from large pools of 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 that account for dynamic and shared resources 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.

Previous approaches for dynamic reconfiguration have involved *fine-grained migration*, which moves an application's computational components (such as actors, agents or processes) to utilize unused cycles [12, 14] 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 [4, 24]. Coarse-grained migration can prove highly expensive when resources change availability frequently, as in the case of web computing [3, 18] 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 reconfiguration is limited by the granularity of the applications' entities, and cannot adapt to the heterogeneity of memory hierarchies and data distribution. Existing load balancing approaches (e.g. [6, 11]) 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 Sect. 4.1). This application is run on a dynamic cluster consisting of five processors. Figure 1 shows the performance of different approaches to dynamic 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 due to context switching. It is also not scalable as the number of components required for this scheme increases exponentially as processors

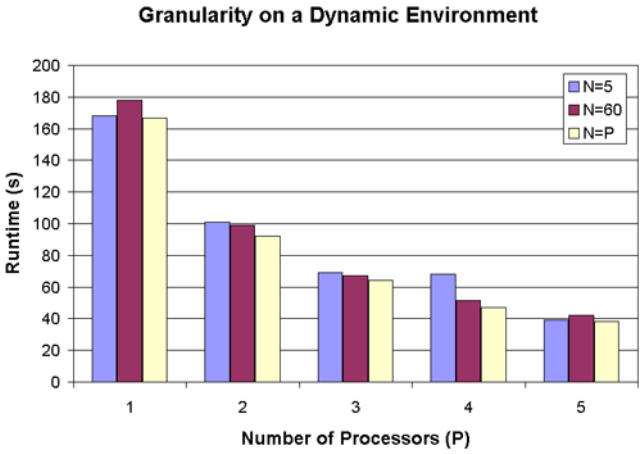

Fig. 1 Different possibilities for data distribution on a dynamic environment with N entities and P processors. With $N = 5$, data imbalances occur, degrading performance. $N = 60$ allows even data distribution, but suffers from increased context switching. $N = 5$ and $N = 60$ can be accomplished using fine-grained migration, while $N = P$ requires malleability

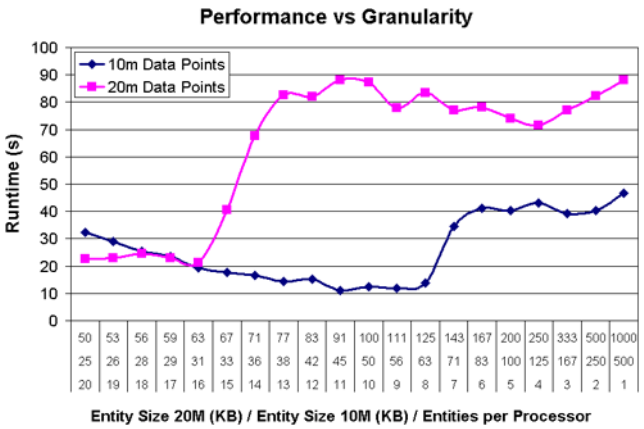

Fig. 2 A heat diffusion application (see Sect. 4.2), run with different data sizes and granularities on 20 processors. Decreasing granularity allows the entities to execute entirely within a 64 KB L1 cache, greatly increasing performance. This figure also illustrates how overly decreasing granularity can degrade performance by incurring additional overhead

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 by enabling entity computation and data usage to execute in lower levels of memory hierarchy. For example, in Fig. 2, the runtime for a scientific application modeling heat diffusion over a solid (for a more detailed description see Sect. 4.2) 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 2.27 GHz Opteron processors with 64 KB L1 cache. As the data per entity decreases, computation can be done within L1 cache, instead of within L2 cache, with over a 10× performance increase for 20 million data points and 320 entities, and an over 5× 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 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 and 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.

This work introduces a new type of application reconfiguration, malleability, that can be used in conjunction with existing methods of reconfiguration, such as migration. Malleability has been implemented in both MPI [17] and the SALSA programming language [25]. The IOS middleware [15] has been extended to autonomously reconfigure applications using malleability. In Sect. 2, the language and library extensions for SALSA and MPI to allow for malleability are described. Section 3 briefly describes the autonomous middleware that is used to automate malleability. Application case studies are presented in Sect. 4 that illustrate how malleability has been achieved for two representative iterative applications. The middleware and algorithms for malleability are evaluated in Sect. 5. Related work is discussed in Sect. 6. The paper concludes with a discussion and avenues of future work in Sect. 7.

2 Language extensions

In order to make malleability accessible to a distributed application's designer, the problem of dynamic granularity is approached from two directions: (i) Providing language-level constructs and libraries to enable the developer to make preexisting entities malleable with minimal effort and to facilitate the development of new malleable entities, and (ii) Providing extensible middleware that can use profiled application-level and environmental information to make reconfiguration decisions, enabling malleable applications to improve performance on unknown and dynamic environments. Both the SALSA programming language and MPI have been extended with library support for enabling malleability.

2.1 Language constructs and libraries

Fine-grained migration can be accomplished by checkpointing and locking mechanisms because the state of the entities being migrated does not need to change. In some cases this process can be entirely automated, as in the SALSA [25] programming language. In non-trivial cases, malleability not only requires redistribution of data, which can be dependent on entity behavior, but also redirection of references when new entities are created or removed. In order to make malleability accessible to the developer, language constructs and libraries are needed to help automate this process.

For the purposes of this work, groups of entities are defined as malleable if their granularity can change by data redistribution, increase by merging, or decrease by splitting, i.e. a group of M entities can split or merge into a group of N entities, with a redistribution of the same data. Group malleability allows for a wider range of reconfiguration possibilities, from very coarse-grained reconfiguration using most or all entities in an application to very fine-grained reconfiguration using small subsets of an application's entities.

2.2 Malleability in SALSA

SALSA uses the Actor model of computation [2]. Actors encapsulate state and process into a single entity and communicate asynchronously. Actor semantics enable transparent checkpointing. An actor can always checkpoint in between processing messages. Additionally, encapsulated state and distributed memory provide clean separation of how data is distributed among entities, which allows malleability to be clearly defined. This is in contrast to shared memory systems, which violate state encapsulation properties. Finally, the execution environment provided by SALSA can run on heterogeneous architectures, allowing malleability to be tested on a wide range of architectures and environments.

Malleability is performed using extensible *directors* that coordinate the malleability process for a group of malleable

Table 1 MalleableActor API

Return type	Method name	Parameters
long	getDataSize	()
Object	getData	(long size, Options options)
void	receiveData	(Object data, Options options)
void	redirectReference	(String referenceName, Reference newTarget)
void	handleMalleabilityMessage	(Message message)

actors. Actors are made malleable by implementing the *MalleableActor* behavior (see Table 1), which consists of methods that allow directors to redirect references to accommodate newly created and removed actors, and redistribute data between malleable actors participating in reconfiguration. This allows directors to implement various algorithms to perform malleability. Provided directors use a generic protocol to perform malleability operations. Reconfiguration using malleability consists of three basic parts: (i) locking all participants in the reconfiguration, (ii) redistributing data to newly created participants or from removed participants, and (iii), 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. 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 4 describes implementations of data redistribution and reference redirection for sample applications. (5) After all redirection and redistribution are 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.

2.3 Malleability in MPI

Unlike SALSA, MPI does not provide inherent support for migration. In previous work [16], the Process Checkpointing and Migration (PCM) library was developed to enable process migration in MPI. PCM has been extended with additional library support for malleability. As in the SALSA implementation, a director is responsible for initiating data redistribution and updating all the necessary references. Due to the nature of iterative applications in MPI, reconfiguration can only occur at barrier or synchronization points. This preserves the correctness of the algorithm by ensuring that no messages are in transit while reconfiguration takes place. The director is the master process, which is usually process with rank 0.

PCM provides fours classes of services, environmental inquiry services, checkpointing services, global initialization and finalization services, and collective reconfiguration services. Table 2 shows the classification of the PCM API calls. Malleability has been implemented in MPI for data parallel programs with a two-dimensional data structure and a linear communication. Common data distributions are allowed such as block, cyclic, and block-cyclic distributions.

MPI_PCM_INIT is a wrapper for MPI_INIT. The user calls this function at the beginning of the program. MPI_PCM_INIT is a collective operation that takes care of initializing several internal data structures. It also reads a configuration file that has information about the port number and location of the PCM daemon (PCMD), a runtime system that provides checkpointing and global synchronization between all running processes.

Migration and malleability operations require the ability to save and restore the current state of the process(es) to be reconfigured. PCM_ Store and PCM_Load provide storage and restoration services of the local data. Checkpointing is handled by the PCMD runtime system that ensures that data is stored in locations with reasonable proximity to their destination.

At startup, an MPI process can have three different states: (1) PCM_STARTED, a process that has been initially started in the system (for example using mpiexec), (2) PCM_MIGRATED, a process that has been spawned because of a migration, and (3) PCM_SPLITTED, a process

Table 2 The PCM API

Service type	Function name
Initialization	MPI_PCM_Init
Finalization	PCM_Exit, PCM_Finalize
Environmental Inquiry	PCM_Process_Status, PCM_Comm_rank, PCM_Status, PCM_Merge_datacnts
Reconfiguration	PCM_Reconfigure
	PCM_Split, PCM_Split_Collective
	PCM_Merge, PCM_Merge_Collective
Checkpointing	PCM_Load, PCM_Store

that has been spawned because of a split operation. A process that has been created as a result of a reconfiguration (migration or split) proceeds to restoring its state by calling PCM_Load. This function takes as parameters information about the keys, pointers, and data types of the data structures to be restored. An example includes the size of the data, the data buffer and the current iteration number. Process ranks may also be subject to changes in case of malleability operations. PCM_ Comm_rank reports to the calling process its current rank. Conditional statements are used in the MPI program to check for its startup status. An illustration is given in Fig. 6.

The application probes the PCMD system to check if a process or a group or processes need to be reconfigured. Middleware notifications set global flags in the PCMD system. To prevent every process from probing the runtime system, the director process probes the runtime system and broadcasts any reconfiguration notifications to the other processes. This provides a callback mechanism that makes probing non-intrusive for the application. PCM_status returns the state of the reconfiguration to the calling process. It returns different values to different processes. In the case of a migration, PCM_MIGRATE value is returned to the process that needs to be migrated, while PCM_RECONFIGURE is returned to the other processes. PCM_Reconfigure is a collective function that needs to be called by both the migrating and non-migrating processes. Similarly PCM_SPLIT or PCM_MERGE are returned by the PCM_status function call in case of a split or merge operation. All processes collectively call the PCM_Split or PCM_Merge functions to perform a malleable reconfiguration.

Split and merge functions change the ranking of the processes, the total number of processes, and the MPI communicators. All occurrences of MPI_COMM_WORLD, the global communicator with all the running processes, should be replaced with PCM_COMM_WORLD. This latter is a malleable communicator since it expands and shrinks as processes get added or removed. All reconfiguration operations happen at synchronization barrier points. The current implementation requires no communication messages to be outstanding while a reconfiguration function is called. Hence, all calls to the reconfiguration PCM calls need to happen either at the beginning or end of the loop.

When a process or group of processes engage in a split operation, they determine the new data redistribution and checkpoint the data to be sent to the new processes. When the new processes are created, they inquire about their new ranks and load their data chunks from the PCMD. The checkpointing system maintains an up-to-date database per process rank. Then all application's processes synchronize to update their ranks and their communicators. The malleable calls return handles to the new ranks and the updated communicator. Unlike a split operation, a merge operation entails removing processors from the MPI communicator. Merging operations for data redistribution are implemented using MPI scatter and gather operations.

3 Autonomous middleware

This work extends the Internet Operating System (IOS) [15], a modular middleware for reconfiguration of distributed applications, to autonomously use malleability. IOS is responsible for the profiling and reconfiguration of applications, allowing 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 [8] and MPI [13]. 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 [15], 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. Having profiling and reconfiguration tools conform to a specific API allows the middleware to receive profiling information and reconfigure applications in a programming language independent manner, by determining what parts of the

application are reconfigured and when they should be reconfigured. Likewise, middleware can profile the environment by interacting with third-party clients, such as the Network Weather Service (NWS) [27] or the Globus Meta Discovery Service (MDS) [7].

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 more efficient use of memory hierarchy and reduced context switching. 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.

4 Application case studies

Two representative applications have been modified to utilize malleability features. 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 [19] based on data derived from the SLOAN digital sky survey [22]. 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 spatially-independent data, because the data needs to be redistributed such that it preserves the adjacency of data points.

4.1 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. Figure 3 contains the algorithm 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*, and the *desiredWorkerData* for each of these workers. 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*.

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

```
MalleableActor[] redistributeData(MalleableActor[] workers, long[] desiredWorkerData)
    long[] workerData = new long[workers.length];
    for (i = 0 ...  workers.length) workerData[i] = workers[i]←getDataSize();
    MalleableActor[] workersToExpand = {}, workersToShrink = {};
    long[] expandDesiredData = {}, expandCurrentData = {}, shrinkDesiredData = {}, shrinkCurrentData = {};
    /*split workers into workersToShrink and workersToExpand, and create arrays with their current and desired sizes*/
    for (i = 1..workers.length)
        if (workerData[i] < desiredWorkerData[i])
            workersToExpand.append(workers[i]);
            expandDesiredData.append(desiredWorkerData[i]);
            expandCurrentData.append(workerData[i]);
        else if (workerData[i] > desiredWorkerData[i])
            workersToShrink.append(workers[i]);
            shrinkDesiredData.append(desiredWorkerData[i]);
            shrinkCurrentData.append(workerData[i]);

    /*send data from workersToShrink to workersToExpand*/
    currentShrink = currentExpand = 1;
    while (currentShrink <= workersToShrink.length and currentExpand <= workersToExpand.length)
        dataToSend = shrinkCurrentData[currentShrink] - shrinkDesiredData[currentShrink];
        dataToReceive = expandDesiredData[currentExpand] - expandCurrentData[currentExpand];
        toSend = min(dataToSend, dataToRecieve);
        workersToExpand←receiveData( workersToShrink←getData(toSend) );
        shrinkCurrentData[currentShrink] -= toSend; dataToSend -= toSend;
        expandCurrentData[currentExpand] += toSend; dataToReceive -= toSend;
        if (dataToSend == 0) currentShrink++;
        if (dataToReceive == 0) currentExpand++;
    return workers;
```

Fig. 3 The data redistribution algorithms for applications with spatially independent data. The middleware creates new workers or specifies 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. Message passing, denoted by the ← operator, can be done asynchronously and messages can be processed in parallel

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.

4.2 Simulation of heat diffusion

The heat application's entities 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.

4.2.1 SALSA implementation

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 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.

Figure 4 contains the algorithms 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. The algorithm iterates through the workers, shifting data left and right as required to achieve the *desiredWorkerData*. 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 leftward as in the algorithm involves at most $2m$ *getData* and *recieveData* 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

```
MalleableActor[] redistributeData(MalleableActor[] workers, long[] desiredWorkerData)
   /*obtain the amount of data at each worker*/
   long[] workerData = new long[workers.length];
   for (i = 0 ...  workers.length) workerData[i] = workers[i]←getDataSize();
   current = 1;
   while (current <= workers.length)
      if (workerData[current] < desiredWorkerData[current])
         /*the current worker needs more data, shift data leftwards to this worker*/
         dataToSend = desiredWorkerData[current] - workerData[current];
         currentTarget = current+1;
         while (dataToSend > 0)
            while (workerData[currentTarget] == 0) currentTarget++;
            toSend = min(dataToSend, workerData[currentTarget]);
            workers[current]←receiveData( workers[currentTarget]←getData(toSend,left), right );
            dataToSend -= toSend; workerData[currentTarget] -= toSend; workerData[current] += toSend;

      else if (workerData[current] > desiredWorkerData[current])
         /*the current worker has too much data, shift data rightwards away from this worker*/
         dataToSend = workerData[current] - desiredWorkerData[current];
         /*find the rightmost worker that data needs to be shifted right to*/
         currentTarget = current;
         totalDesired = desiredWorkerData[currentTarget]; totalCurrent = workerData[currentTarget];
         while (totalDesired < totalCurrent)
            currentTarget++;
            totalDesired += desiredWorkerData[currentTarget];
            totalCurrent += workerData[currentTarget];
         /*find the initial amount of data to sent to the target*/
         dataToSend = 0;
         if (totalCurrent > totalDesired)
            dataToSend = (totalDesired-desiredWorkerData[currentTarget])-(totalCurrent-workerData[currentTarget]);
         else /*totalCurrent == totalDesired*/
            dataToSend = desiredWorkerData[currentTarget] - workerData[currentTarget];
         /*shift data right, towards the target*/
         currentSource = currentTarget-1;
         while (currentTarget > current)
            while (dataToSend > 0)
               toSend = min(workerData[currentSource], dataToSend);
               workers[currentTarget]←receiveData( workers[currentSource]←getData(toSend, right), left );
               dataToSend -= toSend;
               workerData[currentTarget] += toSend;
               workerData[currentSource] -= toSend;
               if (workerData[currentSource] == 0) currentSource--;
            currentTarget--;
            if (currentSource == currentTarget) currentSource--;
            dataToSend = desiredWorkerData[currentTarget]-workerData[currentTarget];

      else current++;
   return workers;
```

Fig. 4 The data redistribution algorithm for applications with two-dimensional spatially dependent data. The middleware creates new workers or specifies 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. Message passing, denoted by the ← operator, can be done asynchronously and messages can be processed in parallel, however messages must be processed by workers in the order that they are sent to that worker

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.

4.2.2 MPI implementation

A sample skeleton of a simple MPI-based application is given in Fig. 5. The structure of the example given is very common in iterative applications. The code starts by performing various initializations of some data structures. Data is distributed by the root process to all other processes in a block distribution. The xDim and yDim variables denote the dimensions of the data buffer. The program then enters the iterative phase where processes perform computations locally and then exchange border information with their neighbors. Figure 6 shows the same application instrumented with PCM calls to allow for migration and malleability. In case of split and merge operations, the dimensions of the data buffer for each process might change. The PCM split and merge take as parameters references to the data buffer and dimen-

Fig. 5 Skeleton of the original MPI code of an MPI application

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

int main(int argc, char **argv) {
   //Declarations
   ....
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &totalProcessors );
   current_iteration = 0;
   //Determine the number of columns for each processor.
   xDim = (yDim-2) / totalProcessors;
   //Initialize and Distribute data among processors
   ...
   for(iterations=current_iteration; iterations<TOTAL_ITERATIONS; iterations++){
      // Data Computation.
      ...
      //Exchange of computed data with neighboring processes.
      // MPI_Send() || MPI_Recv()
      ...
   }
   // Data Collection
   ...
   MPI_Finalize();
   return 0;
}
```

sions and update them appropriately. In case of a merge operation, the size of the buffer needs to be known so enough memory can be allocated. The PCM_Merge_ datacnts function is used to retrieve the new buffer size. This call is significant only at processes that are involved in a merge operation. Therefore a conditional statement is used to check whether the calling process is merging or not.

The example shows that it is not complicated to instrument MPI iterative applications with PCM calls. The programmer is required only to know the right data structures that are needed for malleability. With these simple instrumentations, the MPI application becomes malleable and ready to be reconfigured by IOS middleware.

5 Results

Malleability was tested on a variety of architectures and environments to determine its performance. Section 5.1 describes the different environments that were used in the evaluations. Section 5.2 measures the overhead of using the autonomous middleware and implementing *MalleableActor*. Reconfiguration time and scalability of malleability is compared to application stop-restart in Sect. 5.3 for the astronomy and heat applications. Lastly, Sect. 5.4 shows the benefit of using malleability and migration on a cluster with dynamically changing resources.

5.1 Test environments

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

The single- and dual-core Opteron clusters are connected by 10 GB/sec bandwidth, 7 usec latency Infiniband, and 1 GB/sec bandwidth, 100 µsec latency Ethernet. Intra-cluster communication on the AIX cluster is with 1 GB/sec bandwidth, 100 µsec latency Ethernet, and it is connected to the opteron clusters over the RPI campus wide area network.

5.2 Middleware and library 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. Figure 7 shows the overhead of the middleware services and extending the `MalleableActor` interface with the heat and astronomy applications. 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).

To evaluate the overhead of the PCM profiling and status probing, we have run the heat diffusion problem with and without PCM instrumentation on a cluster of 4 dual-processor nodes. We varied the granularity of the application and recorded the execution time of the application. Figure 8 shows that the overhead of the PCM library does not exceed 20% of the application's running time. This is mainly profiling overhead. The library supports tunable profiling,

```
#include "mpi.h"
#include "pcm_api.h"
...
MPI_Comm PCM_COMM_WORLD;
int main(int argc, char **argv) {
   //Declarations
   ....
   int current_iteration, process_status;
   PCM_Status pcm_status;
   //declarations for malleability
   double *new_buffer;
   int merge_rank, mergecnts;
   MPI_PCM_Init( &argc, &argv);
   PCM_COMM_WORLD = MPI_COMM_WORLD;
   PCM_Init(PCM_COMM_WORLD);
   MPI_Comm_rank( PCM_COMM_WORLD, &rank );
   MPI_Comm_size( PCM_COMM_WORLD, &totalProcessors );

   process_status = PCM_Process_Status();
   if(process_status == PCM_STARTED){
      current_iteration = 0;
      //Determine the number of columns for each processor.
      xDim = (yDim-2) / totalProcessors;
      //Initialize and Distribute data among processors
      ...
   }
   else{
      PCM_Comm_rank(PCM_COMM_WORLD, &rank);
      PCM_Load(rank, "iterator",&current_iteration);
      PCM_Load(rank, "datawidth", &xDim);
      prevData = (double *)calloc((xDim+2)*yDim,sizeof(double));
      PCM_Load(rank, "myArray",prevData);
   }

   for(iterations=current_iteration; iterations<TOTAL_ITERATIONS; iterations++){
      pcm_status = PCM_Status(PCM_COMM_WORLD);
      if(pcm_status == PCM_MIGRATE){
        PCM_Store(rank,"iterator",&iterations,PCM_INT,1);
        PCM_Store(rank,"datawidth",&xDim,PCM_INT,1);
        PCM_Store(rank,"myArray",prevData,PCM_DOUBLE, (xDim+2)*yDim);
        PCM_COMM_WORLD = PCM_Reconfigure(PCM_COMM_WORLD,argv[0]);
      }
      else if(pcm_status == PCM_RECONFIGURE)
      {
        PCM_Reconfigure(&PCM_COMM_WORLD,argv[0]);
        MPI_Comm_rank(PCM_COMM_WORLD, &rank);
      }
      else if(pcm_status == PCM_SPLIT){
        PCM_split( prevData, PCM_DOUBLE, &iterations, &xDim, &yDim, &rank, &totalProcessors, &PCM_COMM_WORLD, argv[0]);

      }else if(pcm_status == PCM_MERGE){
        PCM_Merge_datacnts(xDim, yDim, &mergecnts, &merge_rank, PCM_COMM_WORLD);
        if(rank == merge_rank)
          /*Reallocate memory for the data buffer*/
          new_buffer = (double*)calloc(mergecnts, sizeof(double));
          PCM_Merge( prevData, MPI_DOUBLE, &xDim, &yDim, new_buffer, mergecnts,&rank,&totalProcessors, &PCM_COMM_WORLD);
          if(rank == merge_rank)
          prevData = new_buffer;
       }

      // Data Computation.
      ...
      //Exchange of computed data with neighboring processes.
      // MPI_Send() || MPI_Recv()
      ...
   }
   // Data Collection
   ...
   PCM_Finalize(PCM_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Fig. 6 Skeleton of the malleable MPI code with PCM calls

whereby the degree of profiling can be decreased by the user to reduce its intrusiveness.

5.3 Malleability evaluation

Reconfiguration time was measured by making clusters available and unavailable to the heat and astronomy applications (see Fig. 9). 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 and removed from the 4×1 Opteron cluster to measure reconfiguration time over a WAN, and the 4×2 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 re-

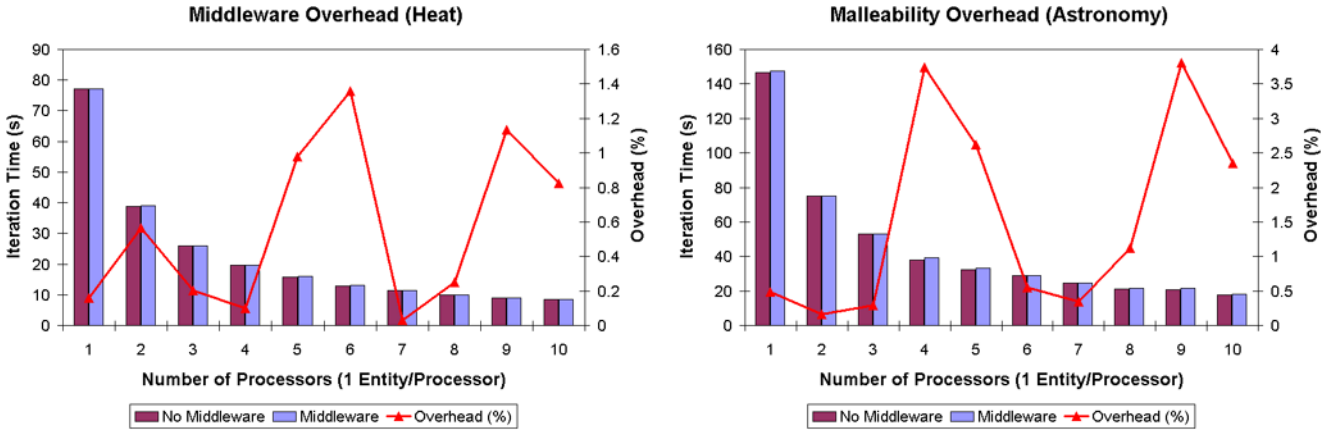

Fig. 7 Overhead of using autonomous middleware and the *MalleableActor* interface for the heat and astronomy applications. The application was tested with the same configurations with different amounts of parallelism, with and without the middleware and *MalleableActor* interface

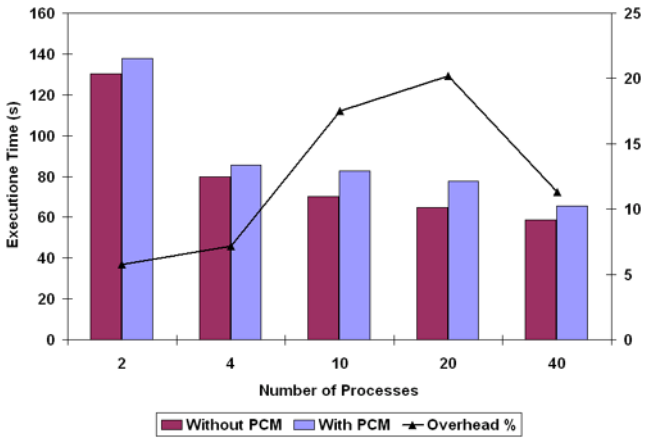

Fig. 8 Overhead of the PCM library with malleability

configuration algorithms and the middleware sent less data to the AIX cluster than the 4 × 2 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 4 × 2 Opteron cluster. Malleability is shown to be more scalable than application stop-restart as data size increased. Over the LAN, splitting went from being 46× to 73× faster as data size increased from 100 MB to 300 MB, while merging improved 38× to 57×. Malleability is shown to be even more scalable over a WAN, as split improved reconfiguration time 34× to 103× and merge improved 23× to 67× as data size increased from 100 MB to 300 MB.

The heat application also gained significant reconfiguration time improvements using split and merge. Over a LAN, split improved from 30× to 45× faster, and merge improved from 34× to 61× faster than stop-restart as data size increased from 100 MB to 300 MB. Due to the additional synchronization required by the spatially-dependent data reconfiguration algorithms, split and merge did not improve as significantly over the WAN. Reconfiguration time using split increased 9× to 13× and increased 7× to 9× using merge, as data increased from 100 MB to 300 MB.

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.

5.4 Malleability on dynamic environments

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 10 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.

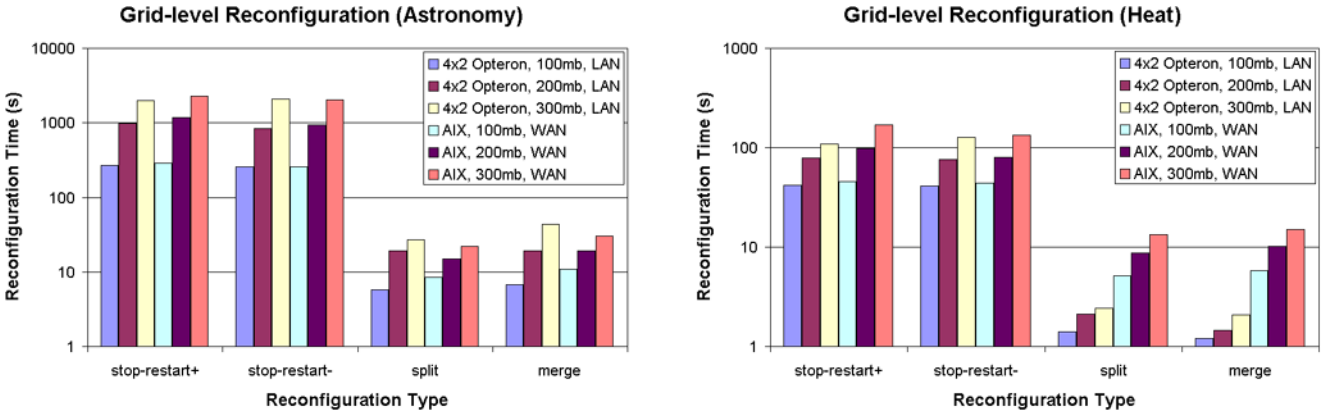

Fig. 9 Performance of grid-level reconfiguration using the heat and astronomy applications with different data sizes over local and wide area networks. For reconfiguration over a WAN, the AIX cluster was added and removed from 4×1 opteron cluster, while the 4×2 opteron was added and removed for reconfiguration 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

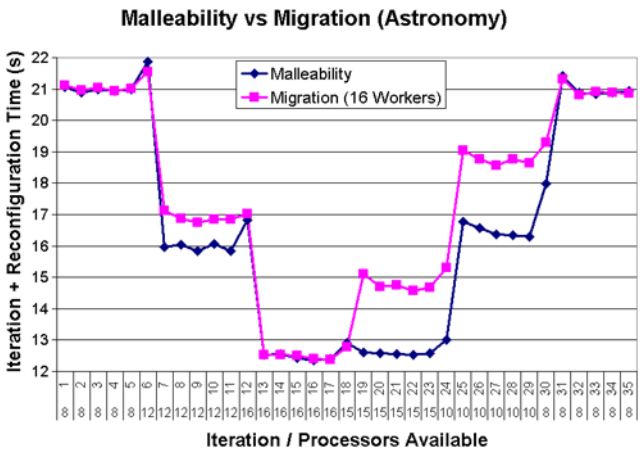

Fig. 10 Autonomous reconfiguration using malleability and migration compared to autonomous reconfiguration only using migration. Every 6 iterations, the environment changed, from 8 to 12 to 16 to 15 to 10 to 8 processors. The last 8 processors removed were the initial 8 processors. A non-reconfigurable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

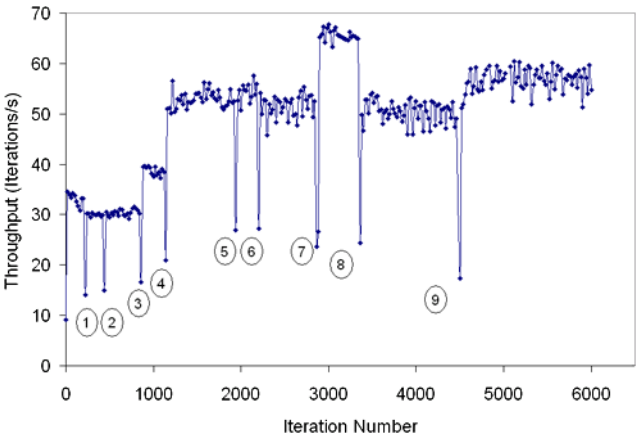

Fig. 11 Gradual adaptation using malleability and migration as resources leave and join

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.

Malleability was also run using the MPI heat application on a dynamic environment (see Fig. 11). When the execution environment experiences small load fluctuations, a gradual adaptation strategy is needed. The MPI heat application was launched on a dual-processor machine with 2 processes. Two binary split operations occurred at events 1 and 2. The throughput of the application decreased because of the decrease of the granularity of the processes on the hosting machine. At event 3, another dual-processor node was made available to the application. Two processes migrated to the new node. The application experienced an increase in throughput as a result of this reconfiguration. A similar situation happened at events 5 and 6, which triggered two split operations and two migrations to another dual-processor node. A node left at event 8 which caused two processes to be migrated to one of the participating machines. A merge operation happened at event 9 in the node with excess processors improving the application's throughput.

6 Related work

Several recent efforts have focused on middleware-level technologies for the emerging computational grids. Dynamic reconfiguration in grid environments includes the GrADS project [4] which includes SRS [24], a library which allows stop and restart of applications in grid environments using the Globus Toolkit [9] based on dynamic performance evaluation. Adaptive MPI (AMPI) [5, 10] 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 [20] 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 [23] 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. Our approach is different in that we do not need to over-allocate resources initially. Such a strategy, though potentially very useful, may be impractical in grid environments where resources join and leave and where an initial over-allocation may not be possible. We allow new nodes that become available to join the computational grid to improve the performance of running applications during their execution.

Other efforts have focused on application checkpointing and restart as a mechanism to allow applications to adapt to fault-prone environments. Examples include CoCheck [21] and starFish [1]. Both CoCheck and starFish support checkpointing for fault-tolerance, while we provide this feature to allow process migration and hence load balancing. Our work differs in the sense that we support migration at a finer granularity. Process checkpointing is a non-functional operational concern that is needed to allow dynamic reconfiguration. To be able to migrate MPI processes to better performing nodes, processes need to save their state, migrate, and restart from where they left off. Application-transparent process checkpointing is not a trivial task and can be very expensive, as it requires saving the entire process state. Semi-transparent checkpointing provides a simple solution that has been proved useful for iterative applications [20, 24]. API calls are inserted in the MPI program that informs the middleware of the important data structures to save. This is an attractive solution that can benefit a wide range of applications and does not incur significant overhead since only relevant state is saved.

As far as we know, this work is novel in that it is the first presentation of a generic framework for autonomous reconfiguration using dynamic entity granularity refinement. The selected related work is far from comprehensive. For a more in depth discussion of middleware for autonomous reconfiguration over dynamic grid environments, the reader is referred to [26].

7 Discussion

This paper describes a framework for dynamic application granularity, or *malleability*, which enables applications to dynamically redistribute data and add and remove processing entities. Malleability is not limited to farmer-worker applications where there is no inter-worker communication, nor to applications with spatially independent data. MPI and the SALSA programming language were extended with language-level abstractions and libraries to facilitate the use of malleability. A protocol to prevent malleability from creating data inconsistencies is given. Two iterative scientific applications are made malleable using these abstractions and libraries: an astronomy application with a farmer-worker communication topology and spatially independent data, and a heat application with a doubly-linked list communication topology and spatially dependent data. Efficient linear time algorithms are given for concurrent data redistribution of both spatially independent and dependent data, and are evaluated using representative applications.

Reconfiguration using malleability is shown to be approximately 10 to 100 times faster than application stop-restart for the applications and environments tested. These results show that dynamic malleability is significantly more efficient and scalable than application stop-restart and can be done efficiently with respect to application runtime. Additionally, dynamic malleability is compared to migration on a dynamic cluster, running up to 15% faster in the different configurations tested. This work argues that dynamic and autonomous malleability is a valuable asset for iterative applications running on dynamic environments.

IOS's generic interface allows for various languages and programming architectures to utilize the autonomous malleability strategies presented in this work. Additionally, the director/malleable entity strategy presented allows for different data distribution algorithms to be implemented for diverse data representations. Modular director and malleable actor implementations allow research into more efficient data redistribution algorithms. Ideally, applications will only need to extend library-provided malleable actors, and be made autonomously malleable with a minimum amount of code modification.

This work is part of ongoing research into making applications adaptive and highly responsive to changes in their distributed execution environments using middleware. As distributed environments grow in size and become increasingly heterogeneous and dynamic, new reconfiguration

methods and policies, such as those described in this paper, will become a necessity for distributed applications, allowing applications to become more efficient and fully utilize existing grid computing environments.

Acknowledgements This work has been partially supported by the following grants: NSF CAREER CNS Award No. 0448407, NSF SEI(AST) No. 0612213, NSF MRI No. 0420703, IBM SUR Award 2003, and IBM SUR Award 2004.

References

1. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In: Proceedings of The Eighth IEEE International Symposium on High Performance Distributed Computing, p. 31. IEEE Computer Society (1999)
2. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
3. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002)
4. Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., Wolski, R.: The GrADS project: Software support for high-level grid application development. Int. J. High-Perform. Comput. Appl. **15**(4), 327–344 (2002)
5. Bhandarkar, M.A., Kale, L.V., de Sturler, E., Hoeflinger, J.: Adaptive load balancing for MPI programs. In: Proceedings of the International Conference on Computational Science—Part II, pp. 108–117. Springer (2001)
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94), Santa Fe, New Mexico, November 1994, pp. 356–368
7. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), August 2001
8. Desell, T., Maghraoui, K.E., Varela, C.: 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
9. Foster, I., Kesselman, C.: The Globus project: A status report. In: Antonio, J. (ed.) Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98), pp. 4–18. IEEE Computer Society (1998)
10. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, October 2003
11. Lan, Z., Taylor, V.E., Bryan, G.: Dynamic load balancing of SAMR applications on distributed systems. Sci. Progr. **10**(4), 319–328 (2002)
12. Litzkow, M., Livny, M., Mutka, M.: Condor—a hunter of idle workstations. In: Proceedings of the 8th International Conference of Distributed Computing Systems, June 1988, pp. 104–111
13. Maghraoui, K.E., Flaherty, J., Szymanski, B., Teresco, J., Varela, C.: Adaptive computation over dynamic and heterogeneous networks. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Wasniewski, J. (eds.) Proc. of the Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM'2003), Czestochowa, Poland, September 2003. Lecture Notes in Computer Science, vol. 3019, pp. 1083–1090. Springer, Berlin (2003)
14. Maghraoui, K.E., Desell, T., Szymanski, B.K., Teresco, J.D., Varela, C.A.: Towards a middleware framework for dynamically reconfigurable scientific computing. In: Grandinetti, L. (ed.) Grid Computing and New Frontiers of High Performance Processing. Elsevier (2005)
15. Maghraoui, K.E., Desell, T.J., Szymanski, B.K., Varela, C.A.: The internet operating system: Middleware for adaptive distributed computing. Int. J. High Perform. Comput. Appl. (IJHPCA) **10**(4), 467–480 (2006), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms
16. Maghraoui, K.E., Szymanski, B., Varela, C.: An architecture for reconfigurable iterative mpi applications in dynamic environments. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Wasniewski, J. (eds.) Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005), ser. LNCS, no. 3911, Poznan, Poland, September 2005, pp. 258–271
17. Message Passing Interface Forum, MPI: A message-passing interface standard, Int. J. Supercomput. Appl. High Perform. Comput. **8**(3/4), 159–416 (1994)
18. Pande, V., et al.: Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. Biopolymers **68**(1), 91–109 (2002), Peter Kollman Memorial Issue
19. Purnell, J., Magdon-Ismail, M., Newberg, H.: 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, pp. 475–484. Springer (2005)
20. Sievert, O., Casanova, H.: A simple MPI process swapping architecture for iterative applications. Int. J. High Perform. Comput. Appl. **18**(3), 341–352 (2004)
21. Stellner, G.: Cocheck: Checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium, pp. 526–531. IEEE Computer Society (1996)
22. Szalay, A., Gray, J.: The world-wide telescope. Science **293**, 2037 (2001)
23. Taura, K., Kaneda, K., Endo, T.: Phoenix: a parallel programming model for accommodating dynamically joininig/leaving resources. In: Proc. of PPoPP, pp. 216–229. ACM (2003)
24. Vadhiyar, S.S., Dongarra, J.J.: SRS—a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process. Lett. **13**(2), 291–312 (2003)
25. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Not. OOPSLA'2001 Intriguing Techn. Track Proc. **36**(12), 20–34 (2001), http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf
26. Varela, C.A., Ciancarini, P., Taura, K.: Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. Sci. Program. J. **13**(4), 255–263 (2005), Guest Editorial
27. Wolski, R., Spring, N.T., Hayes, J.: The Network Weather Service: A distributed resource performance forecasting service for metacomputing. Future Gener. Comput. Syst. **15**(5–6), 757–768 (1999)

Travis Desell is a Ph.D. candidate in the Department of Computer Science at Rensselaer Polytechnic Institute (RPI). He also received a BS degree in computer science from RPI. Current research interests involve finding new methodologies for application scheduling and load balancing for grid applications, as well as applying decentralized learning strategies to such environments. Other research interests include visualization of large scale systems and peer-to-peer networks.

Kaoutar El Maghraoui is a PhD Candidate in the Department of Computer Science at Rensselaer Polytechnic Institute (RPI). She received the BS degree in Computer Science and the MS degrees in Computer Networks from Alakhawayn University in Ifrane, Morocco, in 1999 and 2001 respectively. Before starting her PhD program, she worked as a Computer Science lecturer in the School of Science and Engineering at Alakhawayn University. Her current research interests lie in the area of resource management and middleware technologies for grid computing. In particular, she is focusing on the challenges associated with adapting distributed application to the dynamics of computational grids. She is a member of the IEEE Computer Society and the ACM.

Carlos A. Varela is an Assistant Professor at the Department of Computer Science and Founding Director of the Worldwide Computing Laboratory, at RPI. He received his Ph.D., M.S. and B.S. degrees in Computer Science at the University of Illinois at Urbana-Champaign, in 2001, 2000 and 1992, respectively. He was a Research Staff Member at IBM T.J. Watson Research Center before joining RPI. Dr. Varela is the Information Director of the ACM Computing Surveys journal and has served as Guest Editor of the Scientific Programming journal. Dr. Varela has given lectures about his research on web-based computing, concurrent and distributed systems, and active object oriented programming languages, internationally, including seminars in the U.S., Switzerland, Japan, Colombia, Canada, Germany, Australia, Netherlands, France, the Czech Republic, and Italy. In 2005, Dr. Varela received the National Science Foundation CAREER award. His current research interests include adaptive middleware and programming technology for distributed computing over wide-area networks. For more information on his group's research, please visit http://wcl.cs.rpi.edu/.