CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2008)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1362

Malleable iterative MPI applications

K. El Maghraoui[1,*,†], Travis J. Desell[2], Boleslaw K. Szymanski[2] and Carlos A. Varela[2]

[1] *IBM T.J. Watson Research Center, Yorktown Heights, New York, NY 10598, U.S.A.*
[2] *Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590, U.S.A.*

SUMMARY

Malleability enables a parallel application's execution system to split or merge processes modifying granularity. While process migration is widely used to adapt applications to dynamic execution environments, it is limited by the granularity of the application's processes. Malleability empowers process migration by allowing the application's processes to expand or shrink following the availability of resources. We have implemented malleability as an extension to the process checkpointing and migration (PCM) library, a user-level library for iterative message passing interface (MPI) applications. PCM is integrated with the Internet Operating System, a framework for middleware-driven dynamic application reconfiguration. Our approach requires minimal code modifications and enables transparent middleware-triggered reconfiguration. Experimental results using a two-dimensional data parallel program that has a regular communication structure demonstrate the usefulness of malleability. Copyright © 2008 John Wiley & Sons, Ltd.

Received 10 December 2007; Revised 9 April 2008; Accepted 7 May 2008

KEY WORDS: dynamic reconfiguration; malleability; MPI

1. INTRODUCTION

Application *reconfiguration* mechanisms are becoming increasingly popular as they enable distributed applications to cope with dynamic execution environments such as non-dedicated clusters and grids. In such environments, traditional application or middleware models that assume dedicated resources or fixed resource allocation strategies fail to provide the desired high performance.

*Correspondence to: K. El Maghraoui, IBM T.J. Watson Research Center, Yorktown Heights, New York, NY 10598, U.S.A.
†E-mail: kelmaghr@us.ibm.com

Contract/grant sponsor: NSF IIS; contract/grant number: 0612213
Contract/grant sponsor: NSF CAREER CNS; contract/grant number: 0448407

Reconfigurable applications enjoy higher application performance because they improve system utilization by allowing more flexible and efficient scheduling policies [1]. Hence, there is a need for new models targeted at both the application-level and the middleware-level that collaborate to adapt applications to the fluctuating nature of shared resources.

Feitelson and Rudolph [2] classify parallel applications into four categories from a scheduling perspective: *rigid*, *moldable*, *evolving*, and *malleable*. Rigid applications require a fixed allocation of processors. Once the number of processors is determined, the application cannot run on a smaller or larger number of processors. Moldable applications can run on various numbers of processors. However, the allocation of processors remains fixed during the runtime of the application. In contrast, both evolving and malleable applications can change the number of processors during execution. In the case of evolving applications, the change is triggered by the application itself. While in malleable applications, it is triggered by an external resource management system. In this paper, we further extend the definition of malleability by allowing the parallel application not only to change the number of processors in which it runs but also to change the granularity of its processes. We demonstrated in previous work [3] that adapting the process-level granularity allows for more scalable and flexible application reconfiguration.

Existing approaches to application malleability have focused on processor virtualization (e.g [4]) by allowing the number of processes in a parallel application to be much larger than the number of available processors. This strategy allows flexible and efficient load balancing through process migration. Processor virtualization can be beneficial as more and more resources join the system. However, when resources slow down or become unavailable, certain nodes can end up with a large number of processes. The node-level performance is then impacted by the large number of processes it hosts because the small granularity of each process causes unnecessary context-switching overhead and increases inter-process communication. On the other hand, having a large process granularity does not always yield the best performance because of the memory-hierarchy. In such cases, it is more efficient to have processes with data that can fit in the lower level of memory caches' hierarchy. To illustrate how the granularity of processes impacts performance, we have run an iterative application with different numbers of processes on the same dedicated node. The larger the number of processes, the smaller the data granularity of each process. Figure 1 shows an experiment where the parallelism of a data-intensive iterative application was varied on a dual-processor node. In this example, having one process per processor did not give the best performance, but increasing the parallelism beyond a certain point also introduces a performance penalty.

Load balancing using only process migration is further limited by the application's process granularity over shared and dynamic environments [3]. In such environments, it is impossible to predict accurately the availability of resources at application's startup and hence determine the right granularity of the application. Hence, an effective alternative is to allow applications' processes to expand and shrink opportunistically as the availability of the resources changes dynamically. Over-estimating by starting with a very small granularity might degrade the performance in the case of a shortage of resources. At the same time, under-estimating by starting with a large granularity might limit the application from potentially utilizing more resources. A better approach is therefore to enable dynamic process granularity changes through malleability.

Message passing interface (MPI) is widely used to build parallel and distributed applications for cluster and grid systems. MPI applications can be moldable. However, MPI does not provide explicit support for malleability and migration. In this paper we focus on the operational aspects

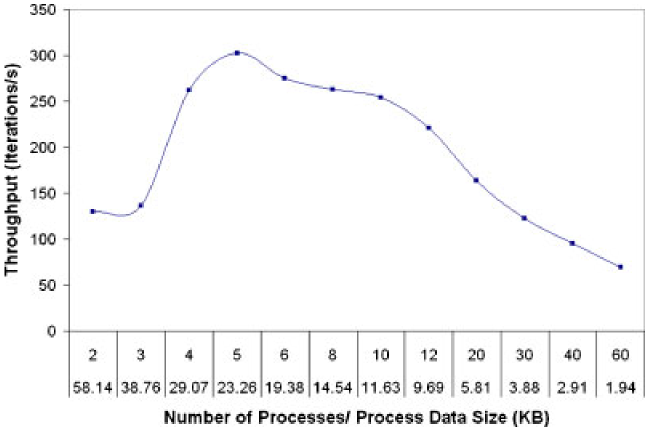

Figure 1. Throughput as the process data granularity decreases on a dedicated node.

of making iterative MPI applications malleable. Iterative applications are a broad and important class of parallel applications that include several scientific applications such as partial differential equation solvers, particle simulations, and circuit simulations. Iterative applications have the property of running as slow as the slowest process. Therefore, they are highly prone to performance degradations in dynamic and heterogeneous environments and will benefit tremendously from dynamic reconfiguration. Malleability for MPI has been implemented in the context of the Internet Operating System (IOS) [5,6] to shift the concerns of reconfiguration from the applications to the middleware. IOS is designed with generic interfaces that allow for various languages and programming architectures to utilize the same autonomous reconfiguration strategies. In previous work [7], we showed how malleability can be implemented in SALSA, a language that implements the Actor programming model, using the same middleware infrastructure.

The remainder of the paper is organized as follows. Section 2 presents the adopted approach of malleability in MPI applications. Section 3 introduces the process checkpointing and migration (PCM) library extensions for malleability. Section 4 discusses the runtime system for malleability. Split and merge policies are presented in Section 5. Section 6 evaluates performance. A discussion of related work is given in Section 7. Section 8 wraps the paper with concluding remarks and discussion of future work.

2. DESIGN DECISIONS FOR MALLEABLE APPLICATIONS

There are operational and meta-level issues that need to be addressed when deciding how to reconfigure applications through malleability and/or migration. Operational issues involve determining how to split and merge the application's processes in ways that preserve the semantics and correctness of the application. The operational issues are heavily dependent on the application's programming model. On the other hand, meta-level issues involve deciding when should a process be split or merged, how many processes to split or merge, and what is the proper mapping of the processes to the physical resources. These issues render programming for malleability and migration a complex task. To facilitate application's reconfiguration from a developer's perspective, middleware technologies need to address meta-level reconfiguration issues. Similarly, libraries need to be developed to address the various operational issues at the application-level. This separation of concerns allows the meta-level reconfiguration policies built into middleware to be widely adopted by various applications.

Several design parameters come to play when deciding how to split and merge an application's parallel processes. Usually there is more than one process involved in the split or merge operations. The simplest scenario is performing binary split and merge, which allows a process to be split into two processes or two processes to merge into one. Binary malleable operations are more intuitive since they mimic the biological phenomena of cell division. They are also highly concurrent since they could be implemented with minimal involvement of the rest of the application. Another approach is to allow a process to be split into N processes or N processes to merge into 1. This approach, in the case of communication-intensive applications, could increase significantly the communication overhead and could limit the scalability of the application. It could also easily cause data imbalances. This approach would be useful when there are large fluctuations in resources. The most versatile approach is to allow for collective split and merge operations. In this case, the semantics of the split or merge operations allow any number of M processes to be split or merged into any other number of N processes. Figures 2 and 3 illustrate example behaviors of split and merge operations. In the case of the M to N approach, data are redistributed evenly among the resulting processes when splitting or merging. What type of operation is more useful depends on the nature of applications, the degree of heterogeneity of the resources, and how frequently the load fluctuates.

While process migration changes mapping of an application's processes to physical resources, split and merge operations go beyond that by changing the communication topology of the application, the data distribution, and the data locality. Splitting and merging cause the communication topology of the processes to be modified because of the addition of new or removal of old processes, and the data redistribution among them. This reconfiguration needs to be done atomically to preserve application semantics and data consistency.

We provide high-level operations for malleability based on the MPI paradigm for SPMD data parallel programs with regular communication patterns. The proposed approach is high level because in that the programmer is not required to specify when to perform split and merge operations

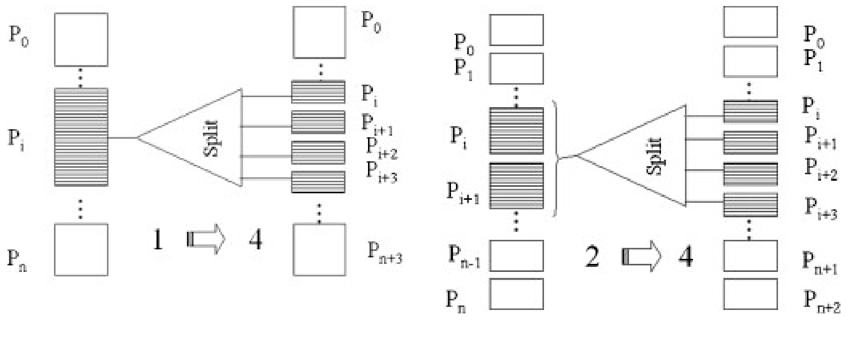

Figure 2. Example M to N split operations.

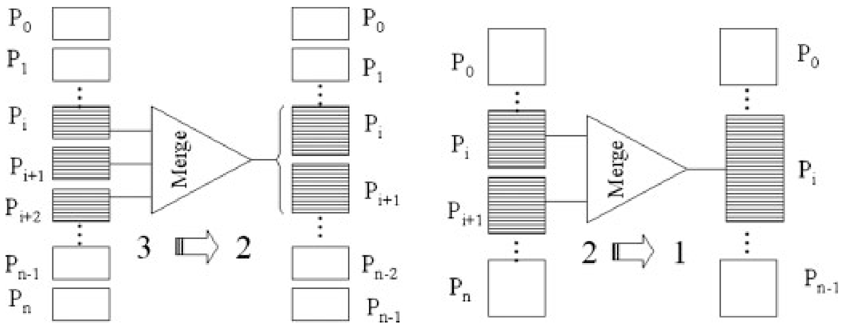

Figure 3. Example M to N merge operations.

and some of the intrinsic details that are involved in re-arranging the communication structures explicitly: these are provided by the PCM library. The programmer needs, however, to specify the data structures that will be involved in the malleability operations. Since there are different ways of subdividing data among processes, programmers also need to guide the split and merge operations for data-redistribution.

3. MODIFYING MPI APPLICATIONS FOR MALLEABILITY

In previous work [5], we have designed and implemented an application-level checkpointing application programming interface (API) called PCM and a runtime system called PCM daemon (PCMD). Few PCM calls need to be inserted in MPI programs to specify the data that need to be checkpointed, to restore the process to its previous state in the case of migration, to update the data distribution structure and the MPI communicator handles, and to probe the runtime system for reconfiguration requests. This library is semi-transparent because the user does not have to worry about when or how checkpointing and restoration are done. The underlying PCMD infrastructure takes care of all the checkpointing and migration details. This study extends the PCM library with malleability features.

PCM is implemented entirely in the user-space for portability of the checkpointing, migration, and malleability schemes across different platforms. PCM has been implemented on top of MPICH2 [8], a free available implementation of the MPI-2 standard.

3.1. The PCM API

PCM has been extended with several routines for splitting and merging MPI processes. We have implemented split and merge operation for data parallel programs with a 2D data structure and a linear communication structure. Figure 4 shows the parallel decomposition of the 2D data structure and the communication topology of the parallel processes. Our implementation allows for common data distributions such as block, cyclic, and block–cyclic distributions.

PCM provides fours classes of services: environmental inquiry services, checkpointing services, global initialization and finalization services, and collective reconfiguration services. Table I shows

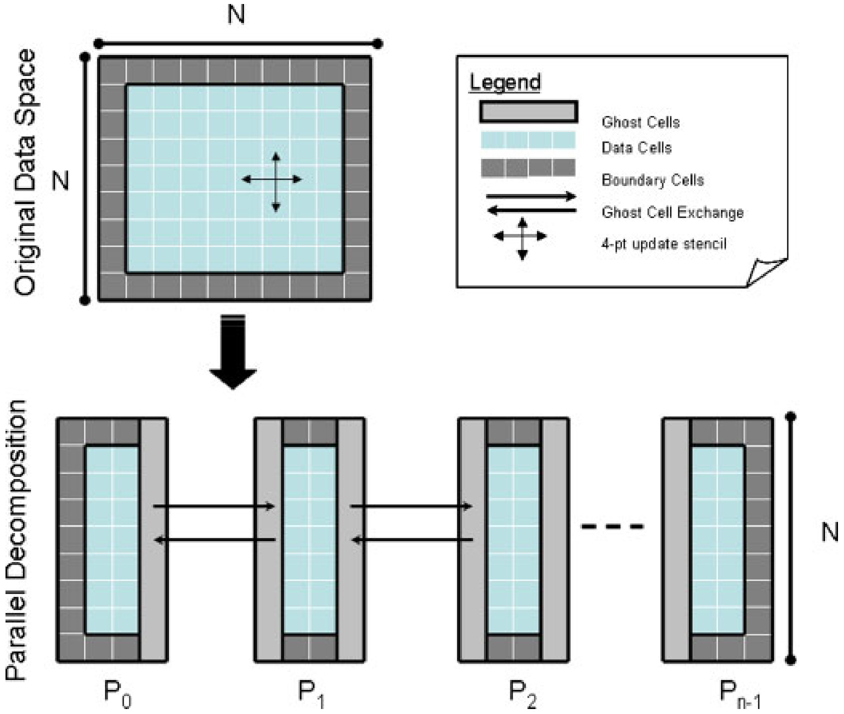

Figure 4. Parallel domain decomposition of a regular 2D problem.

Table I. 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_Merge PCM_Split_Collective PCM_Merge_Collective
Checkpointing	PCM_Load, PCM_Store

the classification of the PCM API calls. 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 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 are stored in locations with reasonable proximity to their destination.

Upon 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 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 the 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.

The running application probes the PCMD system to check whether a process or a group of processes need to be reconfigured. Middleware notifications set global flags in the PCMD system. To prevent every process from probing the runtime system, the root process (usually process with rank 0) 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, whereas 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 is returned by the PCM_Status function call in the case of a split or merge operation. All processes collectively call the PCM_Split or PCM_Merge functions to perform a malleable reconfiguration.

We have implemented the 1 to N and M to N split and merge operations. PCM_Split and PCM_Merge provide the 1 to N behavior, whereas PCM_Split_Collective and PCM_Merge_Collective provide the M to N behavior. The values of M and N are transparent to the programmer. They are provided by the middleware that decides the granularity of the split operation.

Split and merge functions change the ranks 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. Every data chunk is associated with a unique key that is constructed from the process's rank and a data qualifier. Every PCMD maintains a local database that stores checkpoints for the processes that are running in its local processor. The data associated with the new processes to be created are migrated to their target processors' PCMD databases. When the new processes are created, they inquire about their new ranks and load their data chunks from their local PCMD using their data chunk keys. 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 processes from the MPI communicator. Merging operations for data redistribution are implemented using the MPI scatter and gather operations.

3.2. Instrumenting an MPI program with PCM

Figure 5 shows a sample skeleton of an MPI-based application with a very common structure in iterative applications. The code starts by performing various initializations of some data structures. Data are 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. Figures 6 and 7 show the same application instrumented with PCM calls to

```
#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_Barrier( MPI_COMM_WORLD );

   MPI_Finalize();
   return 0;
}
```

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

```
#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;

    PCM_MPI_Init(&argc, &argv);
    PCM_COMM_WORLD = MPI_COMM_WORLD;
    PCM_Init(PCM_COMM_WORLD);
    MPI_Comm_rank(PCM_COMM_WORLD, &rank );
    MPI_Comm_size(PCM_COMM_WORLD, &totalProcessors );
    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);
    }
            ...
            ...
}
```

Figure 6. Skeleton of the malleable MPI code with PCM calls: initialization phase.

allow for migration and malleability. In the 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 dimensions and update them appropriately. In the case of a merge operation, the size of the buffer needs to be known so that enough memory can be allocated. The PCM_Merge_datacnts function is used to retrieve the new buffer size. This call is only meaningful for 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 variable merge_rank will have a valid process rank in the case the calling process is merging, otherwise it has the value -1.

The example shows how to instrument MPI iterative applications with PCM calls. The programmer is required only to know the right data structures that are needed for malleability. A PCM-instrumented MPI application becomes malleable and ready to be reconfigured by the IOS middleware.

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

Figure 7. Skeleton of the malleable MPI code with PCM calls: iteration phase.

4. THE RUNTIME ARCHITECTURE

IOS [6] provides several reconfiguration mechanisms that allow (1) analyzing profiled application communication patterns, (2) capturing the dynamics of the underlying physical resources, and (3) utilizing the profiled information to reconfigure application entities by changing their mappings to physical resources through migration or malleability. IOS adopts a decentralized strategy that avoids the use of any global knowledge to allow scalable reconfiguration. An IOS system consists of collection of autonomous agents with a peer-to-peer topology.

MPI/IOS is implemented as a set of middleware services that interact with running applications through an MPI wrapper. The MPI/IOS runtime architecture consists of the following components: (1) the PCM-enabled MPI applications, (2) the wrapped MPI that includes the PCM API, the

PCM library, and wrappers for all MPI native calls, (3) the MPI library, and (4) the IOS runtime components.

4.1. The PCMD runtime system

Figure 8 shows an MPI/IOS computational node running MPI processes. A PCMD interacts with the IOS middleware and MPI applications. A PCMD is started in every node that actively participates in an application. A PCM dispatcher is used to start PCMDs in various nodes and is used to discover the existing ones. The application initially registers all MPI processes with their local daemons. The port number of a daemon is read from a configuration file that resides in the same host.

Every PCMD has a corresponding IOS agent. There can be more than one MPI process in each node. The daemon consists of various services used to achieve process communication profiling, checkpointing, migration, and malleability. The MPI wrapper calls record information pertaining to how many messages have been sent and received and their source and target process ranks. The profiled communication information is passed to the IOS profiling component. IOS agents keep monitoring their underlying resources and exchanging information about their respective loads.

When a node's used resources fall below a predefined threshold or a new idle node joins the computation, a Work-Stealing Request Message (WRM) is propagated among the actively running nodes. The IOS agent of a node responds to work-stealing requests if it becomes overloaded and its decision component decides according to the resource-sensitive model that process(es) need(s) to be migrated. Otherwise, it forwards the request to an IOS agent in its set of peers. The

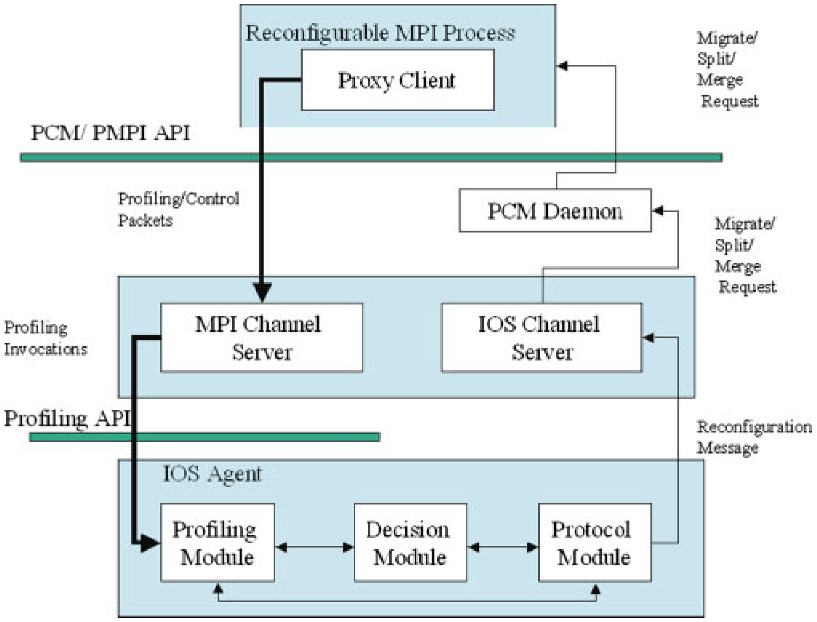

Figure 8. The PCM/IOS runtime environment.

DOI: 10.1002/cpe

decision component then notifies the reconfiguration service in the PCMD, which then sends a migration, split, or merge request to the desired process(es). At this point, all active PCMDs in the system are notified about the event of a reconfiguration. This causes all processes to cooperate in the next iteration until migration is completed and application communicators have been properly updated. Although this mechanism imposes some synchronization delay, it ensures that no messages are being exchanged while process migration is taking place and avoids incorrect behaviors of the MPI communicators.

4.2. The profiling architecture

MPI processes need to send periodically their communication patterns to their corresponding IOS profiling agents. To achieve this, we have built a profiling library that is based on the MPI profiling interface (PMPI). The MPI specification provides a general mechanism for intercepting calls to MPI functions using name shifting. This allows the development of portable performance analyzers and other tools without access to the MPI implementation source code. The only requirement is that every MPI function be callable by an alternate name (`PMPI_Xxxx` instead of the usual `MPI_Xxxx`.). The built profiling library intercepts all communication methods of MPI and sends any communication event to the profiling agent.

All profiled MPI routines call their corresponding `PMPI_Xxxx` and, if necessary, PCM routines. Figure 9 shows the library structure of the MPI/IOS programs. The instrumented code is linked with the profiling library PMPI, the PCM library, and a vendor MPI implementation's library. The generated executable passes all profiled information to the IOS run-time system and also communicates with the local PCMD. The latter is responsible for storing local checkpoints and passing reconfiguration decisions across a socket API from the IOS agent to the MPI processes.

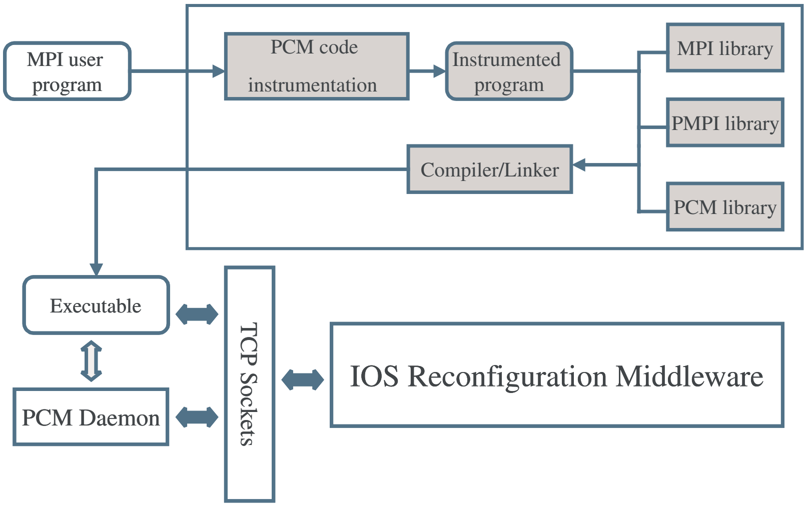

Figure 9. Library and executable structure of an MPI/IOS application.

DOI: 10.1002/cpe

5. MALLEABILITY POLICIES

5.1. Transfer policy

The purpose of the transfer policy is to determine when to transfer load from one agent to another and how much load needs to be transferred, whenever a WRM is sent from an agent n_j to an agent n_i. We identify the load of a given agent simply by the number of running application's processes hosted by this agent. We denote by Nb_i the number of application's processes running on agent n_i before a given reconfiguration step and by Na_i the number of application's processes running on agent n_i after a given reconfiguration step, since processes may have different granularities. We measure the number of processes in units of the process with the smallest data sizes. For example, if two processes are running and one of them has twice the size of the other, the total number of processes will be reported as 3. This accounts for the heterogeneous sizes of processes and simplifies our analysis. Let APW_i be the percentage of the available CPU-processing power of agent n_i and UPW_j be the percentage of the used CPU-processing power of agent n_j. Let PW_i and PW_j be the current processing powers of agents n_i and n_j, respectively. We use the Network Weather Service [9] to measure the values of APW and UPW. The transfer policy tries to adjust the load between two peer agents based on their relative machine performances as shown in the equations below:

$$N_{\text{total}} = Nb_i + Nb_j = Na_i + Na_j \tag{1}$$

$$\frac{Na_i}{APW_i * PW_i} = \frac{Na_j}{UPW_j * PW_j} \tag{2}$$

$$Na_i = \frac{APW_i * PW_i}{APW_i * PW_i + UPW_j * PW_j} * N_{\text{total}} \tag{3}$$

$$Na_j = \frac{UPW_j * PW_j}{APW_i * PW_i + UPW_j * PW_j} * N_{\text{total}} \tag{4}$$

$$N_{\text{transfer}} = Na_j - Nb_j \tag{5}$$

Equation (5) allows us to calculate the number of processes that need to be transferred to remote agent n_i to achieve load balance between n_i and n_j. All the processes in host n_j are ranked according to a heuristic decision function [6] that calculates their expected gain from moving from agent n_i to agent n_j. Only the processes that have a gain value greater than a threshold value θ are allowed to migrate to the remote agent. So the number of processes that will migrate can be less than N_{transfer}. The goal of the gain value is to select the candidate processes that benefit the most from migration to the remote host.

5.2. Split and merge policies

5.2.1. The split policy

The transfer policy discussed above shows how the load in two agents needs to be broken up to reach pair-wise load balance. However, this model will fail when there are not enough processes to

DOI: 10.1002/cpe

make such load adjustments. For example, assume that a local node n_l has only one entity running. Assume also that n_r, a node that is three times faster than n_l, requests some work from n_l. Ideally, we want node n_r to have three times more processes or load than node n_l. However, the lack of enough processes prevents such adjustment. To overcome this situation, the entity running in node n_l needs to be split into enough processes to send a proportional number remotely.

Let N_{total} be the total number of processes running in both n_l and n_r. Let n be the desired total number of processes, l be the desired number of processes at local node n_l, and r be the desired number of processes at node n_r. APW_r and PW_r denote the percentage of the available processing power and the current processing power of node n_r, respectively. UPW_l and PW_l denote the percentage of the used processing power and current processing power of node a_l. The goal is to minimize n subject to constraints shown in the set of Equations (6)–(12) and to solve for n, l, and r in the set of positive natural numbers IN^+

$$n - l - r = 0 \tag{6}$$

$$\frac{APW_r * PW_r}{APW_r * PW_r + UPW_l * PW_l} * n - r = 0 \tag{7}$$

$$\frac{UPW_l * PW_l}{APW_r * PW_r + UPW_l * PW_l} * n - l = 0 \tag{8}$$

$$n \geqslant N_{\text{total}} \tag{9}$$

$$n \in IN^+ \tag{10}$$

$$r \in IN^+ \tag{11}$$

$$l \in IN^+ \tag{12}$$

A split operation happens when $n > N_{\text{total}}$ and the entity can be split into one or more processes. In this case the number of processes in local node will be split refining the granularity of the application's processes running in the local node n_l.

5.2.2. The merge policy

The merge operation is a local operation. It is triggered when a node has a large number of running processes and the operating system's context switching is large. To avoid a thrashing situation that causes processes to be merged and then split again upon receiving a WRM, merging happens only when the surrounding environment has been stable for a while. Every peer in the virtual network tries to measure how stable it is and how stable its surrounding environment is.

Let $S_i = (APU_{i,0}, APW_{i,1}, \ldots, APW_{i,k})$ be a time series of the available CPU-processing power of an agent n_i during different consecutive k measurement intervals. Let avg_i denote the average available CPU-processing power of series S_i (see Equation (13)). We measure the stability value σ_i of agent n_i by calculating the standard deviation of the series S_i (see Equation (14))

$$avg_i = \frac{1}{k} * \sum_{t=0}^{k} APW_{i,t} \tag{13}$$

$$\sigma_i = \sqrt{\frac{1}{k} * \sum_{t=0}^{k} (APW_{i,t} - avg_i)^2} \tag{14}$$

$$avg = \frac{1}{p} * \sum_{i=0}^{p} \sigma_i \tag{15}$$

$$\sigma = \sqrt{\frac{1}{p} * \sum_{i=0}^{p} (\sigma_i - avg)^2} \tag{16}$$

A small value of σ_i indicates that there has not been much change in the CPU utilization over previous periods of measurements. This value is used to predict how the utilization of the node is expected to be in the near future. However, the stability measure of the local node is not enough since any changes in the neighboring peers might trigger a reconfiguration. Therefore, the node also senses how stable its surrounding environment is. The stability value is also carried in the WRMs within the machine performance profiles. Therefore, every node records the stability values σ_j of its peers. The nodes periodically calculate σ (see Equation (16)), the standard deviation of the σ_j's of its peers.

A merging operation is triggered only when σ_i and σ are small, $\sigma_i < \varepsilon_1$ and $\sigma < \varepsilon_2$. In other words, the local node attempts to perform a merge operation when possible if its surrounding environment is expected to be stable and the context-switching rate of the host operating system is higher than a given threshold value. The OS context-switching rates can be measured using tools such as the Unix `vmstat` command.

6. PERFORMANCE RESULTS

6.1. Application case study

We have used a fluid dynamic problem that solves heat diffusion in a solid for testing purposes. This applications is representative of a large class of highly synchronized iterative mesh-based applications. It has been implemented using C and MPI and has been instrumented with PCM library calls. We have used a simplified version of this problem to evaluate our reconfiguration strategies. A 2D mesh of cells is used to represent the problem data space. The cells are uniformly distributed among the parallel processors. At the beginning, a master process takes care of distributing the data among processors. For each iteration, the value of each cell is calculated based on the values of its neighbor cells. So each cell needs to maintain a current version of them. To achieve this, processors exchange values of the neighboring cells, also referred to as ghost cells. To sum up, every iteration consists of doing computation and exchanging ghost cells from the neighboring processors.

For the experimental test bed we used a heterogeneous cluster that consists of four dual-processor SUN Blade 1000 machines with a processing speed of 750M cycles s^{-1} and 2 GB of memory and 18 single-processor SUN Ultra 10 machines with a processing speed of 360M cycles s^{-1} and 256 MB of memory. The SUN Blade machines are connected with high-speed gigabit ethernet, whereas the SUN Ultra machines are connected with 100 MB ethernet. For comparative purposes, we used MPICH2 [8], a free implementation of the MPI-2 standard. We run the heat simulation for 1000 iterations with 1000 × 1000 mesh and a total data size of 7.8 MB.

DOI: 10.1002/cpe

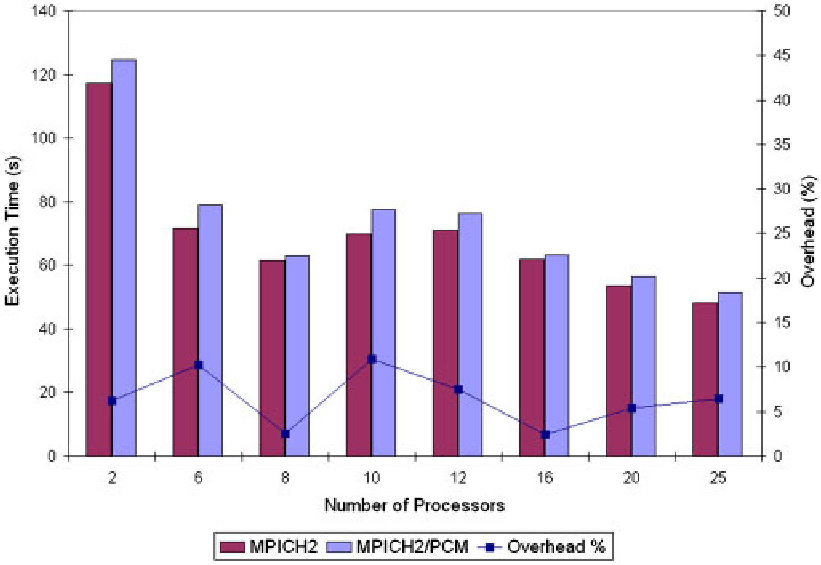

Figure 10. Overhead of the PCM library.

6.2. Overhead evaluation

To evaluate the overhead of the PCM profiling and status probing, we have run the heat diffusion application with the base MPICH2 implementation and with the PCM instrumentation. We ran the simulation with 40 processes on different numbers of processors. Figure 10 shows that the overhead of the PCM library does not exceed 11% of the application's running time. The measured overhead includes profiling, status probing, and synchronization. The library supports tunable profiling, whereby the degree of profiling can be decreased by the user to reduce its intrusiveness.

For a more in-depth evaluation of the cost of reconfiguration and the overhead of the PCM/IOS reconfiguration, we conducted an experiment that compares a reconfigurable execution scenario with a baseline MPICH2 execution scenario. In the conducted experiments, the application was started on a local cluster. Artificial load was then introduced in one of the participating machines. Another cluster was made available to the application. The baseline implementation using MPICH2 was not able to reconfigure the running application, while the PCM/IOS implementation managed to reconfigure the application by migrating the affected processes to the second cluster. The experiments in Figures 11 and 12 show that in the studied cases, reconfiguration overhead was negligible. In all cases, it accounted for less than 1% of the total execution time. We also used an experimental testbed that consisted of two clusters that belong to the same institution. So the network latencies were not significant. The reconfiguration overhead is expected to increase with larger latencies and larger data sizes. However, reconfiguration will still be beneficial in the case of large-scale long-running applications. Figure 12 shows the breakdown of the reconfiguration cost. The overhead measured consisted mainly of the costs of checkpointing, migration, and the synchronizations involved in re-arranging the MPI communicators. Owing to the highly synchronous nature of this

DOI: 10.1002/cpe

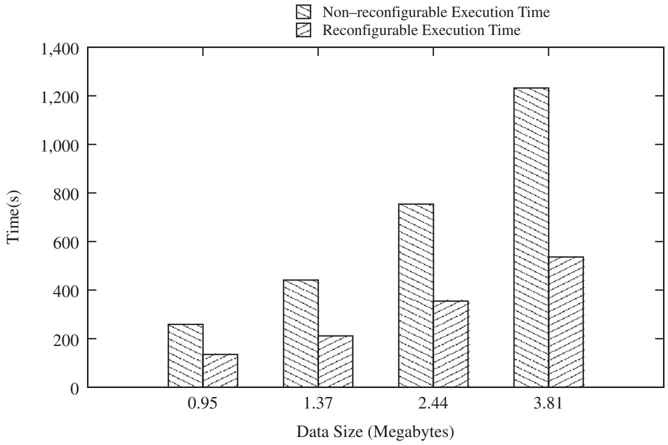

Figure 11. Total running time of reconfigurable and non-reconfigurable execution scenarios for different problem data sizes for the heat diffusion application.

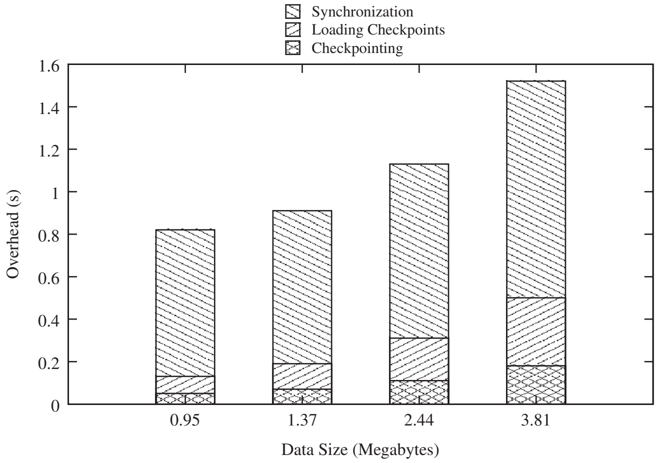

Figure 12. Breakdown of the reconfiguration overhead for the experiment of Figure 11.

application, communication profiling was not used because a simple decision function that takes into account the profiling of the CPU usage was enough to yield good reconfiguration decisions.

6.3. Split/merge features

An experiment was set up to evaluate the split and merge capabilities of the PCM malleability library. The heat diffusion application was started initially on eight processors with a configuration of one process per processor. Then, eight additional processors at iteration 860 were made available.

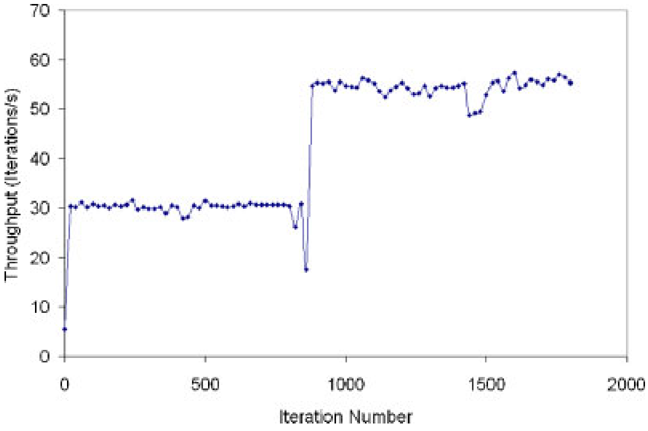

Figure 13. Expansion and shrinkage capabilities.

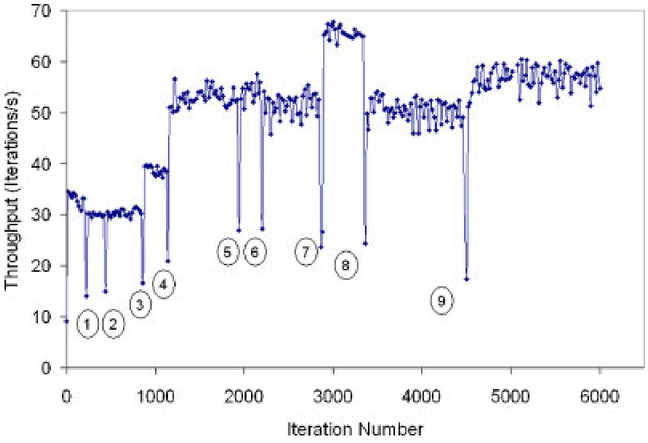

Figure 14. Adaptation using malleability and migration.

The eight additional processes were split and migrated to harness the newly available processors. Figure 13 shows the immediate performance improvement that the application experienced after this expansion. The sudden drop in the application's throughput at iteration 860 is due to the overhead incurred by the split operation. The collective split operation was used in this experiment because of the large number of resources that have become available. The small fluctuations in the throughput are due to the shared nature of the cluster used for experiments.

6.4. Gradual adaptation with malleability and migration

The experiment shown in Figure 14 illustrates the usefulness of having the 1 to N split and merge operations. When the execution environment experiences small load fluctuations, a gradual

DOI: 10.1002/cpe

adaptation strategy is needed. The heat application was launched on a dual-processor machine with two processes. Two binary split operations occurred at events 1 and 2. The throughput of the application decreased a bit because of the decrease in 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 then two migrations to another dual-processor node at event 7. An increase in throughput was noticed after the migration at event 7 due to a better distribution of work. A node left at event 8 caused two processes to be migrated to one of the participating machines. A merge operation happened at event 9 in the node with excess processes, which improved the application's throughput.

7. RELATED WORK

Malleability for MPI applications has been ma inly addressed through processor virtualization, dynamic load balancing strategies, and application stop and restart.

Adaptive MPI (AMPI) [4] is an implementation of MPI built on top of the Charm + + runtime system, a parallel object-oriented library with object migration support. AMPI leverages Charm++ dynamic load balancing and portability features. Malleability is achieved in AMPI by starting the applications with a very fine process granularity and relying on dynamic load balancing to change the mapping of processes to physical resources through object migration. The PCM/IOS library and middleware support provide both migration and process granularity control for MPI applications. Phoenix [10] is another programming model that allows virtualization for a dynamic environment by creating extra initial processes and uses a virtual name space and process migration to balance load and scale applications.

The EasyGrid middleware [11] embeds a hierarchical scheduling system into MPI applications with the aim of efficiently orchestrating the execution of MPI applications in grid environments. In this study a hybrid of static and dynamic scheduling policies are utilized to map MPI processes to grid resources initially. The number of MPI processes in this scheme remains the same throughout the execution of the application. PCM/IOS allows for more flexible scheduling policies because of the added value of split and merge capabilities.

In [1], the authors propose virtual malleability for message passing parallel jobs. They apply a processor allocation strategy called the Folding by JobType (FJT) that allows MPI jobs to adapt to load changes. The folding technique reduces the partition size in half, duplicating the number of processes per processor. In contrast to our work, the MPI jobs are only simulated to be malleable by using moldability and the folding technique.

Process swapping [12] 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. The process granularity in this approach is fixed. Our approach is different in that we do not need to over-allocate resources initially. The over-allocation strategy in process swapping may not be practical in highly dynamic environments where an initial prediction of resources is not possible because of the constantly changing availability of the resources. Dyn-MPI [13] is another system that extends iterative MPI programs with adaptive execution features in non-dedicated environment through data redistribution and the possibility of removing badly performing nodes. In contrast to

DOI: 10.1002/cpe

our scheme, Dyn-MPI does not support the dynamic addition of new processes. In addition, Dyn-MPI relies on a centralized approach to determine load imbalances while we utilize decentralized load balancing policies [6] to trigger malleable adaptation.

Checkpointing and application stop and restart strategies have been investigated as malleability tools in dynamic environments. Examples include CoCheck [14], starFish [15], and the SRS library [16]. Stop and restart are expensive especially for applications operating on large data sets. The SRS library provides tools to allow an MPI program to stop and restart where it left off with a different process granularity. Our approach is different in the sense that we do not need to stop the entire application to allow for change of granularity.

8. CONCLUSIONS AND FUTURE WORK

The paper describes the PCM library framework for enabling MPI applications to be malleable through split, merge, and migrate operations. The implementation of malleability operations is described and illustrated through an example of a communication-intensive iterative application. Different techniques for split and merge are presented and discussed. Collective malleable operations are more appropriate in dynamic environments with large load fluctuations, whereas individual split and merge operations are more appropriate in environments with small load fluctuations. Our performance evaluation has demonstrated the usefulness of malleable operations in improving the performance of iterative applications in dynamic environments.

This paper has mainly focused on the operational aspect of implementing malleable functionalities for MPI applications. The performance evaluation experiments that we conducted were done using small- to medium-sized clusters. Future work should address the scalability aspects of our malleable reconfiguration. IOS reconfiguration decisions are all based on local or neighboring node information and use decentralized protocols. Therefore, we expect our scheme to be scalable in larger environments. Future work aims also at improving the performance of the PCM library, and thoroughly evaluating the devised malleability policies that decide when it is appropriate to change the granularity of the running application, what is the right granularity, and what kind of split or merge behavior to select. Future work includes also devising malleability strategies for non-iterative applications.

ACKNOWLEDGEMENTS

This work has been partially supported by NSF IIS No. 0612213 Grant and NSF CAREER CNS Award No. 0448407. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Utrera G, Corbalán J, Labarta J. Implementing malleability on MPI jobs. *IEEE PACT*. IEEE Computer Society: Silver Spring, MD, 2004; 215–224.
2. Feitelson DG, Rudolph L. Towards convergence in job schedulers for parallel supercomputers. *JSSPP* (*Lecture Notes in Computer Science*, vol. 1162), Feitelson DG, Rudolph L (eds.). Springer: Berlin, 1996; 1–26.

3. Desell T, Maghraoui KE, Varela C. Malleable components for scalable high performance computing. *Proceedings of HPDC'15 Workshop on HPC Grid Programming Environments and Components* (*HPC-GECO/CompFrame*), Paris, France, June. IEEE Computer Society: Silver Spring, MD, 2006; 37–44.
4. Huang C, Zheng G, Kalé L, Kumar S. Performance evaluation of adaptive MPI. *PPoPP '06*: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press: New York, NY, U.S.A., 2006; 12–21.
5. Maghraoui KE, Szymanski B, Varela C. An architecture for reconfigurable iterative MPI applications in dynamic environments. *Proceedings of the Sixth International Conference on Parallel Processing and Applied Mathematics* (*PPAM'2005*) (*Lecture Notes in Computer Science*, vol. 3911), Poznan, Poland, September, Wyrzykowski R, Dongarra J, Meyer N, Wasniewski J (eds.). Springer: Berlin, 2005; 258–271.
6. Maghraoui KE, Desell TJ, Szymanski BK, Varela CA. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications* (*IJHPCA*), *Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms* 2006; **20**(4):467–480.
7. Desell T, Maghraoui KE, Varela C. Malleable applications for scalable high performance computing. *Cluster Computing* 2007; **10**(3):323–337.
8. Argone National Laboratory. MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2 [4 April 2008].
9. Wolski R. Dynamically forecasting network performance using the network weather service. *Cluster Computing* 1998; **1**(1):119–132.
10. Taura K, Kaneda K, Endo T. Phoenix: A parallel programming model for accommodating dynamically joining/leaving resources. *Proceedings of PPoPP*. ACM: New York. 2003; 216–229.
11. Nascimento AP, Sena AC, Boeres C, Rebello VEF. Distributed and dynamic self-scheduling of parallel MPI grid applications: Research articles. *Concurrency and Computation*: *Practice and Experience* 2007; **19**(14):1955–1974.
12. Sievert O, Casanova H. A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications* 2004; **18**(3):341–352.
13. Weatherly DB, Lowenthal DK, Nakazawa M, Lowenthal F. Dyn-MPI: Supporting MPI on non dedicated clusters. *SC '03*: *Proceedings 2003 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society: Washington, DC, U.S.A., 2003; 5.
14. Stellner G. Cocheck: Checkpointing and process migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society: Silver Spring, MD. 1996; 526–531.
15. Agbaria A, Friedman R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society: Silver Spring, MD, 1999; 31.
16. Vadhiyar SS, Dongarra J. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters* 2003; **13**(2):291–312.