Dynamic Malleability in MPI Applications

Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590, USA
{*elmagk,deselt,szymansk,cvarela*}*@cs.rpi.edu*

Abstract

Malleability enables a parallel application's execution system to split or merge processes modifying the parallel application's 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 PCM (Process Checkpointing and Migration) library, a user-level library for iterative MPI applications. PCM is integrated with the Internet Operating System (IOS), a framework for middleware-driven dynamic application reconfiguration. Our approach requires minimal code modifications and enables transparent middleware-triggered reconfiguration. We present experimental results that demonstrate the usefulness of malleability.

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 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 that are being made available by computational grids. Reconfigurable applications offer improved application performance. They also offer better overall system utilization since they allow more flexible and efficient scheduling policies [11]. 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 grid resources.

Feitelson and Rudolph [3] provide an interesting classification of 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 by the user, 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 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 [2] 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 [5]) by allowing the number of processes in a parallel application to be much larger than the available number of 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 of the small granularity of each process which 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

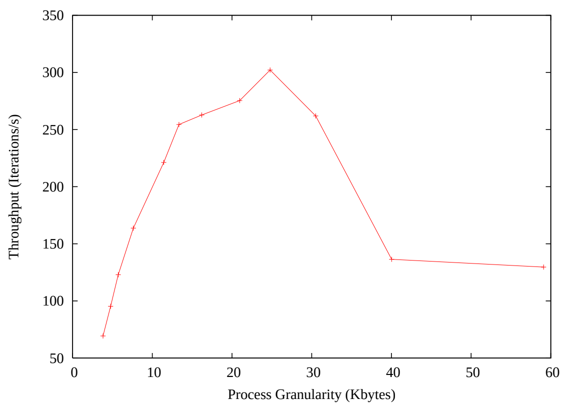

Figure 1. Throughput of an iterative application as the data granularity of the processes increases on a dedicated dual-processor node.

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 node 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 [2]. 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 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. The best approach is therefore to enable dynamic process granularity changes through malleability.

MPI (Message Passing Interface) 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 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, heat-wave 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 IOS [7, 6] to shift the concerns of reconfiguration from the applications to the middleware.

The rest of the paper is organized as follows. Section 2 presents the adopted approach of malleability in MPI applications. Section 3 introduces the PCM library extensions for malleability. Section 4 discusses the runtime system for malleability. Section 5 presents performance evaluation. A discussion of related work is given in Section 6. Section 7 wraps the paper with concluding remarks and discussion of future work.

2 Design Decisions for Malleable Applications

There are operational and behavioral issues that need to be addressed when deciding how to reconfigure applications though 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 model. On the other hand, behavioral issues decide when should a process split or merge, how many more processes to split and how many should 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 reconfiguration behavioral issues. Similarly, libraries need to be developed to address the various operational issues at the application-level. This separation of concerns allows the middleware-level reconfiguration policies 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 merge 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 a minimal involvement of the rest of the application. Another approach is to allow a process to merge into N processes and 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

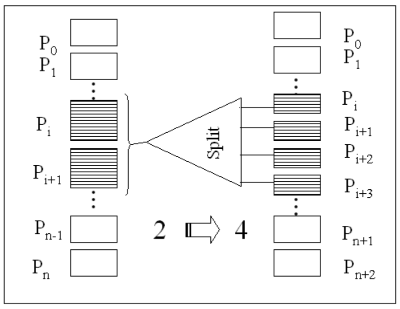

Figure 2. An example of the M to N split operation.

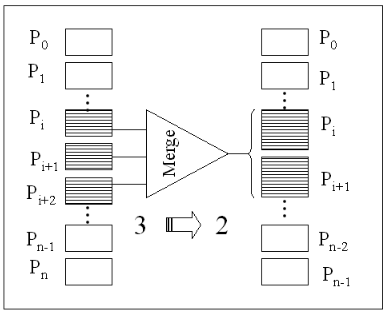

Figure 3. An example of the M to N merge operation.

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 split or merge 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 is redistributed evenly among the resulting processes when splitting. 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 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 causes 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.

In this work we address split and merge for SPMD data parallel programs with regular communication patterns. We provide high-level operations for malleability based on the MPI paradigm. Our approach is high level in that the programmer is not required to specify when to perform split and merge operations and some of the intrinsic details in-

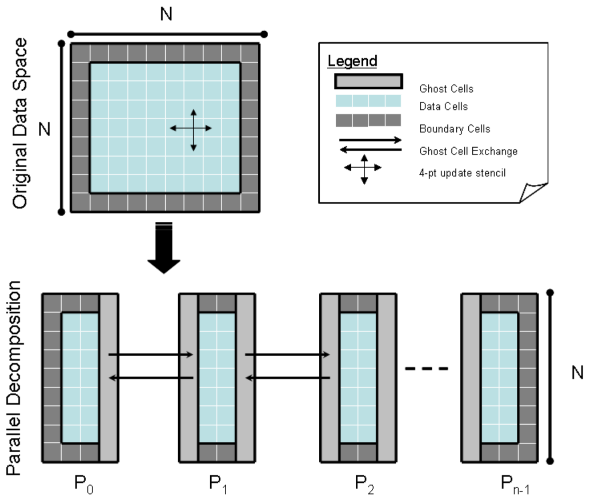

Figure 4. Parallel domain decomposition of a regular 2-dimensional problem

volved in re-arranging the communication structures explicitly: these are provided by the built-in PCM library. The programmer needs only to specify the domain decomposition for the particular application in hand. There are different ways of subdividing data among processes, it is imperative for programmers to guide the split and merge operations for data-redistribution.

3 Modifying MPI Applications for Malleability

PCM (Process Checkpointing and Migration) [7] is a library that allows iterative MPI programs to be dynamically reconfigurable by providing the necessary tools for checkpointing, and migration of MPI processes. PCM is implemented entirely in the user-space to allow portability of the used checkpointing and migration scheme across different platforms.

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 like 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

Table 1. 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

services. Table 1 shows 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 PCM daemon, 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.

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 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 Figure 6.

The running 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 root process which is 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, 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.

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, while PCM_Split_Collective and PCM_Merge_Collective provide the N to M behavior. The middleware in notified about which form of malleability operation to use implicitly. The values of M and N are transparent to the programmer. They are provided by the middleware which decides the granularity of the split operation.

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.

```
#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.

3.2 An Example Application

A sample skeleton of a simple MPI-based application is given in Figure 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 dimensions 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 instru-

```
#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
   ...
   MPI_Barrier( PCM_COMM_WORLD );

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

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

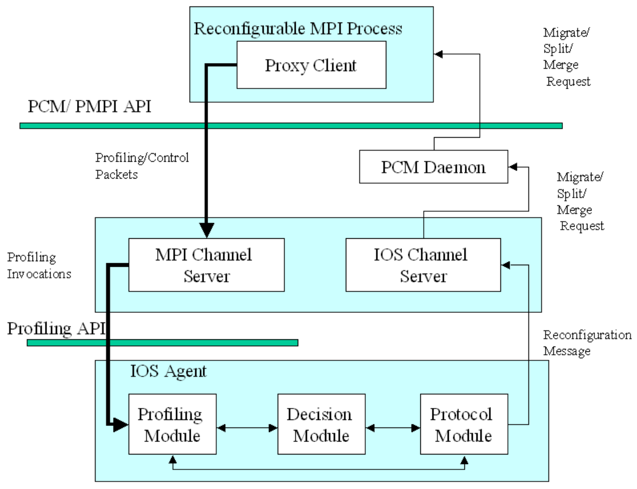

Figure 7. Architecture of the PCM/IOS run-time environment.

mentations, the MPI application becomes malleable and ready to be reconfigured by IOS middleware.

4 Middleware Services for Malleability

The PCM Daemon (PCMD) exists during the entire duration of the application and spans across several reconfigurations. The PCMD is responsible for handling checkpointing services to the MPI processes running in their cluster and forwarding reconfiguration requests. The PCMD needs to be launched by the user before running the MPI application. The location of the central PCMD and its port number should be written in a configuration file that needs to be accessible to the application prior to its startup. Every node that potentially could host an MPI process, also needs to have a local IOS agent.

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. 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.

The PCM library provides also profiling services that are based on the MPI profiling interface (PMPI). The profiling library gathers information about the communication topology of MPI processes, the size of data being transfered, and the iteration times. The profiled information is sent periodically to the IOS agent to help analyze the performance of the running process, detect any performance degradations, and decide how to reconfigure the application to improve

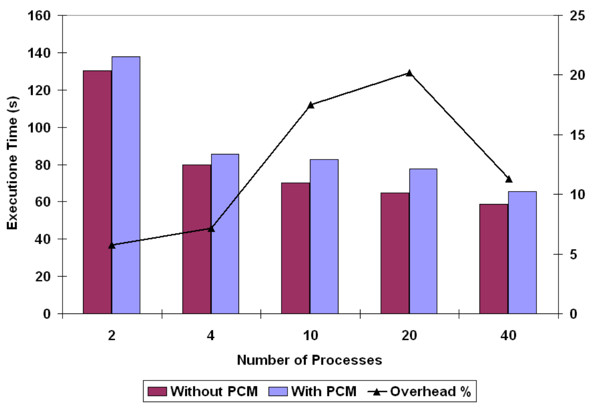

Figure 8. Overhead of the PCM library with malleability

performance. The interactions between the reconfigurable MPI processes and the IOS middleware are shown in Figure 7. MPI/IOS transparently leverages the dynamic reconfiguration features of IOS modules.

5 Performance Results

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 the large class of highly synchronized iterative mesh-based applications. The application 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 two-dimensional mesh of cells is used to represent the problem data space. The mesh initially contains the initial values of the mesh with the boundary values. 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 exchange of ghost cells from the neighboring processors.

Overhead Evaluation. 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

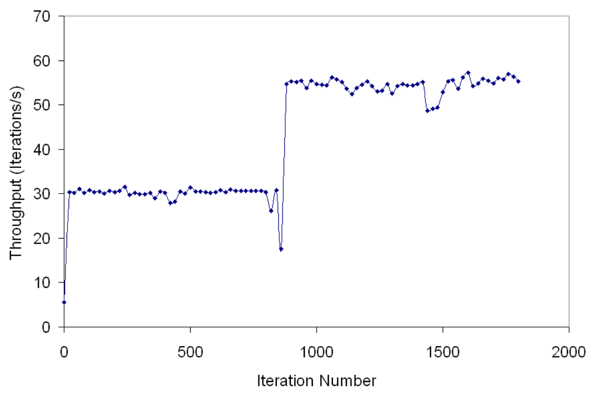

Figure 9. The experiment illustrates the expansion and shrinkage capability of the PCM library.

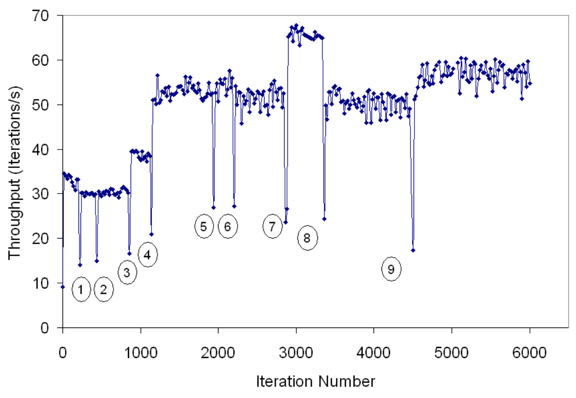

Figure 10. Gradual adaptation using malleability and migration as resources leave and join

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

Split/Merge Features. An experiment was setup to evaluate the split and merge capabilities of the PCM malleability library. The heat diffusion application was started initially with 8 processors. Then, 8 additional processors at iteration 860 were made available. 16 additional processes were split and migrated to harness the newly available processors. Figure 9 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. All the experiments performed were run on shared cluster environments which explains the small fluctuations in the application's throughput for a given application's configuration.

Gradual Adaptation with Malleability and Migration. The following experiment illustrates the usefulness of having the 1 to N split and merge operations. When the execution environment experiences small load fluctuations, a gradual adaptation strategy is needed. The 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 a bit 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, which improved the application's throughput.

6 Related Work

Malleability for MPI applications has been mainly 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 a another programming model which allows virtualization for a dynamic environment by creating extra initial processes and using a virtual name space and process migration to load balance and scale applications.

In [11], 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 moldabilty and the folding technique.

Process swapping [8] 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.

Checkpointing and application stop and restart strategies have been investigated as malleability tools in dynamic environments. Examples include CoCheck [9], starFish [1], and the SRS library [12]. Stop and restart is 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.

7 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 semantics of split and merge are presented and discussed. Collective malleable operations are more appropriate in dynamic environments with large load fluctuations, while 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. PCM/IOS is still an ongoing research project. More work needs to be done to improve the performance of the PCM library and its scalability, and to devise autonomous middleware-level policies that decide when it is appropriate to change the granularity of the running application, what is the right granularity, and what kind or split or merge behavior to select. We plan to extend the IOS middleware with malleability policies. Future work include also devising malleability strategies for non-iterative applications.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31. IEEE Computer Society, 1999.

[2] T. Desell, K. E. Maghraoui, and C. Varela. Malleable Components for Scalable High Performance Computing . In *Proceedings of the HPDC'15 Workshop on HPC Grid programming Environments and Components (HPC-GECO/CompFrame)*, pages 37–44, Paris, France, June 2006. IEEE Computer Society.

[3] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In D. G. Feitelson and L. Rudolph, editors, *JSSPP*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 1996.

[4] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[5] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of adaptive mpi. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21, New York, NY, USA, 2006. ACM Press.

[6] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 20(4):467–480, 2006.

[7] K. E. Maghraoui, B. Szymanski, and C. Varela. An architecture for reconfigurable iterative mpi applications in dynamic environments. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005)*, number 3911 in LNCS, pages 258–271, Poznan, Poland, September 2005.

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

[9] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

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

[11] G. Utrera, J. Corbalán, and J. Labarta. Implementing malleability on mpi jobs. In *IEEE PACT*, pages 215–224. IEEE Computer Society, 2004.

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