MPI-Mitten: Enabling Migration Technology in MPI

Cong Du and Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{ducong, sun}@iit.edu

Abstract

Group communications are commonly used in parallel and distributed environment. However, existing migration mechanisms do not support group communications. This weakness prevents migration-based proactive fault tolerance, among others, to be applied to MPI applications. In this study, we propose distributed migration protocols with group membership management to support process migration with group changing. We design and implement a process migration enabling MPI library, named MPI-Mitten, to verify the protocols and enhance current MPI platforms for reliability and usability. MPI-Mitten is based on MPI standard and can be applied to any MPI-2 implementations. Experimental results show the proposed distributed process migration protocols are solid and the MPI-Mitten system is effective and is uniquely supporting migration-based fault tolerance.

1. Introduction

The advances in scientific computing platforms and architectures challenge the traditional parallel programming model. With recent advances on the massively parallel technologies, more and more supercomputers were built with large amount of commodity hardware to achieve high performance and low cost-to-performance ratio. With the most recent upgrade, IBM Blue Gene/L [1], the current leading supercomputer scales up to a peak computing capability in excess of 280.6 teraflops with 65536 dual-processor PowerPC commodity nodes. With large amount of commodity hardware in supercomputing, reliability becomes a major concern in HPC society. The application failure rate increases with the number of computing nodes. Commodity hardware, even high reliable hardware, which performs well in small clusters, may have reliability issues because the hardware mean time between failures (MTBF) decreases linearly as the computing nodes increases. The accumulated error rate limits the scalability of applications written in traditional static group communication model, which is most widely used in parallel computing. Fault tolerant techniques are demanded to migrate these applications to modern massively parallel supercomputers.

The requirements of fault tolerance have changed for current parallel environments. Conventional checkpointing approaches provide fault tolerance by periodically saving the application state to reliable storages. Periodic recording is costly in both accessing time and storage space. The storage needs to be both reliable and efficient; however, it is very expensive for massively parallel supercomputer to maintain such storages for with tens of thousand processes.

With the advances in hardware sensing, proactive fault tolerance has emerged as a new approach of fault tolerance. Companies such as Intel and Sun are providing products with hardware failure prediction functionalities. Analytical [23] and data mining [11] techniques are also used to support proactive fault tolerance. With proper warning, a process can be migrated from its fault-imminent host to a fault-free host before the failure occurs. Traditionally used for load balancing, process migration has shown its increasingly important role in fault tolerance. Existing works on process migration do not support proactive fault tolerance. Some of them are based on sequential applications [21][25]; others support only point-to-point communication [7] or depend on checkpoint/restart model [19].

Group communication is a key feature in high performance parallel computing platforms such as MPI. MPI standards define many collective primitives and allow these primitives to be optimized for various hardware platforms to achieve better performance. However, the static group communication model poses great difficulties in dynamic process management and fault tolerance. In this paper, we propose new

communication protocols to manage the group membership during a migration. We design and implement Migration Technology Enabled MPI (MPI-Mitten), a high-level portable process migration-enabling library, to verify the newly proposed protocols and enhance the current MPI platforms in reliability and usability. Experimental results show the proposed dynamic process management system is feasible and efficient.

In next section, we give an overview of related work on current fault tolerance techniques and their communication protocols for MPI applications. Section 3 describes the problems and our distributed migration protocols. We describe the design and implementation of the MPI-Mitten in section 4. In section 5, we present the experimental tests and result analysis. The conclusion and future work are discussed in Section 6.

2. Related Work

Most fault tolerant MPI implementations are based on checkpoint/restart model. Examples of such implementations include Lam/MPI [22], MPICH-VCL [4] and Cocheck [24]. However, these platforms are not appropriate to proactive fault tolerance. First, periodically checkpointing applications with tens of thousands processes are expensive. Second, all processes, including non-faulty processes, have to be restarted from the previous checkpoint when a failure occurs. Third, complicated and expensive synchronization protocols are introduced to avoid a domino effect where coordinating processes need to rollback repeatedly trying to reach a consistent global state. Rather than restarting a complete application from the previous consistent checkpoint in checkpoint-based rollback recovery, the pessimistic log-based rollback recovery protocols, which are implemented in MPICH-V1 [3] and MPICH-V2 [5], bring the restarted process forward to the current consist global state by replaying the nondeterministic events logged. Each nondeterministic event is logged to a stable storage before the event affects the application state. The block waiting incurs much performance overhead during normal execution. FT-MPI uses a different method to handle failures based on HARNESS distributed computing framework [10]. Starfish MPI [2] provides failure detection and recovery at run time but it uses low-level strict atomic communication to maintain the communication state. MPI-FT [6] supports fault tolerance with all communicators building with pre-defined spare processes, which are utilized when there is a failure.

Our method is novel compared with other fault tolerant MPI implementations, which follow the checkpoint/restart model and do not support proactive fault tolerance with process migration. They are either independent implementation of a MPI platform [10][2] [22] or based on a specific implementation of MPI [16]. None of them is compatible with existing parallel programming environments. Our protocol, however, is implemented as a portable communication library, which is a high-level add-on layer to various existing implementations. The performance optimization for hardware and communication channels is preserved to maximize the performance. In this way, we save much effort in redesigning a complete fault tolerant platform from scratch.

3. Distributed Migration Protocols

There are three major challenges in supporting proactive fault tolerance of parallel applications. They are how to manage and update the communication group; how to synchronize the processes and maintain a consistent global communication state; and how to collect the execution state, memory state and I/O state and restore them to a new process.

We solve the third problem in our previous work [9]. In this section, we present our protocols to synchronize the processes, update the communication group view and maintain a consistent global communication state.

3.1. Problem Description

A collection of processes forms a communication group *G* and communicates by point-to-point and collective communication. Processes are identified by their ranks within group *G*. Communication operations include point-to-point and collective primitives. The point-to-point primitives involve two peers and can be blocked or unblocked, buffered or non-buffered. All the processes in group *G* participate collective operations. The communication channels are bidirectional First-In-First-Out (FIFO) for each communication peer and communication tag. Dynamic group management functions, including process spawning, merging, splitting etc., and one-sided

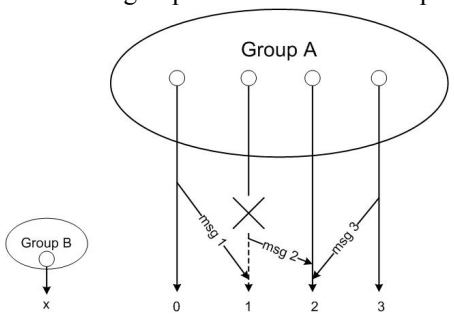

Figure 1. Communication group and messages

communication functions, including *put*, *get etc.,* are supported by the underlying platforms. These operations are defined in MPI-2 standards and are

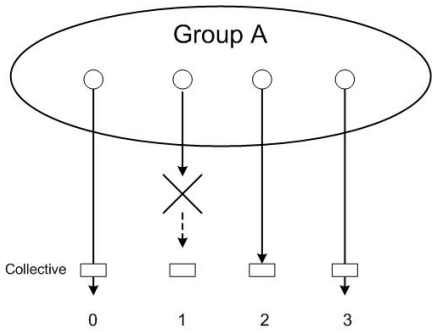

Figure 2. Collective Communication

supported by several general MPI-2 platforms [12][15].

Without loss of generality, we have two assumptions. First, the process to-be-migrated can properly receive the migration signal from an external runtime scheduling system. The migration signal can be delivered in different ways including Unix signal, monitoring daemon, or shared file system depending on the systems. The choice can be made considering the characteristics of the platform with parallel applications running on it. Second, the group of processes can designate a destination machine where to spawn a new process. When a group of processes spawn a child process, it may optionally use an information argument to tell the runtime environment where and how to start the process. For simplicity, we presented one migration inside one communication group in this paper. However the protocols do not impose limitation to one group. The MPI application with multiple groups can be similarly migrated.

As shown in Figure 1, there is an imminent failure detected on the host of *process 1*. As a member of Group A, the absence of *process 1* causes failures to all communication operations it participates including all collective communication operations and some point-to-point operations. To replace *process 1* in Group A with a new process on a stable host, all processes are notified about the replacement and update their local group representation. All messages transmitted through the old communication group, such as *msg 1,* have to be correctly delivered to the new group. Another scenario is that one process is waiting for a message through the old group, but the sender is migrated and the message is sent through the new group. For example, *process 2* is waiting for *msg 2* from *process 1* through group A; however *process 1* is replaced by a new *process x* and *process x* is going to send *msg 2* through the new communication group.

Collective communications encounter similar problem in updating a group. As shown in *Figure 2*, all processes participate in a collective operation but the collective operations are asynchronous. That is, when a process is migrating because of a predicted fault, the other processes, such as *process 0* and *process 3,* may have finished this collective operation and all the results have been committed. Replacing the communication group at this time results in an inconsistent collective operation.

3.2. Protocols

Based on our previous experiences in process migration [7] on PVM, we propose communication protocols for group communication based on common parallel programming paradigms, such as the ones adopted in MPI.

All the processes in a group need to synchronize to create a new process and update the group membership information. We divide the synchronization process into two phases: collective synchronization and point-to-point synchronization. We define a *superstep* as the execution block between any two collective operations. Within a *superstep*, processes can send messages only through point-to-point (pt2pt) communication channels. After receiving a migration signal, collective synchronization protocol brings all the processes in a group to the same *superstep*. Then point-to-point synchronization wakes up all processes waiting for messages through the old communication group, drains the communication buffer and preserves the messages in transmission. After synchronization, all the processes coordinate to spawn a new process, create a new group, and update the group information. The preserved communication state, together with local process states, is transmitted to the new process for continuous execution.

The group information updating, and local process state management are shown in the distributed migration protocol given by Figure 3. All processes are initialized to asynchronously receive the migration command from other members. Once a process P_j receives a migration command, it will distribute it to all other processes. Then all processes in the same group coordinate to reach a synchronization point where all processes have a consensus to spawn a new process as the destination. All processes in the group including the newly spawned process collectively replace the migrating process with the newly spawned process in a new communicator C_{new}. After migration, communicator $C_{new} = \{<P_0, P_1, P_{j-1}, P_n, P_{j+1}...P_{n-1}>\}$ replaces the original communicator $C = \{<P_0, P_1, ... P_{n-1}>\}$. The migrating process P_j and the spawned process P_n coordinate to collect, transmit and restore the local process state and communication state. After

migration, P_j exits and P_n replaces P_j in continuous execution.

The key point to migrate a parallel process with group communication is how to reach a synchronization point while all the processes are running asynchronously. The synchronization protocols are shown in Figure 4 and Figure 5. The synchronization is performed in two steps. First all the processes coordinate to reach the same *superstep* and then reach a synchronization point within the *superstep*.

$P_0, P_1, \ldots P_{n-1}$: Initiation
Initiation()
Begin
Asynchronously waiting for migration notification;
End

P_j: On migration signal to migrate to machine *m*
OnMigration()
Begin
  Send a migration notification (*MIG_CMD* P_j) to each process P_i in application $App = \{P_0, P_1, \ldots P_{n-1}\}$
  CollectiveSyn();
End

P_i : Reaching a synchronization point where $P_i \in \{P_0, P_1, \ldots P_{n-1}\}$;
P_n : After Initiated
CommUpdate()
Begin:
  If $(i < n)$ **then**
    Spawn a new process P_n and establishing an intercommunicator $C_{inter} = \{<P_0, P_1, \ldots P_{n-1}>, <P_n>\}$ from communicator $C = \{<P_0, P_1, \ldots P_{n-1}>\}$;
  Else if $(i == n)$ **then**
    Process initiation
    Get the parent intercommunicator C_{inter};
  End if
  Establish a new intracommunicator $C_{new} = \{<P_0, P_1, P_{j-1}, P_n, P_{j+1} \ldots P_{n-1}>\}$;
  Replace C with C_{new};
  If $(i == j)$ **then**
    Send local communication state to P_n;
    Local process state collection;
    Send process state to P_n;
    Process finalization
  Else if $(i == n)$ **then**
    Receive local communication state to from P_j;
    Receive process state to P_n;
    Local process state restoration;
    Reset migration and synchronization flags;
  Else
    Repeat pending pt2pt operation *op*;
    Reset migration and synchronization flags;
  End if
End

Figure 3. **Distributed Migration Protocol**

MPI provides more collective communication operations than any other parallel communication platform. By optimizing the performance of collective operations according to each system and communication infrastructure, MPI can achieve higher performance. However because collective communication operations' implementation details are transparent to the user, the group membership imposes great difficulty on high-level group membership management. In our protocol, the processes are allowed to execute asynchronously for better performance. The processes synchronize only when some process is commanded to migrate. One-sided communication is used to asynchronously obtain the process state from other processes. A one-sided communication window is created, which is visible to

P_j: After distributing the migration cmd;
CollectiveSyn()
Begin
  Lock *super_step* for read;
  step = 0;
  For i = 0 **to** n-1 **do**
    If ($super\_step_i > step$) **then**
      $step = super\_step_i$;
    End if
  End for
  For i = 0 **to** n-1 **do**
    If (i <> j) then
      Send *step* to P_j
    End if
  End for
  Unlock *super_step*
End

P_i: Before entering a collective communication; where $P_i \in \{P_0, P_1, \ldots P_{j-1}, P_{j+1} \ldots P_{n-1}\}$;
BeforeCollective()
Begin
  Test migration *cmd*;
  If ($cmd == MIG\_CMD\ P_j$ **) then**
    Recv *migration cmd;*
    Recv *global_superstep* from P_j;
  End if
  If ($superstep == global\_superstep$ **) then**
    P2Psyn();
  End if
  Increase *superstep*;
End

P_i: After a communication operation; where $P_i \in \{P_0, P_1, \ldots P_{n-1}\}$;
AfterOp()
Begin
  If (P_i is in synchronization phase **and**
  $superstep == global\_superstep$) **then**
    P2Psyn();
End

Figure 4. **Collective Synchronization Protocol**

all processes. In this window, each process records its collective communication step. After broadcasting the migration command, the migrating process checks the current *superstep* of each process and determines the maximum step as the global *superstep*. The global *superstep* is sent to each process and all processes keep execution until all of them reach the global *superstep*.

```
Pi: reaching the global super step where Pi ∈ {P0, P1,…Pn-1};
P2PSyn()
Begin
  If ( i == j) then
    Lock current_op for read;
    For i = 0 to n-1 do
      If Pi is block waiting for Px, then
        If ( x == j ) then
          wake Px;
        Else
          send MIG_WAKE i to Px;
        End if
      End if
      For i = 0 to n-1 do
        If ( i <> j ) then
          send MIG_END to Px;
        End if
      End for
    End for
    Unlock current_op;
  Else
    Receive cmd from Pj;
    While ( cmd == MIG_WAKE Px ) Do
      wake Px;
    End while
    Assert (cmd == MIG_END);
  Endif
  If Pi is woken up by Py,
    Mark the prevous pt2pt operation op as pending;
  Endif
  Drain local communication channel and save the data as local communication state;
  CommUpdate()
End
```

Figure 5. **Pt2pt Synchronization Protocol**

As shown in Figure 5, after all the processes are within the same global *superstep*, the migrating process initiates the distributed point-to-point synchronization protocol. To break the deadlock caused by one process block waiting for the messages from the processes which is directly or indirectly blocked by the migration (shown in Figure 1), the migrating process P_j checks the dependency of each process and notifies the corresponding process to wake up the blocked process. Another one-sided communication window *current_op* is used to record the process's current operation. After all the processes are woken up, the local communication channels are drained and all pending messages are stored as the local communication state. Then all processes collectively spawn new process and update their local group information. The migrating process transmits memory and communication states to the new process, and finalizes its communication channels. The new process then resumes execution as P_j in new group. If a process is woken up from a blocking point-to-point operation, this operation is repeated in the new group. After migration, the processes should first lookup the local communication state for corresponding message before they actually receive messages from their communication channel.

4. MPI-Mitten and its Implementation

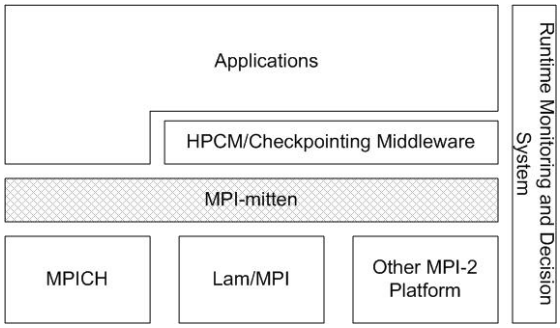

Figure 6. **MPI-Mitten and HPCM**

Migrating processes over MPI platform is challenging. The initial impetus for developing Message Passing Interface was that each Massively Parallel Processor (MPP) vendor created their own message-passing API to expedite the parallel applications running on their own hardwares. However, the performance was optimized on the expenses of the interoperability, flexibility and manageability. MPI was initiated by a broad group of parallel computer users, vendors, and software writers. There are at least 28 MPI implementations publicly available as listed at [13]. To achieve portability, MPI-Mitten shall not rely on any particular implementation besides MPI standards.

We implement our protocols on MPI platform with dynamic process management and one-sided communication. We manage the group membership with dynamic process management features supported by MPI-2. First all the processes in the same static group reach a consensus to spawn a new process on designate machine with MPI_Comm_spawn, and then the processes in the group and the newly spawned process can communicate with an intercommunicator. The newly spawned process can get the handler of the intercommunicator by MPI_Comm_get_parent. The processes work together to merge the intercommunicator into an intracommunicator. Then

MPI_Comm_split is utilized to remove the migrating process from the intracommunicator and reorder the rank of each process. We use MPI_Win_get and MPI_Win_put for one-sided communication and MPI_Win_lock and MPI_Win_unlock for synchronization and mutual exclusion. All these operations are defined in MPI-2.0 standard and there are several MPI-2 implementations [15][12] supporting or planning to support them.

Because MPI-2.0 is not fully supported by various MPI implementations, some limitations are imposed to the implementation of MPI-Mitten on these platforms. Popular MPI implementations, Lam/MPI and MPICH, do not support multithread level MPI_THREAD_MULTIPLE defined by MPI-2.0 standard. Instead, Lam/MPI supports multithread level MPI_THREAD_SERIALIZED, which may cause deadlock if used in synchronizing the processes. Similarly, MPICH-2 is not thread-safe and can support multithread to MPI_THREAD_FUNNELED or MPI_THREAD_SERIALIZED level when kernel (as opposed to user) threads are used MPICH2. Another option using error handler in synchronization is not possible because the MPI standard does not define the communication state after an error occurs. MPI applications are assumed to be error-free. After an error is detected, the state of MPI communicator is undefined. That is, using a user-defined error handler or MPI_ERRORS_RETURN, does not allow users to continuously use the communicator after an error is detected [14] [15]. We solve synchronization problems by allowing the processes to be locked, and we detect the locks and break the locks.

The *superstep* shown in figure 3-4 is a parameter controlling the level of synchronization. In figure 3-4, it is set to the value of infinity for an ideal non-synchronization scenario. In implementation, this value can be set to a finite value to allow specific level of synchronization. Some level of synchronization can help reduce the cost reaching a synchronization point but it increases the cost for normal execution. Because the communication among the processes implicitly imposes synchronization to communication peers, setting *superstep* to infinity does not mean that synchronization cost is unlimited. Moreover, users can always tradeoff between the normal execution cost and synchronization cost to reach a balance.

The protocols are implemented as a portable library, named MPI-Mitten. MPI-Mitten features:

Portability - There is no need to modify current MPI implementations and the fault tolerance is achieved by adding a thin high-level library.

Scalability - The distributed migration protocols are distributed without designated central control. The implementation is scalable and does not introduce single failure point.

Minimized overhead - Synchronization is only performed during migration when it is necessary. The protocols do not introduce extra overhead to point-to-point operations. The semantics of collective operations are preserved and any optimization to a particular implementation is still effective.

The layered system architecture of MPI-Mitten is shown in Figure 6. MPI-Mitten is a thin high-level library sitting between the MPI layer and the application layer. MPI-Mitten enhances the application with dynamic management and fault tolerance. An external runtime system is provided to monitor the status of processes and decide when and where to migrate a process when it detects an imminent failure.

MPI-Mitten can be used on both homogeneous and heterogeneous platforms. It uses HPCM middleware to preserve local process execution state and maintain active disk IO channels. We choose HPCM to preserve the local process state because of its heterogeneity and efficiency. HPCM (High Performance Computing Mobility) is a middleware supporting user-level heterogeneous process migration of legacy codes written in C, Fortran or other stack-based programming languages via denoting the source code. HPCM is customized to different scenarios on heterogeneous and homogeneous platforms. Live variable analysis, pipelining and memory block analysis [9] are performed to reduce the migration cost.

MPI-Mitten is implemented on the HPCM middleware but our method is general and works with other process migration or checkpointing middleware. HPCM middleware [9] and its runtime monitoring and decision system [8] are not built-in components of MPI-Mitten. Other checkpointing packages to preserve the local process state at kernel level [26] or user level [20] can also be hooked up to MPI-Mitten.

5. Experiments

To verify the practical feasibility, effectiveness, and scalability of our distributed migration protocols, we have implemented the MPI-Mitten library and tested it on the sunwulf Computer Farm in the Scalable Computing Software (SCS) laboratory at the Illinois Institute of Technology. Sunwulf is composed of one Sun Enterprise 450 server node (sunwulf node), 64 Sun Blade workstations 100 (hpc-1 to hpc-64) and 20 Sun Fire V210R (hpc-65 to hpc-84) compute nodes. The Sun Enterprise 450 server has four CPUs, 8M cache and 4GB memory. Each CPU is 480 MHz. The Sun Blade computing node has one 500-MHz CPU, 256K L2 cache, and 128M memory. The Sun Fire V210R computing node has two 1GHz CPUs, 1M L2 cache

and 2GB memory. All the systems are running SunOS 5.9 operating system. All the Sun Fire 210R servers are connected with a Gigabits Ethernet. The maximum bandwidth is 89.1M bytes/s. Other communication channels within the workstations or between the servers and the workstations are 100Mbps internal Ethernet. The maximum bandwidth is 11.8M bytes/s. The workstations are organized as a "fat tree" structure.

We test two typical MPI benchmarks. One is IS benchmark from NAS Parallel Benchmark 3.1 [18]; the other is mpptest [17]. NAS IS benchmark tests a sorting operation. It fills a vector with random integers, and then computes a rank for each number. NAS IS benchmark features significant data communication especially collective communication so we use it to test the effectiveness of the protocols and the normal execution overhead. Mpptest is a program that measures the performance of some MPI message passing routines in a variety of situations [17]. Mpptest is a communication intensive benchmark testing the communication performance of MPPs. We use Mpptest as an application with a combination of different MPI operations such as point-to-point operations and collective operations.

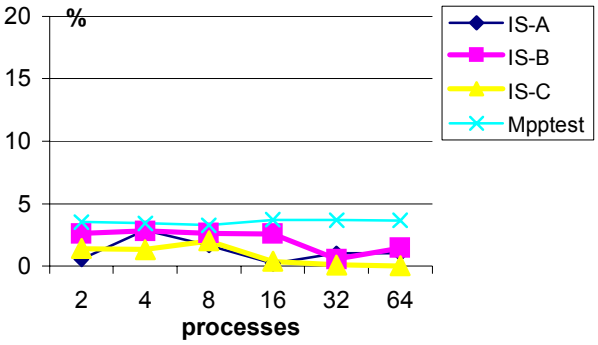

Figure 7. **MPI-Mitten Overhead**

The first experiment tests the system overhead during normal execution when there is no failure detected. Figure 7 shows the MPI-Mitten overhead for NAS IS benchmark and Mpptest benchmark. Because NAS IS and Mpptest benchmark have more collective operations, their experiment results show a worse case scenario. The tests are performed for data size from A to C where C has maximum problem size. We perform the test on 2 to 64 processes. The overall overhead is less than 4% and average overhead is 1.22% for NAS IS and 3.55% for Mpptest. The overhead is caused by the "test" operations and synchronization operations before and after communication primitives. Because the migration signal is through asynchronous communication primitives, this operation does not introduce much overhead.

As shown in Figure 8, the second experiment tests the efficiency and scalability of dynamic migration on a predicted failure. We test NAS IS and Mpptest with 2 to 32 processes and find that the synchronization time is almost constant while the number of processes increase. The average synchronization time is 2.21 seconds for NAS IS and 2.42 for mpptest. When the processes increase to 32, the problem partition size decreases, so the total migration cost decreased. The group management time increases slowly because there is more time spent on the initiation of a new process at the destination machine. In this experiment, we observe that MPI_Comm_spawn and the new process initiation contribute 69.3% to 81.8% of the group management cost. The group membership management cost depends on the implementation of MPI platform. Optimizing the MPI process initialization may improve the group management performance.

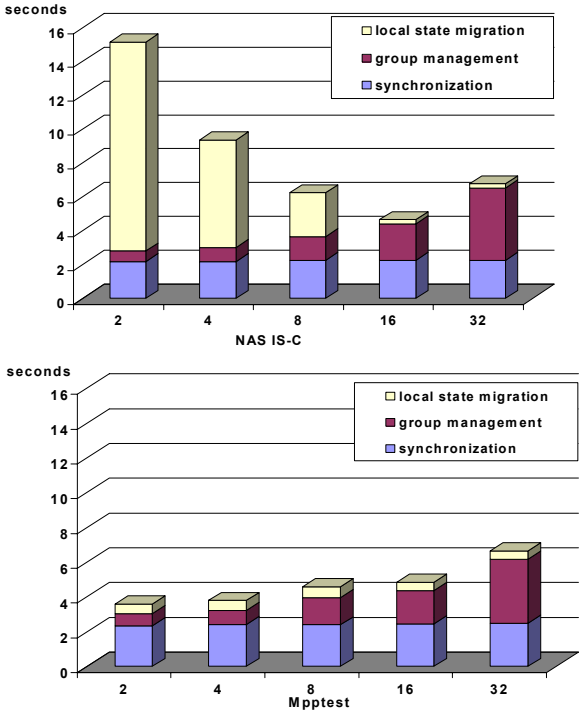

Figure 8. **Migration Timing**

6. Conclusion and Future Work

We have presented a novel method to provide proactive fault tolerance to MPI application based on process migration. Three different protocols, distributed migration protocol, collective synchronization protocol and point-to-point synchronization protocol, are proposed to synchronize MPI processes and perform migration. We implemented the protocols on MPICH-2 platform and

developed MPI-Mitten, a prototype MPI library enabling MPI applications to migrate for proactive fault tolerance, load balancing and other management demands. Several important implementation issues are identified and addressed. Experimental results confirm the feasibility and scalability of the newly proposed protocols and the efficiency of MPI-Mitten. The synchronization cost and normal execution cost of MPI-Mitten is low. Though the implementation is on MPI platform, the protocols' assumptions are general and well stated. The protocols can also be used to other group communication based parallel environments.

We proposed and implemented the protocols to enable the fault tolerance and dynamic management of MPI applications. Currently we did not consider some advanced features of MPI such as topology and MPI-IO. To promote utilization of MPI-Mitten to more applications, we plan to extend our current work to more complicated scenario in the future.

Acknowledgments

This research was supported in part by national science foundation under NSF grant SCI-0504291, CNS-0406328, EIA-0224377, and ANI-0123930.

References

[1] N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor et al. "An overview of the BlueGene/L supercomputer". In Proc. Supercomputing (SC2002), Baltimore, MD, Nov. 2002.

[2] A. Agbaria and R. Friedman. "Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations". In 8th International Symposium on High Performance Distributed Computing, 1999.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, et. al, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in Supercomputing (SC2002), Baltimore, Nov. 2002.

[4] A. Bouteiller, H.-L. Bouziane, P. Lemarinier, T. Hérault, and F. Cappelo, "Hybrid preemptive scheduling for mpi applications on the grids," in 5th IEEE/ACM Workshop on Grid Computing, Nov. 2004.

[5] A Bouteiller, F Cappello, T Herault, G Krawezik, et. al. "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging", in Supercomputing (SC2003), Phoenix, AZ, Nov. 2003.

[6] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. "Mpi/ft: Architecture and taxonomies for fault-tolerant, messagepassing middleware for performance-portable parallel computing". In Proc. IEEE International Symposium on Cluster Computing and the Grid, Melbourne, Australia, May 2001.

[7] K. Chanchio, X-H. Sun, "Communication state transfer for the mobility of concurrent heterogeneous computing," IEEE Trans. on Computers, vol. 53, No. 10, pp:1260-1273, 2004.

[8] C. Du, S. Ghosh, S. Shankar, and X.-H. Sun, "A runtime system for autonomic rescheduling of MPI programs," in Proc. International Conference of Parallel Processing, Montreal, Canada, August 2004.

[9] C. Du, X.-H. Sun and K. Chanchio, "HPCM: A pre-compiler aided middleware for the mobility of legacy code," in Proc. IEEE Cluster Computing Conference, Hong Kong, Dec. 2003. Software available at: http://archive.nsf-middleware.org/NMIR4/contrib/download.asp.

[10] Fagg, G., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J. "Scalable fault tolerant MPI: extending the recovery algorithm," Euro PVM/MPI, Sorrento (Naples), Italy, Sep, 2005.

[11] G. Hamerly, C. Andelkan, "Bayesian approaches to failure prediction for disk drives," In Proc. 18th International Conference on Machine Learning, Williams College, MA, Jun. 2001.

[12] "LAM/MPI Parallel Computing," http://www.lam-mpi.org/

[13] "MPI Implementation List," http://www.lam-mpi.org/mpi/implementations/fulllist.php?show_inactive=1

[14] "MPI: A message-passing interface standard", http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[15] "MPICH2", http://www-unix.mcs.anl.gov/mpi/mpich2/

[16] "MPICH-V", http://www.lri.fr/~bouteill/MPICH-V/

[17] "MPPTEST",http://www-unix.mcs.anl.gov/mpi/mpptest

[18] "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Software/NPB/

[19] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh, "The design and implementation of Zap: A system for migrating computing environment", in Proc. USENIX 5th OSDI, Dec. 2002.

[20] J. S Plank, M Beck, G Kingsley, K Li, "Libckpt: transparent checkpointing under Unix," USENIX, 1995.

[21] P. Smith and N. Hutchinson, "Heterogeneous process migration: The Tui system," Software - Practice and Experience, vol 28, No.6, pp.611-639, 1998.

[22] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman. "The LAM/MPI checkpoint/restart framework: system-Initiated checkpointing". In LACSI Symposium. Santa Fe, NM. October 2003.

[23] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in Proc. Intl. conf. on dependable systems and networks(DSN'04), Florence, Italy, Jun. 2004

[24] G. Stellner. Cocheck: "Checkpointing and process migration for MPI," In Proc. IPPS, Honolulu, Hawaii, April 1996.

[25] M. M. Theimer and B. Hayes, "Heterogeneous process migration by recompilation," in Proc. 11th IEEE International Conference on Distributed Computing Systems, Arlington, TX, Jun. 1991.

[26] P Tullmann, J Lepreau, B Ford, M Hibler, "User-level checkpointing through exportable kernel state," in Proc. Intl. Workshop on Object Oriented Operating System, 1996.