ABARIS: An Adaptable Fault Detection/Recovery Component Framework for MPIs

Hideyuki Jitsumoto[1], Toshio Endo[1], Satoshi Matsuoka[1,2]

[1]Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, 152-8552 JAPAN

[2]National Institute of Informatics
2-1-1 Hitotsubashi, Chiyoda-ku,
Tokyo, 101-8430 JAPAN

{jitsumo0, matsu}@is.titech.ac.jp, endo@gsic.titech.ac.jp

Abstract

Long-running MPI applications on clusters and grids that are prone to node and network failures, motivates the use of fault tolerant MPI implementations. However, previous fault tolerant MPIs lack the ability to allow the user to easily choose appropriate fault recovery strategies according to the execution environment, independent of the application codes—rather, the user often had to hard-code restoration strateties in accordance to diverse sets of fault patterns, which could be numerous: for instance, if the fault is transient to a particular process, we merely have to restart the process on the same computing node; on the other hand, if the fault is due to repetitive hardware unreliability, we must migrate the process to a new node in its recovery. ABARIS is our new Fault/Recovery model aware component framework for MPI, where users can customize MPI fault detection and recovery algorithms according to their application and execution environmental requirements by merely selecting appropriate fault/recovery components, independent of the application code. Currently, the ARABIS framework prototype is implemented on top of MPICH-P4MPD. Preliminary evaluation of the prototype using NPB on our MPI fault simulator demonstrates that overhead compared to the original MPICH-P4MPD is almost negligible (less than 1%) under normal execution, and when faults occur, appropriate selections and pairings of fault model and recovery method components for corresponding to the execution environment is significant to the overall execution time.

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas, 18049028, 2006. We would lile to thank INRIA Grand-Large project team and ANL MPICH Group for their advice on FT for MPI.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

1 Introduction

Modern scientific applications on large scale MPPs, large clusters, and girds usually employ MPI[10]LAM[4] as an underlying communication substrate, and often have execution times ranging from days to months. As the systems scale, these systems have become increasingly fragile; an ideal MPI implementation thus must embody fault tolerance properties that support long execution times in fragile underlying execution environments, while being easy and flexible to use as well as being portable and adaptable of their execution environment and fault characteristics. Previous fault tolerant MPI implementations such as LAM/MPI[16] and MPICH-V[3] are easy to use for the user in that fault tolerance is largely transparent to the programmer, but their recovery protocol is fixed and thus not adaptable. This may not be desirable, as for example, the desirable recovery method is obviously different when a fault is transient software one confined in a single process, versus a repetitive one caused by a faulty hardware. In the latter case, we must be migrated the recovered process to a new physical node, but such a strategy may turn out to be wasteful or even unfeasible, especially if the fault is of the former, i.e., the transient one. Moreover, such a fault may be misdetected by the fault detector to be a node failure and unnecessary migration might occur whereas it might turn out that the fault was really with the network switch. FT-MPI[8], allows the end programmer to adapt so that appropriate recovery protocol would be selected, but it is the responsibility of the user to not only make the selection but also implement the recovery protocol itself embedded in the user program, making the system harder to use.

ABARIS, our fault tolerant framework for MPI, achieves independency/adaptability and ease of use simultaneously by allowing the user (not necessarily the programmer) to select a pair of predefined fault model and recovery com-

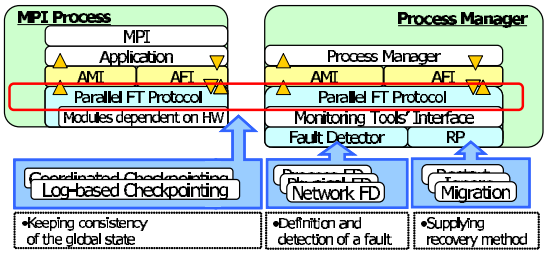

Figure 1. ABARIS Components

ponents, depending on the execution environment. Fault model components select appropriate fault recovery protocols (e.g., ignore/restart/migrate) per each fault occurrence. Moreover, a programmer can customize the components in an object-oriented fashion to adapt to the characteristics of execution environments in a fine-grained manner, if necessary.

The current prototype of ARABIS is implementation on top of MPICH-P4MPD. Preliminary evaluation of our prototype implementation using NPB on our fault simulator demonstrates that overhead compared to the original MPICH-P4MPD is almost negligible (less than 1% under fluctuating time measurements) under normal execution. The validity of our framework is demonstrated by the fact that users need not modify their original MPI programs, and when faults occur, appropriate selections and pairings of fault model and recovery method components for corresponding to the execution environment is significant to the overall execution time, in a controlled environment by modifying the failure occurrence rates of nodes.

2 The ABARIS Framework

2.1 Overview of the ARABIS Architecture

ABARIS consists of three main components (Figure 1): the algorithm component that maintains the consistency of the state of an MPI process (PFTP: Parallel Fault Tolerant Protocol component), the fault definition and detector component (FD: Fault Detector component) and the recovery work component (RP: Recovery Protocol component). ABARIS also embodies other supplementary components such as monitoring tools component, and node-local components for implementing fault tolerance, such as the checkpointing component.

Components interfaces in ABARIS can be categorized into two types: first is the ABARIS Fault tolerant Interface (AFI), called from the underlying MPI system or other ABARIS components to implement some fault tolerant function for every component. To be more specific, ABARIS assumes an MPI implementation where process-managers (PMs) exist for each rank, and every component implements a set of interfaces handling the three kinds of messages, which is between two PMs with different ranks, between a PM and an MPI process with the same rank and between two MPI processes. A PFTP component also implements an interface for handling time-interval events that corresponds to the starting point of recovery.

Another category is the ABARIS MPID Interface (AMI), which is called by every ABARIS component on the use of MPI or PM functions, such as sending a message between PMs or MPI processes. For example, at each MPISends and MPIReceives, an AFI is called transparently by the underlying MPI system. Then, the implementation of the AFI does some work to perform bookkeeping function for fault tolerance, e.g., message-logging. Finally, the AFI implementation invokes an AMI for message transfer etc. that use the underlying messaging devices used on various MPI systems.

When a fault occurs, ABARIS starts recovery according to the following procedure:

1. The FD components are called from PFTP component with some intervals and compare their fault model to the information supplied from the monitoring tools.
2. If the information matches its embodied model, the FD component selects an appropriate RP component to start fault recovery.
3. The PFTP component initiates recovery by using the selected RP component above, while maintaining messaging consistency between the processes.

In general, the base implementations of these components are intended to be done by trained programmers in parallel and distributed systems, resulting an arsenal of various components. Once that is achieved, end-programmers and system administrators can customize fault tolerance and recovery models easily to adapt to their requirements by mere selection of the appropriate components. We will demonstrate the actual realization of such a scenario in the latter sections.

2.2 Components and Recovery Models

2.2.1 The Parallel Fault Tolerant Protocol (PFTP) Component

The PFTP component supplies various functionalities to maintain the global consistent state of MPI programs. A global state of a message passing system is a collection of individual states of the communication channels and their processes. As such, when the global system is said to be

Table 1. Recovery Protocol

Protocol	IGN.	RES.	MIG.	PRO.
Recovery cost	0	medium	large	small
Necessary nodes	0	0	each fault	large
Process FT	No	Yes	Yes	Yes
Physical FT	No	No	Yes	Yes

consistent, it is necessary to guarantee that if the state of a particular process indicates message having being received, then the state of the corresponding sender should reflect sending of that message other than the consistency of each processes state[7]. One of the simple PFTP implementation we have in ARBAIS is coordinated checkpointing, which we employ here as an example. The coordinated checkpointing PFTP component is tasked at performing two actions. First, the PFTP component creates a checkpoint for its (client) MPI processes at prescribed time intervals which users define. When creating a checkpoint, the PFTP component drains all the messages traveling on the connections between MPI processes by sending a "drainage" (or "bulldozing" in other literatures) packet just prior to the actual checkpointing. As a result of this draining, all sending messages will guaranteed to have been received. Secondly, on fault recovery, the PFTP component recovers the state of all the MPI processes simultaneously for keeping the consistency of each MPI process state.

2.2.2 The Recovery Protocol (RP) component

These components represent various different strategies for recovery from faults. In particular, each component embeds its own policy on deciding how the MPI processes are to be recovered, and on which nodes they should be recovered to. On ABARIS, the recovery protocol component roughly categorized into 4 types below. Here, having to perform checkpointing is a prerequisite for RESTART and MIGRATION, and process replication of process is a prerequisite for PROMOTE.

IGNORE a process ignores the fault

RESTART a process restarts on the node it had been assigned to prior to fault occurrence.

MIGRATE a process migrates to the node it had not been assigned to prior to fault occurrence.

PROMOTE a standby, replicated process is promoted to becoming the primary process in place of the failed process.

We show a partial list of properties of these protocols (Table 1). For recovery, MIGRATE has the largest overhead due to the cost of process image migration. PROMOTE requires less overhead but will require many extra standby nodes for redundancy, whereas MIGRATE does not need such extra nodes as process migration occurs only on process failure. In this fashion, an end-user (not necessarily the programmer) can prioritize either the overall running time or the resource usage, according to his requirements and the property of the underlying system, as well as the frequency of faults. For example, qualitatively PROMOTE would be advantageous in an environment that embodies numerous available nodes. Another example would be that, we could simply say IGNORE if the application facilitates its own fault tolerance mechanism.

2.2.3 The Fault Detector (FD) component

This component defines faults and mechanisms for their detections. It continuously compares information from the monitoring tools to some thresholds prescribed for the FD component and decides on which RP to choose out of the above 4 types. FD uses information such as whether the node is currently working or not, how many failures have occurred in the past and how many processes executes on the node.

Here are some examples of the simple fault models we have defined for ARABIS:

Process Fault This model indicates that a failure occurred only within a user process, and the node accommodating the process is alive. We can implement the model by examining the status of user process by a heartbeat technique, and status of the nodes from a monitoring tool such as the Ganglia Cluster Toolkit[13].

Physical Fault This model says that a failure occurred in the hardware resource itself. We can easily decide whether this model holds as is with the case of the Process Fault with appropriate node monitoring.

Network Fault This model indicates that a failure occurred in the network resource. His model is difficult to implement merely on the nodes themselves, but rather require information from the underlying switches in the network with the knowledge of network topologies. For most clusters this is not a problem since switches and their topologies are well-known by the administrators. For grids and other large systems, there are various proposed methods to detect switch topologies[12]

Repeated or cascaded fault occurrences, has to be treated in a special fashion in our framework. For example, a

node that experiences a uncorrectable memory ECC failures could cause faults in a repeated fashion for the same process, but in somewhat of a irreproducible way. Initially, such faults may show up as Process Faults. However, with repeated faults the component can direct further investigation to see whether ECC memory error has occurred, and if it has, the users can choose the Physical Fault to avoid any more faults. Our architecture can deal with such repeated faults by a FD which embodies a repetition threshold and would delegate a decision to other FD upon reaching the threshold.

3 Prototype Implementation

We have implemented a prototype of our ABARIS framework on MPICH. Among several device modules that MPICH provides[5], we have chosen P4MPD, because its daemon-based architecture conform to ABARIS architecture well. Each ABARIS component is implemented as a dynamic library. Set of components to be loaded actually is specified by a configuration file.

The current prototype implementation does not deal with real faults; instead, we have implemented a simulator that simulates multiple kinds of faults and recovery.

3.1 Adaptation of MPICH-P4MPD to ABARIS

In MPICH-P4MPD device architecture, a daemon process called MPD runs on each node, as described in Figure 3. All MPDs are connected in a ring topology. MPDs and mpdmans, which are described below, play roles of process manager (PM) in section 2.1.

MPICH-P4MPD invokes MPI processes as follows:

1. When a user invoke a mpirun process, it sends a 'invoking message' to its local MPD.
2. The invoking message is transmitted via the MPD ring. When a MPD receives the message, it invokes an mpdman process. After that, mpirun process makes a connection to its local mpdman.
3. All the mpdman processes composes a ring topology by connection.
4. Each mpdman invokes its corresponding MPI process and starts application.

Figure 3 describes modification of MPICH-P4MPD for supporting ABARIS. White boxes represent components that originate from original MPICH-P4MPD. As described in the figure, ABARIS components are embedded both in MPI processes and process managers (MPDs and mpdmans). The original MPICH-P4MPD components are

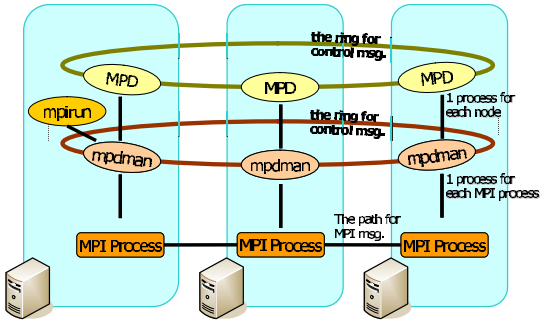

Figure 2. Structure of MPICH-P4MPD architecture

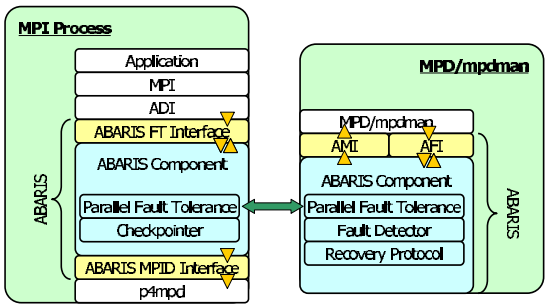

Figure 3. Prototype ABARIS implementation on MPICH-P4MPD

modified so that they invoke ABARIS components via AFI. By doing so, ABARIS components can capture all messages among all MPI processes and process managers. In addition, mpdman is modified so that it invokes PFTP component periodically. Conversely, ABARIS components call original components via AMI. AMI is mainly used to conduct raw communication and obtain information related to MPI such as process rank.

3.2 Implementation of ABARIS components

3.2.1 PFTP component

We have implemented a simple PFTP component that adopts coordinated checkpointing algorithm[7]. This component creates global consistent states at intervals specified in a configuration file. The global consistent state is created as follows (Figure 4).

1. One of mpdmans send 'start-cp' messages to all the mpdmans.

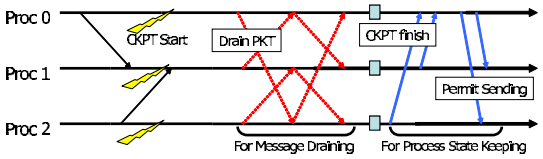

Figure 4. Simple Coordinated Checkpointing

2. When an mpdman receives the message, the mpdman forwards it to its local MPI process.

3. When an MPI process receives the message, it stops application work and sends 'drainage' messages to all other processes that have connections with it.

4. After receiving 'drainage' messages from all other (connected) processes, the MPI process creates a local checkpoint. This is conducted with Zandy's ckpt library [17].

5. After creating a local checkpoint, each MPI process restarts application. While local computation is restarted independently, message passing is suspended until the counterpart process also finishes checkpointing.

A global state, which consists of local checkpoints of all MPI processes, is consistent. This issue is discussed in[7].

3.2.2 RP component

As RP components, we have implemented prototypes of RESTART and MIGRATE components. Currently, they do not recover not recover the processes fully, since we are executing in a fault simulator environment (by all means it does not undermine their capabilities or performances); Instead, they simulate effects of faults and recovery by suspending application processes until they are (virtually) recovered. The suspending time, denoted by $T_{recover}$, is calculated as follows. Let T_{elapse} be elapsed time from the last checkpoint to fault occurrence time. And let T_{load} be the time to load a checkpoint from disk and $T_{transfer}$ be the network transfer time of checkpoint. T_{load} and $T_{transfer}$ are predetermined through a preliminary experiment. Then $T_{recover}$ is calculated as $T_{recover} = T_{elapse} + T_{load} + T_{transfer}$.

The difference between RESTART and MIGRATE components appears in $T_{transfer}$. In RESTART component, which always restarts processes locally, $T_{transfer}$ is set to zero.

3.2.3 FD component

We have implemented the following basic FD components, which are based on fault models described in Section 2.2.3. They are invoked when the system finds a process is (virtually) dead.

Process FD This component checks the state of node that the failed process have resided in. If the node is alive, this case is regarded as process fault and RESTART protocol is selected.

Physical FD If this component finds the node dead (physical fault), it selects MIGRATE protocol.

Network FD This component checks the state of network switch that the process is hanging on. If the switch is found unstable, MIGRATE protocol is selected.

Repeated FD It checks the number of process faults for each node. If the number is over a predefined threshold, it selects MIGRATE protocol to avoid any more faults.

In Section 4, we describe how we cascade these basic components. Note that states of nodes and switches used by the components are generated by our fault injector tool. Currently, a process of rank 0 checks states of all processes. Since it introduces bottleneck, we plan to implement a distributed monitoring method for our next production version.

3.2.4 Fault Injector

To provide node states to FD components, we have implemented a fault injector tool that works as a monitoring tool. This tool maintains a data record that consists of process rank, physical fault rate and process fault rate for each node. In the beginning, each node is assigned its record according to a scenario. Then the tool creates faults randomly, by referring to the fault rates. With this tool, MIGRATE protocol is simulated by exchanging the records between nodes, instead of actual process migration.

4 Performance Evaluation

4.1 Experimental Condition

Experiments are run on 32 nodes of a 256-node cluster and distribute one process for each node. Each node is equipped with two AMD Opteron(tm) 242 Processors, running at 1.6GHz, 2GB main memory (DDR SDRAM), two 250GB IDE ATA100 hard drives, and two GbE Network Interface(we use only one GbE). Each 20-node is connected to same switch which provide GbE (Dell Power connect 5224), and a switch are connected by 4Gbps link which composed of 4 ports trunk link. All these nodes are operating under Linux 2.6.12. The tests and benchmarks are compiled with GCC 3.3.5 (with flag -O3).

Table 2. The performance comparisons of the original MPICH-P4MPD and the prototype implementation

CG			
	Original	prototype impl.	d-ratio(%)
AVE	1569.379	1561.933	0.474455
STDEV	138.4851	124.31552	
MG			
	Original	prototype impl.	d-ratio(%)
AVE	6620.598	6616.103	0.067894
STDEV	232.2076	142.39843	
EP			
	Original	prototype impl.	d-ratio(%)
AVE	175.438	174.952	0.277021
STDEV	0.161109	0.4793078	

For all the experiments, we assume a single checkpoint server connected by high bandwidth link. In addition we assume, first, checkpoint images are saved on local hard drive on each node. Then, the checkpoint images are collected by checkpoint server when the all MPI processes finish checkpointing.

4.2 The overhead of ABARIS Framework

We evaluate the overhead of ABARIS framework. In order to conduct this measurement, we compare the performance of NAS Parallel Benchmark (NPB 2.4) CG/MG/EP CLASS-C [2] on original MPICH-P4MPD to one on the prototype implementation without any ABARIS component (This mean is we only measure the overhead of modify of AFI calling.). Table 2 presents the performance (Mop/s) and the standard deviation of the performance about NPB 2.4 CG/MG/EP. Besides, it also presents degradation ratio (%) of MPICH-P4MPD to the prototype implementation. Regarding CG and MG, the difference between the performance of MPICH-P4MPD and the prototype implementation is much smaller than each standard deviation. So, the overhead of ABARIS framework can be considered to be statistically negligible. While the performance difference of EP is larger than standard deviations, but the degradation ratio of MPICH-P4MPD and the prototype implementation is only 0.2 %. So the overhead of ABARIS framework can also be considered to be statistically negligible about EP.

4.3 ABARIS Framework performance validation

We evaluate our ABARIS framework when faults occur. In order to measure this point, we simulate the fault occurrence and the recovery by FD components, RP components, and the fault injector as mentioned in section 3.2. In this evaluation we use NPB 2.4 CG CLASS-C on 8 nodes. First, we measure the size and the load time of checkpoint to fix parameters, $T_{transfer}$ and T_{load}, for RP components. As a result, we get the size of checkpoint is 158MB and the load time of checkpoint is 0.38 sec. So, we fix that $T_{transfer}$ is 1.325 sec. (the theoretical time to send 158MB by 1Gbps link without a congestion), and T_{load} is 0.38 sec. Regarding the other parameters, we fix the intervals of checkpointing at 30 sec. and the interval of fault detection at 10 sec. Moreover we assume faults don't happen during the recovery due to the limitation of current implementation. We use two models. Both models, the nodes are connected the switch by a 1Gbps link, and all processes are allocated on group (A) when the evaluation starts. We present that it is important to use an appropriate pair of fault model and recovery method for each characteristics of environment in terms of the performance comparison of two set of FD components while changing failure rate.

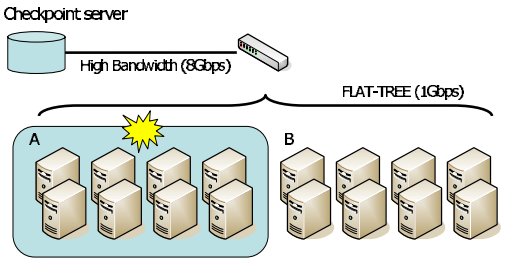

Figure 5. Frequent process fault model

4.3.1 Frequent Process Fault

In this model (figure 5), we assume the group (A) has high probability of process fault because an OS or a job scheduler kills a process. Moreover we assume there are high bandwidth link between the checkpoint server and the switch. We compare the set of Process Fault FD and Physical Fault FD (PROC'nPHY), and the set of Process Fault FD, Physical Fault FD and Repeated Fault FD (REPEATED). Figure 6 clearly demonstrates REPEATED model is suitable for the this environment. The difference of performance is from 3% to 30%. In addition, the number of migration node is suppressed when the number of nodes is smaller than 8 nodes, except when the failure is injected every 30 sec. This means it would be desirable for a fault tolerant MPI system to change their fault model and recovery method. Figure 6 also presents the performance with 30 sec. fault intervals is higher than the one with 60 sec. fault intervals. The reason for this is the number of the fatal

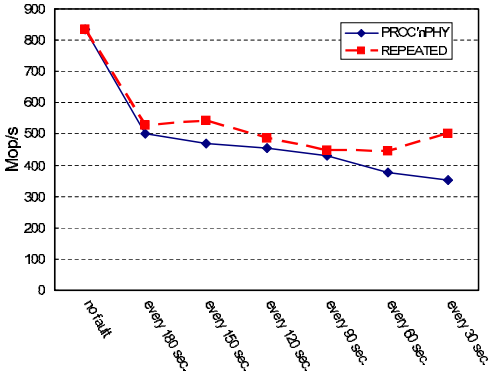

Figure 6. Performance comparion of PROC'nPHY with REPEATED

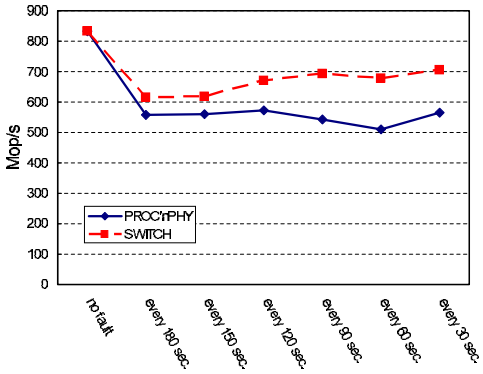

Figure 8. Performance comparion of PROC'nPHY with SWITCH

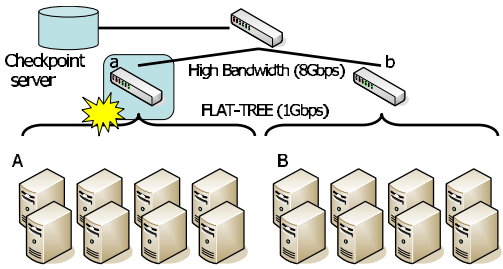

Figure 7. Frequent physical fault model

errors is suppressed owing to the fault occurring simultaneously on each node in between each fault detection interval the fault detection intervals. In practice, there were 9 occurrences of fatal errors detections when the failure was injected every 60 sec. On the other hand, there were 7.8 occurrences of fatal errors detections when the failure was injected every 30 sec.

4.3.2 Frequent Physical Fault

In this model(Figure 7), we assume the group (A) has high probability of physical faults because the switch (a) is faulty, in some bad condition with intermittent faults in its transfer. Moreover we assume there are high bandwidth link between the checkpoint server, the switch (a) and the switch (b). We compare the set of Process Fault FD and Physical Fault FD (PROC'nPHY), and the set of Process Fault FD, Physical Fault FD and Network Fault FD (SWITCH). Figure 8 clearly demonstrates SWITCH model is suitable for this environment. While PROC'nPHY checks the node state on the basis of the local information of the node, SWITCH checks the node state on the basis of the information including switch state. SWITCH hypothesizes that the health nodes which are connected faulty switch will cause the faults soon. Thus SWITCH moves the processes from the nodes which are connected faulty switch earlier than PROC'nPHY. The difference of performance is from 9% to 25%. This result indicates that it might be necessary to give up the use of a perfectly healthy node. In other words, this result indicates the prediction of fault is quite effective.

5 Related work

LAM/MPI[16] has been extended to support fault tolerance and process migration with coordinated checkpoint using the Chandy-Lamport algorithm [6]. However, LAM/MPI does not support other fault tolerance algorithms, such as message logging. In addition, it does not provide automatic recovery mechanism because it does not embody a fault detector.

Egida[15] focuses on designing, implementing and comparing several automatic fault tolerance protocols for MPI applications. With this framework, pessimistic and causal logging algorithms have been compared. However, its coverage is limited to logging based protocols.

More recently, MPICH-V[3] system support a wide range of fault tolerance protocols. Its generic framework covers coordinated, uncoordinated, pessimistic logging and causal logging, etc. By comparing multiple protocols, MPICH-V team has shown that coordinated checkpoint achieves much better performance than message logging both in fault-free environment and fragile environment. The differnce between our research and MPICH-V is that ABARIS is designed to be an extensive fault tolerant component framework to support not only multiple fault toler-

ance protocols, but also multiple recovery protocols based on fault models.

FT-MPI[8] handles failures at the MPI communicator level. It provides programming interfaces to exploit errors returned by MPI instructions when faults occur. Although FT-MPI can deal with faults in a flexible manner, its main drawback is lack of orthogonality, transparency, and portability; users need to add special fault tolerance code to the application.

OpenMPI[9] is a project to combine technologies and resources of several existing projects (FT-MPI, LA-MPI[1], LAM/MPI, PACX-MPI[11] and so on). According to their web site, they plan to provide multiple fault tolerance protocols, while current release does not have such support.

6 Conclusion and Future work

We proposed ABARIS framework, which has a fault/recovery model aware component framework for MPI. We have implemented a prototype of our ABARIS framework on MPICH. Then we measure the overhead of ABARIS frame work, and this result shows the overhead is statistically negligible. Next, to demonstrate the validity of ABARIS framework, we evaluate the performance of our prototype implementation by means of two environment models and simulator of fault/recovery occurrence. As a result, it is found that using appropriate fault/recovery model on individual execution environment achives low overhead.

As future work, we also plan to select ABARIS component and its parameter automatically and dynamically because it may still be difficult for end-programmers to select appropriate ABARIS components for their execution environments. For that purpose, we need to design performance model based on existing research such as [14] that manages the checkpoint intervals. In addition, we started to implement ABARIS on MPICH-2. In this version, we are implementing ABARIS framework on the layer which virtualizes communication channel. Thus users can obtain device-independent fault tolerant MPI based on ABARIS.

References

[1] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, and T. S. Woodall. Architecture of LA-MPI, a network-fault-tolerant MPI. In *IPDPS*, 2004.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V: a multiprotocol fault tolerant mpi. In *International Journal of High Performance Computing and Applications.*, 2005.

[4] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] R. M. Butler and E. L. Lusk. Monitors, messages, and clusters: the p4 parallel programming system.

[6] K. M. Chandy and L.Lamport. Distributed snapshots : Determining global states of distributed systems,. In *Transactions on Computer Systems, vol. 3(1). ACM,*, pages 63–75, February 1985.

[7] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[8] G. Fagg and a Dongarra. FT-MPI: Faulttolerant mpi,supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting 2000 ,Springer-Verilag, Berlin, Germany*, pages 346–353, 2000.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[11] R. Keller, B. Krammer, M. S. Mueller, M. M. Resch, and E. Gabriel. MPI development tools and applications for the grid,. In *Workshop on Grid Applications and Programming Tools, held in conjunction with the GGF8 meetings, Seattle, WA, USA,*, June 2003.

[12] B. Lowekamp, D. O'Hallaron, and T. Gross. Topology discovery for large ethernet networks. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications(SIGCOMM '01)*, pages 237–248, 2001.

[13] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. In *Parallel Computing, 30,*, July 2004.

[14] D. Nurmi, R. Wolski, and J. Brevik. Model-based checkpoint scheduling for volatile resource environments. Technical Report 2004-25, University of California Santa Barbara, Department of Computer Science, Santa Barbara, CA, 93106, 2004.

[15] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low overhead fault tolerance. *In Proceedings of the 29th Fault-tolerant Computing Symposium (FTCS-29), Madison, Wisconsin,*, pages 48–55, June 1999.

[16] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[17] V. C. Zandy. ckpt: A process checkpoint library, 2002. http://www.cs.wisc.edu/~zandy/ckpt.