PAPER

MPICH-GF: Transparent Checkpointing and Rollback-Recovery for Grid-enabled MPI Processes

Namyoon WOO[†a)], ***Member***, **Hyungsoo JUNG**[†b)], **Heon Y. YEOM**[†c)], ***Nonmembers***, **Taesoon PARK**[††d)], ***Member***, ***and*** **Hyungwoo PARK**[†††e)], ***Nonmember***

SUMMARY Fault-tolerance is an essential feature of the distributed systems where the possibility of a failure increases with the growth of the system. In spite of extensive researches over two decades, fault-tolerance systems have not succeeded in practical use. It is due to the high overhead and the unhandiness of the previous fault-tolerance systems. In this paper, we propose MPICH-GF, a user-transparent checkpointing system for grid-enabled MPICH. Our objectives are to fill the gap between the theory and the practice of fault-tolerance systems, and to provide a checkpointing-recovery system for grids. To build a fault-tolerant MPICH version, we have designed task migration, dynamic process management, and atomic message transfer. MPICH-GF requires no modification of application source codes, and it affects the MPICH communication characteristics as less as possible. The features of MPICH-GF are that it supports the direct message transfer mode and that all of the implementation has been done at the lower layer, that is, the abstract device level. We have evaluated MPICH-GF using NPB applications on Globus middleware.

key words: *Fault-tolerance, Checkpoint, Consistent recovery, MPI, Grid computing*

1. Introduction

A 'computational grid' (shortly a grid) is a specialized instance of distributed systems, which includes a heterogeneous collection of computers in different domains connected by networks [15], [16]. The grid has attracted considerable attentions for utilizing ubiquitous computational resources with a view of a single system image. It has been believed to benefit many computation-intensive parallel applications. Unfortunately, distributed systems such as the grid are not reliable enough to guarantee the completion of parallel processes in a determinate time because of their inherent failure factors. Even a single local failure can be fatal to parallel processes since it could nullify all of the computation results that have been executed in cooperation with one another. In order to increase the reliability of distributed systems, it is essential to provide fault-tolerance.

Checkpointing / rollback-recovery is a well-known technique for fault-tolerance. Checkpointing is an operation to store the states of processes into the stable storage for the purpose of recovery or migration [11]. A process can resume its previous state at any time with the latest checkpoint file. Periodic checkpointing can minimize the computation loss incurred by failures. Although several stand-alone checkpoint toolkits [22], [30], [37] have been proposed, they are not sufficient for parallel computing due to the following reasons. First, they cannot restore communication states such as sockets or shared memory. Second, they do not consider the causal relationship among the states of processes. Process states may be dependent on one another in the message-passing environment, hence, the recovery without consideration of dependency may tangle the global process states [8], [26]. Consistent recovery algorithms for message-passing processes have been extensively studied for over two decades. However, implementation of the algorithms seems another issue because most of researches assume the followings:

- A process, by itself, detects a failure and revives.
- Both revived and survived processes can communicate after recovery without any procedure of channel reconstruction.
- Checkpointed binary files are always available.

Indeed, there have been many efforts to make theories practical [3], [5], [9], [12], [17], [21], [23], [27], [29], [32], [33], [35]. Their approaches take different strategies in the following context: the system-level checkpointing versus the application-level checkpointing, the user-transparency versus the convenience of implementation, and the direct message transfer mode versus the indirect one. These strategies affect the architecture of fault-tolerance system significantly. The previous frameworks remain to be proofs or evaluation tools of the recovery algorithms and to the best of our knowledge there are only a few systems which succeed in practical use on the specific platform [10], [17].

Our goal in this paper is to construct a practical fault-tolerance system for message-passing applications on the grid. We have integrated rollback-recovery algorithms with the message-passing programming model. We present our MPICH-GF that is based

[†]The authors are with the School of Computer Science and Engineering, Seoul National University.
[††]The author is with the Department of Computer Engineering, Sejong University.
[†††]The author is with the Supercomputing Center, KISTI.
a) E-mail: nywoo@dcslab.snu.ac.kr
b) E-mail: jhs@dcslab.snu.ac.kr
c) E-mail: yeom@snu.ac.kr
d) E-mail: tspark@sejong.ac.kr
e) E-mail: hwpark@kisti.re.kr

on MPICH-G2 [19], the grid-enabled MPI implementation. MPICH-GF is completely transparent to users, so that the applications do not have to be re-written. Any in-transit messages during checkpointing are never lost whether they are blocking operations or non-blocking operations. One of the main implementation issues is to realize dynamic process management that is not specified in the original MPI standard (version 1). In order to allow a new MPI instance to communicate with the running processes, we have implemented the `MPI_Rejoin` function. MPICH-GF supports a coordinated checkpointing protocol, and other consistent recovery algorithms (e.g. message logging) are under development. MPICH-GF operates on Linux kernel ver 2.4 with Globus toolkit 2.2 *.

The rest of this paper is organized as follows. In Section 2, we present the concept of consistent recovery and the related works of the fault-tolerance system. Section 3 describes the operation of the original MPICH-G2. We propose the MPICH-GF architecture in Section 4 and address implementation issues in Section 5. The experimental results of MPICH-GF with Nas Parallel Benchmarks are shown in Section 6. We conclude this paper in the final section.

2. Background

2.1 Consistent Recovery

In the message-passing environment, states of processes have dependency with one another, through message-receipt events. A consistent system state is the one in which for every message-receipt event reflected in the system state, the corresponding sending-event should be reflected [17]. If a process rolls back to a past state while another process whose current state is dependent on the lost state does not roll-back, inconsistency occurs.

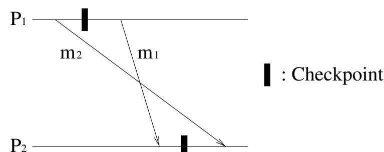


Fig. 1 Inconsistent global checkpoint

Figure 1 shows a simple example of two processes whose local checkpoints do not form a consistent global checkpoint. Suppose that two processes should roll-back to their latest local checkpoints. Process P_1 has not sent the message m_1 yet while P_2 has marked m_1 as being received. In this case, m_1 becomes an *orphan* message that causes inconsistency. Message m_2 is considered as a *lost* message, in the sense that P_2 waits for the arrival of m_2 which P_1 has marked as being already sent. An in-transit message gets lost on recovery time if it has not been recorded anywhere during checkpointing. Both of these messages cause an abnormal execution of processes during recovery.

Extensive researches on consistent recovery have been conducted [11]. Approaches to the consistent recovery can be categorized into the coordinated checkpointing, the communication induced checkpointing and the message logging protocols. In the coordinated checkpointing protocol, processes synchronize before local checkpointing, so that consistency in every global checkpoint can always be kept [8], [20]. On failure, all the processes roll-back to the latest global checkpoints. The protocol is simple, but it is a common idea that the coordination would not scale up. Communication induced checkpointing (CIC) allows processes to checkpoint independently as well as prevents the *domino effect* using the information piggybacked on the message [2], [36]. In [2], Alvisi *et al.* dispute against the belief that CIC would be scalable since CIC generates enormous forced checkpoints. Message logging records messages with checkpoint files in order to replay them on recovery. It minimizes the amount of lost computation with high storage overhead.

2.2 Related Works

Our concerns in categorizing the related works are whether they support direct or indirect message transfer and at which layer they implement fault-tolerance module, since those affect the performance and user-transparency issues respectively. Figure 2 shows our categorization.

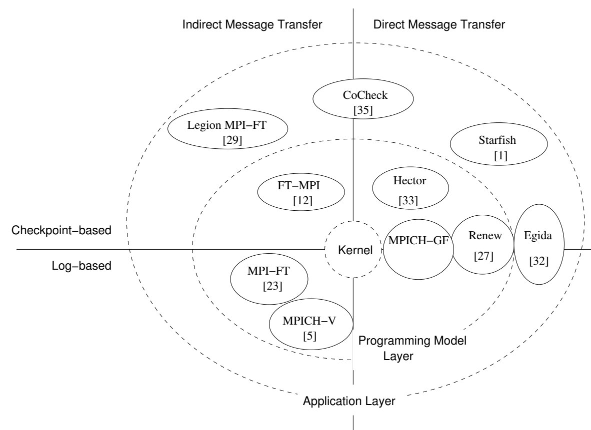


Fig. 2 The related works

With direct message transfer, application processes communicate with the other sibling processes directly, that is, client-to-client communication. With

*Argonne National Laboratory has proposed the Globus Toolkit for the framework of the grid which has been taken as the *de-facto* standard of grid services [14]

indirect message transfer, messages are transferred through intermediates like either local or remote daemons: PVM and LAM-MPI [6] are examples. In these systems, processes do not have to know the physical addresses of the other siblings, instead they maintain the connections with the intermediaries. Therefore, the recovery of communication context is relatively easy. Since the intermediaries can moderate the message transfer to be mutually exclusive to the checkpointing procedure, it is convenient to design the atomicity of the message transfer. For those reasons, fault-tolerance developers are favorable to the indirect message transfer mode. CoCheck [35] is a thin library over programming models, that supports coordinated checkpointing protocol for PVM and tuMPI. While CoCheck for tuMPI supports the direct transfer mode, the PVM version exploits PVM daemon to transfer messages. MPICH-V [5] is a fault-tolerant MPICH that supports pessimistic message logging, where all the messages are transferred to the remote *Channel Memory* (CM) servers that log and replay them. CMs are assumed to be stable so that revived processes can recover simply by reconnecting to CMs. The literature admits that it costs twice the message latency. FT-MPI [12] proposed by Fagg and Dongarra supports MPI-2's dynamic task management that is based on PVM or HARNESS daemons. In Li and Tsay's LAM-MPI based implementation, messages are transferred via a multicast server on each node [21]. MPI-FT from Cyprus University [23] adopts message logging. An observer processor copies all the messages and reproduces them on recovery, which requires a high storage cost for all the messages. MPI/FT proposed by Batchu *et al.* [3] adopts task redundancy for fault-tolerance. It has a central coordinator that relays messages to all the redundant processes. Legion MPI-FT [29] is the first effort to build fault-tolerance system on the grid, based on the LAM-MPI with daemon mode. It supports the coordinated checkpointing protocol where processes interchange complex control messages to ensure the absence of in-transit messages.

Since the increase in message delay is inevitable with the indirect message transfer mode, we have approached the direct one in favor of the performance issue. The followings are some previous research results supporting direct message transfer mode. Starfish [1] is a heterogeneous checkpointing toolkit based on Java virtual machine, which makes it possible for the processes to migrate among heterogeneous platforms. The limits of this system are that they have to be written in OCaml and that byte codes run more slowly than native codes. Egida [32] is an object-oriented toolkit that employs both communication induced checkpointing and message logging for MPICH. An event handler hijacks events of MPI operations from application codes in order to perform the appropriate actions for the consistent recovery algorithms. The current version of Egida can handle process failures only, not hardware failures. Hector [33] exists as a movable MPI library and several executables. It supports coordinated checkpointing; Before checkpointing, every process closes its channel connection to ensure that there is no in-transit message left in the network. Processes have to reconstruct the channel after checkpointing. RENEW [27] proposed by Neves *et al.* has a user-level reliable communication layer built upon the UDP layer, in order to log messages as well as to prevent the loss of in-transit messages. This technique requires additional memory copies at the sender side.

The related works can be further classified as their vertical locations in architecture: above or below programming models, or at kernel level. A kernel level approach is the most powerful in a sense that the range of checkpoint toolkit's reach is broad, but the integration with programming model is recommended for the abstraction of message events at the higher level as [2] describes. Fault-tolerance implementations above programming models are not efficient in abstracting the non-blocking message events, because the non-blocking communication function calls at the programming model do not coincide with the actual message delivery. For example, Egida above the programming model replaces non-blocking operations with blocking ones for the atomic message transfer. To avoid checkpointing during message transfer, those higher-level approaches require either coarse grain of checkpointing timing or the user-defined checkpointing location. We assume that the lower level approach can handle the non-blocking functions more precisely.

The locations of fault-tolerance module are dependent primarily on what kind of checkpoint toolkits are employed. Recently application-level checkpointing techniques are proposed for the purpose of the migration among heterogeneous nodes or of the decrease in checkpoint size [4], [28], [29], [31], [34]. These approaches burden users with decision of when to checkpoint, what to store in checkpoint file, and how to recover with the stored information. Although the system-level checkpoint file is not heterogeneous itself, we believe that user transparency is preferable for we are afraid that application developers may not accept such a programming effort.

3. MPICH-G2

Message Passing Interface (MPI) is the *de-facto* standard specification for parallel programming that abstracts low-level message-passing primitives away from developers [13]. Among several MPI implementations, MPICH [18] is the most popular for good performance and portability, which are due to the direct message transfer and the abstraction of low-level operations – the Abstract Device Interface (ADI) – respectively. MPICH version 1.2.3 includes about fifteen abstract

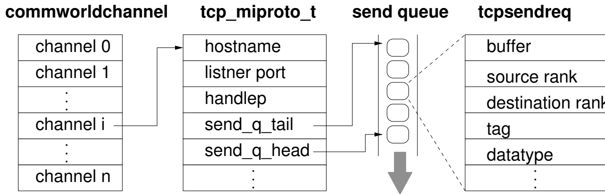


Fig. 3 `commworldchannel` and a send queue structures

devices and especially MPICH with *globus2*, a device for the grid, is called MPICH-G2 [19].

Collective communication in MPICH-G2 is implemented as a combination of point-to-point (shortly P2P) communications. MPICH has P2P communication primitives with the following semantics †: Blocking Send/Receive, Non-blocking Send/Receive, and Polling. The blocking operation submits a request to the kernel and waits until the kernel delivers the message. The non-blocking operation only registers its request and exits without waiting for the message delivery. A polling function performs actual message delivery. In MPICH-GF, the blocking operation consists of the non-blocking operation and the polling function.

The procedure of channel establishment in *globus2* device is as follows; Each MPI process opens a listener socket in order to accept a request for channel opening. The receiver opens another socket on the request and constructs a channel for two processes. At the MPI initialization, the master process with rank 0 collects the listener information of the others and broadcasts them. Figure 3 shows `commworldchannel`, the structure of process group table ††. The i-th entry in `commworldchannel` contains the information of a channel to the process with rank i. The pair of *hostname* and *port* is the address of a listener socket. *handlep* contains the real channel information. If *handlep* is null, the channel has not been opened yet. Send-operation pushes a request into the send-queue and registers this request to `globus_io` module.

There are two receive-queues in MPICH: the unexpected queue and the posted queue. The former is for messages that arrive earlier than the corresponding receive calls, the latter is for receive-requests that are waiting for the corresponding messages to arrive. If a receive-function is called, it examines the unexpected queue first whether the message has already arrived. If the corresponding message exists in the unexpected queue, it is delivered. Otherwise, the receive-request is enqueued into the posted queue.

† We exclude rendezvous operations in this paper, for we have rarely found their uses in applications

†† In Figure 3, we abstract the structure to show only the values of our concern

4. MPICH-GF

In this section, we present MPICH-GF's structure and checkpointing/recovery protocol implementation.

4.1 Structure

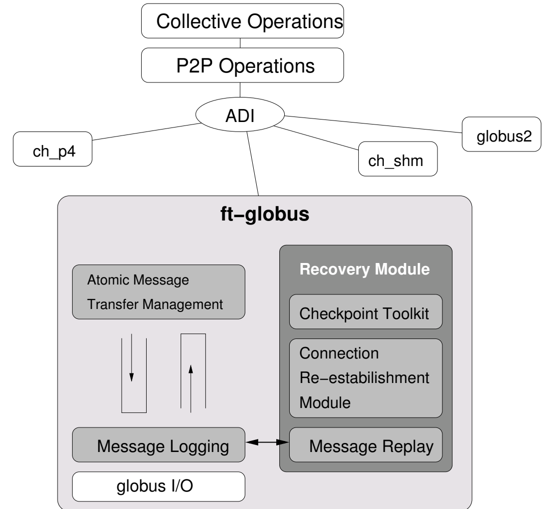


Fig. 4 The structure of *ft-globus* device

Figure 4 describes the MPICH-GF library structure. Although the fault-tolerance module has been implemented at the abstract device level, please note that it is still user-level. Our own abstract device, *ft-globus*, is originated from *globus2* abstract device. It contains a checkpoint toolkit, an atomic message transfer management, and a connection re-establishment module. Primitives of message logging/replaying are included, although this paper does not cover the message logging protocols. The checkpoint toolkit dumps the user-level memory image into the stable storage on the request. Messages that are in kernel memory or at network are assumed as in-transit messages. The atomicity of message transfer at the abstract device level realizes a fine grained checkpoint timing compared to the higher-level approach, because it narrows the code area that should be exclusive against the checkpointing procedure. The details of both atomic message transfer and connection re-establishment are described in Section 5.

4.2 Process Managers

Figure 5 describes how MPI processes are launched on the globus middleware. There are three main Globus modules that concern the process execution: DUROC (Dynamic Updated Request Online Co-allocator), GRAM (Globus Resource Allocation Management) job managers and a gatekeeper. DUROC distributes a user request to local GRAM modules, then a

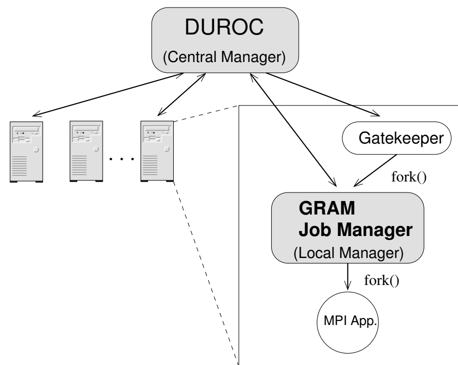


Fig. 5 The GRAM architecture

gatekeeper on each node checks whether the user is authenticated or not. If (s)he is an authenticated user, the gatekeeper launches a GRAM job manager. Finally, the GRAM job manager forks and executes requested processes. Though neither DUROC nor GRAM job manager controls processes dynamically, Globus has the fundamental framework of hierarchical process management. For dynamic process management, we've expanded DUROC and GRAM job manager's capabilities. From this context, we use the terms, *central/local manager* instead of DUROC and GRAM manager respectively for the convenience.

Those managers detect process/hardware failures and recover the failed processes without any user intervention. Since each local manager forks an application process, it receives a `SIGCHLD` signal when its forked application process terminates. Upon receiving the signal, the manager checks whether the termination was through normal `exit()`, by calling the system call, `waitpid()`. If that is the case, it assumes that the process completes its execution successfully. Otherwise, the local manager regards it as a process failure and notifies the central manager. It is possible for the local manager to fail as well. The central manager monitors all the local managers with periodic queries. If a local manager does not answer, the central manager assumes that the local manager has failed or hardware/network has failed. Then, it re-submits the request to GRAM module in order to restore the failed processes from the checkpoint files.

While the application messages are transferred among MPI processes directly, all control messages pass through managers.

4.3 Coordinated Checkpointing

For consistent recovery, the coordinated checkpointing protocol is employed. Figure 6 shows the procedure of coordinated checkpointing. The central manager initiates global checkpointing periodically, then local managers signal processes with `SIGUSR1` to be ready for checkpointing. On receipt of `SIGUSR1`, the signal handler executes a barrier-like function before checkpointing. By performing the barrier, two things can be guaranteed; One is that there is no orphan message between any two processes and the other is that there is no in-transit message because barrier messages push messages that are previously issued to the receiver. Since channels are built on TCP sockets, FIFO property is kept. Pushed messages are stored at the receiver's unexpected queue in the user-level memory so that a checkpoint file can include them. These in-transit messages are valid on recovery at the receiver's user-level memory. This technique is similar to *Ready Message* of CoCheck [35]. After coordination, each process generates the checkpoint file and informs the local manager of the successful completion of checkpointing. The central manager checks whether all the checkpoint files have been generated. If so, it confirms the collection of checkpoint files as a new version of global checkpoint.

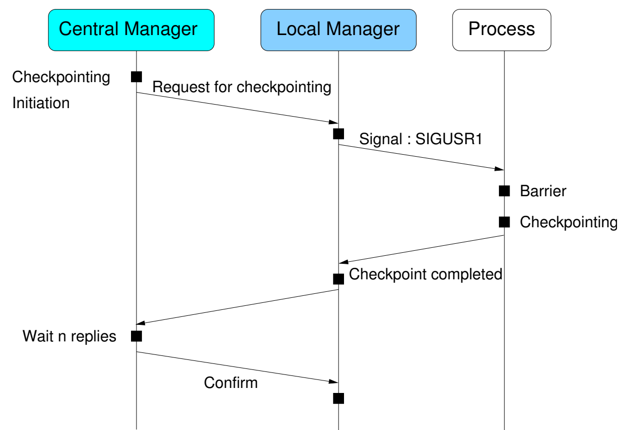


Fig. 6 Coordinated checkpointing protocol

4.4 Consistent Recovery

In coordinated checkpointing, a single failure results in the roll-back of all the processes to the consistent global checkpoint. The central manager broadcasts both the failure notification and the roll-back request. Our first approach to recovery was to kill all the processes and to recreate them by submitting sub-requests to the gatekeeper on each node. Since this approach takes too much time in re-authenticating requests and in reconstructing all the channels, it has been discarded. To improve the efficiency of recovery, we keep the survived processes without terminating them. Recovery is performed by dumping each checkpoint file to the memory using *exec()*. This technique affects only the user-level memory and the channel status remaining at the kernel side is unchanged. After dumping the memory, the survived processes can communicate without any channel reconstruction. Only the channels to failed processes should be reconstructed.

5. Implementation Issues

5.1 Communication Channel Reconstruction

The original MPI version 1 only specifies the static process group management, where once a process group and channels are built, they cannot be altered during run time. Since a recovered process instance is regarded as new instance at the view of the survived processes, it cannot join with the process group.

We have designed a new function called `MPI_Rejoin` in order to reconstruct the communication channels of restored processes. Before a restored process resumes the computation, it calls `MPI_Rejoin()` in order to re-initiate its channel information and to update its listener information of the other's `commworldchannel`. `MPI_Rejoin` informs the following values:

- *global rank*: the logical process ID of the previous run.
- *hostname*: the address of the current node where the process is restored.
- *port number*: the new listener port number.

The central manager collects this information and broadcasts it to every local manager. The survived processes renew the listener information of the restored process according to the information from its local manager. Figure 7 describes the interaction among managers and processes for the *MPI_Rejoin()* call. The restored process sets its handles null to re-initiate channels so that it can consider as if it has not created any channel to the others.

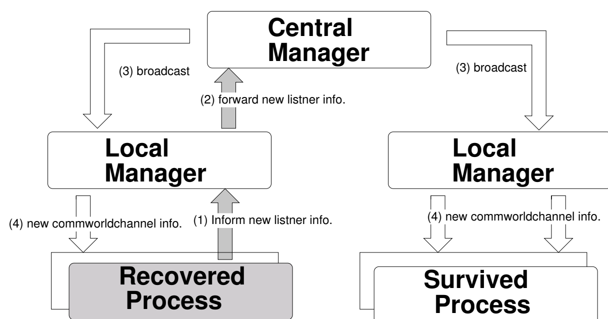


Fig. 7 Communication channel reconstruction protocol

5.2 Atomicity of Message Transfer

Messages in MPICH are sent being divided into the header and the payload. If checkpointing is performed after receiving a header but before receiving the corresponding payload, a partial message loss may happen. We design the atomicity of message transfer in order to store and restore the communication context safely. In other words, checkpointing is not performed while the message transfer is in progress. In MPICH-GF, the communication operations and the checkpointing procedure are mutually exclusive as shown in Figure 8.

```
MPID_function() {
    ...
    requested_for_ckpt = FALSE
    ft_globus_mutex = 1;
    ...
    (FUNCTION BODY)
    ...
    if (requested_for_ckpt == TRUE) then
        do_checkpoint();
    endif
    ft_globus_mutex = 0;
}

signal_handler(){
    if( ft_globus_mutex == 1) then
        requested_for_ckpt = TRUE;
    else
        do_checkpointing();
    return;
}
```

Fig. 8 Atomicity of message transfer

Each mutually exclusive area for send and receive operations has been implemented in different levels. We set the whole send-operation area as a critical section. The process status in checkpoint files should be either that any send-operation has not been called, or that a message has been sent completely. The kernel does not process any non-blocking send-request until a polling function is called. Since we do not want a checkpoint file to contain send-requests in the send-queue, MPICH-GF replaces non-blocking send-operations into blocking send-operations to ensure that no send-request exists in checkpoint files. If a process restores send-queue entries that are bound to the old physical channel information, they cannot be sent correctly. This replacement does not affect the correctness of computation.

The critical sections of the receive-operations are narrower than that of the send-operations. If the whole receive-operations are implemented as critical sections, the deadlock situation may happen in coordinated checkpointing as shown in Figure 9. In the figure, a sender is waiting for all the processes to enter the coordination procedure while a receiver is waiting for the arrival of the requested message. At the receiver side, the checkpointing procedure cannot start until the receive-call finishes. The blocking receive is a combination of the non-blocking receive and a loop of polling function. Actual message deliveries from the kernel to the user memory are done by polling. To prevent the deadlock, we set the polling function in the

loop as a critical section. The checkpoint file may contain receive-requests in the receive-queue, which does not matter because they are not related with physical channel information. In order to match arrived messages and the receive-requests, MPI checks only sender's rank and the message tag. So the recovered process can receive messages corresponding to the restored receive-requests. Finally, the non-blocking receive does not need to be replaced with the blocking one.

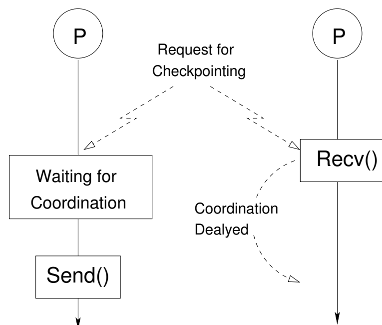


Fig. 9 Deadlock of blocking operation in coordinated checkpointing

6. Experimental Results

In this section, we present the performance results for five Nas Parallel Benchmark applications [25] – LU, BT, CG, IS and MG – executing on a cluster of four Intel Pentium III 800 MHz PCs with 256MB memory connected by ordinary 100 Mbps Ethernet. Linux kernel version 2.4 and Globus toolkit version 2.2 have been installed. The central manager queries local managers for their status every five seconds.

Table 1 shows the characteristics of the applications. Application IS uses all-to-all collective operations only. Though MG and LU use anonymous receive-operations with `MPI_ANY_SOURCE` option, the corresponding senders for the receive calls are determined statically by message tagging. LU is the most communication-intensive application, while the single message size is the smallest among the five applications. Each process of CG has two neighbors that it communicates intensively. The message size of CG is relatively large. BT is also a communication intensive application. each process has six neighbors in all directions of a 3D cube, and performs non-blocking operations extensively.

The checkpoint sizes are almost the same as the size of user-level memory that each process occupies at the checkpointing moment. If the available physical memory size is smaller than the checkpoint size, the execution time increases extremely due to the extensive disk swapping effect. We couldn't perform the upper class (C) of each application, because they require more memory than our cluster PCs are equiped with. Judging from the comparison to the executable binary sizes in Table 1, the dominant parts of a checkpoint file are the heap and the stack.

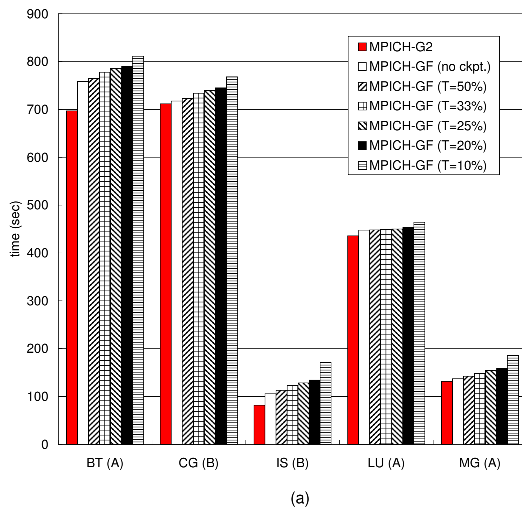


Fig. 10 Total execution time

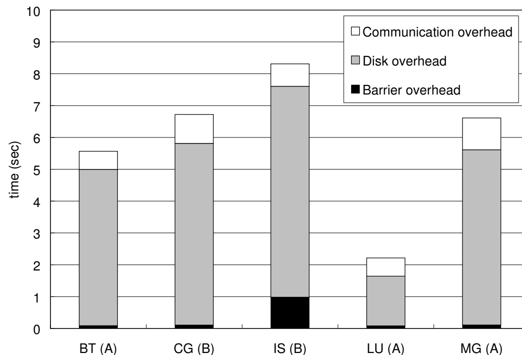


Fig. 11 Composition of the single checkpointing cost

Figure 10 shows the total execution time without any failure event. The first and second bars in the figure are for the original MPICH-G2 and for the MPICH-GF without checkpointing respectively. The difference between those two bars indicates the monitoring overhead and the effect of replacing non-blocking send-operations with blocking ones. The difference is outstanding with BT application which performs non-blocking operations frequently. In order to decrease this difference, we are designing the mechanism of handling the non-blocking send-operations for the next version of MPICH-GF. The other five bars show the execution time using MPICH-GF with 1, 2, 3, 4, and 9 times checkpointing respectively. LU shows the little difference among them except the last bar, since it has the

App. (Class)	Description	Comm. Pattern	Aver. Message Size (KB)	Number of Sent Messages per process	Executable file size (KB)	Aver. Checkpoint Size (MB)
BT (A)	Navier-Stokes Equation	3D Mesh	112.3	2440	1018.9	86.8
CG (B)	Conjugate gradient method	Chain	144.9	7990	993.2	125.9
IS (B)	Integer Sort	All-to-all	2839.2	106	901.1	154.9
LU (A)	LU Decomposition	Mesh	3.8	31526	1054.7	18.1
MG (A)	Multiple Grid	Cube	35.4	3046	952.3	124.2

Table 1 Characteristics of the NPB applications used in the experiments

small checkpoint file size. With the other applications, the overload is proportional to the number of checkpointing.

Figure 11 shows the average cost of single checkpointing and the composition. The checkpointing cost consists of barrier (coordination) cost, the disk writing cost, and the communication cost. The barrier cost is pretty small in all cases except IS. Because the message of IS is the biggest among all applications, it takes more time to handle the in-transit messages of IS. We have measured the barrier cost separately with up to 64 nodes; The cost does not exceed one second. The disk cost is dominant in single checkpointing and is almost proportional to the checkpoint file size as expected. We use the option `O_SYNC` at file open for the synchronous disk write that is considerably time-consuming. We assume the communication cost as the network delay among the hierarchical managers. It is the same for all the applications.

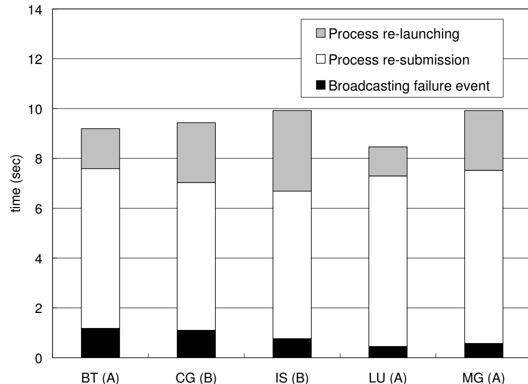


Fig. 12 Recovery cost

Figure 12 shows the recovery cost that is measured at the central manager from the failure detection to the completion of the process re-join. The recovery is performed as follows: failure broadcast, process re-submission, and process re-launching. The job re-submission includes another user-authentication, which is necessary in Globus middleware and is time-consuming. For the improvement, the next version of MPICH-GF will eliminate this redundant authentication by adopting multi-level hierarchical manager system. The process re-launching is primarily for the disk read of a checkpoint file. It also includes the re-join procedure.

Our experimental results show that the overload of fault-tolerance is mainly from the disk I/O operations rather than the scalability issues with the small number of nodes. Although more considerate experiments with larger size cluster are required, we expect that the disk I/O cost would be still dominant to some extent.

7. Conclusions

In this paper, we have presented the feasibility, architecture and evaluation of fault-tolerant system for grid-enabled MPICH. The proposed system, MPICH-GF, minimizes the loss of computation of processes by periodic checkpointing and guarantees the consistent recovery. While previous researches yiled the performance issues by approaching to the higher level implementation, MPICH-GF respects the communication characteristic of MPICH by the lower level approach. Consideration on communication context at the abstract device realizes a fine grain of checkpoint timing. Another feature of MPICH-GF is user-transparency for the user-convenience. It does not require any modification of application source codes or MPICH upper layer. All of our jobs have been accomplished at the user level.

The next version of MPICH-GF will solve the following problems. First, the weakness of our system is the exposure of the central manager to a single point of failure. Currently, we are implementing the replication technique on the managers for the high availability, though our research focuses mainly on checkpointing techniques. Second, to improve the performance, non-blocking send-operations should be managed in a more considerate way. The eliminating the redundant user-authentication from the recovery procedure is also required. Finally, implementations of message logging protocols are under going.

Acknowledgement

The ICT at Seoul National University provides research

facilities for this study.

References

[1] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," *Proc. IEEE Symp. on High Performance Distributed Computing*, pp. 167–176, August 1999.

[2] L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel, "An analysis of communication induced checkpointing," *Symp. on Fault-Tolerant Computing*, pp. 242–249, 1999.

[3] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte, "MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing", *Proc. 1st Int'l Symp. on Cluster Computing and the Grid*, May 2001.

[4] A. Beguelin, E. Seligman, and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations", *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, 1997.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. F. Magniette, V. Néri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes", *SuperComputing 2002*, pp. 1–18, 2002.

[6] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI", *Proc. Supercomputing Symp.*, pp. 379–386, Toronto, Canada, 1994.

[7] R. Butler and E. L. Lusk, "Monitors, messages, and clusters: The p4 parallel programming system", *Parallel Computing*, vol. 20, no. 4, pp. 547–564, 1994.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", *ACM Trans. on Computing Systems*, vol. 3, no. 1, pp. 63–75, Aug. 1985.

[9] Y. Chen, K. Li, and J. S. Plank, "CLIP: A checkpointing tool for message-passing parallel programs", *Proc. SC97: High Performance Networking and Computing*, Nov. 1997.

[10] I. B. M. Corporation, "IBM Loadleveler: Users Guide", Sept. 1993.

[11] E. N. Elnozahy, L. Alvisi, Y.-M.Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[12] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world", *PVM/MPI 2000*, pp. 346–353, 2000.

[13] M. P. I. Forum, "MPI: A Message Passing Interface Standard", May 1994.

[14] I. Foster and C. Kesselman, "The globus project: A status report", *Proc. the Heterogeneous Computing Workshop*, pp. 4–18, 1998.

[15] I. Foster and C. Kesselman, "The Grid: Blueprint for a Future Computing Infrastructure", Morgan Faufmann Publishers, 1999.

[16] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations", *Journal of Supercomputer Applications*, vol. 15, no. 3, 2001.

[17] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids", *Proc. 10th IEEE Symp. on High Performance Distributed Computing (HPDC10)*, Aug. 2001.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI Message Passing Interface Standard" *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[19] N. T. Karnois, B. Toonen, and I. Foster, "MPICH-G2: A grid-enabled implementation of the message passing interface", *Journal of Parallel and Distributed Computing*. vol. 63, no. 5, pp. 551–563, May 2003.

[20] R. Koo and S. Toueg, "Checkpointing and rollback recovery for distributed systems". *IEEE Trans. on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, 1987.

[21] W.-J. Li and J.-J. Tsay, "Checkpointing message-passing interface(MPI) parallel programs", *Pacific Rim Int'l Symp. on Fault-Tolerant Systems (PRFTS)*, 1997.

[22] M. J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the unix kernel" *USENIX Conference Proceedings*, pp. 283–290, San Francisco, CA, Jan. 1992.

[23] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "Portable fault tolerance scheme for MPI", *Parallel Processing Letters*, vol. 10, no. 4, pp. 371–382, 2000.

[24] K. Z. Meth and W. G. Tuel, "Parallel checkpoint/restart without message logging", *Proc. of the 2000 Int'l Workshops on Parallel Processing*, 2000.

[25] NASA Ames Research Center. "Nas parallel benchmarks", *Technical report*, http://science.nas.nasa.gov/Software/NPB/, 1997.

[26] R. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots", *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165–169, 1995.

[27] N. Neves and W. K. Fuchs, "RENEW: A tool for fast and efficient implementation of checkpoint protocols", *Symp. on Fault-Tolerant Computing*, pp. 58–67, 1998.

[28] G. T. Nguyen, V. D. Tran, and M. Kotocová, "Application recovery in parallel programming environment", *European PVM/MPI*, pp. 234–242, 2002.

[29] A. Nguyen-Tuong, "Integrating Fault-Tolerance Techniques in Grid Applications", PhD thesis, University of Virginia, USA, 2000.

[30] J. S. Plank, M. beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix", *USENIX Winter 1995 Technical Conference*, Jan. 1995.

[31] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing", *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[32] S. Rao, L. Alvisi, and H. M. Vin, "Egida: An extensible toolkit for low-overhead fault-tolerance", *Symp. on Fault-Tolerant Computing*, pp. 48–55, 1999.

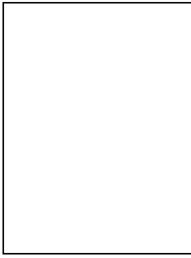
[33] S. H. Russ, J. Robinson, B. K. Flachs, and B. Heckel, "The hector distributed run-time environment", *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1102–1114, Nov. 1998.

[34] L. M. Silva and J. G. Silva, "System-level versus user-defined checkpointing" *Symp. on Reliable Distributed Systems*, pp. 68–74, 1998.

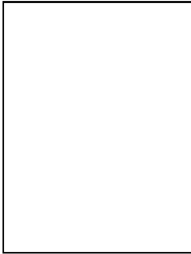
[35] G. Stellner, "CoCheck: Checkpointing and process migration for MPI", *Proc. the Int'l Parallel Processing Symp.*, pp. 526–531, Apr. 1996.

[36] J. Tsai, S.-Y. Kuo, and Y.-M. Wang, "Theoretical analysis for communication-induced checkpointing protocols with rollback dependency trackability", *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 963–971, 1998.

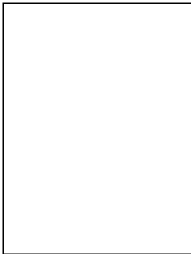
[37] V. Zandy, B. Miller, and M. Livny, "Process hijacking", *Eighth Int'l Symp. on High Performance Distributed Computing*, pp. 177–184, Aug. 1999.



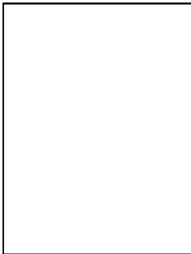
Namyoon Woo received his BS and MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1997 and 1999 respectively. Currently he is a PhD candidate at Seoul National University. His research interests include fault tolerance, distributed systems, grid computing, multiple task scheduling and user-level communications.



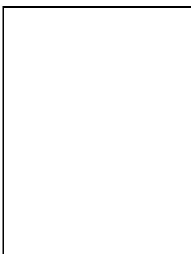
Hyungsoo Jung received his BS in mechanical engineering from Korea University, Seoul, Korea, in 2002 and MS degree in computer science from Seoul National University, Seoul, Korea, in 2004. Currently he is a PhD candidate at Seoul National University. His research interests are computer systems: operating system, networking, compilers, and parallel programming languages for distributed systems.



Heon Y. Yeom is an Associate Professor with the Department of Computer Science and Engineering, Seoul National University. He received his BS degree in computer science from Seoul National University in 1984 and received the MS and PhD degree in computer science from Texas A&M University in 1986 and 1992, respectively. From 1986 to 1990, he worked with Texas Transportation Institute as a Systems Analyst and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems, multimedia systems and transaction processing.



Taesoon Park received the BS degree in computer engineering from Seoul National University, Seoul, Korea, in 1987; and the MS and the PhD degrees in computer science from Texas A&M University, College Station, TX in 1989 and 1994, respectively. She is currently an Assistant Professor in Sejong University, Seoul, Korea. Her research interests are in the areas of fault-tolerant and distributed computing systems.



Hyoungwoo Park is a Present Head of Grid Technology Research Department, KISTI Supercomputing Center, Korea. He received his BA in electronic engineering from Seoul City University, Korea in 1985, and his MA and PhD in Information Engineering and Information Communication from SungKyunKwan University, Korea in 1996 and 2001 respectively. His research interests are grid computing and NGI.