Process Migration for MPI Applications based on Coordinated Checkpoint

Jiannong Cao, Yinghao Li
Department of Computing
The Hong Kong Polytechnic University
Kowloon, Hung Hom
Hong Kong, China PR
{csjcao, c2990103}@comp.polyu.edu.hk

Minyi Guo
Department of Computer Software
The University of Aizu
Aizu-Wakamatsu City,
Fukushima 965-8580, Japan
minyi@u-aizu.ac.jp

Abstract

A lot of research has been done on fault-tolerance for MPI applications, some on checkpoint/restart, and some on network fault-tolerance. Process migration, however, has not gained widespread use due to the additional complexity of the requirement that the knowledge about the new location of a migrated process has to be made known to every other process in the application. Here we present a simple yet effective method of process migration based on coordinated checkpointing of MPI applications. Migration is achieved by checkpointing the application, modifying the process location information in the checkpoint files, and restarting the application. Checkpoint/restart and migration are transparent to MPI applications. Performance evaluation results showed that the additional checkpoint/restart capability has little impact on application performance, and the migration method scales well on a large number of nodes.

Keywords: process migration, checkpoint/restart, coordinated checkpoint, MPI

1 Introduction

In recent years, the parallel computing community has seen a trend of building high-performance computers with clusters of workstations instead of traditional massive parallel processor (MPP) architectures. The cluster architecture offers a more cost effective way to build high-performance computers. Many of these clusters, however, built with commodity hardware and software, present great challenges on reliability and scalability.

The Message Passing Interface (MPI) [18] is a *de facto* standard for communication among the nodes running a parallel program on a distributed memory system. However, the MPI standard itself does not specify any checkpoint or process migration behavior to support reliable and scalable execution of applications. Many widely used MPI implementations have not been designed to be fault-tolerant.

In this paper, we present a simple method and prototype implementation for process migration based on coordinated checkpoints for MPI applications. Our solution has the following features:

Portability: The system is an extension to LAM/MPI [10], a widely used and open-source MPI implementation, and Berkeley Lab's Checkpoint/Restart (BLCR) [3] library, a kernel level checkpoint system. LAM/MPI integrates with BLCR through a checkpoint interface, which allows checkpoint/restart support to be extended to other checkpoint libraries. Our prototype system does not modify LAM/MPI or BLCR, but only modifies the process location information in the checkpoint files produced by them, so it can be used with existing installations. The system can also be modified to work with other checkpoint systems.

Transparency: The LAM/MPI integration with BLCR is transparent to MPI programs. Source code of MPI programs does not need to be modified and migration is involuntary. To perform process migration, our system needs process location information of every MPI process in the application. In the current implementation, some information can only be obtained by calling a function at the beginning of the MPI program. This restriction can later be eliminated by a more thorough study of the LAM/MPI runtime system. Apart from that, MPI programs are totally unaware of the migration.

Simplicity: Since our system is based on coordinated checkpoint, process migration is performed by merely modifying the checkpoint files in between checkpointing and restarting. Synchronization is dealt with at the checkpoint

stage. Such approach does not require modification of either the MPI implementation or the checkpoint/restart library. The algorithm for processing checkpoint files is simple and fast.
Performance: According to the experiments, adding coordinated checkpoint/restart to LAM/MPI has insignificant impact on message passing performance. Noticeable application latency occurs only during checkpointing. The percentage of increases in running time can be further reduced by adopting a suitable time interval between checkpoints according to the scale of the application.

Our prototype system proves that the concept of process migration for MPI applications based on coordinated checkpoints is feasible. The performance is promising and the implementation is simple and straightforward. Because of these features, our system can be adopted easily to work with other MPI implementations with coordinated checkpoint capability, thus benefiting users with immediate need of process migration for MPI applications.

2 Background and Related Work

In this section we present the issues in checkpoint/restart and process migration for parallel programs, as well as some previous work in related fields. We also present considerations when choosing an MPI implementation to build our prototype system.

2.1 Process Migration

Process migration is the act of transferring a process from one machine to another for continuing its execution [4]. Process migration is particularly useful for long-running MPI applications when the cost of restarting the job on a different set of nodes from the beginning is not acceptable.

Migration of processes in an MPI application is more difficult than that of individual processes because MPI processes are more tightly coupled. If an MPI process is migrated while the application is still running, every other process that communicates with the migrated one must be informed of its new location. If there are pending messages involving the migrating process between the start point of migration and the point when notification of peers is completed, the application will fail.

One solution is to use a communication broker to coordinate message passing. Only the broker is aware of the location of processes and all messages are sent to the broker to be dispatched to their destination. This approach, however, will greatly reduce communication bandwidth and increase latency.

The other solution is to coordinate the processes to reach a consistent global state. Before a process migration can commence, requests are sent to all MPI processes and they interact with each other to guarantee that their local state will result in a consistent global state where no pending message exists. This is analogous to a coordinated checkpoint. The CL protocol of MPICH-V and LAM/MPI use this method [2] [15].

2.2 Checkpointing

In the context of message-passing parallel applications, a *global state* is a collection of the individual states of all participating processes and of the status of the communication channels. A *consistent global state* is one that may occur during a failure-free, correct execution of a distributed computation [15]. Within a consistent global state, if a given process has a local state indicating that a particular message has been received, then the state of the corresponding sender must indicate that the message has been sent [9]. A *consistent global checkpoint* is a set of local checkpoints, one for each process, forming a consistent global state. Any consistent global checkpoint can be used to restart process execution upon failure.

Checkpoint/restart implementations can be broadly categorized into two classes, system level and user level [19]. System level implementations require the library be compiled into the operating system kernel or as a loadable kernel module, since they need to access data in the system kernel. Berkeley Lab's Checkpoint/Restart (BLCR) library is an implementation of such kind. It supports transparent checkpoint/restart of multi-threaded applications on Linux. User level libraries require minimal modification of the operating system kernel. But most of them require either pre-processing of the source code, or linking the object code with their library routine. Also, since user space programs can not access the system kernel, these implementations usually have fewer capabilities than system level implementations. File descriptor, sockets, etc. may not be properly saved upon checkpoints. Condor [17] and Libckpt [7] are examples of user level checkpoint/restart libraries.

There are several tools that support checkpoint/restart in MPI. CoCheck [5] is an independent application for checkpointing PVM and MPI applications. However, it was built to work with tuMPI, a research purpose MPI implementation that has not gained widespread use. Another consideration is that the MPI version of CoCheck is no longer supported by its developers. Starfish [1] provides failure detection and recovery at the runtime level for dynamic and static MPI-2 programs. However, Starfish only supports MPI applications written in the OCaml programming language, which limited its practical usage. MPICH-V [14] is an extension to the popular MPICH [13] implementation and features three different protocols, V1, V2, and CL. V1 adopts uncoordinated checkpoint and remote message pessimistic logging through channel memories. Processes are checkpointed independently and stored on checkpoint servers. All communication traffic goes through the channel memory servers for logging, so the number of such servers is directly proportional to the throughput of message passing traffic. These servers also lead to additional latency during message passing. V2 adopts sender based pessimistic message logging instead of channel memory servers. A communication daemon runs on each node and messages go through it for logging. Direct point-to-point communication is achieved in this protocol, but the use of these daemons still yield latency. CL protocol of MPICH-V adopts the Chandy-Lamport's algorithm [9] for coordinated checkpoint. MPICH-V integrates the Condor checkpoint library [11] for checkpoint/restart of MPI processes.

Most of the tools described above are designed for MPI implementations that have not gained widespread industrial usage. LAM/MPI, on the other hand, has gained a wide range of users. It also has outstanding performance compared to other implementations [12]. LAM/MPI has built-in support for coordinated checkpointing, which provides us with a platform for process migration based on checkpoint files. Currently, it integrates with BLCR as its checkpoint library, but the modular design of LAM/MPI makes it easy to integrate other checkpoint libraries. To enable checkpoint/restart support in LAM/MPI, one has to use a "cr-aware" communication module (one that can accept checkpoint requests and coordinate with processes to reach a consistent global state). At the moment, crtcp (TCP socket with checkpoint/restart) and gm (Myrinet interconnect) module support this. Support for InfiniBand and other high-speed interconnects are going underway.

Previous research work indicates that, in some cases, it is possible to boot LAM/MPI on a set of nodes, start the MPI application, checkpoint, and reboot LAM/MPI on another set of nodes with the same topology [15]. In this approach, however, LAM node (as opposed to real computer) is the unit of migration rather than processes. This approach also requires the modification of checkpoint files. Rebooting the LAM/MPI increases process migration cost and complexity.

3 Design and Implementation

This section presents the design and implementation of our prototype system. The main idea of our system is to perform coordinated checkpointing of the MPI application, modify process location related information in the checkpoint files, and restart the application using the modified checkpoint files. The system currently supports process migration for MPI-1 applications. It is built on top of, but not limited to, LAM/MPI and BLCR. Currently, MPI-2 dynamic processes can not be checkpointed by LAM/MPI, thus can not be migrated using our system.

In a failure-free (in which no failure actually occurs) environment, coordinated checkpoint approach has smaller overhead compared to uncoordinated approach. So for smaller scale parallel programs which have less probability of encountering a failure during execution, this approach has little impact on performance. For bigger applications that run for a long time, the longer time for recovering from failures (compared to log-based approaches) is still reasonable comparing to the total time needed for the application. Therefore coordinated checkpoint and process migration based on coordinated checkpoint are adopted.

3.1 Process Migration based on Coordinated Checkpointing

Our prototype system is based on the coordinated checkpoint approach of LAM/MPI. The process of a checkpoint/restart in LAM/MPI is summarized below [15].

Sequence of events at checkpoint:

1. **mpirun**: receives a checkpoint request from a user or batch scheduler.
2. **mpirun**: propagates the checkpoint request to each MPI process.

3. **mpirun**: indicates that it is ready to be checkpointed.
4. **each MPI process**: coordinates with the others to reach a consistent global state in which the MPI job can be checkpointed. For example, processes using TCP for MPI message passing drain in-flight messages from the network to achieve a consistent global state.
5. **each MPI process**: indicates that it is ready to be individually checkpointed.
6. **underlying checkpointer**: saves the execution context of each process to stable storage.
7. **each MPI process**: continues execution after the checkpoint is taken.

Sequence of events at restart:

1. **mpirun**: restarts all the process from the saved process images.
2. **each MPI process**: sends its new process information to `mpirun`.
3. **mpirun**: updates the global list containing information about each process in the MPI job and broadcasts it to all processes.
4. **each MPI process**: receives information about all the other processes from `mpirun`.
5. **each MPI process**: re-builds its communication channels with the other processes.
6. **each MPI process**: resumes execution from the saved state.

Since the checkpoint files contain execution context of MPI processes in a consistent global state, modifying the process location information in these files is equal to notifying all processes of the migration event. Modifying the checkpoint files has two additional benefits. First, multiple migrations can be performed simultaneously by modifying multiple variables in the checkpoint files. Second, checkpoint files can be used to restart the application in case of failures.

At the moment, there are several limitations on checkpoint/restart support in LAM/MPI. The system can not checkpoint dynamically spawned processes in MPI-2. BLCR only supports Linux kernel version 2.4 on several architectures. Our system's migration ability is also affected by these limitations.

Our system assumes a stable, shared storage so that the checkpoint files of processes on different nodes can be managed and processed in a centralized manner. Shared storage also assures that checkpoint files of all the processes are available to all nodes in the system, so that the checkpoint of a process can be restarted on another node which also has access to the checkpoint files.

3.2 Location Sensitive Information

To perform process migration, the following location information in all of the checkpoint files must be modified.

Checkpoint file names: The LAM/MPI and BLCR checkpoint system will generate one checkpoint file for the `mpirun` process and one checkpoint file for each of the MPI processes. The checkpoint file of the `mpirun` process is named in the form of "context.PID", where PID is the process ID of `mpirun`. The checkpoint files of the MPI processes are named in the form of "`context.PID1-nNID-PID2`", where `PID1` is the process ID of `mpirun`, `NID` is the node ID on which the process runs, and `PID2` is the ID of the MPI process. To reflect the migration, the `NID` field in the file name of the process checkpoint to migrate is changed to its new node ID.

Process location: LAM/MPI use type `struct _gps` to identify the location of a process in the LAM universe [8]. The definition of this type is given below:

```
struct _gps {
   int4 gps_node;
   int4 gps_pid;
   int4 gps_idx;
   int4 gps_grank;
};
```

The individual elements are:

- `gps_node`: The node ID in the LAM universe where the process is running. This will be an integer in [0, N), where N is the number of nodes in the LAM universe.
- `gps_pid`: The POSIX PID of the process that invoked MPI_INIT.
- `gps_idx`: The index of the process in the local LAM daemon's process table.
- `gps_grank`: The "global rank" of the process. This is the integer rank of this process in MPI_COMM_WORLD.

Each process in a running MPI application, including the `mpirun` process, keeps local copies of `_gps` for every MPI process in the MPI application. Since the checkpoint library stores execution context of processes in the checkpoint files, these process location structures can also be found in the checkpoint files. One key step in our migration prototype is to search in the checkpoint files `_gps` of process we want to migrate, and replace gps_node with the node number to migrate to.

LAM/MPI uses a daemon on each node for process control, meta-environment control, and, in some cases, message passing. Due to some implementation limitations, the field gps_idx is no longer useful when the MPI application is checkpointed, stopped and restarted. So we choose to neglect this field when modifying process location. This issue will later be addressed through our coordination with the LAM/MPI team.

Checkpoint of mpirun: The mpirun process keeps a string for each MPI process as the command to restart that process. The string is in the form of "path filename", where path is the absolute path to the executable of restart command of the checkpoint/restart library (in the case of BLCR, cr_restart) and filename is the file name of the checkpoint of that process. Since we rename the checkpoint file to reflect process migration, this filename will also be modified.

Checkpoint of process to migrate: In the checkpoint file of the process to migrate, there is a string "<nNID/PID/RANK>crtcp: ". This is a debug message used by the crtcp module (TCP communication with checkpoint/restart support). NID is the node ID the process is running on; PID is the POSIX PID of the process and RANK is the global rank of the process. The system replaces NID with the node ID of the target node for migration.

There is also a string in the form of "TMP/lam-USER@HOST". TMP is the system temporary directory, by default "/tmp"; USER is the username of the user running MPI applications; HOST is the hostname of the node. This is the path to the temporary directory of LAM/MPI. When a process is migrated to another node, hostname of the node is also changed. Therefore we need to replace HOST with the hostname of the target node of migration.

3.3 Obtaining the Process Location

To perform process migration on a process in an MPI application, we need the complete _gps structure of that process. It is more convenient to construct a table containing _gps structure of every MPI process when the MPI application is started. The table can later be used to locate the _gps structure of any process and do migration on that process.

Our system uses a wrapper program of mpirun command to start MPI applications. The wrapper delegates all its command line arguments to mpirun, and starts a modified version of mpitask (a utility in LAM/MPI that displays information of running MPI processes) in another process to obtain the process location table. Our modified version of mpitask generates process ID in addition to node ID and index in the original version. Due to some technical difficulties, the rank of running processes, however, can not be obtained. An additional function call is placed right after MPI_INIT in the MPI program. This function gathers rank, node ID and process ID of every process and passes the information to the wrapper program. Based on these two sources of information, the wrapper program can produce a complete table of _gps and write it in a file for later reuse.

3.4 Processing the Checkpoint Files

We provide a program to automate the task of processing checkpoint files. It accepts a task file as input and modifies the checkpoint files according to the migration tasks in the task file.

The structure of the file is very simple. It contains the process ID of mpirun and a set of migration tasks. Each task is composed of four members of the _gps structure of the process to be migrated and the node ID it will be migrated to. The task file can be produced by manually modifying the process location table file described in the previous section.

The program scans parts of the checkpoint files for relevant variables and replaces them with designated values. It also renames checkpoint files of nodes to reflect the migration.

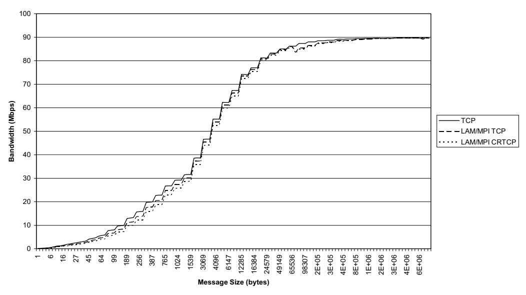
4 Performance Evaluation

Three sets of experiments were conducted to measure the performance of our prototype system. The first set measures communication performance using NetPIPE [16] (A Network Protocol Independent Performance Evaluator). The second set measures overhead of checkpoint/restart using High-Performance Linpack [6]. The third set measures processing time of checkpoint files. The test platform is a 4-node Linux mini-cluster. Each node has a Pentium III 500Mhz processor; 256 MB of memory and Fast Ethernet interconnect.

4.1 Communication Performance

NetPIPE is a program that performs ping-pong tests, bouncing messages of increasing size between two processes across a network to measure communication performance. We ran NetPIPE on top of TCP sockets, LAM/MPI's TCP RPI module (TCP point-to-point channel without checkpoint/restart support), and CRTCP module (TCP with checkpoint/restart). As seen in figure 1, the TCP communication module of LAM/MPI causes slight overhead than the native TCP socket communication. The CRTCP module has more overhead than its counterpart with no checkpoint/restart support. But the discrepancy is almost negligible.

Figure 1: Communication Performance



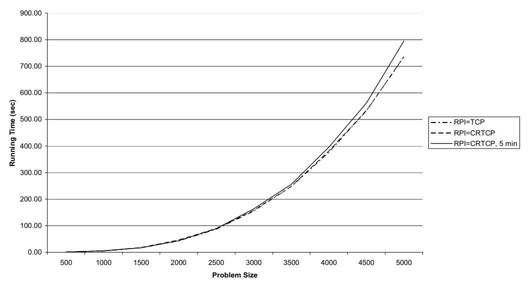
4.2 Checkpoint/Restart Restart

High-Performance Linpack is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers, commonly used to benchmark parallel computers. Since the problem size can be defined by the user, it offers us a convenient way to evaluate the performance of checkpoint when dealing with processes of different sizes. When the problem size reaches 5000, the checkpoint file of each process is almost as big as 180MB. MPI process that occupies this amount of memory is not uncommon in large-scale applications.

We conducted the experiment using problem sizes from 500 to 5000 with an interval of 500. The same experiment is carried out three times, the first time with TCP module, the second with CRTCP module, and the third with CRTCP module and a 5 minutes checkpoint interval. As shown in figure 2, experiments using TCP and CRTCP module without checkpoint have almost the same execution time. The experiment with CRTCP module and checkpoint interval of 5 minutes, however, yields extra execution time of about 6%. The percentage of extra execution time increases as the problem size increases. It reaches 8% when the problem size is 5000. The execution time is also expected to increase drastically when the number of MPI processes increases.

The performance bottleneck in these experiments lies in the file system, since the four nodes in our testing cluster use a shared network file system located on one of the nodes. Each node is equipped with only one SCSI hard disk and all of them are connected with Fast Ethernet. The performance is greatly reduced when all processes are transferring and storing checkpoint files concurrently to the file server. This issue can be improved by using some high-performance cluster file system.

Figure 2: Checkpoint Overhead



4.3 Processing Time of Checkpoint Files

The processing time of a single checkpoint file is independent of the file size because the processing program only scans several designated areas of fixed size in the file. The total processing time is only affected by the product of the number of processes and the number of migration tasks. Experiments show that migrating one, two, three, or four processes out of four requires about 1.8, 5.2, 5.8, or 6.0 seconds respectively. The speed can be further increased by adjusting the algorithm for processing the checkpoint files.

We used HPL again to test the integrity of the system, since it involves extensive message passing traffic and heavy computation that resembles many real live applications. Only two lines of code are added to enable process migration in HPL. Our system succeeded in migrating arbitrary process to any node within the LAM universe. It can also migrate processes to nodes added at a later time. Even processes on the master node (the node from which the MPI program is initiated) can be migrated, result in a system with no single point of failure.

5 Future Work

While the prototype discussed in this paper proved the feasibility of the idea, future work on several directions is planned. The first priority is to refine the system for practical use, such as eliminating the need for modifying the MPI programs, and taking `gps_idx` into account when processing the checkpoint file. The next step is to increase the speed of processing checkpoint files, especially when dealing with a large number of MPI processes. The algorithm for processing checkpoint files will also be adjusted to reduce processing time, possibly by omitting checkpoints of processes that are not affected by the migration, and by locating the variables of interest more effectively through in depth study of the structure of checkpoint files. The long term goal is to integrate the system with some monitoring and scheduling system to build a fault-tolerant MPI runtime environment. The system should be able to recover from node failures and to perform processes migration when more powerful nodes are added or when system load of the working nodes change.

6 Conclusions

This paper presents a simple yet effective approach to process migration of MPI applications. A prototype system is built for LAM/MPI and BLCR. It performs process migration by modifying the process location information in the coordinated checkpoint files. Performance tests were conducted and results show that adding checkpoint/restart support to MPI leads to trivial performance penalty, and the processing time of checkpoint files is insignificant. In a failure-free environment, the execution time of a checkpoint and migration enabled application is almost identical to one without it.

7 References

[1] A. M. Agbaria, and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, p. 31, 1999.*

[2] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated Checkpoint versus Message Log for Fault Tolerant MPI. *IEEE International Conference on Cluster Computing (CLUSTER'03), p. 242, Dec 01-04, 2004, Hong Kong.*

[3] Berkeley Lab Checkpoint/Restart (BLCR). http://ftg.lbl.gov/checkpoint.

[4] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys (CSUR), Vol. 32, Issue 3, p. 241-299, 2000.*

[5] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium, p. 526-531, 1996.*

[6] HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl.

[7] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. *Technical Report: UT-CS-94-242, 1994.*

[8] J. M. Squyres, B. Barrett, A. Lumsdaine. Request Progression Interface (RPI) System Services Interface (SSI) Modules for LAM/MPI, API Version 1.0.0 / SSI Version 1.0.0.

[9] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS), Vol. 3, Issue 1, p. 63-75, 1985.*

[10] LAM/MPI Parallel Computing. http://www.lam-mpi.org.

[11] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. *University of Wisconsin-Madison Computer Sciences Technical Report #1346, April 1997.*

[12] MPI Performance on Coral. http://www.icase.edu/coral/mpi/MPIonCoral.html.

[13] MPICH - A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[14] MPICH-V – Introduction. http://www.lri.fr/~gk/MPICH-V.

[15] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. Los Alamos Computer Science Institute *(LACSI) Symposium, Oct 2003.*

[16] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. *http://www.scl.ameslab.gov/netpipe.*

[17] The Condor Project Homepage. http://www.cs.wisc.edu/condor.

[18] The Message Passing Inteface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi.

[19] Y. M. Wang, Y. Huang, K.P. Vo, P. Y. Chung, and C. Kintala. Checkpointing and Its Applications. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, p. 22, 1995.*