MPICkpt: A Transparent Checkpointing Tool for MPI

C. R. Dow, J. S. Chen, J. C. Chen, and M. C. Hsieh
Department of Information Engineering
Feng Chia University, Taichung, Taiwan
{crdow, jschen, jcchen, mchsieh}@crab.iecs.fcu.edu.tw

Abstract

This work designs and implements MPICkpt, a transparent and multifunctional checkpointing system for MPI. In addition to employing two kinds of checkpointing techniques, i.e. coordinated and independent, MPICkpt includes various optimized checkpointing functions, e.g. forked, incremental, and compressed functions, to reduce the checkpointing overhead. The user can either take checkpoints in some specified locations of the source program or let the system do so automatically and in a fully transparent manner. In addition, a checkpointing visualization facility is developed and implemented to evaluate the effectiveness of various checkpointing techniques. Experimental results indicate that the performance of checkpointing depends on how the applications function.

Keywords: Parallel/distributed systems, MPI, fault-tolerance, checkpointing, rollback recovery and visualization tools.

1. Introduction

Checkpointing is extensively used in many distributed/parallel applications, such as fault-tolerance [5, 9], debugging [8, 10, 13], and mobile computing [1, 3, 25]. During normal execution, the state of each process is periodically saved on stable storage as a checkpoint. When a failure occurs, each process can then roll back to its previous checkpoint by reloading the saved state. In a message passing system, checkpointing consists of two approaches: coordinated and independent. The former involves a scenario in which a process coordinates other processes to save their state simultaneously. The latter does not require system-wide coordination and, therefore, may scale better [9].

The MPI (Message Passing Interface) standard [28-29] largely focuses on providing a portable, efficient, and feasible interface for message passing. Although various systems and tools have been developed and implemented for MPI, e.g. compilers, debuggers and performance evaluation tools, few checkpointing systems have been developed for MPI. In general, most checkpointing systems are based on coordinated checkpointing techniques. However, demonstrating that coordinated checkpointing is more appropriate than independent checkpointing techniques for various applications is impossible. For instance, independent checkpointing is more appropriate than coordinated checkpointing with respect to mission-critical service-providing applications [27]. Therefore, a comprehensive checkpointing system should provide coordinated and independent checkpointing facilities.

A checkpointing system is *transparent* if the user does not need to change the source code of an application program to take a checkpoint. Checkpointing transparency allows not only for the concealment of checkpoints, but also for the user to complete an application program without knowledge of communication patterns and behaviors of the program. Transparency conceals and renders anonymous the resources of checkpointing that are not directly relevant to the task-in-hand from the user. Although user level checkpointers can advance the portability, transparency is difficult to achieve in a user level checkpointing library [23].

In this work, these problems are resolved by designing and implementing MPICkpt, a transparent and multifunctional checkpointing system for MPI. Among the range of facilities that MPICkpt supports includes coordinated and independent checkpointing functions, sequential, forked, incremental, and compressed facilities, a fully transparent checkpointing facility, a graphical user interface, and a checkpointing visualization tool. The rest of this paper is organized as follows. Section 2 discusses background material useful in this work. Section 3 describes our system architecture. Section 4 presents an independent checkpointing algorithm implemented in MPICkpt. Next, Section 5 describes how to implement MPICkpt. Section 6 then illustrates the MPICkpt prototype and its functionalities. Section 7 summarizes the experimental studies performed on the MPICkpt prototype. Finally, conclusions and areas for future work are discussed in Section 8.

2. Related Work

Checkpoint-based rollback-recovery techniques [1, 3-5, 9, 18-21, 27] can be classified into three categories: coordinated checkpointing [1, 25], independent checkpointing [4, 9, 18], and communication-induced checkpointing [9, 19]. Coordinated checkpointing is not susceptible to the domino effect since the processes always restart from the most recent checkpoint. Also, recovery and garbage collection are both simplified and stable storage overhead is lower than that for independent checkpointing. The main disadvantage is the sacrifice of process autonomy in taking checkpoints. Independent (or uncoordinated) checkpointing allows each process to decide independently when to take checkpoints. The main advantage is the lower runtime overhead during normal execution. The main disadvantage is the possibility of the domino effect which may cause a large amount of useful work to be undone regardless of how many checkpoints have been taken. Communication-induced checkpointing is another way to avoid the domino effect in independent checkpointing protocols. A system-wide constrain on the checkpoint and communication pattern is specified to guarantee recovery line progression. Sufficient information is piggybacked on each message so that the receiver can examine the information and decide whether to take an adaptive checkpoint.

The MPI standard has been widely implemented, such as in MPICH [6] and LAM [7]. Among the many tools (or products) designed for MPI include p2d2 [13], ARCH [2], Para++ [30], and MPIgdb [8]. p2d2 is a debugger server that promotes portability of the user interface code by isolating the system dependent code. ARCH is an object-oriented tool for parallel programming on machines using the MPI communication library. Para++ provides a C++ interface to the MPI and PVM [11] message passing libraries. This approach attempts to overload input and output operators to perform communication. MPIgdb, an integrated debugging system for MPI on cluster of workstations, provides interactive and traced-based cyclical debugging functions.

Several checkpointing systems have been developed and implemented, including libckpt [23], ickp [24], CLIP [22], and CoCheck [26]. Libckpt is a portable checkpointing tool for Unix. Several optimizations have been implemented in libckpt to enhance the performance of checkpointing. Ickp is a library that enables users of the Intel iPSC/860 to preserve the execution state of their programs to disk. Ickp is an important piece of work as it is the first checkpointer ever written for a multicomputer. CLIP consists of two user-level libraries, libNXckpt and libMPIckpt, thereby allowing NX and MPI application programmers to write fault-tolerant code running on Intel Paragon multi-computers. CoCheck (Consistent Checkpoints) provides an effective means of creating consistent checkpoints of parallel applications which can be used to migrate processes of a parallel application to new hosts when a part of the machines becomes unavailable.

3. System Architecture

MPICkpt largely focuses on integrating fault tolerance to the MPI applications. MPICkpt supports two kinds of checkpointing techniques, coordinated and independent, and various optimized checkpointing techniques to reduce checkpointing overhead and ease the process of checkpointing. MPICkpt consists of four major components: a multi-functional checkpointing facility, a highly transparent checkpointing mechanism, a graphic user interface, and a checkpointing visualization facility.

Owing to that MPICkpt is developed for MPI, the MPICkpt communication world is based on the MPI communication world. Each node contains a monitor process used to set up checkpoints, perform the garbage collection, and ensure the consistency of checkpoints. A profiling technique is used to wrap the original MPI functions to pack some checkpointing information. The processes can communicate with each other or access the file system server by calling MPI functions. When constructing the MPI Communication World, a MPI-based application can also trigger the MPICkpt Communication World. In addition, MPICkpt provides a graphical user interface to facilitate the user in performing various checkpointing actions.

Checkpoints can be set up, either by the user to take checkpoints in some specified locations or by the checkpointing system to take checkpoints periodically after a fixed time and without changing the source code. According to Fig. 1, the MPICkpt system supports coordinated and independent checkpointing functions. For coordinated checkpointing, MPICkpt supports sync-and-stop technique to form a consistent global checkpoint. MPICkpt also provides bound elapsed progress when failures occur. For the independent checkpointing function, MPICkpt supports an adaptive independent checkpointing technique, as described in Section 4. Moreover, among the various checkpointing techniques that MPICkpt provides include sequential, forked, incremental, and compressed techniques to support different requirements. Sequential checkpointing is an approach for which checkpointing overhead is essentially identical to checkpoint latency. The process continues with execution only after the state is completely saved on stable storage. Incremental checkpointing can further reduce the overhead and avoid rewriting portions of the process states that do not change between consecutive checkpoints. Forked checkpointing is another approach to reduce checkpoint overhead. In this approach, a process wanting to take a checkpoint forks a child process to save its state on stable storage. These optimized techniques can be performed, independent or combined with each other, for coordinated or independent checkpointing.

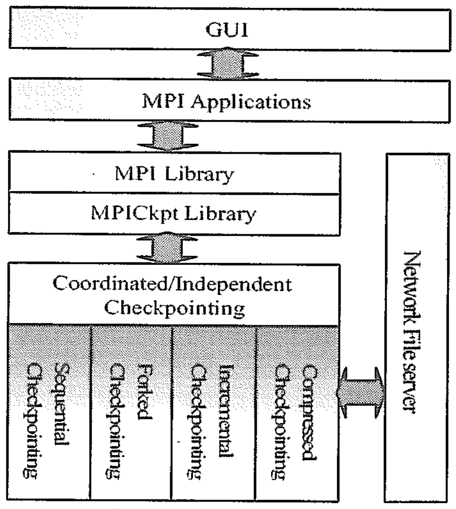


Figure 1: MPICkpt Architecture

4. Adaptive Independent Checkpointing

Conventional independent checkpointing techniques are susceptible to the domino effect and are difficult to perform garbage collection. In this section, we present an adaptive independent checkpointing algorithm [18] that has been implemented in MPICkpt. The algorithm can guarantee no useless checkpoints and a consistent global checkpoint always exists. Furthermore, the algorithm is effective and has low overhead in the length of piggybacked information. The algorithm uses a Boolean vector to determine whether or not to take an adaptive checkpoint for eliminating useless checkpoints. Every bit of this Boolean vector corresponds to each node in a distributed system. Process P_i, upon taking a new checkpoint and received at least one message in the previous checkpoint's interval, sets the values of all bits of this Boolean vector to *true* except bit *i*. This node then sends the computational message in a piggybacked form with this vector to other nodes. This control information is used to inform other nodes that a new adaptive checkpoint may be necessary to prevent useless checkpoints. Each node maintains the following data structure:

maybe_useless: an array of *n* Boolean values at each node. The vector $maybe\_useless_i$ records whether a useless checkpoint may occur in process P_i or not. Process P_i, upon sending a message to process P_j, piggybacks this information with the normal message to inform process P_j whether an adaptive checkpoint must be taken to prevent useless checkpoints in process P_i.

send_to: a Boolean flag at each node. *send_to$_i$* denotes that at least a message has been sent from process P_i after checkpointing.

has_delivery: a Boolean flag at each process, *has_delivery$_i$* denotes that at least a message has been delivered at process P_i..

```
var maybe_useless_i[n], send_to_i, has_delivery_i: boolean;
Actions for the initialization by P_i
    ∀k do maybe_useless_i[k] := false enddo;
    take_checkpoint;
Actions taken when P_i sends a message to P_j
    send_to_i := true;
    send( P_i, message, maybe_useless_i) to P_j;
    maybe_useless_i[j] := false;
Actions for node P_i when receiving a message from P_j
    receive(P_j, message, maybe_useless_j) from P_j;
    if send_to_i = true and maybe_useless_j[i] = true then take_checkpoint; endif;
    has_delivery_i := true;
    ∀k, k≠i do maybe_useless_i[k] := maybe_useless_i[k] OR maybe_useless_j[k]\
    enddo;
    deliver( message);
Action for P_i to take a checkpoint
    take_checkpoint;
procedure take_checkpoint is
    if has_delivery_i = true then ∀k ≠ i do maybe_useless_i[k] := true enddo; endif;
```

Figure. 2. Adaptive independent checkpointing algorithm

Figure 2 formally describes the algorithm. The algorithm contains four major phases: the initialization phase, message-sending phase, message-receiving phase, and checkpointing phase. Figure 3 illustrates an adaptive independent checkpointing example. Each process P_i is assumed to take a checkpoint $C_{i,1}$ initially and the Boolean vector *maybe_useless* is set to *false*. Process P_0 must set all bits of *maybe_useless$_0$* flag to *true* except itself after taking a periodic checkpoint $C_{0,2}$ because process P_0 takes this checkpoint after receiving message m_1 from process P_2. Checkpoint $C_{0,2}$ is useless because messages m_1, m_2 and m_3 construct a non-causal path. To prevent useless checkpoints, the rewinding path can be disrupted by taking an adaptive checkpoint $C_{1,2}$ in
process P_1. Before receiving message m_4 at process P_1, an adaptive checkpoint is unnecessary because *send_to$_1$* flag is *false*. Process P_0 does not need to take an adaptive checkpoint although *send_to$_0$* is *true* but *maybe_useless$_1$*[0] is *false*. Furthermore, Process P_1 must take an adaptive checkpoint before receiving message m_6 because *send_to$_1$* flag is *true* and *maybe_useless$_2$*[1] vector is *true*.

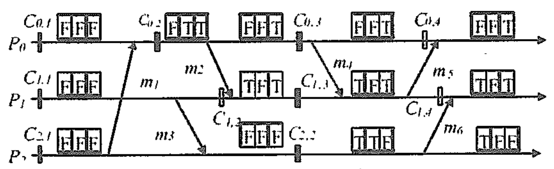


Figure 3 : An example

5. Implementation

This section describes how to implement MPICkpt. Implementation of the MPICkpt library is discussed first. The transparency facility is then implemented by applying the profiling technique to enhance MPI functions without changing the application programs. The implementation of the MPICkpt user interface and the checkpointing visualization is finally described.

5.1. MPICkpt Library

The MPICkpt library includes libckpt-based functions and MPI-based functions, among others. For libckpt-based functions, this work modifies and extends the libckpt source code to make it applicable not only for sequential machines but also for parallel/distributed environments. The incremental and forked checkpointing functions of MPICkpt are implemented mainly on the basis of libckpt. For MPI-based procedures, profiling techniques by MPICH are used to repack communication functions of the MPI library. In this implementation, checkpointing information deemed necessary to transit is packed into MPI-based communication functions. To accumulate information for visualization, some MPE [28] routines are also packed in MPI-based communication functions. Functions are implemented to provide a facility of sync-and-stop coordinated checkpointing. This work also implements the independent checkpointing algorithm described earlier and, then, extends the sequential, forked, and incremental checkpointing routines of the libckpt library for distributed/parallel programs.

Table 1 compares the MPICkpt library and the libckpt library. The source code of the MPICkpt library is 6608 lines, which is 1.6 times the size of the libckpt library. Five new files (2232 lines) are created and seven files are modified to wrap MPI functions, implement coordinated and independent checkpointing algorithms, extend the sequential, forked, and incremental checkpointing techniques, and accumulate data for checkpointing visualization.

	# files	Source code (lines)	Library size (bytes)
MPICkpt	23	6608	207K
Libckpt	18	4503	137K

Table 1 : Comparison of the MPICkpt library and the libckpt library

5.2. Transparency Facility

The wrappergen and profiling wrapper generator of the MPICH system are used to implement the transparency facility of MPICkpt. The user can write 'meta' wrappers for various MPI functions. The MPI profiling interface attempts to ensure that various applications can relatively easy interface their code to MPI implementations on different machines. Since MPI is a machine independent standard with many different implementations, expecting the user to modify the MPI program in order to use the MPI tools would be unrealistic. Therefore, the user must be provided with a tool without modifying the original source programs.

Figure 4 depicts the resolution of the MPICkpt calls, which contains four layers: application, MPICkpt, MPI&Libckpt, and physical layers. The application layer provides an environment for *normal* MPI applications. The MPICkpt layer provides the feature of fault tolerance for MPI applications. Additional information about checkpointing is packed (or unpacked) in this layer and some information for visualization is also accumulated here. The MPI&Libckpt layer includes MPI and libckpt libraries. In this layer, checkpointing mechanisms are achieved and wrapped function calls are performed. Notably, the physical layer provides the mechanisms of communication and storage.

To make the checkpointing transparent for distributed programs, the same source file can be complied to include the MPICkpt versions of the library, depending on the state of PROFLIB macro. Importantly, the standard MPI library must be built in such a manner that the MPI functions can be included one at a time. Consider a situation in which applications call an MPI function. If the function is wrapped then the call is linked to the profiling library, execute the extra operation, and call the mpi library to perform the original function. Otherwise, the call links to the mpi libray and execute the original mpi function. Figure 4 also depicts the flow of message passing during execution of MPI applications. When P_i sends a computational message Msg_i to P_j, the message of function MPI_Send() is packed, i.e. new message $Message_i$ consists of the computational message Msg_i and piggybacked information $Piggy_i$. Then, PMPI_Send() is called to transmit message $Message_i$. Upon message $Message_i$ arriving to P_j from P_i, P_j performs a function, MPI_Receive(). The profiling process of MPI_Receive() is described as follows. (1) Function PMPI_Receive() is called to receive message $Message_i$. (2) Message $Message_i$ is unpacked to computational message Msg_i and piggybacked information $Piggy_i$. (3) Function checkpoint_here() provided by libckpt is called to take an adaptive checkpoint if the condition of a useless checkpoint is identified from the piggybacked information $Piggy_i$. (4) Function MPI_Receive() is completed and, then, message Msg_i is delivered by P_j.

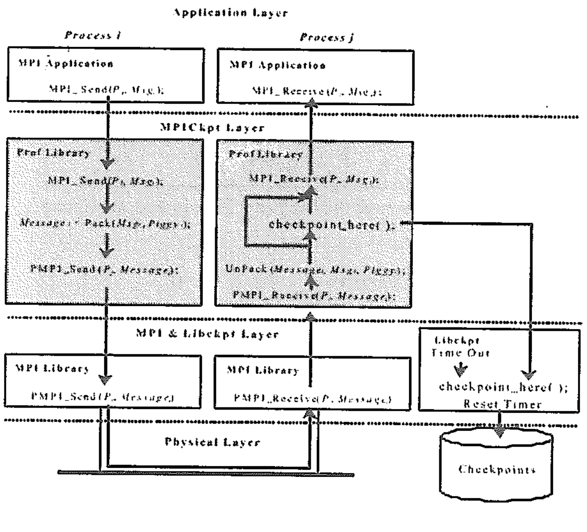


Figure 4: Resolution of MPICkpt calls

Notably, libckpt is not completely transparent because the initial procedure in C language of libckpt must be changed from main() to ckpt_target() to gain control of the program as it starts. However, MPICkpt applies profiling techniques to conceal all extra operations in the MPICkpt layer and achieve fully transparency. For MPI applications, MPI_Init() that is used to initialize the MPI execution environment must be included in the beginning of the application program and the ckpt_target() procedure is wrapped into MPI_Init() to achieve fully transparency.

Figure 5 presents an illustrative example of profiling actions in a MPICkpt function call. This function packs some information to the PMPI_Send() function, accumulates the cumulative amount of data spent, and collects information to upshot-style log file for visualization.

```
static int totalBytes;
static double totalTime;
char *Piggy;
int MPI_Send(void *Msg, const int count, MPI_Datatype datatype, int dest,
             int tag, MPI_comm, comm.)
{
  double tstart =MPI_Wtime; /* Pass on all arguments */
  int extent;
  buff= strcat (Msg, Piggy); /* Merge two strings */
  MPE_Log_event(1, 0, "start Send"); /* Start Logging */
  int result = PMPI_Send(buffer, count, datatype, dest, tag, comm);
  MPE_Log_evvnt(2, 0, "end Send"); /* Log information for visualization */
  MPI_Type_size(datatype, &extent); /* Compute Size */
  totalByte += count * extent;
}
```

Figure 5: A profiling example

5.3. User Interface and Checkpointing Visualization

The MPICkpt user interface is implemented on Sun 4, SPARC machines under the X Window environment. Tcl/Tk, a simple scripting language for controlling and extending applications, is used to construct the major part of the MPICkpt's user interface. Expectk [17], a toolkit for wrapping character-oriented interactive programs in GUIs, can also be used to combine multiple programs together for achieving synergism.

For checkpointing visualization, the program's run-time data are stored on the log file. This upshot-style log file can be used to monitor the program's performance. In this manner, the user can effectively observe the program's execution and checkpointing status through the visualization tool. upshot [12]. MPICH can create a customized logfile for upshot by calling various mpe logging routines. A profiling library automatically logs all calls to MPI functions. The profiling library generates logfiles which are files of timestamped events [16]. During program execution, calls to MPI_Log_event are made to store events of certain types in memory; these memory buffers are collected and merged in parallel during MPI_Finalize. MPI_Pcontrol is used to suspend and restart logging operations and analyze the logfile produced at the end of the program execution with a variety of tools.

6. MPICkpt Prototype

In this section, we present a novel prototype of MPICkpt. The MPICkpt prototype runs on a cluster of Sun-SPARC workstations. The user can control the system through the graphical user interface. In addition, the prototype allows the user to make various checkpointing options.

The checkpointing facility is provided by the MPICkpt library. The user can allow the system to either take a checkpoint automatically after a fixed time interval or insert the checkpoint_here() command into the program's source. The user does not need to modify his/her programs to use the MPICkpt checkpointing system. MPICkpt is a highly transparent checkpointing system. MPICkpt also provides a checkpointing visualization tool. Moreover, the user can use the visualization tool to observe the status of execution and checkpointing.

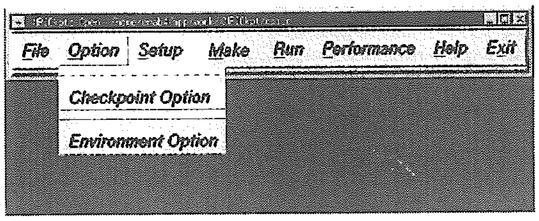
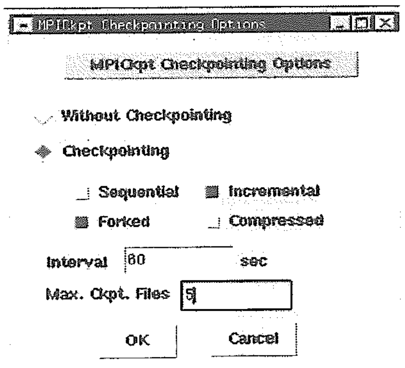


Figure 6: A snapshot of the MPICkpt prototype

MPICkpt is a fully transparent checkpointing system, implying that the user can easily take checkpoints without any additional programming effort. MPICkpt provides sequential style options, from file selection, checkpointing and environment selections, configuration setup, making an executable program, and running the program. Herein, MPICkpt is not treated merely as a checkpointing tool. MPICkpt also provides ' Performance' and ' Help' functions to help the programmer visualize and debug a program. According to Fig. 6, the main window of our MPICkpt checkpointing system has the following components which imply an order the user may normally apply.

- File: Operations to load and save files and to quit the system.
- Options: System options can be divided into checkpoint options and environment options. The former provides various checkpointing techniques and the latter is used to setup the execution environment.
- Setup: The user can setup the system configuration.
- Make: A terminal window is provided and displays the process of making a specified executable program.
- Run: MPICkpt provides an option to setup the number of nodes to be used for the parallel/distributed program.
- Performance: It provides visualization functions to let the user observe the checkpointing status and message passing.
- Help: A Help function and the MPIMan function are provided to the user.



MPICkpt Checkpointing Options

MPICkpt Checkpointing Options

Without Checkpointing

[✓ Checkpointing]

Sequential [✓ Incremental]

[✓ Forked] Compressed

Interval 60 sec

Max. Ckpt. Files 5

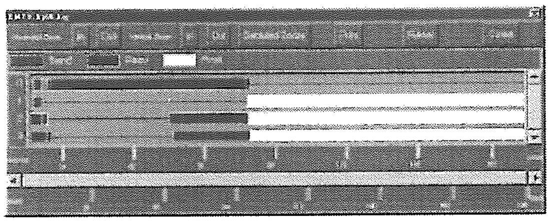
OK Cancel

Figure 7: Checkpoint Options

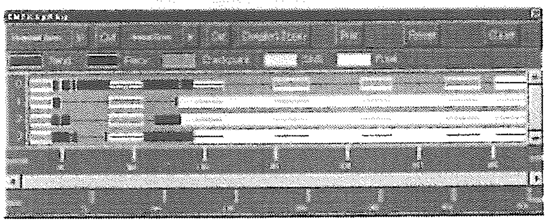
Figure 7 depicts the prototype for checkpointing options. The user selects independent checkpointing with 60 seconds per checkpoint interval, incremental and forked checkpointing facilities to take a checkpoint, and at most five checkpoints preserved for each process, i.e. garbage collection is performed every five checkpoints. Figure 8 presents illustrative examples of matrix multiplication for various checkpointing techniques using Upshot. Upshot uses parallel time lines for processes and displays various interactions between processes. Above figures contain the message-passing, checkpointing, and piggybacked information. Figure 8(a) depicts the visualization of matrix multiplication without checkpointing. This figure indicates that node 0 communicates frequently with other nodes. Figure 8(b) depicts the visualization of matrix multiplication with coordinated checkpointing. According to this figure, MPICkpt uses the sync-and-stop (SNS) technique for coordinated checkpointing. Figure 8(c) depicts the visualization of matrix multiplication with independent checkpointing. This figure reveals not only the checkpointing process, but also the actions of information piggybacking (Piggy).

7. Experimental Results

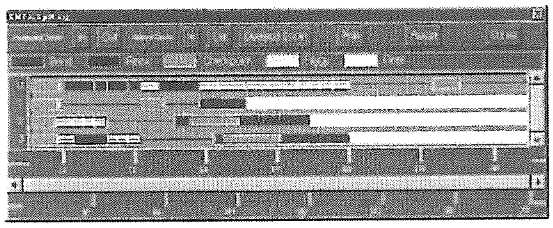
This section summarizes the experimental results of MPICkpt. To evaluate the effectiveness of MPICkpt, some MPI applications are tested on the MPICkpt system. Our experimental results include various applications, such as cpi, mm, pingpong and laplace.



(a) Matrix Multiplication without checkpointing



(b) Matrix Multiplication with coordinated checkpointing



(c) Matrix Multiplication with independent checkpointing

Figure 8 : Matrix m Multiplication with and without checkpointing

The application program ' cpi' used to compute the value of π is based on the ' mpi.c' of mpich's testing program. Herein, we modify the loop to increase the execution time.

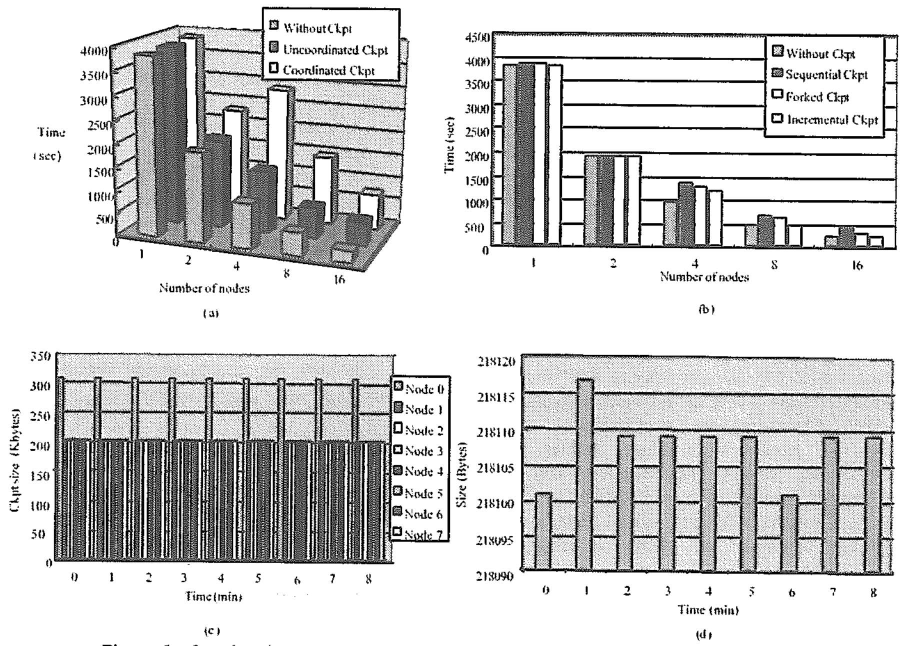


Figure 9 : Overhead comparisons of program cpi for various checkpointing techniques

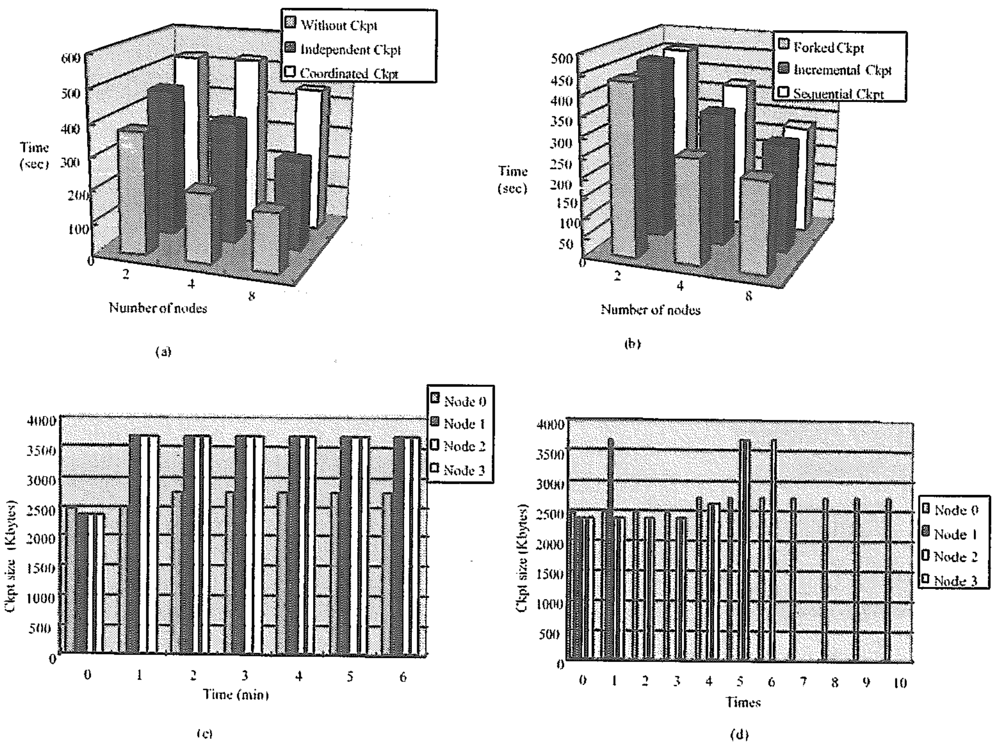
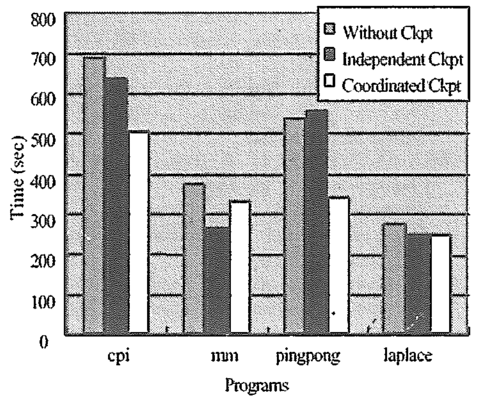
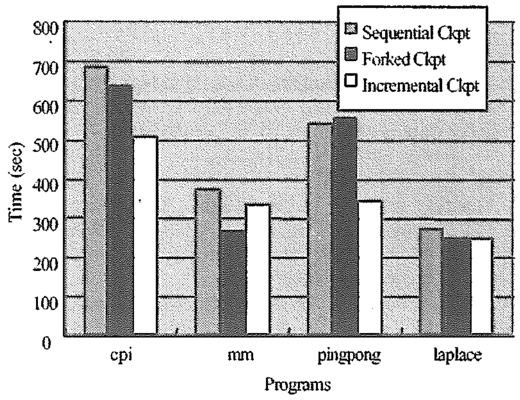


Figure 10: Overhead comparisons of program mm for various checkpointing techniques



(a)



(b)

Figure 11: Execution time for various checkpointing techniques

The 'mm' application program is used to compute matrix multiplication that assigns the task to nodes 1 to n-1 and returns the result to node 0 after each node completes its task. The application program 'pingpong' sends and receives message that is used to analyze the performance of system environments. The application 'laplace' is used to solve the Laplace equations that is designed by SPMD style to compute 48X48 matrix operation.

For the experimental platform, sixteen SUN SPARC classic workstations under the NIS network are used. The workstations can share the network file system between the network of workstations. In the experiments, the checkpoint interval is set to 1 minute and the maximum file number is 5, i.e. the garbage collection is performed every five checkpoints.

Figure 9 compares the various checkpointing techniques in terms of overhead of program cpi. In this experiment, checkpoints are taken by sequential checkpointing and the average execution time of coordinated/independent checkpointing is 1.53/1.12 times of the technique without checkpointing. This figure indicates that independent checkpointing has a lower overhead than coordinated checkpointing for program cpi. Figure 9(b) depicts the overhead of the independent algorithm with various optimized checkpointing techniques. According to this figure, incremental checkpointing has a lower overhead than forked and sequential checkpointing. Figure 9(c) depicts the checkpoint size with 8 nodes for coordinated checkpointing. Figure 9(d) depicts the average checkpoint size of Figure 9(c).

Figure 10 compares various checkpointing algorithms with respect to the overhead of program mm. In this experiment, checkpoints are taken by sequential checkpointing and the average execution time of coordinated/independent checkpointing is 1.95/1.45 times of the execution time of the technique without checkpointing. This figure indicates that independent checkpointing has a lower overhead than coordinated checkpointing for program mm. Figure 10(b) depicts the overhead of independent algorithm with various optimized checkpointing techniques. According to this figure, forked checkpointing has a lower overhead than incremental and sequential checkpointing. Figure 10(c) depicts the checkpoint size with four nodes for coordinated checkpointing. Figure 10(d) depicts the average checkpoint size with four nodes for independent checkpointing. According to Fig. 10(d), some columns have no data for checkpoint size because these nodes have already completed their tasks and are waiting for completion of the program.

Figure 11 reveals that the checkpointing performance depends on how the MPI applications run. Moreover, Fig. 11(a) indicates that the overhead of independent checkpointing is lower than coordinated checkpointing in application programs cpi and mm and the overhead of coordinated checkpointing is lower than independent checkpointing in application programs pingpong and laplace. Furthermore, Fig. 11(b) indicates not only that the overhead of incremental checkpointing is lower than forked checkpointing for programs cpi and pingpong, but also that the overhead of forked checkpointing is lower than incremental checkpointing for program mm. Therefore, more effective means can be adopted to take our checkpoint based on how the programs function. If the communication is infrequent, independent checkpointing algorithms can be chosen. In contrast, coordinated checkpointing algorithms should be selected.

8. Conclusions

This work presents MPICkpt, a fully transparent checkpointing tool for parallel/distributed programs. Some novel features of MPICkpt are as follows: (1) MPICkpt is designed for MPI; (2) MPICkpt provides consistent checkpointing algorithms based on coordinated checkpointing algorithms and independent checkpointing algorithms; (3) MPICkpt provides various checkpointing techniques, including sequential, forked, incremental, and compressed techniques to reduce checkpointing overhead; (4) MPICkpt is a highly transparent checkpointing system that can take checkpoints without changing the source code of application programs; and (5) MPICkpt also provides checkpointing information by a graphical visualization tool. The user can easily and effectively observe the checkpointing process of the parallel/distributed program. Currently, we are transplanting MPICkpt to Sun Enterprise 10000 and implementing other independent checkpointing techniques and automatic failure detection and recovery mechanisms.

Acknowledgment

The authors would like to thank the National Science Council of the Republic of China for financially supporting this research under Contract No. NSC 88-2213-E-035-039.

Reference

[1] A. Acharya and B. R. Badrinath, "Checkpointing Distributed Applications on Mobile Computers," Proc. Third Int'l Conf. Parallel and Distributed Information Systems (September 1994), 73-80.

[2] J. M. Adamo, "ARCH, An Object-Oriented Library for Asynchronous and Loosely Synchronous System Programming," Tech. Rep. ncstrl.cornell.tc/95-228, Institution Cornell University, Theory Center, 1995.

[3] S. Alagar and S. Venkatesan, "Causal Ordering in Distributed Mobile Systems," IEEE Transactions on Computers, Vol. 46, No. 3 (March 1997), 353-361,.

[4] R. Baldoni, J. M. Helary, A. Mostefaoui, M. Raynal, "On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems," Tech. Rep. RR-2564, IRISA, July 1995.

[5] A. Beguelin, E. Seligman, P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," Tech. Rep. CMU-CS-96-157, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, August 1996.

[6] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "Users' Guide to Mpich, a Portable Implementation of MPI," Argonne National Laboratory, October 1995.

[7] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," Ohio Supercomputer Center, 1994.

[8] C. R. Dow and Y. G. Gou, "A Parallel/Distributed Debugger for MPI," Proceedings of 1997 Workshop on Distributed System Techniques and Applications, Tainan, Taiwan, pp. 556-561, May 1997.

[9] E. N. Elnozahy, D. B. Johnson, and Y.M. Wang, "A Survey of Rollback Protocols in Message-Passing Systems" Tech. Rep. CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1997.

[10] J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints," Proc. IEEE Int'l Conf. Distributed Computing Systems, pp. 134-141, 1990.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, "PVM 3 User's Guide and Reference Manual," Oak Ridge National Laboratory, May 1993.

[12] V. Herrarte, E. Lusk, "Studying Parallel Program Behavior with Upshot," Tech. Rep. ANL – 91/15, Argonne National Laboratory, Argonne, IL 60439, 1991.

[13] R. T. Hood, "The p2d2 Project: Building a Portable Debugger," Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, May 1996.

[14] J. L. Kim and T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 8 (August 1993), 955-960.

[15] R. Koo and S. Toueg, "Checkpointing and Rollback Recovery for Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-13, No. 1 (January 1987), 23-31.

[16] L. Lamport, "Time, Clock and the Ordering of Events in Distributed Systems," Comm. ACM, Vol. 21, No. 7 (July 1978), 558-565.

[17] D. Libes, "X Wrapper for Non-Graphic Interactive Programs," Proceeding of X hibition 94, San Jose, California, June 1994.

[18] C. M. Lin and C. R. Dow, "Efficient Independent Checkpointing techniques for Message-Passing Programs," Tech. Rep., Department of Information Engineering and Computer Science, Feng-Chia University, January 1998.

[19] D. Manivannan and M. Singhal, "A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing," In Proc. IEEE, Int. Conf. Distributed Comput. Syst., pp. 100-107, 1996.

[20] D. Manivannan, R. Netzer, and M. Singal, "Finding Consistent Global Checkpoints in a Distributed Computation," IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 6 (June 1997), 623-627.

[21] R. H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 2 (February 1995), 165-169.

[22] J. S. Plank, Y. Chen, and K. Li, "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs," Tech. Rep., Department of Computer Science, University of Princeton, 1996.

[23] J. S. Plank, M. Beck, G. Kingsley, "Libckpt: Transparent Checkpointing Under Unix," USENIX Winter 1995 Technical Conference, New Orleans, Louisiana, January 16-20, 1995.

[24] J. S. Plank and K. Li, "Performance Results of Ickp -- A Consistent Checkpointer on the iPSC/860," Scalable High Performance Computing Conference, pp. 686-693, Knoxville, TN, May, 1994.

[25] R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 10 (October 1996), 1035-1048.

[26] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," 10th International Parallel Processing Symposium, April 1996.

[27] Y. M. Wang, "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," IEEE Transactions on Computers, Vol. 46, No. 4 (April 1997), 456-468.

[28] Message Passing Interface Forum, "MPI: a Message-Passing Interface Standard," Tech. Rep. CS-94-230, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994.

[29] Message Passing Interface Forum. "MPI2: Extension to the Message-Passing Interface," November 1995.

[30] Para++: C++ Bindings for Message Passing Libraries, INRIA RT-0174, June 1995.