Adaptive Checkpointing (Invited Paper)

Zizhong Chen
Department of Mathematical and Computer Sciences
Colorado School of Mines, Golden, USA
Email: zchen@mines.edu

Abstract— **Checkpointing is a typical approach to tolerate failures in today's supercomputing clusters and computational grids. Checkpoint data can be saved either in central stable storage, or in processor memory (as in diskless checkpointing), or local disk space (replacing memory with local disk in diskless checkpointing). But where to save the checkpoint data has a great impact on the performance of a checkpointing scheme. Fault tolerance schemes with higher efficiency usually choose to save the checkpoint data closer to the processor. However, when failures are handled from application level, the storage hierarch of a platform is often not available at the fault tolerance scheme design time. Therefore, it is often difficult to decide which checkpointing schemes to choose at the application design time. In this paper, we demonstrate that, a good fault tolerance efficiency can be achieved by adaptively choosing where to store the checkpoint data at run time according to the specific characteristics of the platform. We analyze the performance of different checkpointing schemes and propose an efficient adaptive checkpointing scheme to incorporate fault tolerance into high performance computing applications.**

Index Terms— **adaptivity, checkpointing, diskless checkpointing, fault tolerance, parallel and distributed computing, high performance computing**

I. Introduction

Checkpointing is a typical approach to tolerate failures in supercomputing clusters and computational grids [8]. Checkpoints can often be taken either from the system level or from the application level. However, when checkpoints are taken from application level, most fault tolerance schemes proposed in literature are *non-adaptive* in the sense that the fault tolerance schemes incorporated in applications are either designed without incorporating system environments (such as the amount of available memory and the local and network I/O bandwidth, etc) or designed only for a specific system environment. From the application point of view, fault tolerant high performance applications need to be able to achieve high performance under different system environments with as low performance overhead as possible. In order to achieve high reliability and survivability with low performance overhead, the fault tolerance schemes in such applications need to be adaptable to different (or dynamic) system environments.

In this paper, we demonstrate that, a good fault tolerance efficiency can be achieved by adaptively choosing where to store the checkpoint data at run time according to the specific characteristics of the platform. we analyze the performance of different checkpoint schemes and propose an efficient adaptive checkpointing scheme to incorporate fault tolerance into parallel applications. Applying this scheme to self-adaptive numerical software such as LAPACK for Clusters [2] will result in self-adaptive fault tolerant numerical libraries. Applications that call this type of self-adaptive fault tolerant numerical libraries will be able to survive certain processor failures transparently with very low performance overhead.

The rest of this paper is organized as follows. Section II defines the problem we are targeting. Section III analyzes the performance of several static checkpointing schemes. Section IV presents a self adapting application level checkpointing scheme for high performance computing. In Section V, experimental and simulation results are presented. Section VI concludes the paper and discusses future work.

II. Application Level Fault Tolerance

To define the problem we are targeting and clarify the differences with the system level fault tolerance approaches, in this section we first specify the type of failures we are focusing on and then briefly introduce FT-MPI, a fault tolerant version Message Passing Interface that supports application level fault tolerance.

Assume the target computing systems have many nodes which are connected by network connections. Each node has its own memory and local disk. There is at least one processor on each node and only one application process on each processor. Assume the target application is optimized to run on a fixed number of processes. Unlike in traditional algorithm-based fault tolerance which assumes a failed process continues to work but produce incorrect results, in this work we assume a *fail-stop* failure model. In a fail-stop failure model, the failed process is assumed to stop working and all data associated with the failed process are assumed to be lost. The surviving processes can neither send nor receive any message from the failed processes.

Current parallel programming paradigms for high-performance distributed computing systems are typically based on the Message-Passing Interface (MPI) specification [9]. However, the current MPI specification does

This paper is partly based on "Self Adaptive Application Level Fault Tolerance for Parallel and Distributed Computing", Z. Chen, et al., Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, Long Beach, CA, USA, March 26-29, 2007. IEEE Computer Society Press.

This work was supported in part by NSF grants OCI-0905019 and Microsoft.

Manuscript received March 15, 2009; revised August 10, 2009; accepted October 1, 2009.

doi:10.4304/jcm.5.1.81-87

not specify the behavior of an MPI implementation when one or more process failures occur during runtime. MPI gives the user the choice between two possibilities of how to handle failures. The first one, which is the default mode of MPI, is to immediately abort all the processes of the application. The second possibility is just slightly more flexible, handing control back to the user application without guaranteeing that any further communication can occur.

FT-MPI [6] is a fault tolerant version of MPI that is able to provide basic system services to support fault survivable applications. FT-MPI implements the complete MPI-1.2 specification, some parts of the MPI-2 document and extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI can survive the failure of n-1 processes in a n-process job, and, if required, can re-spawn the failed processes. However, the application is still responsible for recovering the data structures and the data of the failed processes.

III. Static Checkpointing

When building fault tolerant applications with FT-MPI, many fault tolerance schemes can be used. In this section, we analyze both the performance and the storage requirement of different checkpointing schemes that can be used with FT-MPI. To simplify the analysis, we only discuss the case to tolerate single processor failure.

Assume the checkpointing is performed in a parallel system with p processors and the size of checkpoint on each processor is m bytes. It takes $\alpha + \beta x$ to transfer a message of size x bytes between two processors regardless of which two processors are involved and. α is often called latency of the network. $\frac{1}{\beta}$ is called the bandwidth of the network. Assume the rate to calculate the XOR of two arrays is γ seconds per byte. We also assume that it takes $\alpha + \beta x$ to write x bytes of data into the stable storage. The I/O band width to local disk is assumed to be $\frac{1}{\zeta}$. Our default network model is the duplex model where a processor is able to concurrently send a message to one partner and receive a message from a possibly different partner. The more restrictive simplex model permits only one communication direction per processor. We also assume that disjoint pairs of processors can communicate each other without interference each other.

A. CSSC: Central Stable Storage Checkpoint

Today's long running scientific applications typically tolerate failures by checkpoint/restart approaches in which all process states of an application are periodically saved into stable storage. The advantage of this approach is that it is able to tolerate the failure of the whole system. However, in this approach, if one process fails, usually all surviving processes are aborted and the whole application is restarted from the last checkpoint. The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints to stable storage [12]. The checkpoint of an application on a, say, ten-thousand-processor computer implies that all critical data for the application on all ten thousand processors have to be written into stable storage periodically, which may introduce an unacceptable amount of overhead into the checkpointing system. As the number of processors in the system increases, the total number of process states that need to be written into the stable storage also increases linearly. Therefore, the fault tolerance overhead increases linearly. Figure 1. shows how a typical checkpoint/restart approach works.

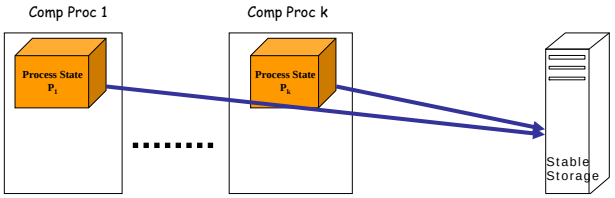

Figure 1. Central stable storage checkpoint scheme

When the size of checkpoint on each processor is m bytes, the total size of checkpoint for all processors is pm bytes. Therefore, the amount of stable storage needed for the central stable storage checkpoint scheme is pm bytes.

Without the support of a parallel file systems, to write all checkpoint data on all processors into the central stable storage, the time T_{cssc} it takes can be estimated by

$$\begin{aligned} T_{cssc} &= \alpha + p\beta m \\ &\approx p\beta m \end{aligned} \quad (1)$$

when m is relatively large.

When there is a parallel file system in the system, p in the above fomula represents the number of processors each file server serves.

B. MBPC: Memory-Based Parity Checkpoint

Diskless checkpointing [12] is a technique to save the state of a long running computation on a distributed system without relying on stable storage. Memory-based parity checkpoint is one form of the diskless checkpointing. With memory-based parity checkpoint, each processor involved in the computation stores a copy of its state locally, either in memory. Additionally, encodings of these checkpoints are stored in local memory of some processors. When a failure occurs, each live processor may roll its state back to its last local checkpoint, and the failed processor's state may be calculated from the local checkpoints of the surviving processors and the checkpoint encodings. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, memory-based parity checkpoint removes the main source of overhead in checkpointing on distributed systems [12]. Figure 2. demonstrates how memory-based parity checkpoint scheme works.

By breaking up a large message of size m into s smaller segments and sending these smaller messages through the

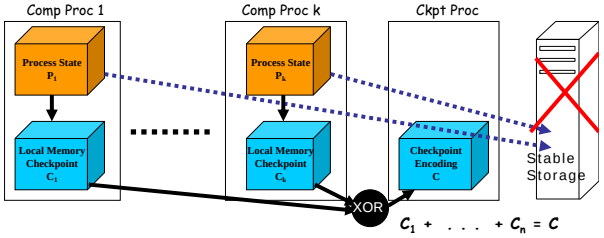

Figure 2. Memory-based parity checkpoint scheme

network by a pipelining style, the time to perform diskless checkpoint can be modeled by

$$\begin{aligned} T_{mbpc} &= (p-1+\frac{m}{s})(\alpha+\beta s+\gamma s) \\ &= (p-1)\alpha+(\beta+\gamma)m \\ &\quad +2\sqrt{(p-1)\alpha(\beta+\gamma)m} \\ &\geq (\beta+\gamma)m \qquad (2) \\ &\quad \cdot \left(1+2\sqrt{\frac{(p-1)\alpha}{(\beta+\gamma)m}}+\frac{(p-1)\alpha}{(\beta+\gamma)m}\right) \\ &\approx (\beta+\gamma)m \qquad (3) \end{aligned}$$

when m is large and p is relatively small.

Note that the equility in (2) can actually be achieved when

$$s=\sqrt{m\alpha/(p-1)/(\beta+\gamma)}.$$

If the size of checkpoint for each processor is m bytes, the memory overhead for the memory-based parity checkpoint scheme is m bytes.

C. DBPC: Disk-Based Parity Checkpoint

Many applications, such as HPL benchmark [4], achieve higher efficiency when most of the processor memory is used for the application. Reserving memory for checkpointing purpose often degrades the performance. However, if there is a local disk associated with the processor, free local disk storage can be used to store the checkpoint data. The checkpointing algorithm works the same way as the memory-based parity checkpoint except that the local disk are used to replace the memory. Figure 3. shows how disk-based parity checkpoint scheme works.

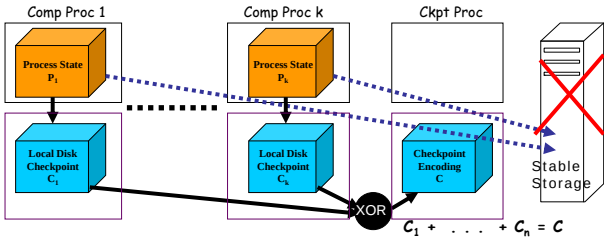

Figure 3. disk-based parity checkpoint scheme

By using the same pipelined XOR calculating algorithm as in memory-based parity checkpoint scheme, the time to perform checkpointing can be approximated by

$$\begin{aligned} T_{dbpc} &= \alpha+(\beta+\gamma+\zeta)m \\ &\approx (\beta+\gamma+\zeta)m \qquad (4) \end{aligned}$$

when m is relatively large and p is relatively small.

In the disk-based parity checkpoint scheme, if the size of checkpoint for each processor is m bytes, the local disk storage overhead for the disk-based parity checkpoint scheme is m bytes.

D. MBCM: Memory-Based CheckPoint Mirroring

When processor memory is used to store the checkpoint data, another possibility is to organize all computation processors as pairs (assume there are even number of computation processors). The two processors in a pair are neighbors of each other. Each processor first takes a local in-memory checkpoint and, at the same time, sends a copy of its local checkpoint to its neighbor processor. Figure 4. shows how memory-based checkpoint mirroring scheme works.

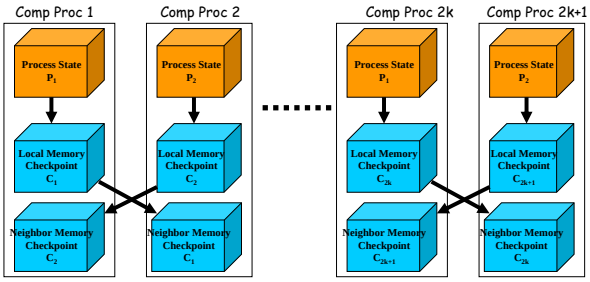

Figure 4. memory-based checkpoint mirroring scheme

Under the duplex network model where a processor is able to concurrently send a message to one partner and receive a message from a possibly different partner, the time to perform checkpoint mirroring can be approximated by

$$\begin{aligned} T_{mbcm} &= \alpha+\beta m \\ &\approx \beta m \qquad (5) \end{aligned}$$

when m is relatively large.

As shown in Figure 4., if the size of checkpoint for each processor is m bytes, the memory overhead for the memory-based checkpoint mirroring scheme is $2m$ bytes.

E. DBCM: Disk-Based CheckPoint Mirroring

When there is a local disk associated with each processor, the local disk can be used to replace the memory used in the memory-based checkPoint mirroring scheme. This scheme can be called as disk-based checkPoint mirroring scheme.

Under the duplex network model, the time to perform disk-based checkpoint mirroring can be approximated by

$$\begin{aligned} T_{dbcm} &= \alpha+\beta m+\zeta m \\ &\approx (\beta+\zeta)m \qquad (6) \end{aligned}$$

when m is relatively large.

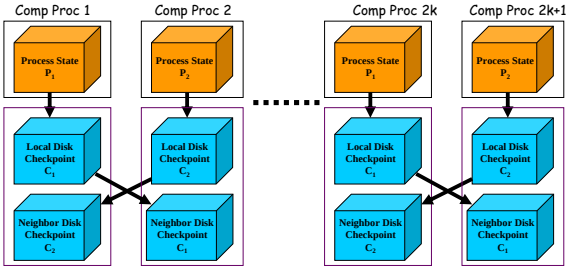

Figure 5. disk-based checkpoint mirroring scheme

In the disk-based checkPoint mirroring scheme, if the size of checkpoint for each processor is m bytes, the local disk storage overhead for the disk-based checkPoint mirroring scheme is $2m$ bytes.

IV. Adaptive Checkpointing

From Section 3, we have seen that Each fault tolerance scheme has its own advantages and disadvantages. However, different systems have different resource characteristics. What is the best way to incorporate different fault tolerance schemes into applications so that the reliability and survivability is as high as possible while the performance overhead is as low as possible?

From the application point of view, it is desirable that fault tolerant high performance applications is able to achieve both high performance and high reliability (survivability) with low fault tolerance overhead no mater under which kind of system environments it is running. To achieve this goal, the best strategy would be to adaptively choose the fault tolerance schemes in applications based on different (or dynamic) system environments that the applications are running.

The key idea of our recovery framework is the adaptivity of our checkpoint scheme to different system environments. Our adaptive scheme is similar to Vaidya's two-level recovery scheme in that both schemes combine the central stable storage checkpoint scheme with other higher efficiency checkpoint schemes such as diskless checkpointing. However Vaidya's recovery technique is static. He consider the availability of the memory and the local disk storage at the software design time, but after the design is finished, the software will never need to check the information of the hardware architecture (such as number of available processors, amount of memory and local disk storage) again. Thus we classify his scheme as static scheme. However, in our scheme, the software will have to check the information of the hardware architecture (such as number of available processors, amount of memory and local disk storage) to decide the optimal checkpoint scheme. Thus, we regard our scheme as adaptive rather than static.

The application of this self-adaptive fault tolerance framework to self-adaptive numerical software such as LFC will result in self-adaptive fault tolerant numerical libraries. Applications that use this kind of self-adaptive fault tolerant numerical libraries is able to survival certain processor failures transparently with very low performance overhead.

A. An Example Self Adaptive Recovery Scheme

What checkpoint scheme is the best for a specific system is often affected by many factor such as the amount of available storage of each type, the overhead of each checkpoint scheme, the failure rate and distribution of the system, the characteristics of the application, and the number of available processors for this application, etc. At the present stage, we only consider

- The size of checkpoint;
- The amount of available memory;
- The amount of available local disk storage;
- The amount of central stable storage;

If one type of storage is not available in the system, then we assume there are zero bytes of that type of storage in the system. We also assume that a node failure also means that both its memory and its local disk becomes unavailable.

The five candidate basic checkpoint schemes that we consider at the present time are

- **CSSC:** central stable storage checkpoint scheme
- **DBPC:** disk-based parity checkpoint scheme
- **DBCM:** disk-based checkpoint mirroring scheme
- **MBPC:** memory-based parity checkpoint scheme
- **MBCM:** memory-based checkpoint mirroring scheme

In the self-adaptive checkpointing scheme We first check the amount available free memory, free local diskless storage, and free central stable storage. We then use this information to choose the best basic recovery scheme. At the present time, we decide the checkpoint frequency manually. If we can somehow check the MTBF (or the failure rate) of the system in the future, we will use it to decide the checkpoint frequency.

It has been shown both from our experiments and in literature [3], [10]–[14] that the memory-based checkpoint mirroring scheme usually performs better than memory-based parity checkpoint scheme which usually performs better than disk-based checkpoint mirroring scheme. Disk-based checkpoint mirroring scheme usually performs better than disk-based parity checkpoint scheme which performs better than central stable storage checkpoint scheme. Based on this fact, we propose to use the algorithm in Figure 6. to decide which simple checkpoint scheme to choose. By making decisions at run time, we get the opportunity to know more information about the platform the application will execute than making decisions at the application design time. Therefore, we get the opportunity to make better decisions. This is why we can get better performance in a self adapting fault tolerance scheme.

B. Performance Analysis

In this section, we analyze the overhead of the proposed self adapting application level fault tolerance scheme.

```
if ( there is enough local and neighbor memory for checkpoint mirroring ) {
    use MBCM: memory-based checkpoint mirroring;
} else if ( there is enough local and neighbor memory for parity checkpoint ) {
    use MBPC: memory-based parity checkpoint;
} else if ( there is enough local and neighbor disk for checkpoint mirroring ) {
    use DBCM: disk-based checkpoint mirroring;
} else if ( there is enough local and neighbor disk for parity checkpoint ) {
    use DBPC: disk-based parity checkpoint;
} else if ( there is enough central stable storage to store checkpoint data ) {
    use CSSC: central stable storage checkpoint;
} else {
    there is no enough storage to store checkpoint data;
}
```

Figure 6. A simple self adaptive fault tolerance scheme

The fault tolerance overhead for the self adapting fault tolerance scheme includes two part: the overhead for gathering system information and making a decision on which simple scheme to use and the overhead for actually performing the checkpoint. Assume the time to gather system information and make a decision is $T_{decision}(p)$ and the time to perform the checkpoint is $T_{adaptive-checkpoint}(p, m)$.

The self-adapting checkpoint scheme make a decision on which simple checkpoint scheme to choose according the size of the checkpoint and the amount of each type of storage available. Consider a simplified case where each processor have the same amount freely available memory, and local disk storage. Let S_m denote the amount of the local free memory for a processor, S_d denote the amount of the local free disk storage of a processor, S_c denote the amount of central stable storage. Assume $S_m \leq S_d \leq \frac{1}{p}S_c$, then the time for adaptive checkpointing $T_{adaptive-checkpoint}(p, m)$ can be modeled by formula (7).

$T_{decision}(p)$ is the time for gathering system information and making a decision, which is often negligible compared with the overhead to perform the actually checkpoint. Compared to basic non-adaptive schemes such as the central stable storage checkpoint scheme in which the time for one checkpoint is $p\beta m$, the adaptive scheme usually has better performance unless $\frac{1}{p}S_c < c$. When $\frac{1}{p}S_c < c$, there is no enough storage to store any checkpoint.

Schemes with low fault tolerance overhead tend to use local (or neighbor) memory or local (or neighbor) disk instead of central stable storage to store checkpoint data. However, it is usually unclear what is the amount of local storage that can be used to store the checkpoint data until the program execution time. By postponing the time to make decisions to the program execution time, we get the opportunity to use as much local and neighbor storage as possible to store the checkpoint data. Therefore, we are able to get better performance by adapting the fault tolerance scheme to system environments at run time.

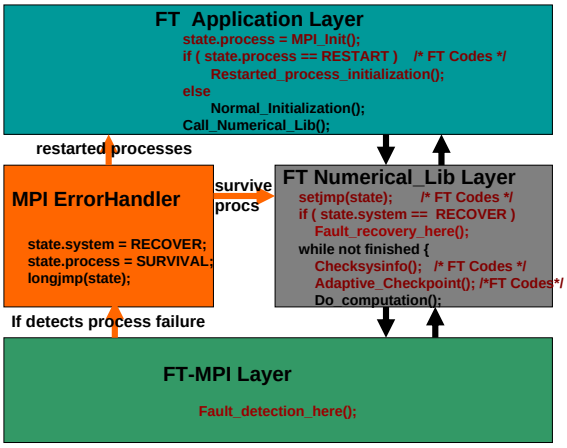

Figure 7. The control flow of a typical fault tolerant application

C. Incorporating fault tolerance into FT-MPI Applications

Handling fault-tolerance typically consists of three steps: 1) failure detection, 2) notification, and 3) recovery. The only assumption the FT-MPI specification makes about the first two points is that the run-time environment discovers failures and all remaining processes in the parallel job are notified about these events. The recovery procedure is considered to consist of two steps: recovering the MPI library and the run-time environment, and recovering the application. The latter one is considered to be the responsibility of the application. In the FT-MPI specification, the communicator-mode discovers the status of MPI objects after recovery, and the message-mode ascertains the status of ongoing messages during and after recovery. Figure 7. shows a typical fault tolerant application structure.

V. Performance Evaluation

In this section, we evaluate the performance of the proposed self adapting fault tolerance scheme.

A. Experimental Results

In this subsection, we compare the time for one checkpoint of the following two checkpoint schemes

$$T_{adaptive-checkpoint}(p,m) = \begin{cases} T_{decision}(p) + \beta m, & \text{if } c \le \frac{1}{2}S_m \\ T_{decision}(p) + (\beta+\gamma)m, & \text{if } \frac{1}{2}S_m < c \le S_m \\ T_{decision}(p) + (\beta+\zeta)m, & \text{if } S_m < c \le \frac{1}{2}S_d \\ T_{decision}(p) + (\beta+\gamma+\zeta)m, & \text{if } \frac{1}{2}S_d < c < S_d \\ T_{decision}(p) + p\beta m, & \text{if } S_d < c \le \frac{1}{p}S_c \end{cases} \quad (7)$$

- **MBPC:** The neighbor memory-based parity checkpoint scheme
- **SSAC:** The simple self adapting checkpointing scheme which choose only a single basic scheme (choose the best one) from the five basic schemes at run time according to the amount of different storage available.

The application we used to perform experiment is the PCG code described in [1]. The number of simultaneous processor failures we want to survive is one. The total number of processors we used in PCG is sixteen. The programming environment we used is FT-MPI [5]–[7]. All experiments were performed on a cluster of 32 Pentium IV Xeon 2.4 GHz dual-processor nodes. Each node of the cluster has 2 GB of memory and runs the Linux operating system. The nodes are connected with a Gigabit Ethernet. The timer we used in all measurements is MPI_Wtime.

TABLE I.
PERFORMANCE OF A SIMPLE SELF ADAPTING CHECKPOINTING SCHEME FOR PCG

Size of checkpoint (MBytes)	T_ SSAC (Seconds)	T_ MBPC (Seconds)
100	2.21	2.55
200	4.55	5.09
300	6.56	7.66
400	8.91	10.10
500	10.58	12.61
600	15.30	15.20
700	17.85	17.75
800	20.40	20.11

Table 1 reports the time for performing one checkpoint for both the SSAC and the MBPC schemes. By changing the input problem size in PCG, we varied the amount of data that need to be checkpointed from 100 MBytes to 800 MBytes. The results in Table 1 indicate that the SSAC scheme performs better than the MBPC scheme when the size of checkpoint is less than 500 MBytes. However, when the size of checkpoint is larger than 500 MBytes, the SSAC scheme performs approximately the same as the MBPC scheme. This is because, when the size of checkpoint is less than 500 MBytes, the SSAC scheme detects that a processor can store both a copy of its own checkpoint data and a copy of its neighbor processor's checkpoint data in its local memory. Therefore, the use the memory-based checkpoint mirroring scheme (which has lower performance overhead but high memory overhead than MBPC) is recommended. However, when the size of checkpoint is larger than 500 MBytes, the SSAC scheme detects that there is no enough local memory for a processor to store both its own checkpoint data and his neighbor processor's checkpoint data, therefore, choose to store only its own checkpoint data in his local memory and at the same time store the parity of all local checkpoint data into the memory of another dedicate processor, which is exactly what the MBPC scheme does.

B. Simulation Results

In this subsection, we simulate the performance of the self-adaptive checkpointing scheme by choosing appropriate parameters in formula (7) of Section IV.B.

Figure 8. shows a simulated result for the performance of the self-adapting fault tolerance scheme. In this simulation, $\beta = 20\times10^{-9}$, $\gamma = 5\times10^{-9}$, $\zeta = 13\times10^{-9}$, $p = 5$, $T_{decision}(p) = 0.1$ seconds, $S_m = 400$ MBytes, and S_d =MBytes. From Figure 8., we can see that the self-adapting fault tolerance scheme always choose the best available simple checkpoint schemes according to the size of the checkpoint and the amount of storage available. In particular, if the application designer is not sure whether there are enough memory or local disk storage to store the checkpoint at the application design time and conservately choose the central stable storage checkpoint approach as the fault tolerance scheme, Figure 8. demonstrates that the fault tolerance overhead can be several times higher than the self-adapting fault tolerance scheme.

VI. CONCLUSION AND FUTURE WORK

In this paper, we analyzed the performance of different checkpoint schemes and proposed an efficient adaptive checkpointing scheme to incorporate fault tolerance into MPI applications. Experimental results demonstrated that good fault tolerance efficiency can be achieved by adaptively choosing where to store the checkpoint data at run time according to the specific characteristics of the platform. In the future, we would like to apply the idea to heterogeneous environments.

ACKNOWLEDGMENT

We thank Ming Yang, Guillermo Francia, III, Monica Trifas, and Jack Dongarra for numerous discussions concerning this work.

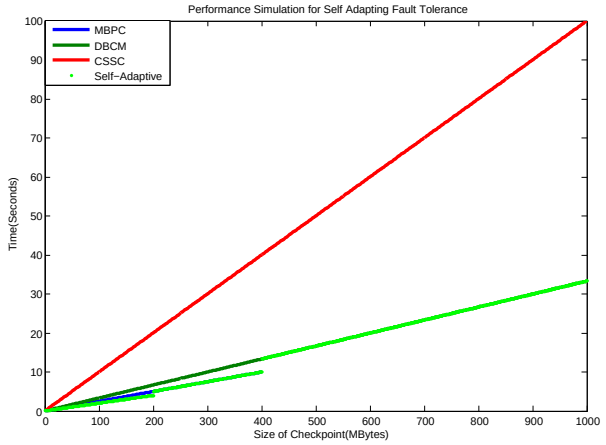

Figure 8. Performance simulation for the self-adaptive checkpointing scheme

REFERENCES

[1] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 14-17, 2005, Chicago, IL, USA*. ACM, 2005.

[2] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003.

[3] T. Chiueh and P. Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *FTCS*, pages 370–379, 1996.

[4] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/

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

[6] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference, Heidelberg, Germany*, 2004.

[7] G. E. Fagg, E. Gabriel, Z. Chen, , T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. *Submitted to International Journal of High Performance Computing Applications*, 2004.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, San Francisco, 1999.

[9] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical Report ut-cs-94-230, University of Tennessee, Knoxville, Tennessee, USA, 1994.

[10] J. S. Plank and K. Li. Faster checkpointing with n+1 parity. In *FTCS*, pages 288–297, 1994.

[11] J. S. Plank. Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques. In *15th Symposium on Reliable Distributed Systems*, pages 76–85, 1996.

[12] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[13] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In *EUROMICRO'98*, pages 395–402, 1998.

[14] N. H. Vaidya. A case for two-level recovery schemes. *IEEE Trans. Computers*, 47(6):656–666, 1998.

Zizhong Chen received a B.S. degree in mathematics from Beijing Normal University, P. R. China, in 1997, and M.S. and Ph.D. degrees in computer science from the University of Tennessee, Knoxville, in 2003 and 2006, respectively. He is currently an assistant professor of computer science at Colorado School of Mines. His research interests include high performance computing, parallel, distributed, and grid computing, fault tolerance and reliability, numerical linear algebra algorithms and software, and computational science and engineering. He is a member of the IEEE.