Self Adaptive Application Level Fault Tolerance for Parallel and Distributed Computing

Zizhong Chen[1], Ming Yang[1], Guillermo Francia, III[1], and Jack Dongarra[2]

[1]Jacksonville State University
MCIS Department
Jacksonville, AL 36265 USA
{zchen, myang, gfrancia}@jsu.edu

[2]University of Tennessee, Knoxville
Department of Computer Science
Knoxville, TN 37996
dongarra@cs.utk.edu

Abstract

Most application level fault tolerance schemes in literature are non-adaptive in the sense that the fault tolerance schemes incorporated in applications are usually designed without incorporating information from system environments such as the amount of available memory and the local or network I/O bandwidth. However, from an application point of view, it is often desirable for fault tolerant high performance applications to be able to achieve high performance under whatever system environment it executes with as low fault tolerance overhead as possibile.

In this paper, we demonstrate that, in order to achieve high reliability with as low performance penalty as possible, fault tolerant schemes in applications need to be able to adapt themselves to different system environments. We propose a framework under which different fault tolerant schemes can be incorporated in applications using an adaptive method. Under this framework, applications are able to choose near optimal fault tolerance schemes at run time according to the specific characteristics of the platform on which the application is executing.

1. Introduction

As the number of processors in modern high performance distributed computer systems continues to grow, the issue of fault tolerance is becoming more and more important. Even making generous assumptions on the reliability of a single processor or link, it is clear that as the processor count in high end clusters grows into the hundreds of thousands, the mean-time-to-failure of these clusters will drop from a few years to a few days, or less. The current DOE ASCI computer (IBM Blue Gene L) is designed with 131,000 processors [1]. The mean-time-to-failure of some nodes or links for this system is reported to be only six days on average [1]. In recent years, the trend of the high performance computing has been shifting from the expensive massively parallel computer systems to the clusters of commodity off-the-shelf systems [5]. While the commodity off-the-shelf cluster systems have excellent price-performance ratio, there is a growing concern with the fault tolerance issue in such system. The recently emerging computational grids [9] environments have further exacerbated the problem. However, many computational science programs are now designed to run for days or even months. Therefore the mean-time-between-failures (MTBF) of such kind of high performance computing systems are significantly shorter than the running time of many computational science programs. Modern computational science programs need to be able to tolerant failures.

Due to the large process state of such kind of applications, the relatively low I/O bandwidth between memory and the central network disk, and the high enough frequency of failures, for these systems, the classical system-level fault tolerance approaches is often either impractical (an application would spend most of its time taking checkpoints) or infeasible (there is no enough time for an application to save its core to disk before the next failure occurs). Therefore the cheaper application level fault tolerance schemes may be deployed as an alternative in such large computational science programs.

However, most application level fault tolerance schemes proposed in literature are *non-adaptive* in the sense that the fault tolerance schemes incorporated in applications are ei-

This research was supported in part by the Los Alamos National Laboratory under Contract No. 03891-001-99 49 and the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

ther 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 propose a framework under which different fault tolerance schemes can be incorporated in applications using an adaptive method. In our framework, applications will be able to choose the best (minimizing the mean execution time of the application) available fault tolerance schemes at runtime (or dynamically) according to different (or dynamic) system environments. Applying this frame work to self-adaptive numerical software such as LAPACK for Clusters [3] will result in self-adaptive fault tolerant numerical libraries. Applications that call this kind 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 2 reviews briefly the existing related literature in checkpointing and rollback recovery. Section 3 explains the motivations of this research. Section 4 presents a self adapting application level fault tolerance scheme for high performance grid computing. In Section 5, some initial experimental results are presented. Section 6 concludes the paper and discusses future work.

2 Fault Tolerance in Parallel and Distributed Systems

Fault tolerance techniques can be divided into two big branches and some hybrid techniques. The first branch is Messaging Logging. In this branch, there are three sub-branches: Pessimistic Messaging Logging, Optimistic Messaging Logging., and Casual Messaging Logging. The second branch is Checkpointing and Rollback recovery. There are also three sub-branches in this branch: Network disk based Checkpointing and rollback recovery, Diskless Checkpointing, and Local Disk based checkpointing.

Our research is mainly concentrated on incorporating fault tolerant techniques into tightly coupled large scale high performance computational intensive applications. These applications are often communication intensive, so checkpoint and rollback recovery approaches generally work better than message logging approaches. In the rest of this section, we confine our literature review to checkpointing and rollback recovery schemes instead of general fault tolerance schemes

Most traditional distributed multiprocessor recovery schemes are designed to tolerante arbitrary number of failures. So they store their checkpoint data in a central stable storage. The central stable storage usually has its own fault tolerance techniques to prevent it from failures. But the bandwidth between the processors and the central stable storage is usually very low. Several experimental studies presented in [13] have shown that the main performance overhead of checkpointing is the time spent on writing the checkpoint data to the central stable storage.

In [11] and [13], Plank proposed to use diskless checkpointing technique as an approach to tolerant single failures with low performance overhead when stable storage is not available. Diskless checkpointing is a technique where processor redundancy, memory redundancy and failure coverage are traded off so that a checkpointing system can operate in the absence of stable storage. Experimental studies presented in [13, 14] have shown that diskless checkpointing have a much better performance than traditional disk based checkpoint techniques.

There are also several papers which compare the performance of different diskless checkpointing schemes. In [4], Chiueh and Deng compare the performance of different diskless checkpointing schemes on a massively parallel SIMD machine. Their experiments were done on a DECmpp 12000 machine. The DECmpp 12000 machine has 8192 processors with each processor owning 64 Kbytes of RAM, but without any local disk for each processor. They implemented three chechpointing schemes (Checkpoint Mirroring, Parity Checkpointing and Partial Party Checkpointing) for a Matrix-matrix Multiplication Application. The XOR operation was done following an O(logN) binary tree fashion. The results of their experiment show that the Checkpoint Mirroring is an order of magnitude faster than the Parity Checkpointing, however introduced twice as much memory overhead as Parity Checkpointing. In [14], Silva also did some experimental studies about diskless checkpointing The experiments were done on an Xplorer Parsytec machine with 8 transputers (T805). Their experimental results show that Checkpoint Mirroring has a much better performance than the n+1 Parity schemes. The drawback is that Checkpoint Mirroring always presents more memory overhead than the n+1 Parity schemes. In [12], Plank also reported that Checkpoint Mirroring has lower performance overhead than Parity checkpointing if the checkpoint data is stored in local disk instead of the memory of the processor.

Local disk can also be used to store the checkpoint data. In [12], Plank applies RAID strategies to deal with local disk checkpoint data so that his checkpoint strategies can yield better performance for a smaller amount of fault coverage than traditional disk based checkpointing. In his paper, coordinated checkpoints are first taken to the local disk of each processor and then Checkpointing Mirroring, n+1 Parity, or Reed-Solomon Coding are used to encode the local checkpoint data to the local disk of other processors. This strategy uses the local disk to replace the memory to tolerate small process failures, which is important to achieve low checkpoint overhead when there is not enough memory to do diskless checkpoint.

To tolerate arbitrary number of failures with low performance overhead, Vaidya proposed a two-level distributed recovery approach in [15]. A two-level recovery scheme tolerates the more probable failures with low performance overhead, while less probable failures maybe tolerated with a higher performance overhead. In his example, the more probable single failures are tolerated with diskless checkpointing (checkpoint mirroring), while the less probable multiple failures are tolerated with traditional disk based checkpointing. In that example, he demonstrated that to minimize the average overhead, it is often necessary to take both diskless checkpoints and disk based checkpoints.

Checkpoint can be done either at the system-level or at the application level. In [14], Silva compared the performance overhead of the system-level checkpointing and the user defined checkpointing. Their experiments were done on an Xplorer Parsytec machine with 8 transputers (T805). Experiments show that the user defined checkpointing schemes have much lower performance overhead than the system-level checkpointing schemes. But the degree of the performance improvement is also dependent on specific applications.

In summary, a review of the existing fault tolerance research demonstrates that

- To tolerate arbitrary number of failures with low performance overhead, a two-level (or multi-level) recovery scheme should be used.
- If enough memory is available, Checkpoint Mirroring should be used rather than Parity Based Checkpointing.
- If there is no enough memory but there is enough local disk storage available, local disk storage can be used to reduce the checkpoint performance overhead.
- To achieve low performance overhead, user defined checkpointing schemes should be used instead of the system-level checkpointing schemes.

3 Motivations for Self Adapting Fault Tolerance

From Section 2, we have seen that the previous fault tolerant research works have produced some very precious result. However, there appears to be a significant gap between the fault tolerant research results and their optimal deployment into applications.

Each fault tolerance scheme has its own advantages and disadvantages. 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 Vaidyas two-level recovery scheme in that both schemes take multi-level checkpoint to tolerate arbitrary number of failures with low performance overhead. However Vaidyas 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 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.

4 A Self Adapting Application Level Fault Tolerance Scheme

In this section, we present a self adapting application level fault tolerance scheme for high performance grid computing.

4.1 Overview

Our goal is to establish a framework under which different fault tolerance schemes can be optimally incorporated in applications using an adaptive method. In our framework, applications will be able to adaptively choose the best (minimizing the mean execution time of the application) available fault tolerance schemes at runtime according to different system environments.

Different fault tolerant schemes require different resources. When designing the fault tolerant application, the application developer may not have an apriori knowledge of the system characteristics of the platform the application will be running on. Therefore, a self adapting application level fault tolerance scheme need to be able to detect system information at run time. The system characteristics that is necessary in determining checkpoint schemes may include

- The number of available processors
- The amount of available memory on each processor
- The amount of available local disk storage on each processor
- Whether there is a central fail free stable storage available
- The I/O bandwidth of the local disk storage and the central stable storage of each processor
- The network bandwidth between processors
- An estimate of the MTBF of the system environments

Different fault tolerant schemes have different degree of reliability. To tolerate the failure of all processors, a central stable storage is usually necessary. However, if we want to tolerate only a small number of processor failures, a central stable storage is usually not necessary. For example, schemes such as neighbour-based diskless checkpointing work fine to tolerate single processor failure. In order to maximize the degree of reliability while maintaining low performance overhead, a multi-level recovery scheme is often desirable in a self adapting application level fault tolerance scheme.

If the failure of all processors need to be tolerated in grid environments, a grid file system such as IBP can be used as a stable storage. In order to achieve low checkpoint overhead, algorithm-based checkpointing method can be used. In order to recover the system environments automatically, FTMPI/HARNESS [6, 7, 8] can be used as the communication library. If the main memory is not enough, consider using the local disk. In order to achieve low memory overhead, we also consider Kims checksum and reverse computation method [10]. In order to achieve transparency, consider and incorporating the fault tolerance in numerical libraries such as LFC. Because we are using an application-level approach, it is also possible to consider the characteristics of the application.

4.2 A Multi-level Self Adaptive Recovery Scheme

Assume a processor can access the following five types of storage in the computing system

- local memory of the processor
- local disk of the processor
- neighbor processors' memory
- neighbor processors' disk
- 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 the bandwidth of these five tpyes storages is strictly decreasing. Assume a node failure also means that both its memory and its local disk becomes unavailable.

Which kind of checkpoint schemes (or combination, or modification of schemes) is best for a specific system is affected by many factors. At the present time, we only consider the following factors:

- The amount of available storage of each kind
- The overhead of each checkpoint scheme (which is mainly dependent on the bandwidth of each storage and the characteristics of that checkpoint schemes)
- The failure distribution of the system.
- The characteristics of the application
- The number of available processors for this application.

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

- **CSSC:** Central Stable Storage Checkpoint scheme
- **NDPC:** Neighbor Disk-based Parity Checkpoint scheme
- **NDCM:** Neighbor Disk-based Checkpoint Mirroring scheme
- **NMPC:** Neighbor Memory-based Parity Checkpoint scheme
- **NMCM:** Neighbor Memory-based Checkpoint Mirroring scheme

```
if ( there is enough central stable storage ) {
    if ( there is enough neighbor disk to check mirroring ) {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes CSSC, NDMC and NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes CSSC, NDMC and NMPC;
        } else {
            use schemes CSSC, NDMC;
        }
    } else if ( there is enough neighbor disk to parity/reverse comp. ) {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes CSSC, NDPC and NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes CSSC, NDPC and NMPC;
        } else {
            use schemes CSSC, NDPC;
        }
    } else {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes CSSC and NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes CSSC and NMPC;
        } else {
            use schemes CSSC;
        }
    }
} else {
    if ( there is enough neighbor disk to check mirroring ) {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes NDMC and NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes NDMC and NMPC;
        } else {
            use schemes NDMC;
        }
    } else if ( there is enough neighbor disk to parity/reverse comp. ) {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes NDPC and NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes NDPC and NMPC;
        } else {
            use schemes NDPC;
        }
    } else {
        if ( there is enough neighbor memory to checkpoint mirroring ) {
            use schemes NMCM;
        } else if ( there are enough neighbor memory to parity/reverse comp. ) {
            use schemes NMPC;
        } else {
            there is no enough storage to do any checkpoint;
        }
    }
}
```

Figure 1. A multi-level self adapting fault tolerance scheme

The multi-level self adaptive recovery scheme is the combination of some of the above five basic schemes. Just as shown in existing research works, on most systems, the performance of these five basic recovery schemes is increasing (but it is also possible in the future to perform experiments to decide the performance of different schemes at run time). Since we also know the degree of fault tolerance of each basic scheme, so which combination to choose is mainly dependent on the availability and the amount of each storage. The checkpoint frequency of each basic scheme is mainly decided by the overhead of the scheme and the failure rate of the system.

Currently, we assume that we can check the availability and the amount of each storage as well as the number of available processors. We use this information to choose the combination of the basic recovery scheme. 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. Otherwise we decide the checkpoint frequency based on the assumption that the total performance overhead of the fault application does not exceed certain percentage (say 5%).

Based on our discussion, we propose to use the algorithm in Figure 1 to decide which combination of checkpoint schemes 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.

5 Performance Evaluation

In this section, we analyze the overhead of the proposed self adapting application level fault tolerance scheme and demonstrate some experimental results.

5.1 Performance Analysis

Let b_{cssc} denote the bandwidth in bytes/sec for a processor to access the central stable storage. Assume there are p processors in the system. Let c denote the size of checkpoint. Then, the time T_{cssc} to perform one checkpoint to the central stable storage can be approximated by

$$T_{cssc} = \frac{p * c}{b_{cssc}}$$

Let b_{ndpc} denote the checkpoint bandwidth for neighbour disk-based parity scheme and T_{ndpc} denote the time to perform one checkpoint in the neighbour disk-based parity scheme, then

$$T_{ndpc} = \frac{c}{b_{ndpc}}$$

Assume b_{ndcm} denote the checkpoint bandwidth for neighbour disk-based mirroring scheme and T_{ndcm} denote the time to perform one checkpoint in the neighbour disk-based mirroring scheme, then

$$T_{ndcm} = \frac{c}{b_{ndcm}}$$

Assume the checkpoint bandwidth for neighbour memory-based parity scheme is b_{nmpc} and T_{nmpc} denote the time to perform one checkpoint in the neighbour memory-based parity scheme, then

$$T_{nmpc} = \frac{c}{b_{nmpc}}$$

If we assume the checkpoint bandwidth for neighbour memory-based mirroring scheme is b_{nmcm} and T_{nmcm} denote the time to perform one checkpoint in the neighbour memory-based mirroring scheme, then

$$T_{nmcm} = \frac{c}{b_{nmcm}}$$

Without loss of generality, in this analysis, we assume $b_{nmcm} > b_{nmpc} > b_{ndcm} > b_{ndpc} > b_{cssc}$. This assumption is also consistent with the experimental results of different basic schemes in literature.

Assume there are five kinds of storage a processor can access in the computing system and

- 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
- the amount of neighbor processors' free memory is S_m
- the amount of neighbor processors' free disk storage is S_d
- S_c denote the amount of central stable storage

Consider a simple adaptive 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. Assume $S_m \leq S_d \leq \frac{1}{p}S_c$. Let $T_{adaptive}$ denote the time to perform one checkpoint in the simple self adapting application level fault tolerance scheme above, then

$$T_{adaptive} = \begin{cases} \frac{c}{b_{nmcm}}, & \text{if } c \le \frac{1}{2}S_m \\ \frac{c}{b_{nmpc}}, & \text{if } \frac{1}{2}S_m < c \le S_m \\ \frac{c}{b_{ndcm}}, & \text{if } S_m < c \le \frac{1}{2}S_d \\ \frac{c}{b_{ndpc}}, & \text{if } \frac{1}{2}S_d < c < S_d \\ \frac{pc}{b_{cssc}}, & \text{if } S_d < c \le \frac{1}{p}S_c \\ \infty, & \text{if } \frac{1}{p}S_c < c \end{cases} \quad (1)$$

Compared to basic non-adaptive schemes such as ”checkpoint to central stable storage” in which the time for one checkpoint is $\frac{pc}{b_{cssc}}$, the simple adaptive scheme always 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 neighbour) memory or local (or neighbour) 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 neighbour 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.

5.2 Experimental Results

In this section, we evaluate the performance of the proposed self adapting fault tolerance scheme experimentally. We compare the time for one checkpoint of the following two checkpoint schemes

- **NMPC:** a Neighbor Memory-based Parity Checkpoint scheme
- **SSAC:** a 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 [2]. 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 [6, 7, 8]. 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 1. Performance of a simple self adapting checkpointing scheme for PCG

Size of checkpoint (MBytes)	T_ SSAC (Seconds)	T_ NMPC (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
900	22.93	22.95
1000	25.50	25.48

Table 1 reports the time for performing one checkpoint for both the SSAC and the NMPCschemes. By changing the input problem size in PCG, we varied the amount of data that need to be checkpointed from 100 MBytes to 1,000 MBytes. The results in Table 1 indicate that the SSAC scheme performs better than the NMPC 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 NMPC 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 neighbour processor's checkpoint data in its local memory. Therefore, the use the Neighbor Memory-based Checkpoint Mirroring scheme (which has lower performance overhead but high memory overhead than NMPC) 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 neighbour 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 dadicate processor, which is exactly what the NMPC scheme does.

6 Conclusion and Future Work

In this paper, we presented a self adapting application level fault tolerance framework for high performance parallel and distributed computing. Within our framework, applications are able to choose near optimal (from the performance point of view) fault tolerance schemes at runtime (or dynamically) according to different (or dynamic) system environments. By making decisions at run time, we get the opportunity to know more information about the platform the application will execute. Thus, we get the opportunity to make better decision. This is why we can get better performance in a self adapting fault tolerance scheme.

Our future plan is to implement and incorporate this fault tolerance technique into the Self Adaptive Numerical Software Effort [3]. We would also like to evaluate this technique on systems with large number of processors.

References

[1] N. R. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Proceedings of the Supercomputing Conference (SC'2002), Baltimore MD, USA*, pages 1–22, 2002.

[2] 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.

[3] 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.

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

[5] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 28th edition. In *Proceedings of the Supercomputing Conference (SC'2006), Pittsburgh PA, USA*. ACM, 2006.

[6] 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.

[7] 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.

[8] 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.

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

[10] Y. Kim. *Fault Tolerant Matrix Operations for Parallel and Distributed Systems.* Ph.D. dissertation, University of Tennessee, Knoxville, June 1996.

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

[12] 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.

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

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

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