Fault Tolerant High Performance Computing by a Coding Approach *

Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou,
Thara Angskun, George Bosilca, and Jack Dongarra
Computer Science Department, University of Tennessee
1122 Volunteer Blvd., Suite 233, Knoxville, TN 37996-3450, USA
{zchen, fagg, egabriel, langou, angskun, bosilca, dongarra}@cs.utk.edu

ABSTRACT

As the number of processors in today's high performance computers continues to grow, the mean-time-to-failure of these computers are becoming significantly shorter than the execution time of many current high performance computing applications. Although today's architectures are usually robust enough to survive node failures without suffering complete system failure, most today's high performance computing applications can not survive node failures and, therefore, whenever a node fails, have to abort themselves and restart from the beginning or a stable-storage-based checkpoint.

This paper explores the use of the floating-point arithmetic coding approach to build fault survivable high performance computing applications so that they can adapt to node failures without aborting themselves. Despite the use of erasure codes over Galois field has been theoretically attempted before in diskless checkpointing, few actual implementations exist. This probably derives from concerns related to both the efficiency and the complexity of implementing such codes in high performance computing applications. In this paper, we introduce the simple but efficient floating-point arithmetic coding approach into diskless checkpointing and address the associated round-off error issue. We also implement a floating-point arithmetic version of the Reed-Solomon coding scheme into a conjugate gradient equation solver and evaluate both the performance and the numerical impact of this scheme. Experimental results demonstrate that the proposed floating-point arithmetic coding approach is able to survive a small number of simultaneous node failures with low performance overhead and little numerical impact.

*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.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming— *Parallel programming*

General Terms

Design, Reliability, Performance

Keywords

High Performance Computing, Message Passing Interface, Fault Tolerance, Floating-Point Arithmetic Coding

1. INTRODUCTION

As the unquenchable desire of today's scientists to run ever larger simulations and analyze ever larger data sets drives the size of high performance computers from hundreds, to thousands, and even tens of thousands of processors, the mean-time-to-failure (MTTF) of these computers are becoming significantly shorter than the execution time of many current high performance computing applications. 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 tens of thousands, the mean-time-to-failure of these clusters will drop from a few years to a few hours, or less. The next generation DOE ASCI computers (IBM Blue Gene L) are being designed with 131,000 processors [1]. The failure of some nodes or links for such a large system is likely to be just a few minutes away [12]. 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[7]. While the commodity off-the-shelf cluster systems have excellent price-performance ratio, due to the low reliability of the off-the-shelf components in these systems, there is a growing concern with the fault tolerance issues in such system. The recently emerging computational grid environments [15] with dynamic resources have further exacerbated the problem. However, driven by the desire of scientists for ever higher levels of detail and accuracy in their simulations, many computational science programs are now being designed to run for days or even months. Therefore, the next generation computational science programs need to be able to tolerate failures.

Today's long running scientific applications typically deal with faults by writing checkpoints into stable storage periodically. If a process failure occurs, then all surviving ap-

plication 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 [21]. 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. The restart of such an application implies that all processes have to be recreated and all data for each process have to be re-read from stable storage into memory or re-generated by computation, which often brings a large amount of overhead into restart. It may also be very expensive or unrealistic for many large systems such as grids to provide the large amount of stable storage necessary to hold all process state of an application of thousands of processes. Therefore, due to the high frequency of failures for the next generation computing systems, the classical checkpoint/restart fault tolerance approach may become a very inefficient way to deal with failures. Alternative fault tolerance approaches need to be investigated.

In this paper, we implement and evaluate an alternative approach to build fault tolerant high performance computing applications so that they can survive a small number of simultaneous processor failures without aborting the whole application. Based on diskless checkpointing [21] and FT-MPI, a fault tolerant version of MPI we developed [9, 10], our fault tolerance approach removes stable storage from fault tolerance and takes an application-level approach, which gives the application developer an opportunity to achieve as low fault tolerance overhead as possible according the specific characteristics of an application. Unlike in traditional checkpoint/restart fault tolerance paradigm, in our fault tolerance framework, if a small number of application processes failed, the survival application processes will not be aborted. Instead, the application will keep all survival processes, and adapt itself to failures.

Despite the use of erasure codes over Galois field has been theoretically attempted [18] before in diskless checkpointing, few actual implementations exist. This probably derives from concerns related to both the efficiency and the complexity of implementing such codes in high performance computing applications [16, 19]. In this paper, we introduce the simple but efficient floating-point arithmetic coding approach into diskless checkpointing and address the associated round-off error issue. We also implement a floating-point arithmetic version of the Reed-Solomon coding scheme into a conjugate gradient equation solver and evaluate both the performance and the numerical impact of this coding scheme. Experimental results demonstrate that the proposed floating-point arithmetic coding approach can survive a small number of simultaneous processor failures with low performance overhead and little numerical impact.

The rest of the paper is organized as follow. Section 2 gives a brief introduction to FT-MPI from the user point of view. Section 3 introduces the floating-point arithmetic coding approach into diskless checkpointing and address the associated round-off error issue. In Section 4, we give a detailed presentation on how to write a fault survivable application with FT-MPI by using a conjugate gradient equation solver as an example. In Section 5, we evaluate both the performance overhead of our fault tolerance approach and the numerical impact of our floating-point arithmetic encoding. Section 6 discusses the limitations of our approach and possible improvements. Section 7 concludes the paper and discusses future work.

2. FT-MPI: A FAULT TOLERANT MPI IMPLEMENTATION

Current parallel programming paradigms for high-performance computing systems are typically based on message passing, especially on the Message-Passing Interface (MPI) specification [17]. However, the current MPI specification does not deal with the case where 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 also 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, however, that any further communication can occur.

2.1 FT-MPI Overview

FT-MPI [10] 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.

Although FT-MPI provides basic system services to support fault survivable applications, prevailing benchmarks show that the performance of FT-MPI is comparable [11] to the current state-of-the-art MPI implementations.

2.2 FT-MPI Semantics

FT-MPI provides semantics that answer the following questions:

1. what is the status of an MPI object after recovery?
2. what is the status of ongoing communication and messages during and after recovery?

When running an FT-MPI application, there are two parameters used to specify which modes the application is running.

The first parameter, the 'communicator mode', indicates what is the status of an MPI object after recovery. FT-MPI provides four different communicator modes, which can be specified when starting the application:

- ABORT: like any other MPI implementation, FT-MPI can abort on an error.
- BLANK: failed processes are not replaced, all surviving processes have the same rank as before the crash and MPI_COMM_WORLD has the same size as before.
- SHRINK: failed processes are not replaced, however the new communicator after the crash has no 'holes' in its list of processes. Thus, processes might have a new rank after recovery and the size of MPI_COMM_WORLD will change.

- REBUILD: failed processes are re-spawned, surviving processes have the same rank as before. The REBUILD mode is the default, and the most used mode of FT-MPI.

The second parameter, the 'communication mode', indicates how messages, which are on the 'fly' while an error occurs, are treated. FT-MPI provides two different communication modes, which can be specified while starting the application:

- CONT/CONTINUE: all operations which returned the error code MPI_SUCCESS will finish properly, even if a process failure occurs during the operation (unless the communication partner has failed).
- NOOP/RESET: all ongoing messages are dropped. The assumption behind this mode is, that on error the application returns to its last consistent state, and all currently ongoing operations are not of any further interest.

2.3 FT-MPI Usage

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. FT-MPI offers for each of those modes several possibilities. This allows application developers to take the specific characteristics of their application into account and use the best-suited method to handle fault-tolerance.

3. THE FLOATING-POINT ARITHMETIC CODING APPROACH

To build fault survivable applications with FT-MPI, application developers have to design their own recovery schemes to recover their applications after failure. Checkpointing, message-logging, algorithm based checkpoint-free schemes such as lossy approach [3, 12] or combinations of these approaches may be used to reconstruct the required consistent state to continue the computation. However, due to its generality and performance, the diskless checkpointing technique [21] is a very promising approach to build fault survivable applications with FT-MPI.

To make diskless checkpointing as efficient as possible it can be implemented at the application level rather than at the system level [19]. There are several advantages to implement checkpointing at the application level. Firstly, the application level checkpointing can be placed at synchronization points in the program, which achieves checkpoint consistency automatically. Secondly, with the application level checkpointing, the size of the checkpoint can be minimized because the application developers can restrict the checkpoint to the required data. This is opposed to a transparent checkpointing system which has to save the whole process state. Thirdly, the transparent system level checkpointing typically write binary memory dumps, which rules out a heterogeneous recovery. On the other hand, application level checkpointing can be implemented such that the recovery operation can be performed in a heterogeneous environment as well.

In traditional diskless checkpointing, the checkpoint data are often treated as bit-streams. However, in typical long running scientific applications, when diskless checkpointing is taken from application level, what needs to be checkpointed is often some numerical data [16]. Although these numerical data can either be treated as bit-streams or as floating-point numbers, compared with treating checkpoint data as numerical numbers, treating them as bit-streams usually has the following disadvantages

1. To survive general multiple process failures, treating checkpoint data as bit-streams often involves the introduction of Galois field arithmetic in the calculation of checkpoint encoding and recovery decoding [18]. If the checkpoint data are treated as numerical numbers, then only floating-point arithmetic is needed to calculate the checkpoint encoding and recovery decoding. Floating-point arithmetic is usually simpler to implement and more efficient than Galois field arithmetic.

2. Treating checkpoint data as bit-streams rules out a heterogeneous recovery. The checkpoint data may have different bit-stream representation on different platforms and even have different bit-stream length on different architectures. The introduction of a unified representation of the checkpoint data on different platforms within an application for checkpoint purposes scarifies too much performance and is unrealistic in practice.

3. In some cases, treating checkpoint data as bit-streams does not work. For example, in [16], in order to reduce memory overhead in fault tolerant dense matrix computation, no local checkpoints are maintained on computation processors, only the checksum of the local checkpoints is maintained on the checkpoint processors. Whenever a failure occurs, the local checkpoints on surviving computation processors are re-constructed by reversing the computation. Lost data on failed processors are then re-constructed through the checksum and the local checkpoints obtained from the reverse computation. However, due to round-off errors, the local checkpoints obtained from reverse computation are not the same bit-streams as the original local checkpoints. Therefore, in order to be able to re-construct the lost data on failed processors, the checkpoint data has to be treated as numerical numbers and the floating point arithmetic has to be used to encode the checkpoint data.

The main disadvantage of treating the checkpoint data as floating-point numbers is the introduction of round-off errors into the checkpoint and recovery operations. Round-off errors is a limitation of any floating-point number calculations. Even without checkpoint and recovery, scientific computing applications are still affected by round-off errors. In practice, the increased possibility of overflows, underflows, and cancellations due to round-off errors in numerically stable checkpoint and recovery algorithms is often negligible.

In this section, we explore the possibility of treating the checkpoint data as floating-point numbers rather than bit-streams. In the following subsections, we discuss how the local checkpoint can be encoded so that applications can survive failures and address the associated round-off error issue.

3.1 Neighbor-Based Encoding

In neighbor-based encoding, a neighbor processor is first defined for each computation processor. Then, in addition to keep a local checkpoint in its memory, each computation processor stores a copy of its local checkpoint in the memory of its neighbor processor. Whenever a computation processor fails, the lost local checkpoint data can be recovered from its neighbor processor.

The performance overhead of the neighbor-based encoding is usually very low. The checkpoint are localized to only two processors: a computation processor and its neighbor. The recovery only involves the failed processors and its neighbors. There is no global communications or encoding/decoding calculations needed in the checkpoint and recovery.

Because no floating point operations are involved in the checkpoint and recovery, no round-off errors are introduced in the neighbor-based encoding.

Depending on how we define the neighbor processor of a computation processor, there are three neighbor-based encoding schemes

3.1.1 Mirroring

The mirroring scheme of neighbor-based encoding is originally proposed in [21]. In this scheme, if there are n computation processors, another n checkpoint processors are dedicated as neighbors of the computation processors. The i-th computation processor simply stores a copy of its local checkpoint data in the i-th checkpoint processor (see Figure 1 (a)).

Up to n processor failures may be tolerated, although the failure of both a computation processor and its neighbor processor can not be tolerated. If we assume that the failure of each processor are independent and identically distributed, then the probability that the mirroring scheme survives k processor failures is

$$\frac{C_n^k 2^k}{C_{2n}^k}.$$

When k is much smaller than n, the probability to survive k failures can be very close to 1.

The disadvantage of the mirroring scheme is that n additional processors are dedicated as checkpoint processors, therefore, can not be used to do computation.

3.1.2 Ring Neighbor

In [23], a ring neighbor scheme was discussed by Silva et al. In this scheme, there are no additional processors used. All computation processors are organized in a virtual ring. Each processor sends a copy of its local checkpoint to the neighbor processor that follows on the virtual ring. Therefore, each processor has two checkpoints maintained in memory: one is the local checkpoint of itself, another is the local checkpoint of its neighbor (see Figure 1 (b)).

The ring neighbor scheme is able to tolerate at least one and up to $\lfloor \frac{n}{2} \rfloor$ processor failures in a n processor job depending on the distribution of the failed processors.

Compared with mirroring scheme, the advantage of the ring neighbor scheme is that there is no processor redundancy in the scheme. However, two copies of checkpoints have to be maintained in the memory of each computation processor. The degree of fault tolerance of the ring neighbor scheme is also lower than the mirroring scheme.

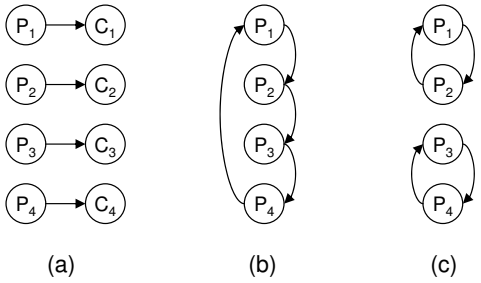

Figure 1: Neighbor-Based Schemes

3.1.3 Pair Neighbor

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 sends a copy of its local checkpoint to its neighbor processor (see Figure 1 (c)).

Like the ring neighbor scheme, there is no processor redundancy used in the paired neighbor scheme and two copies of checkpoints have to be maintained in the memory of each computation processor.

However, compared with the ring neighbor scheme, the degree of fault tolerance for the pair neighbor scheme are improved. Like the mirroring scheme, if we assume that the failure of each processes are independent and identically distributed, then the probability that the pair neighbor scheme survives k failures in a n processor job is

$$\frac{C_{n/2}^k 2^k}{C_n^k}.$$

3.2 Checksum-Based Encoding

The checksum-based encoding is a modified version of the parity-based encoding proposed in [20]. In the checksum-based encoding, instead of using parity, the floating-point number addition is used to encode the local checkpoint data. By encoding the local checkpoint data of the computation processors and sending the encoding to some dedicated checkpoint processors, the checksum- based encoding introduces a much lower memory overhead into the checkpoint system than neighbor-based encoding. However, due to the calculating and sending of the encoding, the performance overhead of the checksum-based encoding is usually higher than neighbor-based encoding schemes. There are two versions of the checksum-based encoding schemes.

3.2.1 Basic Checksum Scheme

The basic checksum scheme works as follow. If the program is executing on N processors, then there is a $N+1$-st processor called the checksum processor. At all points in time a consistent checkpoint is held in the N processors in memory. Moreover a checksum of the N local checkpoints is held in the checksum processor (see Figure 2 (a)). As-

sume P_i is the local checkpoint data in the memory of the i-th computation processor. C is the checksum of the local checkpoint in the checkpoint processor. If we look at the checkpoint data as an array of real numbers, then the checkpoint encoding actually establishes an identity

$$P_1 + \ldots + P_n = C \tag{1}$$

between the checkpoint data P_i on computation processors and the checksum data C on the checksum processor. If any processor fails then the identity (1) becomes an equation with one unknown. Therefore, the data in the failed processor can be reconstructed through solving this equation.

Due to the floating-point arithmetic used in the checkpoint and recovery, there will be round-off errors in the checkpoint and recovery. However, the checkpoint involves only additions and the recovery involves additions and only one subtraction. In practice, the increased possibility of overflows, underflows, and cancellations due to round-off errors in the checkpoint and recovery algorithm is negligible.

The basic checksum scheme can survive only one failure. However, it can be used to construct one dimensional checksum scheme to survive certain multiple failures.

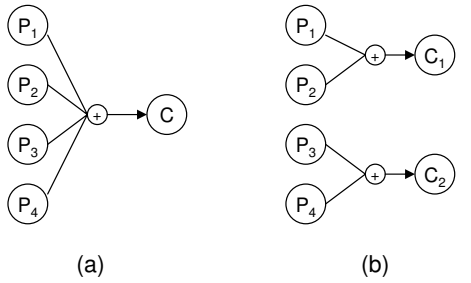

Figure 2: Checksum Based Schemes

3.2.2 One Dimensional Checksum Scheme

The one dimensional checksum scheme works as follow. Assume the program is running on mn processors. Partition the mn processors into m groups with n processors in each group. Dedicate one checksum processor for each group. At each group, the checkpoint are done using the basic checksum scheme (see Figure 2 (b)).

The advantage of this scheme is that the checkpoint are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done parallelly. Therefore, compared with the basic checksum scheme, the performance of the one dimensional checksum scheme is usually better. If we assume that the failure of each processes are independent and identically distributed, then the probability that the one dimensional checksum scheme survives k ($k < m$) failures is

$$\frac{C_m^k (n+1)^k}{C_{m(n+1)}^k}.$$

3.3 Weighted-Checksum-Based Encoding

The weighted checksum scheme is a natural extension to the checksum scheme to survive multiple failures of arbitrary patterns with minimum processor redundancy. It can also be viewed as a version of the Reed-Solomon erasure coding scheme [18] in the real number field. The basic idea of this scheme works as follow: Each processor takes a local in memory checkpoint, m equalities are established by saving weighted checksums of the local checkpoint into m checksum processors. When there are f failures happen, where $f \leq m$, the m equalities becomes m equations with f unknowns. By appropriately choosing the weights of the weighted checksums, the lost data on the f failed processors can be recovered by solving these m equations.

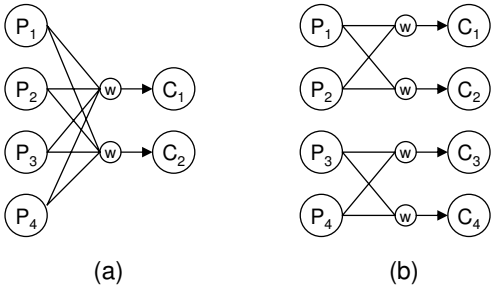

Figure 3: Weighted Checksum Schemes

3.3.1 The Basic Weighted Checksum Scheme

Suppose there are n processors used for computation. Assume the checkpoint data on the i-th computation processor is P_i. In order to be able to reconstruct the lost data on failed processors, another m processors are dedicated to hold m encodings (weighted checksums) of the checkpoint data (see Figure 3 (a)). The weighted checksum C_j on the jth checksum processor can be calculated from

$$\begin{cases} a_{11}P_1 + \ldots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{m1}P_1 + \ldots + a_{mn}P_n & = C_m, \end{cases} \tag{2}$$

where a_{ij}, $i = 1, 2, ..., m$, $j = 1, 2, ..., n$, is the weight we need to choose. Let $A = (a_{ij})_{mn}$. We call A the checkpoint matrix for the weighted checksum scheme.

Suppose that k computation processors and $m - h$ checkpoint processors have failed, then there are $n - k$ computation processors and h checkpoint processors survive. If we look at the data on failed processors as unknowns, then (2) becomes m equations with $m - (h - k)$ unknowns.

If $k > h$, then there are less equations than unknowns. There is no unique solution for (2). The lost data on the failed processors can not be recovered.

However, if $k < h$, then there are more equations than unknowns. By appropriately choosing A, a unique solution for (2) can be guaranteed. Therefore, the lost data on the failed processors can be recovered by solving (2).

Without loss of generality, we assume: (1) the computational processor j_1, $j_2, ..., j_k$ failed and the computational processor j_{k+1}, $j_{k+2}, ..., j_n$ survived; (2) the checkpoint processor i_1, $i_2, ..., i_h$ survived and the checkpoint processor i_{h+1}, $i_{h+2}, ..., i_m$ failed. Then, in equation (2), $P_{j_1}, ..., P_{j_k}$ and $C_{i_{h+1}}, ..., C_{i_m}$ become unknowns after the failure occurs. If we re-structure (2), we can get

$$\begin{cases} a_{i_1 j_1}P_{j_1} + ... + a_{i_1 j_k}P_{j_k} & = C_{i_1} - \sum_{t=k+1}^{n} a_{i_1 j_t}P_{j_t} \\ & \vdots \\ a_{i_h j_1}P_{j_1} + ... + a_{i_h j_k}P_{j_k} & = C_{i_h} - \sum_{t=k+1}^{n} a_{i_h j_t}P_{j_t} \end{cases} \tag{3}$$

and

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1}P_1 + \ldots + a_{i_{h+1}n}P_n \\ & \vdots \\ C_{i_m} & = a_{i_m1}P_1 + \ldots + a_{i_mn}P_n. \end{cases} \quad (4)$$

Let A_r denote the coefficient matrix of the linear system (3). If A_r has full column rank, then $P_{j_1}, ..., P_{j_k}$ can be recovered by solving (3), and $C_{i_{h+1}}, ..., C_{i_m}$ can be recovered by substituting $P_{j_1}, ..., P_{j_k}$ into (4).

Whether we can recover the lost data on the failed processes or not directly depends on whether A_r has full column rank or not. However, A_r in (3) can be any sub-matrix (including minor) of A depending on the distribution of the failed processors. If any square sub-matrix (including minor) of A is non-singular and there are no more than m process failed, then A_r can be guaranteed to have full column rank. Therefore, to be able to recover from any no more than m failures, the checkpoint matrix A has to satisfy *any square sub-matrix (including minor) of A is non-singular.*

How can we find such kind of matrices? It is well known that some structured matrices such as Vandermonde matrix and Cauchy matrix satisfy any square sub-matrix (including minor) of the matrix is non-singular.

However, in computer floating point arithmetic where no computation is exact due to round-off errors, it is well known that, in solving a linear system of equations, a condition number of 10^k for the coefficient matrix leads to a loss of accuracy of about k decimal digits in the solution. Therefore, in order to get a reasonably accurate recovery, the checkpoint matrix A actually has to satisfy *any square sub-matrix (including minor) of A is well-conditioned.*

It is well-known [8] that Gaussian random matrices are well-conditioned. To estimate how well conditioned Gaussian random matrices are, we have proved the following Theorem:

THEOREM 1. *Let $G_{m\times n}$ be an $m \times n$ real random matrix whose elements are independent and identically distributed standard normal random variables, and let $\kappa_2(G_{m\times n})$ be the 2-norm condition number of $G_{m\times n}$. Then, for any $m \geq 2$, $n \geq 2$ and $x \geq |n-m|+1$, $\kappa_2(G_{m\times n})$ satisfies*

$$P\left(\frac{\kappa_2(G_{m\times n})}{n/(|n-m|+1)} > x\right) < \frac{1}{\sqrt{2\pi}}\left(\frac{C}{x}\right)^{|n-m|+1},$$

and

$$E(\ln \kappa_2(G_{m\times n})) < \ln \frac{n}{|n-m|+1} + 2.258,$$

where $5.013 \leq C \leq 6.414$ is a universal positive constant independent of m, n and x.

Due to the length of the proof for Theorem 1, we omit the proof here and refer interested readers to [4] for complete proof.

Note that any sub-matrix of a Gaussian random matrix is still a Gaussian random matrix. Therefore, a Gaussian random matrix would satisfy any sub-matrix of the matrix is well-conditioned with high probability.

Theorem 1 can be used to estimate the accuracy of recovery in the weighted checksum scheme. For example, if an application uses 100,000 processors to perform computation and 20 processors to perform checkpointing, then the checkpoint matrix is a 20 by 100,000 Gaussian random matrix. If 10 processors fail concurrently, then the coefficient matrix A_r in the recovery algorithm is a 20 by 10 Gaussian random matrix. From Theorem 1, we can get

$$E(\log_{10} \kappa_2(A_r)) < 1.25$$

and

$$P(\kappa_2(A_r) > 100) < 3.1 \times 10^{-11}.$$

Therefore, on average, we will loss about one decimal digit in the recovered data and the probability to loss 2 digits is less than 3.1×10^{-11}.

3.3.2 One Dimensional Weighted Checksum Scheme

The one dimensional weighted checksum scheme works as follows. Assume the program is running on mn processors. Partition the mn processors into m groups with n processors in each group. Dedicate another k checksum processors for each group. At each group, the checkpoint are done using the basic weighted checksum scheme (see Figure 3 (b)). This scheme can survive k processor failures at each group. The advantage of this scheme is that the checkpoint are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done parallelly. Therefore, compared with the basic weighted checksum scheme, the performance of the one dimensional weighted checksum scheme is usually better.

4. A FAULT SURVIVABLE ITERATIVE EQUATION SOLVER

In this section, we give a detailed presentation on how to incorporate fault tolerance into applications by using a preconditioned conjugate gradient equation solver as an example.

4.1 Preconditioned Conjugate Gradient Algorithm

The Preconditioned Conjugate Gradient (PCG) method is the most commonly used algorithm to solve the linear system $Ax = b$ when the coefficient matrix A is sparse and symmetric positive definite. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors needs to be kept in memory. In every iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. The pseudo-code for the PCG is given in Figure 4. For more details of the algorithm, we refer the reader to [2].

4.2 Incorporating Fault Tolerance into PCG

We first implemented the parallel non-fault tolerant PCG. The preconditioner M we use is the diagonal part of the coefficient matrix A. The matrix A is stored as sparse row compressed format in memory. The PCG code is implemented such that any symmetric, positive definite matrix using the Harwell Boeing format or the Matrix Market format can be used as a test problem. One can also choose to generate the test matrices in memory according to testing requirements.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

Figure 4: Preconditioned Conjugate Gradient Algorithm

We then incorporate the basic weighted checksum scheme into the PCG code. Assume the PCG code uses n MPI processes to do computation. We dedicate another m MPI processes to hold the weighted checksums of the local checkpoint of the n computation processes. The checkpoint matrix we use is a pseudo random matrix. Note that the sparse matrix does not change during computation, therefore, we only need to checkpoint three vectors (i.e. the iterate, the residual and the search direction) and two scalars (i.e. the iteration index and $\rho^{(i-1)}$ in Figure 4).

The communicator mode we use is the REBUILD mode. The communication mode we use is the NOOP/RESET mode. Therefore, when processes failed, FT-MPI will drop all ongoing messages and re-spawn all failed processes without changing the rank of the surviving processes.

An FT-MPI application can detect and handle failure events using two different methods: either the return code of every MPI function is checked, or the application makes use of MPI error handlers. The second mode gives users the possibility to incorporate fault tolerance into applications that call existing parallel numerical libraries which do not check the return code of their MPI calls. In PCG code, we detect and handle failure events by checking the return code of every MPI function.

The recovery algorithm in PCG makes use of the *longjmp* function of the C-standard. In case the return code of an MPI function indicates that an error has occurred, all surviving processes set their state variable to RECOVER and *jump* to the recovery section in the code. The recovery algorithm consists of the following steps:

1. Re-spawn the failed processes and recover the FT-MPI runtime environment by calling a specific, predefined MPI function.

2. Determining how many processes have died and who has died.

3. Recover the lost data from the weighted checksums using the algorithm described in Section 4.3.1.

4. Resume the computation.

Another issue is that how a process can determine whether it is a survival process or it is a re-spawned process. FT-MPI offers the user two possibilities to solve this problem:

- In the first method, when a process is a replacement for a failed process, the return value of its MPI_Init call will be set to a specific new FT-MPI constant (MPI_INIT_RESTARTED_PROCS).

- The second possibility is that the application introduces a static variable. By comparing the value of this variable to the value on the other processes, the application can detect, whether everybody has been newly started (in which case all processes will have the pre-initialized value), or whether a subset of processes have a different value, since each processes modifies the value of this variable after the initial check. This second approach is somewhat more complex, however, it is fully portable and can also be used with any other non fault-tolerant MPI library.

In PCG, each process checks whether it is a re-spawned process or a surviving process by checking the return code of its MPI_Init call.

The relevant section with respect to fault tolerance is shown in the source code below.

```
/* Determine who is re-spawned */
rc = MPI_Init( &argc, &argv );
if (rc==MPI_INIT_RESTARTED_NODE) {
  /* re-spawned procs initialize */
  ...
} else {
  /* Original procs initialize*/
  ...
}

/*Failed procs jump to here to recover*/
setjmp( env );
/* Execute recovery if necessary */
if ( state == RECOVER ) {
  /*Recover MPI environment*/
  newcomm = FT_MPI_CHECK_RECOVER;
  MPI_Comm_dup(oldcomm, &newcomm);
  /*Recover application data*/
  recover_data (A, b, r, p, x, ...);
  /*Reset state-variable*/
  state = NORMAL;
}

/*Major computation loop*/
do {
  /*Checkpoint every K iterations*/
  if ( num_iter % K == 0 )
      checkpoint_data(r, p, x, ...);
  /*Check the return of communication
    calls to detect failure. If failure
    occurs, jump to recovery point*/
  rc = MPI_Send ( ...)
  if ( rc == MPI_ERR_OTHER ) {
      state = RECOVER;
      longjmp ( env, state );
  }
} while ( not converge );
```

5. EXPERIMENTAL EVALUATION

In this section, we evaluate both the performance overhead of our fault tolerance approach and the numerical impact of our floating-point arithmetic encoding using the PCG code implemented in the last section.

We performed four sets of experiments to answer the following four questions:

1. What is the performance of FT-MPI compared with other state-of-the-art MPI implementations?
2. What is the performance overhead of performing checkpointing?
3. What is the performance overhead of performing recovery?
4. What is the numerical impact of round-off errors in recovery?

For each set of experiments, we test PCG with four different problems. The size of the problems and the number of computation processors used (not include checkpoint processors) for each problem are listed in table 1.

All experiments were performed on a cluster of 64 dual-processor 2.4 GHz AMD Opteron 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: Experiment Configurations for Each Problem

	Size of the Problem	Num. of Comp. Procs
Prob #1	164,610	15
Prob #2	329,220	30
Prob #3	658,440	60
Prob #4	1,316,880	120

5.1 Performance of PCG with Different MPI Implementations

The first set of experiments was designed to compare the performance of different MPI implementations and evaluate the overhead of surviving single failure with FT-MPI. We ran PCG with MPICH-1.2.6 [14], MPICH2-0.96, FT-MPI, FT-MPI with one checkpoint processor and no failure, and FT-MPI with one checkpoint processor and one failure for 2000 iterations. For PCG with FT-MPI with checkpoint, we checkpoint every 100 iterations. For PCG with FT-MPI with recovery, we simulate a processor failure by exiting one process at the 1000-th iteration. The execution time of all tests are reported in table 2.

Figure 5 compares the execution time of PCG with MPICH-1.2.6, MPICH2-0.96, FT-MPI, FT-MPI with one checkpoint processor and no failure, and FT-MPI with one checkpoint processor and one failure for different size of problems. Figure 5 indicates that the performance of FT-MPI is slightly better than MPICH2-0.96. Both FT-MPI and MPICH2-0.96 are much faster than MPICH-1.2.6. Even if with checkpointing and/or recovery, the performance of PCG with FT-MPI is still at least comparable to MPICH2-0.96.

Table 2: PCG Execution Time (in seconds) with Different MPI Implementations

Time	Prob#1	Prob#2	Prob#3	Prob#4
MPICH-1.2.6	916.2	1985.3	4006.8	10199.8
MPICH2-0.96	510.9	1119.7	2331.4	7155.6
FT-MPI	480.3	1052.2	2241.8	6606.9
FT-MPI ckpt	482.7	1055.1	2247.5	6614.5
FT-MPI rcvr	485.8	1061.3	2256.0	6634.0

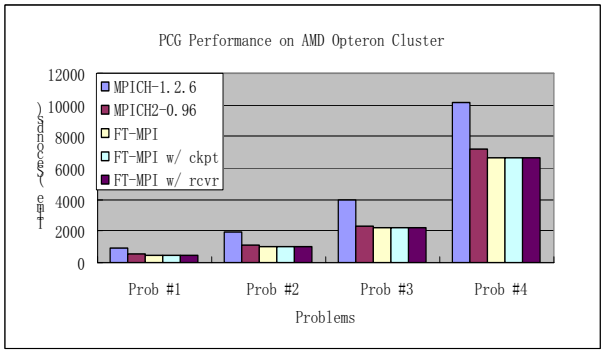

Figure 5: PCG Performance with Different MPI Implementations

5.2 Performance Overhead of Taking Checkpoint

The purpose of the second set of experiments is to measure the performance penalty of taking checkpoints to survive general multiple simultaneous processor failures. There is no processor failures involved in this set of experiments. At each run, we divided the processors into two classes. The first class of processors are dedicated to perform PCG computation work. The second class of processors are dedicated to perform checkpoint. In table 3 and 4, the first column of the table indicates the number of checkpoint processors used in each test. If the number of checkpoint processors used in a run is zero, then there is no checkpoint in this run. For all experiments, we ran PCG for 2000 iterations and checkpoint every 100 iterations.

Table 3: PCG Execution Time (in seconds) with Checkpoint

Time	Prob #1	Prob #2	Prob #3	Prob #4
0 ckpt	480.3	1052.2	2241.8	6606.9
1 ckpt	482.7	1055.1	2247.5	6614.5
2 ckpt	484.4	1057.9	2250.3	6616.9
3 ckpt	486.5	1059.9	2252.4	6619.7
4 ckpt	488.1	1062.2	2254.7	6622.3
5 ckpt	489.9	1064.3	2256.5	6625.1

Table 3 reports the execution time of each test. In order to reduce the disturbance of the noise of the program execution time to the checkpoint time, we measure the time used for checkpointing separately for all experiments. Table 4 reports the individual checkpoint time for each experiment. Figure 6

Table 4: PCG Checkpointing Time (in seconds)

Time	Prob #1	Prob #2	Prob #3	Prob #4
1 ckpt	2.6	3.8	5.5	7.8
2 ckpt	4.4	5.8	8.5	10.6
3 ckpt	6.0	7.9	10.2	12.8
4 ckpt	7.9	9.9	12.6	15.0
5 ckpt	9.8	11.9	14.1	16.8

compares the checkpoint overhead (%) of surviving different numbers of simultaneous processor failures for different size of problems.

Table 4 indicates, as the number of checkpoint processors increases, the time for checkpointing in each test problem will also increases. The increase in time for each additional checkpoint processor is approximately the same for each test problem. However, the increase of the time for each additional checkpoint processor is smaller than the time for using only one checkpoint processor. This is because from no checkpoint to checkpoint with one checkpoint processor PCG has to first set up the checkpoint environment and then do one encoding. However, from checkpoint with k (where $k > 0$) processors to checkpoint with $k + 1$ processors, the only additional work is to perform one more encoding.

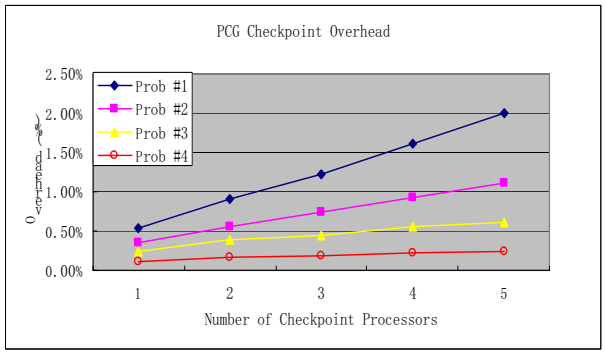

Figure 6: PCG Checkpoint Overhead

Note that we are performing checkpoint every 100 iterations and run PCG for 2000 iterations, therefore, from Table 3, we can calculate the checkpoint interval for each test. Our checkpoint interval ranges from 25 seconds (Prob #1) to 330 seconds (Prob #4). In practice, there is an optimal checkpoint interval which depends on the failure rate, the time cost of each checkpoint and the time cost of each recovery. Much literature about the optimal checkpoint interval [13, 22, 25] is available. We will not address this issue further here.

From figure 6, we can see, even if we checkpoint every 25 seconds (Prob #1), the performance overhead of checkpointing to survive five simultaneous processor failures is still within 2% of the original program execution time, which actually falls into the noise margin of the program execution time. If we checkpoint every 5.5 minutes (Prob #4) and assume a processor fails one after another (one checkpoint processor case), then the overhead is only 0.1%.

5.3 Performance Overhead of Performing Recovery

The third set of experiments is designed to measure the performance overhead to perform recovery. All experiment configurations are the same as previous section except that we simulate a failure of k (k equals the number of checkpoint processors in the run) processors by exiting k processes at the 1000-th iteration in each run.

Table 5 reports the execution time of PCG with recovery. In order to reduce the disturbance of the noise of the program execution time to the recovery time, we measure the time for recovery separately for all experiments. Table 6 reports the recovery time in each experiment. Figure 7 compares the recovery overhead (%) from different number of simultaneous processor failures for different size of problems.

Table 5: PCG Execution Time (in seconds) with Recovery

Time	Prob #1	Prob #2	Prob #3	Prob #4
0 proc	480.3	1052.2	2241.8	6606.9
1 proc	485.8	1061.3	2256.0	6634.0
2 proc	488.1	1063.6	2259.7	6633.5
3 proc	490.0	1066.1	2262.1	6636.3
4 proc	492.6	1068.8	2265.4	6638.2
5 proc	494.9	1070.7	2267.5	6639.7

Table 6: PCG Recovery Time (in seconds)

Time	Prob #1	Prob #2	Prob #3	Prob #4
1 proc	3.2	5.0	8.7	18.2
2 proc	3.7	5.5	9.2	18.8
3 proc	4.0	6.0	9.8	20.0
4 proc	4.5	6.5	10.4	20.9
5 proc	4.8	7.0	11.1	21.5

From table 6, we can see the recovery time increases approximately linearly as the number of failed processors increases. However, the recovery time for a failure of one processor is much longer than the increase of the recovery time from a failure of k (where $k > 0$) processors to a failure of $k+1$ processors. This is because, from no failure to a failure with one failed processor, the additional work the PCG has to perform includes first setting up the recovery environment and then recovering data. However, from a failure with k (where $k > 0$) processors to a failure with $k + 1$ processors, the only additional work is to recover data for an additional processor.

From Figure 7, we can see the overheads for recovery in all tests are within 1% of the program execution time, which is again within the noise margin of a program execution time.

5.4 Numerical Impact of Round-Off Errors in Recovery

As discussed in Section 3, our diskless checkpointing schemes are based on floating-point arithmetic encodings, therefore, introduce round-off errors into the checkpointing system. The experiments in this sub-section are designed to measure

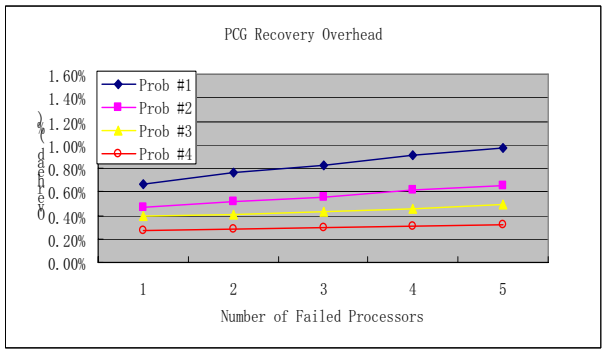

Figure 7: PCG Recovery Overhead

the numerical impact of the round-off errors in our checkpointing system. All experiment configurations are the same as previous section except that we report the norm of the residual at the end of each computation.

Note that if no failures occur, the computation proceeds with the same computational data as without checkpoint. Therefore, the computational results are affected only when there is a recovery in the computation. Table 7 reports the norm of the residual at the end of each computation when there is 0, 1, 2, 3, 4, and 5 simultaneous process failures.

Table 7: Numerical Impact of Round-Off Errors in PCG Recovery

Residual	Prob #1	Prob #2	Prob #3	Prob #4
0 proc	3.050e-6	2.696e-6	3.071e-6	3.944e-6
1 proc	2.711e-6	4.500e-6	3.362e-6	4.472e-6
2 proc	2.973e-6	3.088e-6	2.731e-6	2.767e-6
3 proc	3.036e-6	3.213e-6	2.864e-6	3.585e-6
4 proc	3.438e-6	4.970e-6	2.732e-6	4.002e-6
5 proc	3.035e-6	4.082e-6	2.704e-6	4.238e-6

From table 7, we can see that the norm of the residuals are different for different number of simultaneous process failures. This is because, after recovery, due to the impact of round-off errors in the recovery algorithm, the PCG computations are performed based on different recovered data. However, table 7 also indicates that the residuals with recovery do not have much difference from the residuals without recovery. This is because the PCG algorithm we use are numerically stable for the test problems and, from the floating-point number point of view, all the recovered data do not have much difference from the lost data.

However, if the applications are itself numerically unstable, then the small difference between the lost data and the recovered data can be amplified. Fortunately, most high performance numerical computing applications require numerically stable algorithms. Therefore, our recovery schemes would introduce very little impact on them.

6. DISCUSSION

The size of the checkpoint affects the performance of any checkpointing scheme. The larger the checkpoint size is, the higher the diskless checkpoint overhead would be. In the PCG example, we only need to checkpoint three vectors and two scalars periodically, therefore, the performance overhead is very low.

Diskless checkpointing is good for applications that modify a small amount of memory between checkpoints. There are many such applications in high performance computing field. For example, in typical iterative methods for sparse matrix computation, the sparse matrix is often not modified during the program execution, only some vectors and scalars are modified between checkpoints. For this type of application, the overhead for surviving a small number of processor failures is very low.

Even for applications which modify a relatively large amount of memory between two checkpoints, decent performance results to survive single processor failure were still reported in [16, 19].

The basic weighted checksum scheme implemented in the PCG example has a higher performance overhead than other schemes discussed in Section 3. When an application is executed on large number of processors, to survive general multiple simultaneous processor failures, the one dimensional weighted checksum scheme will achieve a much lower performance overhead than the basic weighted checksum scheme. If processor fails one after another (i.e. no multiple simultaneous processor failures), the neighbor based schemes can achieve even lower performance overhead. It was shown in [6] that a neighbor-based checkpointing was an order of magnitude faster than a parity-based checkpointing, but takes twice as much storage overhead.

Diskless checkpointing could not survive a failure of all processors. Also, to survive a failure occurred during checkpoint or recovery, the storage overhead would double. If an application needs to tolerate these types of failures, a two level recovery scheme [24] which uses both diskless checkpointing and stable-storage-based checkpointing is a good choice.

Another drawback of our fault tolerance approach is that it requires the programmer to be involved in the fault tolerance. However, if the fault tolerance schemes are implemented into numerical softwares such as LFC [5], then transparent fault tolerance can also be achieved for programmers using these software tools.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented how to build fault tolerant high performance computing applications with FT-MPI by a coding approach. We introduced the simple but efficient floating-point arithmetic coding approach into diskless checkpointing and addressed the associated round-off issue. We implemented a floating-point arithmetic version of the Reed-Solomon coding scheme into a conjugate gradient equation solver and evaluated both the performance and the numerical impact of this coding scheme. Experimental results demonstrated that the proposed floating-point arithmetic coding approach is able to survive a small number of simultaneous node failures with low performance overhead and little numerical impact.

For the future, we will evaluate our fault tolerance approach on systems with larger number of processors. We would also like to evaluate our fault tolerance approach with more applications and more coding schemes.

8. 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] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] G. Bosilca, Z. Chen, J. Dongarra, and J. Langou. Recovery patterns for iterative methods in a parallel unstable environment. Technical Report ut-cs-04-538, University of Tennessee, Knoxville, Tennessee, USA, 2004.

[4] Z. Chen and J. Dongarra. Condition numbers of gaussian random matrices. Technical Report ut-cs-04-539, University of Tennessee, Knoxville, Tennessee, USA, 2004.

[5] 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.

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

[7] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 24th edition. In *Proceedings of the Supercomputing Conference (SC'2004), Pittsburgh PA, USA*. ACM, 2004.

[8] A. Edelman. Eigenvalues and condition numbers of random matrices. *SIAM J. Matrix Anal. Appl.*, 9(4):543–560, 1988.

[9] 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.

[10] 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.

[11] 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.

[12] A. Geist and C. Engelmann. Development of naturally fault tolerant algortihms for computing on 100,000 processors. *Submited to J. Parallel Distrib. Comput.*, 2002.

[13] E. Gelenbe. On the optimum checkpoint interval. *J. ACM*, 26(2):259–270, 1979.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

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

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

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

[18] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[19] J. S. Plank, Y. Kim, and J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *J. Parallel Distrib. Comput.*, 43(2):125–138, 1997.

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

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

[22] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *J. Parallel Distrib. Comput.*, 61(11):1570–1590, November 2001.

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

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

[25] J. W. Young. A first order approximation to the optimal checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.