Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing

Zizhong Chen, *Member*, *IEEE*, and Jack Dongarra, *Fellow*, *IEEE*

Abstract—As the number of processors in today's high-performance computers continues to grow, the mean-time-to-failure of these computers is 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 of today's high-performance computing applications cannot survive node failures. Therefore, whenever a node fails, all surviving processes on surviving nodes usually have to be aborted and the whole application has to be restarted. In this paper, we present a framework for building self-healing high-performance numerical computing applications so that they can adapt to node or link failures without aborting themselves. The framework is based on FT-MPI and diskless checkpointing. Our diskless checkpointing uses weighted checksum schemes, a variation of Reed-Solomon erasure codes over floating-point numbers. We introduce several scalable encoding strategies into the existing diskless checkpointing and reduce the overhead to survive k failures in p processes from $2\lceil \log p \rceil . k((\beta + 2\gamma)m + \alpha)$ to $(1 + O(\frac{\sqrt{p}}{\sqrt{m}}))^2 . k(\beta + 2\gamma)m$, where α is the communication latency, $\frac{1}{\beta}$ is the network bandwidth between processes, $\frac{1}{\gamma}$ is the rate to perform calculations, and m is the size of local checkpoint per process. When additional checkpoint processors are used, the overhead can be reduced to $(1 + O(\frac{1}{\sqrt{m}})) . k(\beta + 2\gamma)m$, which is independent of the total number of computational processors. The introduced self-healing algorithms are scalable in the sense that the overhead to survive k failures in p processes does not increase as the number of processes p increases. We evaluate the performance overhead of our self-healing approach by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that our self-healing scheme can survive multiple simultaneous process failures with low-performance overhead and little numerical impact.

Index Terms—Self-healing, diskless checkpointing, fault tolerance, pipeline, parallel and distributed systems, high-performance computing, Message Passing Interface.

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 is becoming significantly shorter than the execution time of many current high-performance computing applications.

Even when 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 days, or less. The current DOE ASCI computer (IBM Blue Gene L) is designed with 1,31,000 processors. 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 [9] has been shifting from the expensive massively parallel computer systems to clusters of commodity off-the-shelf systems [9]. While commodity off-the-shelf cluster systems have excellent price-performance ratio, the low reliability of the off-the-shelf components in these systems leads a growing concern with the fault tolerance issue. The recently emerging computational grid environments [3], [14] 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. To avoid restarting computations after failures, the next-generation high-performance computing applications need to be able to continue execution despite of failures.

Today's long-running scientific applications typically tolerate failures by writing checkpoints into stable storage periodically. If a process failure occurs, then all surviving application 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 [23]. 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 reread from stable storage into memory or

- Z. Chen is with the Department of Mathematical and Computer Sciences, Colorado School of Mines, 1500 Illinois Street, Golden, CO 80401. E-mail: zchen@mines.edu.
- J. Dongarra is with the Department of Electrical Engineering and Computer Science, University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN 37996-3450. E-mail: dongarra@cs.utk.edu.

Manuscript received 27 May 2008; revised 26 Nov. 2008; accepted 23 Feb. 2009; published online 4 Mar. 2009.
Recommended for acceptance by D.R. Avresky, H. Prokop, and D.C. Verma.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-05-0234.
Digital Object Identifier no. 10.1109/TC.2009.42.

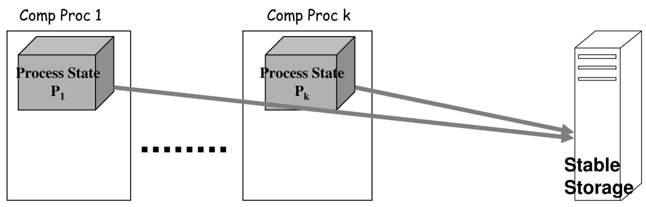

Fig. 1. Tolerate failures by checkpoint/restart approach.

regenerated 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.

Furthermore, 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. Fig. 1 shows how a typical checkpoint/restart approach works.

Due to the high frequency of failures and the large number of processors in next-generation computing systems, the classical checkpoint/restart fault tolerance approach may become a very inefficient way to handle failures. Alternative fault tolerance approaches need to be investigated.

In this paper, we present an alternative self-healing approach that is based on FT-MPI, a fault-tolerant version of MPI [7], [11], [12], and diskless checkpointing [7], [23]. Applications built in our framework are self-healing and can survive multiple simultaneous node or link failures. When part of an application's processes failed due to node or link failure, unlike in classical checkpoint/restart fault tolerance paradigm, the application in our fault tolerance framework will not be aborted. Instead, it will keep all its surviving processes and adapt itself to failures.

Our diskless checkpointing uses weighted checksum schemes, a variation of Reed-Solomon erasure codes over floating-point numbers. Several new encoding strategies are introduced into diskless checkpointing to improve the scalability of the technique. The introduced schemes reduce the fault tolerance overhead to survive k failures in p processes from $2\lceil \log p \rceil \cdot k((\beta + 2\gamma)m + \alpha)$ to $(1 + O(\frac{\sqrt{p}}{\sqrt{m}}))^2 \cdot k(\beta + 2\gamma)m$, where α is the communication latency, $\frac{1}{\beta}$ is the network bandwidth between processes, $\frac{1}{\gamma}$ is the rate to perform calculations, and m is the size of local checkpoint per process. When additional checkpoint processors are used, the overhead can be reduced to $(1 + O(\frac{1}{\sqrt{m}})) \cdot k(\beta + 2\gamma)m$, which is independent of the total number of computational processors. The proposed self-healing schemes are scalable in the sense that the overhead to survive k failures in p processes does not increase as the total number of application processes p increases.

We give a detailed presentation on how to write self-healing high-performance computing applications with FT-MPI using diskless checkpointing and evaluate the performance overhead of our self-healing approach by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that our self-healing approach can survive a small number of simultaneous processor failures with low-performance overhead and little numerical impact.

The rest of this paper is organized as follows: Section 2 introduces FT-MPI, a fault-tolerant version of MPI implementation. In Section 3, we introduce a pipeline-based encoding algorithm to improve the scalability of diskless checkpointing. In Section 4, we present scalable coding strategies to survive multiple simultaneous failures. In Section 5, 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 6, we evaluate both the performance overhead of our fault tolerance approach and the numerical impact of our floating-point arithmetic encoding. Section 7 discusses the limitations of our approach and possible improvements. Section 8 concludes the paper and discusses future work.

2 FT-MPI: A Fault-Tolerant Message Passing Interface

Current parallel programming paradigms for high-performance distributed computing systems are typically based on the Message Passing Interface (MPI) specification [19]. However, the current MPI specification does 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 on how to handle failures. The first one, which is the default mode of MPI, is to immediately abort all surviving 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.

2.1 FT-MPI Overview

FT-MPI [7], [11], [12] 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 and parts of the MPI-2 functionality, and extends some of the semantics of MPI to support self-healing applications. FT-MPI is able to survive the failure of $n-1$ processes in an n-process job, and, if required, can respawn the failed processes. However, fault-tolerant applications have to be implemented in a self-healing way so that they can survive failures.

Although FT-MPI provides basic system services to support self-healing applications, prevailing benchmarks show that the performance of FT-MPI is comparable [13] to the current state-of-the-art nonfault-tolerant MPI implementations.

2.2 FT-MPI Semantics

FT-MPI provides semantics that answer the following questions:

1. What is the status of an MPI communicator after recovery?
2. What is the status of the 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 is *communicator mode* which indicates 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, in this FT-MPI mode, the application aborts itself after failure.
- 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 respawned, and 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 sent but not received while a failure 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 pending messages are dropped. The assumption behind this mode is that on error, the application returns to its last consistent state, and all currently pending operations are not of any further interest.

2.3 FT-MPI Usage

It usually takes three steps to tolerate a failure: 1) failure detection, 2) failure notification, and 3) recovery. The only assumption the FT-MPI specification makes about the first two points is that the runtime environment discovers failures and all remaining processes in the parallel job are notified about these events. The recovery procedure consists of two steps: recovering the MPI runtime environment and recovering the application data. The latter one is considered to be the responsibility of the application developer. 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 these modes several possibilities. This allows application developers to take the specific characteristics of their application into account and use the best-suited method to tolerate failures.

3 A Scalable Encoding Algorithm for Diskless Checkpointing

Diskless checkpointing [23] is a technique to save the state of a long-running computation on a distributed system without relying on stable storage. With diskless checkpointing, each processor involved in the computation stores a copy of its state locally, either in memory or on local disk. In addition, encodings of these checkpoints are stored in local memory or on local disk of some processors which may or may not be involved in the computation. 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, diskless checkpointing removes the main source of overhead in checkpointing on distributed systems [23].

In typical long-running scientific applications, when diskless checkpointing is taken from application level, what needs to be checkpointed is often some numerical data [18]. These numerical data can either be treated as bitstreams or as floating-point numbers. If the data are treated as bitstreams, then bitstream operations such as parity can be used to encode the checkpoint. Otherwise, floating-point arithmetic such as addition can be used to encode the data.

However, compared with treating checkpoint data as numerical numbers, treating them as bitstreams usually has the following disadvantages:

1. To survive general multiple process failures, treating checkpoint data as bitstreams often involves the introduction of Galois Field arithmetic in the calculation of checkpoint encoding and recovery decoding [20]. 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 bitstreams rules out a heterogeneous recovery. The checkpoint data may have different bitstream representation on different platforms and even have different bitstream length on different architectures. The introduction of a unified representation of the checkpoint data on different platforms within an application for checkpoint purposes sacrifices too much performance and is unrealistic in practice.
3. In some cases, treating checkpoint data as bitstreams does not work. For example, in [18], in order to reduce memory overhead in fault-tolerant dense matrix computation, no local checkpoints are maintained on computation processors; only the checksums of the local checkpoints are maintained on the checkpoint processors.

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 error is a limitation of any floating-point number calculation. 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.

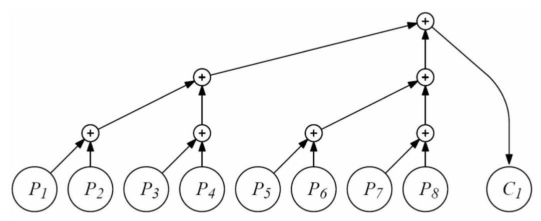

Fig. 2. Encoding local checkpoints using the binary tree algorithm.

In this paper, we treat the checkpoint data as floating-point numbers rather than bitstreams. However, the corresponding bitstream version schemes could also be used if the application programmer thinks they are more appropriate. In the following section, we discuss how the local checkpoint can be encoded so that applications can survive single process failure.

3.1 The Binary Encoding Algorithm for Diskless Checkpointing

Assume that diskless checkpointing is performed in a parallel system with p processors and the size of checkpoint on each processor is m bytes. Also assume that it takes $\alpha + \beta x$ to transfer a message of size x bytes between two processors regardless of which two processors are involved. The variable α is often called latency of the network and $\frac{1}{\beta}$ is called the bandwidth of the network. Assume that the rate to calculate the sum 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. 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 interfering each other.

In classical diskless checkpointing, binary-tree-based encoding algorithm is often used to perform the checkpoint encoding [8], [18], [20], [23], [25]. By organizing all processors as a binary tree and sending local checkpoints along the tree to the checkpoint processor (see Fig. 2 [23]), the time to perform one checkpoint for a binary-tree-based encoding algorithm, $T_{diskless-binary}$, can be represented as

$$T_{diskless-binary} = 2\lceil \log p \rceil \cdot ((\beta + \gamma)m + \alpha). \qquad (1)$$

In high-performance scientific computing, the local checkpoint is often a relatively large message (megabyte level), so $(\beta + \gamma)m$ is usually much larger than α. Therefore, $T_{diskless-binary} \approx 2\lceil \log p \rceil \,.\, (\beta + \gamma)m$.

Note that, in a typical checkpoint/restart approach (see Fig. 1 in Section 1) where βm is usually much larger than α, the time to perform one checkpoint, $T_{checkpoint/restart}$, is

$$\begin{aligned} T_{checkpoint/restart} &= p \cdot (\beta m + \alpha) \\ &\approx p \cdot \beta m. \end{aligned}$$

Therefore, by eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing improves the scalability of checkpointing greatly on parallel and distributed systems.

3.2 A Chain-Pipelined Encoding Algorithm for Diskless Checkpointing

Although the classical diskless checkpointing technique improves the scalability of checkpointing dramatically on parallel and distributed systems, the overhead to perform one checkpoint still increases logarithmically because $T_{diskless-binary} \approx 2\lceil \log p \rceil \cdot (\beta + \gamma)m)$ as the number of processors increases. In this section, we propose a new style of encoding algorithm which improves the scalability of diskless checkpointing significantly. The new encoding algorithm is based on the pipeline idea.

The key idea of pipelining is 1) the segmenting of messages and 2) the simultaneous nonblocking transmission and receipt of data. By breaking up a large message into smaller segments and sending these smaller messages through the network, pipelining allows the receiver to begin forwarding a segment while receiving another segment.

Data pipelining can produce several significant improvements in the process of checkpoint encoding. First, pipelining masks the processor and network latencies that are known to be an important factor in high-bandwidth local area networks. Second, it allows the simultaneous sending and receiving of data, and hence exploits the full duplex nature of the interconnect links in the parallel system. Third, it allows different segments of a large message being transmitted in different interconnect links in parallel after a pipeline is established, hence fully utilize the multiple interconnects of a parallel and distributed system.

When the number of processors is one or two, there is not much that we can improve. Therefore, in what follows, we assume that the number of processors is at least three (i.e., $p \geq 3$).

Assume that there are $p - 1$ computational processors and one checkpoint processor. Let $m[i]$ denotes the data on the ith processor, where $i = 0, 1, 2, \dots,$ and $p - 2$. The task of checkpoint encoding is to calculate the encoding which is $m[0] + m[1] + \cdots + m[p-2]$ and deliver the encoding to the checkpoint processor.

The chain-pipelined encoding algorithm in a p-processor system works as follows. First, organize all $p - 1$ computational processors and the checkpoint processor as a chain. Second, divide the data on each processor into many small pieces. Assume that the data on each processor are divided into t segments of size s. The jth segment of $m[i]$ is denoted as $m[i][j]$. Third, $m[0] + m[1] + \cdots + m[p-2]$ are calculated by calculating $m[0][j] + m[1][j] + \cdots + m[p-2][j]$ for each $0 \leq j \leq t - 1$ in a pipelined way. Fourth, when the jth segment of encoding $m[0][j] + m[1][j] + \cdots + m[p-2][j]$ is available, start to send it to the checkpoint processor.

Fig. 3 demonstrates an example of calculating a chain-pipelined checkpoint encoding for three processors (processor 0, processor 1, and processor 2) and deliver it to the checkpoint processor (processor 3). In step 0, processor 0 sends its $m[0][0]$ to processor 1. Processor 1 receives $m[0][0]$ from processor 0 and calculates $m[0][0] + m[1][0]$. In step 1, processor 0 sends its $m[0][1]$ to processor 1. Processor 1 first concurrently receives $m[0][1]$ from processor 0 and sends $m[0][0] + m[1][0]$ to processor 2, and then calculates

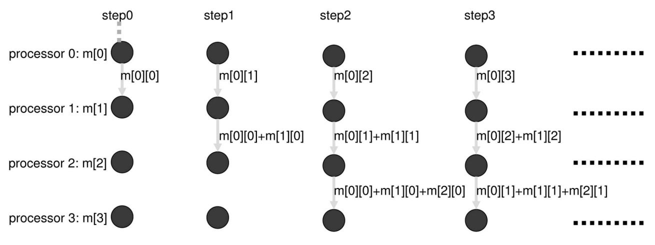

Fig. 3. Chain-pipelined encoding for diskless checkpointing.

$m[0][1] + m[1][1]$. Processor 2 first receives $m[0][0] + m[1][0]$ from processor 1 and then calculates $m[0][0] + m[1][0] + m[2][0]$. As the procedure continues, at the end of step 2, the checkpoint processor will be able to get its first segment of encoding $m[0][0] + m[1][0] + m[2][0]$. From now on, the checkpoint processor will be able to receive a segment of the encoding at the end of each step. After the checkpoint processor receives the last checkpoint encoding, the checkpoint is finished.

3.3 Overhead and Scalability Analysis of the Chain-Pipelined Encoding

In the above chain-pipelined checkpoint encoding, the time for each step can be modeled by $T_{each-step} = \alpha + \beta s + \gamma s$. Since there are p processors in the system, it takes $p-1$ steps for the checkpoint processor to get the encoding of the first segment. After the checkpoint processor gets the encoding of the first segment, the pipeline is established. After the pipeline is established, it takes $t-1$ steps for the checkpoint processor to get the encoding of the rest $t-1$ segments. Therefore, the number of steps to encode and deliver t segments in a p-processor system is $(p-1) + (t-1)$. If we assume that the size of data on each processor is m ($= ts$, where s is the size of the segment), then the total time for encoding and delivery all t segments is

$$\begin{aligned} T_{total}(s) &= [(p-1) + (t-1)](\alpha + \beta s + \gamma s) \\ &= \left(p - 2 + \frac{m}{s}\right)(\alpha + \beta s + \gamma s). \end{aligned}$$

Note that

$$T''_{total}(s) = \frac{2m\alpha}{s^3} \geq 0,$$

and

$$\lim_{s \longrightarrow \infty} T_{total}(s) = \lim_{s \longrightarrow 0} T_{total}(s) = \infty.$$

Therefore, there is a minimum for T_{total} when s changes.

By solving

$$T'_{total}(s) = -\frac{m\alpha}{s^2} + (p-2)(\beta + \gamma) = 0,$$

we get the critical point

$$s_1 = \sqrt{\frac{m\alpha}{(p-2)(\beta+\gamma)}}.$$

If $m > s_1$, then $T_{total}(s)$ reaches its minimum at $s = s_1$. Otherwise, $T_{total}(s)$ reaches its minimum at $s = m$.

When $s = s_1$,

$$\begin{aligned} T_{total}(s) &= (p-2)\alpha + (\beta+\gamma)m + 2\sqrt{(p-2)\alpha(\beta+\gamma)m} \\ &= (\beta+\gamma)m \, . \left(1 + 2\sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}} + \frac{(p-2)\alpha}{(\beta+\gamma)m}\right) \\ &= (\beta+\gamma)m \, . \left(1 + \sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}}\right)^2 \\ &= (\beta+\gamma)m \, . \left(1 + O\left(\frac{\sqrt{p}}{\sqrt{m}}\right)\right)^2. \end{aligned} \tag{2}$$

Therefore, when $m > \sqrt{\frac{m\alpha}{(p-2)(\beta+\gamma)}}$, by choosing an optimal segment size, the chain-pipelined encoding algorithm is able to reduce the checkpoint overhead to tolerate single failure from $2\lceil \log p \rceil \cdot ((\beta+\gamma)m + \alpha)$ to $(1 + O(\frac{\sqrt{p}}{\sqrt{m}}))^2 \cdot (\beta+\gamma)m$.

In diskless checkpointing, the size of checkpoint m is often large (megabytes level). The latency α is often negligible compared with the time to send a large message. When $(\beta+\gamma)m \gg (p-2)\alpha$, both $2\sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}}$ and $\frac{(p-2)\alpha}{(\beta+\gamma)m}$ are negligible. $T_{total} \approx (\beta+\gamma)m$. Therefore, in practice, the number of processors often has very little impact on the time to perform one checkpoint when p is relatively small (i.e., when p satisfies $(\beta+\gamma)m \gg (p-2)\alpha$).

For example, if $p < 100$, $m > 10$ Mbytes, $\alpha = 10^{-5}$ seconds, $\frac{1}{\beta} = 100$ Mbytes/seconds, and $\frac{1}{\gamma} = 1{,}000$ Mbytes/seconds, then $(\beta+\gamma)m > 0.11$, $(p-2)\alpha < 0.001$, and $\sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}} < 0.01$. Therefore,

$$\begin{aligned} T_{total} &= (\beta+\gamma)m \, . \left(1 + \sqrt{\frac{(p-2)\alpha}{(\beta+\gamma)m}}\right)^2 \\ &\approx 0.11(1 + 0.01)^2 \\ &\approx 0.11, \end{aligned}$$

which does not change as p changes.

However, if p does become very large (i.e., p does not satisfy $(\beta+\gamma)m \gg (p-2)\alpha$ any more), then strategies in Sections 4.2 and 4.3 have to be used.

4 Coding to Tolerate Multiple Simultaneous Process Failures

To tolerate multiple simultaneous process failures of arbitrary patterns with minimum process redundancy, a weighted checksum scheme can be used. A weighted checksum scheme can be viewed as a version of the Reed-Solomon erasure coding scheme [20] in the real number field. The basic idea of this scheme works as follows. Each processor takes a local in-memory checkpoint, and M equalities are established by saving weighted checksums of the local checkpoint into M checksum processors. When f failures happen, where $f \leq M$, the M equalities become 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.

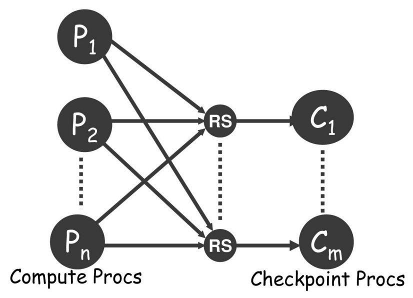

Fig. 4. Basic weighted checksum scheme for diskless checkpointing.

4.1 The Basic Weighted Checksum Scheme

Suppose there are n processors used for computation. Assume that the checkpoint data on the ith 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 Fig. 4).

The weighted checksum C_j on the jth checksum processor can be calculated from

$$\begin{cases} a_{11}P_1 + \cdots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{M1}P_1 + \cdots + a_{Mn}P_n & = C_M, \end{cases} \tag{3}$$

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

Suppose k computation processors and $M - h$ checkpoint processors have failed. Then, there are $n - k$ computation processors and h checkpoint processors that have survived. If we look at the data on the failed processors as unknowns, then (3) becomes M equations with $M - (h - k)$ unknowns.

If $k > h$, then there are fewer equations than unknowns. There is no unique solution for (3), and the lost data on the failed processors cannot be recovered.

However, if $k < h$, then there are more equations than unknowns. By appropriately choosing A, a unique solution for (3) can be guaranteed, and the lost data on the failed processors can be recovered by solving (3).

Without loss of generality, we assume that 1) the computational processors $j_1, j_2, \ldots, j_k$ failed and the computational processors $j_{k+1}, j_{k+2}, \ldots, j_n$ survived; 2) the checkpoint processors $i_1, i_2, \ldots, i_h$ survived and the checkpoint processors $i_{h+1}, i_{h+2}, \ldots, i_M$ failed. Then, in (3), $P_{j_1}, \ldots, P_{j_k}$ and $C_{i_{h+1}}, \ldots, C_{i_M}$ become unknowns after the failure occurs. If we restructure (3), we can get

$$\begin{cases} a_{i_1j_1}P_{j_1} + \cdots + a_{i_1j_k}P_{j_k} & = C_{i_1} - \sum_{t=k+1}^{n} a_{i_1j_t}P_{j_t} \\ & \vdots \\ a_{i_hj_1}P_{j_1} + \cdots + a_{i_hj_k}P_{j_k} & = C_{i_h} - \sum_{t=k+1}^{n} a_{i_hj_t}P_{j_t}, \end{cases} \tag{4}$$

and

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1}P_1 + \cdots + a_{i_{h+1}n}P_n \\ \vdots & \\ C_{i_M} & = a_{i_M1}P_1 + \cdots + a_{i_Mn}P_n. \end{cases} \tag{5}$$

Let A_r denote the coefficient matrix of the linear system (4). If A_r has full column rank, then $P_{j_1}, \ldots, P_{j_k}$ can be recovered by solving (4), and $C_{i_{h+1}}, \ldots, C_{i_M}$ can be recovered by substituting $P_{j_1}, \ldots, P_{j_k}$ into (5).

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 (4) can be any submatrix (including minor) of A depending on the distribution of the failed processors. If any square submatrix (including minor) of A is nonsingular and there are no more than M processes failed, then A_r can be guaranteed to have full column rank. Therefore, to be able to recover from no more than m failures, the checkpoint matrix A has to satisfy the condition that *any square submatrix (including minor) of A is nonsingular.*

How can we find such kind of matrices? It is well known that some structured matrices such as Vandermonde matrix satisfy this condition.

However, in computer floating-point arithmetic where no computation is exact due to round-off errors, it is well known [17] 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 the condition that *any square submatrix (including minor) of A is well conditioned.*

It is well known [10] that Gaussian random matrices are well conditioned. To estimate how well conditioned Gaussian random matrices are, we have proved the following theorem in [5].

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 $0.245 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.414$ are universal positive constants independent of m, n, and x.

Note that any submatrix of a Gaussian random matrix is still a Gaussian random matrix. Therefore, a Gaussian random matrix would satisfy the condition that any submatrix of the matrix is well conditioned with high probability. Due to their nice numerical property, Gaussian random matrices are used as the generator matrices for real number codes in [4].

Theorem 1 can be used to estimate the accuracy of recovery in the weighted checksum scheme. For example, if an application uses $1{,}00{,}000$ processors to perform computation and 20 processors to perform checkpointing, then the checkpoint matrix is a $20 \times 1{,}00{,}000$ Gaussian random matrix. If 10 processors fail concurrently, then the coefficient

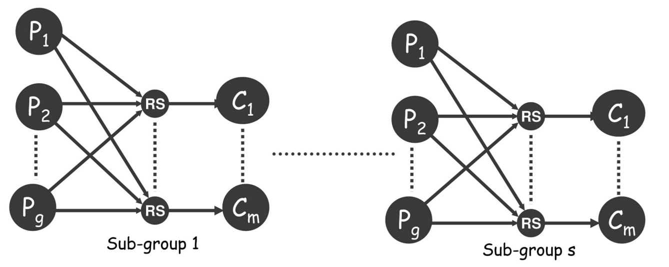

Fig. 5. One-dimensional weighted checksum scheme for diskless checkpointing.

matrix A_r in the recovery algorithm is a 20×10 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 lose about one decimal digit in the recovered data and the probability to lose two digits is less than 3.1×10^{-11}.

Let $T_{diskless\_pipeline}(k,p)$ denote the encoding time to tolerate k simultaneous failures in a p-processor system using the chain-pipelined encoding algorithm and $T_{diskless\_binary}(k,p)$ denote the corresponding encoding time using the binary tree encoding algorithm.

When tolerating k simultaneous failures, k basic encodings have to be performed. Note that, in addition to the summation operation, there is an additional multiplication operation involved. Therefore, the computation time for each number will increase from γ to 2γ. Hence, when the binary tree encoding algorithm is used to perform the weighted checksum encoding, the time for one basic encoding is $2\lceil \log p \rceil \,.\, ((\beta + 2\gamma)m + \alpha)$. Therefore, the time for k basic encodings is

$$\begin{aligned} T_{diskless\_binary}(k,p) &= k \,.\, 2\lceil \log p \rceil \,.\, ((\beta + 2\gamma)m + \alpha) \\ &\approx 2\lceil \log p \rceil \,.\, k(\beta + 2\gamma)m. \end{aligned} \tag{6}$$

When the chain-pipelined encoding algorithm is used to perform the checkpoint encoding, the overhead to tolerate k simultaneous failures becomes

$$\begin{aligned} T_{diskless\_pipeline}(k,p) &= k \,.\, \left(1 + O\left(\frac{\sqrt{p}}{\sqrt{m}}\right)\right)^2 (\beta + 2\gamma)m \\ &= \left(1 + O\left(\frac{\sqrt{p}}{\sqrt{m}}\right)\right)^2 \,.\, k(\beta + 2\gamma)m. \end{aligned} \tag{7}$$

When the number of processors p is not too large, the overhead for the basic weighted checksum scheme $T_{diskless\_pipeline}(k,p) \approx k(\beta + 2\gamma)m$.

However, in today's large computing systems, the number of processors p may become very large. If we do have a large number of processors in the computing systems, either the one-dimensional weighted checksum scheme in Section 4.2 or the localized weighted checksum scheme in Section 4.3 can be used.

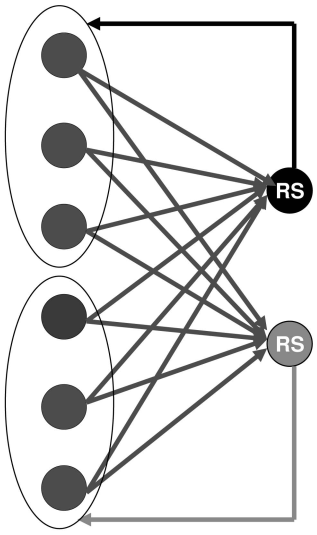

Fig. 6. Localized weighted checksum scheme for diskless checkpointing.

4.2 One-Dimensional Weighted Checksum Scheme

The one-dimensional weighted checksum scheme works as follows. Assume that the program is running on $p = g \times s$ processors. Partition the $g \times s$ processors into s groups with g processors in each group. Dedicate another M checksum processors for each group. In each group, the checkpoints are performed using the basic weighted checksum scheme (see Fig. 5). This scheme can survive M processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each subgroup can be performed in parallel. Therefore, compared with the basic weighted checksum scheme, the performance of the one-dimensional weighted checksum scheme is usually better.

By using a pipelined encoding algorithm in each subgroup, the time to tolerate k simultaneous failures in a p-processor system is now reduced to

$$\begin{aligned} T_{diskless\_pipeline}(k,p) &= T_{diskless\_pipeline}(k,g) \\ &= \left(1 + O\left(\frac{\sqrt{g}}{\sqrt{m}}\right)\right)^2 \,.\, k(\beta + 2\gamma)m \\ &= \left(1 + O\left(\frac{1}{\sqrt{m}}\right)\right) \,.\, k(\beta + 2\gamma)m, \end{aligned} \tag{8}$$

which is independent of the total number of processors p in the computing system. Therefore, the overhead to survive k failures in a p-processor system does not increase as the total number of processors p increases. It is in this sense that the subgroup-based chain-pipelined checkpoint encoding algorithm is a super-scalable self-healing algorithm.

4.3 Localized Weighted Checksum Scheme

The localized weighted checksum scheme works as follows. Assume that we want to tolerate k simultaneous process failures. Divide all processes into subgroups of size $k(k+1)$. In each group, the checkpoint encoding is performed like the basic weighted checksum scheme (see Fig. 6). But each encoding is distributed into $k+1$ processes in the subgroup. Note that there are $k(k+1)$ processes in each subgroup; therefore, all k encodings can be replicated in $k+1$ processes, with each process holding only one encoding. This scheme can survive k processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each subgroup can be done in parallel. Therefore, compared with the basic weighted checksum scheme, the performance of the localized weighted checksum scheme is usually better. Another advantage of the localized weighted checksum scheme is that it does not require dedicated processes to hold the checkpoint encoding.

Let $T_{dc\_pip\_loc}(k,p)$ denote the encoding time for localized weighted checksum scheme and $T_{bcast}(p)$ denote the time to broadcast a message of size m to p processors. The pipeline idea can also be used to broadcast messages. By using a pipelined style of algorithms to broadcast and encoding, the time to perform one checkpoint in the localized weighted checksum scheme is

$$\begin{aligned} T_{dc\_pip\_loc}(k,p) &= T_{diskless\_pipeline}(k, k(k+1)) \\ &\quad + T_{bcast}(k+1) \\ &= \left(1 + O\left(\frac{\sqrt{k(k+1)}}{\sqrt{m}}\right)\right)^2 k(\beta + 2\gamma)m \\ &\quad + \left(1 + O\left(\frac{\sqrt{k+1}}{\sqrt{m}}\right)\right)^2 \cdot \beta m \\ &= \left(1 + O\left(\frac{k}{\sqrt{m}}\right)\right)^2 (k+1)(\beta + 2\gamma)m, \end{aligned} \tag{9}$$

which is also independent of the total number of processors p in the computing system.

5 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.

5.1 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 pseudocode for the PCG is given in Fig. 7. For more details of the algorithm, we refer the reader to [2].

```
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
```

Fig. 7. Preconditioned conjugate gradient algorithm.

5.2 Incorporating Fault Tolerance into PCG

We first implemented the parallel nonfault-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.

We then incorporate the basic weighted checksum scheme into the PCG code. Assume that 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 pseudorandom 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 Fig. 7).

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 pending messages and respawn 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 of incorporating fault tolerance into applications that call existing parallel numerical libraries that do not check the return code of their MPI calls. In the 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. Respawn 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 4.1.
4. Resume the computation.

Another issue is how a process can determine whether it is a surviving process or it is a respawned 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 preinitialized value), or whether a subset of processes have a different value, since each process 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 nonfault-tolerant MPI library.

In PCG, each process checks whether it is a respawned 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 );
```

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

6 Experimental Evaluation

In this section, we evaluate both the performance overhead of the proposed weighted checksum schemes (based on the chain-pipelined encoding) and the numerical impact of our floating-point arithmetic encoding using the PCG code implemented in the last section.

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

1. What is the performance of FT-MPI compared with other state-of-the-art nonfault-tolerant MPI implementations?
2. What is the performance overhead of performing checkpoint as the number of checkpointing processors increases?
3. What is the performance overhead of performing recovery as the number of simultaneous processor failures increases?
4. What is the numerical impact of round-off errors when recovering from multiple simultaneous processor failures?
5. What is the scalability of the pipeline-based self-healing algorithm when the number of simultaneous processor failures is fixed but the total number of application processors increases?

In the first four sets of experiments, we test PCG with four different problems in each set of experiments. The size of the problems and the number of computation processors used (not including 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.

For the fifth set of experiments, we increase the total number of processors for computing, but choose the problems to solve carefully such that the size of checkpoint

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

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

on each processor is always the same in every experiment. By keeping the size of checkpoint per processor fixed, we are able to observe the impact of the total number of computing processors on the performance of the checkpointing.

6.1 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 a single failure with FT-MPI. We ran PCG with MPICH-1.2.6 [16], 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 2,000 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 1,000th iteration. The execution times of all tests are reported in Table 2.

Fig. 8 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 sizes of problems. Fig. 8 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.

6.2 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 are no processor failures involved in this set of experiments. At each run, we divided the processors into two classes. The first class of processors is dedicated to perform PCG computation work. The second class of processors is dedicated to perform checkpoint. In Tables 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 2,000 iterations and checkpoint every 100 iterations.

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. Fig. 11 compares the checkpoint overhead (percent) of surviving different numbers of simultaneous processor failures for different size of problems.

Table 4 indicates that as the number of checkpoint processors increases, the time for checkpointing in each test problem increases roughly linearly. 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.

Note that we are performing checkpoint every 100 iterations and run PCG for 2,000 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 [15], [24], [27] is available. We will not address this issue further here.

From Fig. 9, we can see that even if we checkpoint every 25 seconds (Prob #1), the performance overhead of checkpointing to survive five simultaneous processor failures is still within 2 percent of the original program

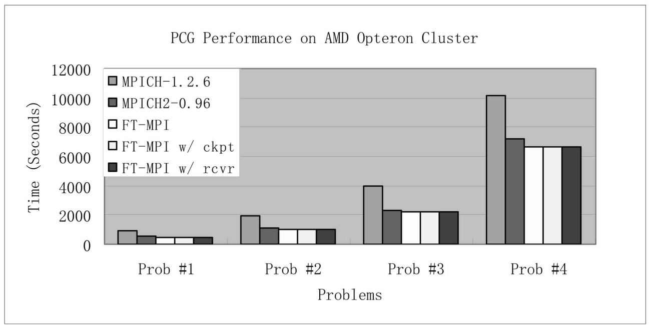

Fig. 8. PCG performance with different MPI implementations.

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

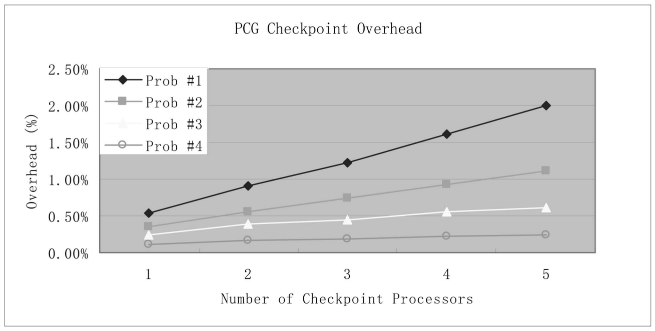

Fig. 9. PCG checkpoint overhead.

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

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 percent.

6.3 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 in the 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 1,000th 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 used for recovery separately for all experiments. Table 6 reports the recovery time in each experiment. Fig. 10 compares the recovery overhead (percent) from different numbers of simultaneous processor failures for different sizes of problems.

From Table 6, we can see that 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 Fig. 10, we can see that the overheads for recovery in all tests are within 1 percent of the program execution time, which is again within the noise margin of the program execution time.

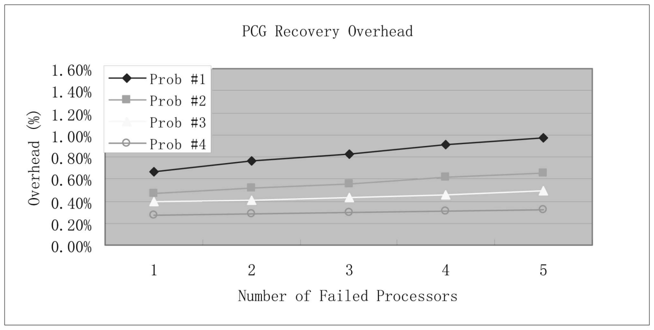

Fig. 10. PCG recovery overhead.

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

6.4 Numerical Impact of Round-Off Errors in Recovery

As discussed in Section 3, our diskless checkpointing schemes are based on floating-point arithmetic encodings, which introduce round-off errors into the checkpointing system. The experiments in this section are designed to measure the numerical impact of the round-off errors in our checkpointing system. All experiment configurations are the same as in the 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 are 0, 1, 2, 3, 4, and 5 simultaneous process failures.

From Table 7, we can see that the norms of the residuals are different for different numbers 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 slightly different recovered data. However, Table 7 also indicates that the residuals with recovery do not have much difference from the residuals without recovery.

6.5 Scalability Test

The fifth set of experiments was designed to test the scalability of the introduced self-healing algorithm (chain-pipeline-based weighted checksum algorithm). Problems to

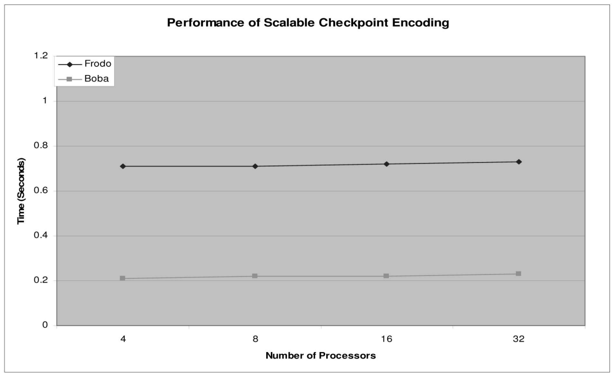

Fig. 11. Performance of pipeline encoding with 8 Mbytes local checkpoint data on each processor.

solve are chosen very carefully so that the size of checkpoint per process is always the same in each experiment. Fig. 11 shows the time to perform one single-failure checkpoint encoding using the chain-pipelined encoding algorithm with 8 Mbytes local checkpoint data on each processor. The experiments are performed on boba and frodo, two Linux clusters, at the University of Tennessee, Knoxville. Fig. 12 reports both the checkpoint and the recovery overhead for tolerating four simultaneous process failures on an IBM RS/6,000 with 176 Winterhawk II thin nodes (each with four 375 MHz Power3-II processors), a parallel platform at Oak Ridge National Laboratory.

Fig. 11 indicates that the time to perform one checkpoint encoding using the chain-pipelined encoding algorithm is almost constant as the number of processors increases. Fig. 12 demonstrates that both the checkpoint overhead and the recovery overhead are very stable as the total number of computing processes increases from 60 to 480. These experimental results are consistent with our theoretical result ($T_{diskless\_pipeline}(k,p) \approx k(\beta + 2\gamma)m.$) in Section 4.1.

7 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 will be. In the PCG example, we only need to checkpoint three vectors and two scalars periodically; therefore, the performance overhead is very low. If the size of checkpoint increases, the overhead will increase proportionally.

The basic weighted checksum scheme implemented in the PCG example has a higher performance overhead than other schemes discussed in Section 5. When an application is executed on a large number of processors, to survive general multiple simultaneous processor failures, the one-dimensional weighted checksum scheme will achieve lower performance overhead than the basic weighted checksum scheme. If processors fail one after another (i.e., no multiple simultaneous processor failures), the neighbor-based schemes can achieve even lower performance overhead. It was shown in [8] that neighbor-based checkpointing is an order of magnitude faster than parity-based checkpointing, but takes twice as much storage overhead.

Diskless checkpointing cannot survive a failure of all processors. Also, to survive a failure occurring during

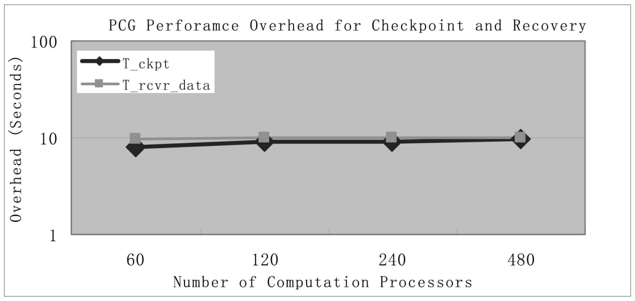

Fig. 12. Scalability of the checkpoint encoding and recovery decoding.

checkpoint or recovery, the storage overhead would double. If an application needs to tolerate these types of failures, a two-level recovery scheme [26], 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 application developers to be involved in the fault tolerance. However, if the fault tolerance schemes are implemented into numerical software packages such as LFC [6], then transparent fault tolerance can also be achieved for application developers using these software tools.

8 Conclusions and Future Work

In this paper, we presented a self-healing framework to build fault-tolerant high-performance computing applications so that they can survive processes failures without aborting themselves. Several checkpoint encoding algorithms were introduced into diskless checkpointing to improve the scalability. The introduced checkpoint encoding algorithms are scalable in the sense that the overhead to survive k failures in p processes does not increase as the number of processes p increases. An example self-healing high-performance computing application was also developed. Experimental results demonstrate that our self-healing approach is able to survive multiple simultaneous processor failures with low-performance overhead and little numerical impact.

In the future, we would like to evaluate our self-healing technique on larger systems and incorporate this technique into more high-performance computing applications.

Acknowledgments

Part of this work was published in the Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), Chicago, Illinois, 15-17 June 2005 [7].

References

[1] N.R. Adiga et al. "An Overview of the BlueGene/L Supercomputer," *Proc. Supercomputing Conf. (SC '02),* pp. 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,* second ed. SIAM, 1994.
[3] F. Berman, G. Fox, and A. Hey, *Grid Computing: Making the Global Infrastructure a Reality.* Wiley, 2003.

[4] Z. Chen and J. Dongarra, "Numerically Stable Real Number Codes Based on Random Matrices," *Proc. Fifth Int'l Conf. Computational Science (ICCS '05),* May 2005.

[5] Z. Chen and J. Dongarra, "Condition Numbers of Gaussian Random Matrices," *SIAM J. Matrix Analysis and Applications,* vol. 27, no. 3, pp. 603-620, 2005.

[6] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche, "Self-Adapting Software for Numerical Linear Algebra and LAPACK for Clusters," *Parallel Computing,* vol. 29, nos. 11/12, pp. 1723-1743, Nov./Dec. 2003.

[7] 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," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '05),* June 2005.

[8] T.C. Chiueh and P. Deng, "Evaluation of Checkpoint Mechanisms for Massively Parallel Machines," *Proc. 26th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS '96),* pp. 370-379, 1996.

[9] J. Dongarra, H. Meuer, and E. Strohmaier, "TOP500 Supercomputer Sites, 24th Edition," *Proc. Supercomputing Conf. (SC '2004),* 2004.

[10] A. Edelman, "Eigenvalues and Condition Numbers of Random Matrices," *SIAM J. Matrix Analysis and Applications,* vol. 9, no. 4, pp. 543-560, 1988.

[11] G.E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," *Proc. Parallel Virtual Machine/Message Passing Interface Conf. (PVM/MPI '00),* pp. 346-353, 2000.

[12] 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," *Proc. Int'l Supercomputer Conf.,* 2004.

[13] 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," *Int'l J. High Performance Computing Applications,* vol. 19, no. 4, pp. 465-477, 2005.

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

[15] E. Gelenbe, "On the Optimum Checkpoint Interval," *J. ACM,* vol. 26, no. 2, pp. 259-270, 1979.

[16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing,* vol. 22, no. 6, pp. 789-828, Sept. 1996.

[17] G.H. Golub and C.F. Van Loan, *Matrix Computations.* The Johns Hopkins Univ. Press, 1989.

[18] Y. Kim, "Fault Tolerant Matrix Operations for Parallel and Distributed Systems," PhD dissertation, Univ. of Tennessee, June 1996.

[19] Message Passing Interface Forum "MPI: A Message Passing Interface Standard," Technical Report ut-cs-94-230, Univ. of Tennessee, 1994.

[20] J.S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Software—Practice & Experience,* vol. 27, no. 9, pp. 995-1012, Sept. 1997.

[21] J.S. Plank, Y. Kim, and J. Dongarra, "Fault-Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 125-138, 1997.

[22] J.S. Plank and K. Li, "Faster Checkpointing with $n+1$ Parity," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS),* pp. 288-297, 1994.

[23] J.S. Plank, K. Li, and M.A. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 10, pp. 972-986, Oct. 1998.

[24] J.S. Plank and M.G. Thomason, "Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems," *J. Parallel and Distributed Computing,* vol. 61, no. 11, pp. 1570-1590, Nov. 2001.

[25] L.M. Silva and J.G. Silva, "An Experimental Study about Diskless Checkpointing," *Proc. EUROMICRO '98 Conf.,* pp. 395-402, 1998.

[26] N.H. Vaidya, "A Case for Two-Level Recovery Schemes," *IEEE Trans. Computers,* vol. 47, no. 6, pp. 656-666, June 1998.

[27] J.W. Young, "A First Order Approximation to the Optimal Checkpoint Interval," *Comm. ACM,* vol. 17, no. 9, pp. 530-531, 1974.

Zizhong Chen received the BS degree in mathematics from Beijing Normal University, People's Republic of China, in 1997, and the MS and PhD 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 the 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.

Jack Dongarra received the Bachelor of Science degree in mathematics from Chicago State University in 1972, the Master of Science degree in computer science from the Illinois Institute of Technology in 1973, and the PhD degree in applied mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as a University Distinguished Professor of Computer Science in the Electrical Engineering and Computer Science Department at the University of Tennessee, has the position of a distinguished research staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), is a Turing fellow in the Computer Science and Mathematics Schools at the University of Manchester, and is an adjunct professor in the Computer Science Department at Rice University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing, and documentation of high-quality mathematical software. He has contributed to the design and implementation of the following open-source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports, and technical memoranda, and he is a coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches. He is a fellow of the AAAS, the ACM, and the IEEE, and a member of the National Academy of Engineering.