Algorithm-Based Recovery for Iterative Methods without Checkpointing

Zizhong Chen
Colorado School of Mines
Golden, CO 80401
zchen@mines.edu

ABSTRACT

In today's high performance computing practice, fail-stop failures are often tolerated by checkpointing. While checkpointing is a very general technique and can often be applied to a wide range of applications, it often introduces a considerable overhead especially when computations reach petascale and beyond. In this paper, we show that, for many iterative methods, if the parallel data partitioning scheme satisfies certain conditions, the iterative methods themselves will maintain enough inherent redundant information for the accurate recovery of the lost data without checkpointing. We analyze the block row data partitioning scheme for sparse matrices and derive a sufficient condition for recovering the critical data without checkpointing. When this sufficient condition is satisfied, neither checkpoint nor rollback is necessary for the recovery. Furthermore, the fault tolerance overhead (time) is zero if no actual failures occur during a program execution. Overhead is introduced only when an actual failure occurs. Experimental results demonstrate that, when it works, the proposed scheme introduces much less overhead than checkpointing on the current world's eighth-fastest supercomputer Kraken.

Categories and Subject Descriptors

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

General Terms

Performance, Reliability

Keywords

Algorithm-Based Recovery, Application Level Fault Tolerance, Checkpointing, Iterative Methods, Rollback Recovery

1. INTRODUCTION

The extreme scale high performance computing (HPC) systems available before the end of this decade are expected to have 100 million to 1 billion CPU cores [5, 33]. Resilience has been widely viewed as a necessity for the exascale HPC applications [19, 20, 21, 33]. Fault tolerance techniques have been identified to be critical to the effective use of these HPC systems [19, 20, 21, 26, 33, 39, 54].

Fail-stop failure [53], where a failed process stops working and all data associated with the failed process are lost, is a very common [35, 39, 54] type of failure in HPC systems such as high end clusters with thousands of processors and computational grids with dynamic computing resources. Fail-stop failures are often tolerated by checkpoint/restart [14, 15, 16, 18, 26, 34, 35, 51, 52, 55, 59]. While checkpoint/restart is a very general technique and can be applied in a wide range of applications, it is sometimes possible to achieve much lower fault tolerance overhead if an *algorithm-based recovery* scheme can be designed to tolerate the failure according to the specific characteristics of an application [8, 24, 25, 28, 29, 30, 31, 32, 44, 46, 47].

Despite there are many other types of failures, this paper focuses on fail-stop failures and presents an algorithm-based recovery scheme for iterative methods. Iterative methods have been widely used to solve linear algebra equations when the co-efficient matrices of the linear systems are sparse [11, 40, 58]. While many iterative software packages such as PETSc [12], Trilinos [43], and HYPRE [36] have been proved to be able to scale to thousands of processors and achieve teraflops level performance, these applications often restart either from the beginning or from a checkpoint if failures occur during the computation.

In this paper, we show that, for many iterative methods, if the parallel data partitioning scheme satisfies certain conditions, the iterative methods themselves will maintain enough inherent redundant information for the accurate recovery of the lost data without checkpointing. Neither checkpoint nor roll-back is necessary. The computation can be restarted from where the failure occurs. Furthermore, the fault tolerance overhead (time) is zero if no actual failures occur in a particular instance of the application execution. Overhead is introduced only when an actual failure occurs.

In cases where the parallel data partitioning scheme does not generate enough inherent redundant information to perform an accurate recovery of all the lost data, it is often possible to slightly modify these iterative methods to incorporate additional redundant information so that all the lost data can still be recovered. Because periodical checkpoint is not necessary during the whole computation and roll-back is not necessary after a failure, the fault tolerance overhead is often lower than checkpointing.

Experimental results demonstrate that the proposed algorithm based recovery schemes introduce much less overhead than checkpointing on the current (as of January 10, 2011) world's eighth-fastest supercomputer Kraken [2] at the National Institute for Computational Sciences.

The rest of the paper is organized as following. Section 2 introduces the background and related work. In Section 3, we analyze the inherent redundant information in parallel sparse matrix-vector multiplication. Section 4 explores the inherent redundant information in the residual relationship. In Section 5, we develop fault tolerant versions of six iterative algorithms including Jacobi, Gauss-Seidel, SOR, SSOR, CG, and BiCGStab methods. In Section 6, we experimentally compare our algorithm-based recovery scheme with checkpointing on kraken.nics.tennessee.edu. Section 7 concludes the paper and discusses the future work.

2. BACKGROUND

Recently, Schroeder and Gibson studied [39, 54] the system logs of 22 HPC systems from 1996 to 2004 in Los Alamos National Laboratory (LANL) and found that the mean-time-to-interrupt (MTTI) for these HPC systems varies from about half a month to less than half a day.

To avoid restarting the computation from the beginning, HPC applications need to be able to tolerate failures. Considerable work has been done to provide fault tolerance capabilities for HPC applications [14, 15, 16, 18, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 44, 46, 47, 51, 52, 55, 59].

Checkpoint/restart has been one of the major approaches to achieve fault tolerance for many years [14, 15, 16, 18, 26, 34, 35, 46, 51, 52, 55, 59]. In HPC field, both LAM/MPI [17] and OPEN MPI [38] implemented a transparent coordinated checkpointing/restarting functionality. Open MPI is able to support both the BLCR checkpoint/restart system [34] and "self" checkpointers. In [7, 18], Cappello et al. implemented MPICH-V which uses both checkpointing and message logging techniques to tolerate failures. MPICH-V provides transparent fault tolerance for MPI programs using four different protocols. By just linking with the MPICH-V library, a non-fault-tolerant application becomes a fault tolerant application.

While many fault tolerance systems have the ability to tolerate the failure of the whole system, it is also desirable to build systems that are able to survive partial failures with reduced overhead. The PVM system [56] developed at the Oak Ridge National Laboratory (ORNL) allows programmers to develop fault tolerant applications that can survive host or task failures. FT-MPI [37], a fault tolerant version of MPI, is also able to provide similar capability to support fault survivable HPC applications. Recently, a collaborative team of researchers (Gupta, Beckman, Park, Lusk, Hargrove, Geist, Panda, Lumsdaine, and Dongarra) developed a coordinated infrastructure for fault-tolerant systems named CIFTS [6, 42]. CIFTS allows system software components to share fault information with each other and aims to give programmers an opportunity to design applications that can adapt to faults in a holistic manner.

When systems are able to survive partial failures, Plank et al. developed the diskless checkpointing technique [50, 51, 52] that removes the I/O bottleneck from checkpointing by storing the checkpoints locally in processor memories and encoding these local checkpoints to dedicated checkpoint processors using Reed-Solomon codes. While diskless checkpointing can not survive the failure of the whole system, it is able to survive partial failures with reduced overhead. Strategies to improve the scalability of diskless checkpointing have been explored in [26, 28].

In order to reduce overhead, fault tolerance can also be addressed from the application level. In [14, 15], Bronevetsky et al. implemented an application-level, coordinated, non-blocking checkpointing system for MPI programs. The system is able to transform non-fault-tolerance C/MPI applications into fault tolerant applications using compiler technologies. Application-level checkpointing schemes are also designed for OpenMP programs in [16].

It is also possible to address the fault tolerance issue from the algorithm level [9, 10, 13, 41, 45, 48, 57]. When failed processes continue working but produce incorrect computing results, Huang and Abraham have developed the algorithm-based fault tolerance (ABFT) technique [45] to detect (and sometimes locate and correct) such miscalculations. Huang and Abraham proved that, for many matrix operations, the checksum relationship in the input checksum matrices is still held in the *final computation results*. Therefore, if the failed process is able to continue their work and finish the computation, the miscalculations can be detected by verifying whether or not the checksum relationship is still held in the final computation results. ABFT was later extended by many researchers (e.g [9, 10, 13, 41]).

When failed processes stop working, recovery often has to be performed in the middle of the computation. In [24, 29, 31, 32], Chen et al. proved that, for the outer product version of the matrix-matrix multiplication, Cholesky factorization, and LU factorization, it is possible to maintain the checksum relationship in the input checksum matrices *throughout the whole computation*. Therefore, whenever a fail-stop failure occurs during the computation, it is possible to use the checksum relationship to recover the lost data on the failed process from where the computation fails. Neither checkpoint nor rollback is necessary for the restart of these computations. This technique is also extended to tolerate multiple simultaneous failures in [22, 23, 30].

In this paper, we call these recovery schemes that directly relay on the specific characteristics of the algorithms as algorithm-based recovery. Algorithm-based recovery for reliable and scalable tree-based overlay networks has been designed by Arnold in [8]. In [49], Ltaief et al. designed an algorithm-based recovery schemes for heat transfer problems. In [44], Hough and Howle explored algorithm-based recovery for several algorithms to solve PDEs. In [47], Langou et al. designed an algorithm-based recovery scheme for iterative methods, called lossy approach, which recovers an **approximation** of the lost data through computing.

While they are challenging to design, algorithm-based recovery schemes often introduce a much less overhead than the more general periodic checkpointing technique. Because no periodic checkpointing is involved during the whole computation and no rollback is necessary after a failure, algorithm-based recovery schemes are often highly scalable and have a very good potential to scale to extreme scale computing and beyond.

3. REDUNDANCIES IN PARALLEL SPARSE MATRIX-VECTOR MULTIPLICATION

Sparse matrix-vector multiplication is one of the most important computational kernels in iterative methods such as iterative linear equation solvers, linear least square problems and eigenvalue solvers [11, 40]. In this section, we present how to recover the lost data by taking advantage of the inherent redundant information from the following sparse matrix-vector multiplication:

$$y = Ax. \tag{1}$$

In a parallel environment, data (such as vectors and matrices) are often partitioned into many pieces and each process often holds only parts of the whole data. Different software packages may choose different data partitioning schemes. The way in which a software package partitions its data often affects both the performance of the software and its compatibility with other existing codes.

In this paper, without loss of generality, we will demonstrate our fault tolerant idea using the default data partitioning scheme in the widely used iterative methods software package PETSc [12].

In PETSc, by default, vectors and matrices are partitioned into contiguous block of rows. In this block row data partitioning scheme, when p processes are used to perform the computation, the dense vector x and y are partitioned into p contiguous blocks (sub-vectors), x_i and y_i, respectively, where x_i and y_i are assigned to the i^{th} process, and $i = 1, 2, \ldots, p$. Accordingly, the sparse matrix A is partitioned as

$$A = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1p} \\ A_{21} & A_{22} & \ldots & A_{2p} \\ \vdots & \vdots & \ldots & \vdots \\ A_{p1} & A_{p2} & \ldots & A_{pp} \end{pmatrix},$$

where $A_{i1}, A_{i2}, \ldots, A_{ip}$ are assigned to the i^{th} process.

In this data partitioning scheme, the sparse matrix-vector multiplication (1) can be rewrite as

$$\begin{cases} y_1 & = A_{11}x_1 + \ldots + A_{1p}x_p \\ \vdots & \\ y_p & = A_{p1}x_1 + \ldots + A_{pp}x_p. \end{cases} \tag{2}$$

To simplify the discussion, let's first consider a simple parallel sparse matrix-vector multiplication algorithm without any optimization on communication.

In the un-optimized algorithm, in order to calculate $y_i = A_{i1}x_1 + \ldots + A_{ip}x_p$ on the i^{th} process, the sub-vector x_j, where $j \neq i$, has to to be sent from the j^{th} process to the i^{th} process. This communication duplicates the sub-vector x_j from the j^{th} process to the i^{th} process.

Therefore, in this un-optimized parallel sparse matrix-vector multiplication, it is possible to maintain multiple copies of the same sub-vector of x in different processes without additional time overhead. When a sub-vector of x on one process is lost, it is possible to recover that lost sub-vector of x by getting it from another process.

Note that, in the above analysis, the sparse matrix-vector multiplication algorithm used is a simple un-optimized algorithm. However, in practice, in order to achieve high performance, the sparse matrix-vector multiplication algorithm are often optimized according the non-zero structure of sparse matrix A.

Now, let's consider the sparse matrix-vector multiplication algorithm that has been optimized according the non-zero structure of the sparse matrix A. Assume when the i^{th} process is calculating $y_i = A_{i1}x_1 + \ldots + A_{ip}x_p$, only the elements (of the sub-vector x_j) that will multiply with a non-zero element in A_{ij}, where $j \neq i$, are sent from j^{th} process to the i^{th} process.

In what follows, we derive a sufficient condition for recovering x without checkpointing. Let S_j denote the set of all elements of the sub-vector x_j, and S_{ji} denote the set of the elements that have been sent from the j^{th} process to the i^{th} process, where $i \neq j$. Let

$$R_j = \left(\bigcup_{i=1}^{j-1} S_{ji}\right) \bigcup \left(\bigcup_{i=j+1}^{p} S_{ji}\right). \tag{3}$$

and

$$R_j^c = S_j - R_j. \tag{4}$$

THEOREM 1. *If the j^{th} process fails and the sub-vector x_j is lost, then x_j can be accurately recovered from the rest of the processes when the following condition is satisfied*

$$R_j = S_j. \tag{5}$$

PROOF. S_{ji} contains elements of x_j that have been sent from the j^{th} process to the i^{th} process. Therefore, we can maintain a copy of S_{ji} on the i^{th} process for all i. When the j^{th} process fails, all other processes are still alive. Therefore, each S_{ji}, where $i \neq j$, is still available. From (3), R_j can be accurately reconstructed using available S_{ji} on the surviving processes. Since $R_j = S_j$, therefore, x_j can be accurately reconstructed. □

Let

$$B = \begin{pmatrix} A_{21} & A_{12} & \ldots & A_{1p} \\ A_{31} & A_{32} & \ldots & A_{2p} \\ \vdots & \vdots & \ldots & \vdots \\ A_{p1} & A_{p2} & \ldots & A_{(p-1)p} \end{pmatrix}, \text{and } B_j = \begin{pmatrix} A_{1j} \\ \vdots \\ A_{(j-1)j} \\ A_{(j+1)j} \\ \vdots \\ A_{pj} \end{pmatrix}. \tag{6}$$

Let $B_{j,k}$ denote the k^{th} column of the matrix B_j. Assume there are m_j columns in the matrix B_j, then x_j is a vector of length m_j. Assume $x_j = (x_{j1}, x_{j2}, \ldots, x_{jm_j})^T$, then $S_j = \{x_{j1}, x_{j2}, \ldots, x_{jm_j}\}$.

THEOREM 2. $R_j = S_j$ *if and only if* $B_{j,k}^T B_{j,k} \neq 0$ *for all* $k = 1, 2, \ldots, m_j$.

PROOF. If $R_j = S_j$, then, for any $x_{jk} \in S_j$, there exists a i_k such that $x_{jk} \in S_{ji_k}$. Note that S_{ji_k} is the set of elements the i_k^{th} process need when performing $A_{i_kj}x_j$. $x_{jk} \in S_{ji_k}$ indicates that at least one element of the k^{th} column of A_{i_kj} is non-zero. Note that $S_j = \{x_{j1}, x_{j2}, \ldots, x_{jm_j}\}$ and $B_j = (A_{1j}, A_{2j}, \ldots, A_{(j-1)j}, A_{(j+1)j} \ldots, A_{pj})^T$, therefore, for any column of B_j, at least one element of the column is nonzero. Therefore, $B_{j,k}^T B_{j,k} \neq 0$ for all $k = 1, 2, \ldots, m_j$.

If $B_{j,k}^T B_{j,k} \neq 0$ for all $k = 1, 2, \ldots, m_j$, then for any column of B_j, at least one element of the column is nonzero.

Therefore, all elements of x_j have to be sent to another process during the matrix-vector multiplication. Hence, $S_j \subseteq R_j$. Note that $R_j \subseteq S_j$, therefore, $R_j = S_j$. $\square$

From *Theorem 1* and *Theorem 2*, it follows that, if the j^{th} process fails, the lost sub-vector x_j can be accurately recovered from the rest of the processes if no column of B_j equals to the zero vector.

THEOREM 3. *If no column of B equals to the zero vector, then the parallel sparse matrix vector multiplication itself contains enough information to accurately recover the lost part of x after any single process failure.*

PROOF. If no column of B equals to the zero vector, then $B_{j,k}^T B_{j,k} \neq 0$ for all $k = 1, 2, \ldots, m_j$, and $j = 1, 2, \ldots, p$. Therefore, according to *Theorem 2*, for any $j = 1, 2, \ldots, p$, $R_j = S_j$. According to *Theorem 1*, for any $j = 1, 2, \ldots, p$, if the j^{th} process fails, the lost sub-vector x_j can be accurately recovered from the rest of the processes. Hence, the lost part of x can be accurately recover after any single process failure. $\square$

If there are some columns of B that equal to the zero vector, then there exists at least one j such that $R_j \neq S_j$. Then $R_j^c = S_j - R_j \neq \emptyset$. Notice that for all $i \neq j$, $R_j^c \cap S_{ji} = \emptyset$. Therefore, the elements in R_j^c are actually these elements of x_j that are not sent out to any other process during the matrix-vector multiplication.

THEOREM 4. *For any $j = 1, 2, \ldots, p$,*

$$R_j^c = \left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\}. \quad (7)$$

PROOF. For any $x_{jt} \in R_j^c$, x_{jt} is not sent out from the j^{th} process to any other process. Therefore, all the t^{th} column of B_j is zero (otherwise, x_{jt} has to be sent out). Hence, $B_{j,t}^T B_{j,t} = 0$. Hence, $x_{jt} \in \left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\}$. Therefore, $R_j^c \subseteq \left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\}$.

For any $x_{jk} \in \left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\}$, x_{jk} will not be used by any other process (except the j^{th} process). Therefore $x_{jk} \notin S_{ji}$ for all $i = 1, 2, \ldots, (j-1), (j+1), \ldots, p$. Hence $x_{jk} \in R_j^c$. Therefore, $\left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\} \subseteq R_j^c$.

Therefore, $R_j^c = \left\{ x_{jk} \mid B_{j,k}^T B_{j,k} = 0 \right\}$. $\square$

The structure of the matrix B defined in (6) depends both on the structure of the original matrix A and on the partition scheme for A. In most iterative methods, the matrix A involved is often non-singular. Therefore, it is often true that no column of A equals to the zero vector. The partition of the matrix A often depends on the partition of the vector x. In the parallel sparse matrix vector multiplication, *wherever the partition does not negatively affect the performance*, we can choose partitions that produce none (or as few as possible) zero columns in B. By minimizing number of the zero columns in B, the amount of data that can be recovered through the existing redundant information is maximized.

When the matrix A can not be partitioned into such a way that no column of B equals to the zero vector, then the parallel sparse matrix vector multiplication algorithm can be slightly modified to incorporate additional redundant information so that all the lost data can be accurately recovered. The simplest modification that is able to achieve this goal is to let the j^{th} process also send R_j^c to the $((j+1)\%p)^{th}$ process while sending $S_{j\ ((j+1)\%p)}$.

Note that the matrix A is often a constant in iterative methods, hence it is possible to avoid the *periodic checkpointing* of A either through recovering using the same way as it is initially constructed or through saving it into disks at the beginning. After the lost part of x is recovered and the lost part of A is reconstructed, the lost part of the result vector y can be recovered by recomputing through the relationship $y = Ax$.

In what follows, without loss of generality, we will assume that, after any single process failure, the lost part of x, y, and A can all be accurately recover from the inherent redundant information in the parallel sparse matrix vector multiplication.

4. REDUNDANCIES IN THE RESIDUAL RELATIONSHIP

When iterative methods are used to solve the linear equation $Ax = b$, for many algorithms (e.g. Conjugate Gradient method and Bi-Conjugate Gradient Stabilized method), the residual vector $r^{(i)}$ for the i^{th} approximate solution $x^{(i)}$ is also computed at each iteration of the algorithm. According to the definition of the residual vector, $r^{(i)}$, $x^{(i)}$ and A satisfy the following residual relationship:

$$r^{(i)} = b - Ax^{(i)}. \quad (8)$$

In a parallel implementation, assuming the data are partitioned among processes using the same way as in Section 3, the residual relationship (8) becomes

$$\begin{cases} r_1^{(i)} & = b_1 - (A_{11}x_1^{(i)} + \ldots + A_{1p}x_p^{(i)}) \\ & \vdots \\ r_p^{(i)} & = b_p - (A_{p1}x_1^{(i)} + \ldots + A_{pp}x_p^{(i)}). \end{cases} \quad (9)$$

Note that, no mater how the residual vector $r^{(i)}$ is computed in the algorithm, the residual relationship (9) is always true.

THEOREM 5. *If the data partition scheme satisfies the condition that B_j defined in Section 3 formula (6) is full rank for all j, then the residual relationship itself contains enough information to accurately recover the lost part of both $x^{(i)}$ and $r^{(i)}$ after any single process failure.*

PROOF. For any j, when the j^{th} processor fails, all data on the j^{th} processor (i.e. i, $r_j^{(i)}$, $x_j^{(i)}, b_j, A_{j1}, \ldots, A_{jp}$) are lost. The loop variable i can be recovered by getting it from any surviving process. Since $b_j, A_{j1}, \ldots, A_{jp}$ is constant and do not change between iterations, they can be reconstructed using the same way as they are created at the beginning of the computation. For $r_j^{(i)}$ and $x^{(i)_j}$, they satisfy the above residual relationship (9). After $b_j, A_{j1}, \ldots, A_{jp}$ are reconstructed, $r_j^{(i)}$ and $x_j^{(i)}$ are the only unknowns in the residual relationship (9).

Let

$$\widetilde{B} = \begin{pmatrix} A_{11} & \dots & A_{1\,j-1} & A_{1\,j+1} & \dots & A_{1p} \\ \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ A_{j-1\,1} & \dots & A_{j-1\,j-1} & A_{j-1\,j+1} & \dots & A_{j-1\,p} \\ A_{j+1\,1} & \dots & A_{j+1\,j-1} & A_{j+1\,j+1} & \dots & A_{j+1\,p} \\ \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ A_{p1} & \dots & A_{p\,j-1} & A_{p\,j+1} & \dots & A_{pp} \end{pmatrix},$$

$$\widetilde{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_{j-1} \\ b_{j+1} \\ \vdots \\ b_p \end{pmatrix}, \; \widetilde{r}^{(i)} = \begin{pmatrix} r_1^{(i)} \\ \vdots \\ r_{j-1}^{(i)} \\ r_{j+1}^{(i)} \\ \vdots \\ r_p^{(i)} \end{pmatrix}, \text{ and } \widetilde{x}^{(i)} = \begin{pmatrix} x_1^{(i)} \\ \vdots \\ x_{j-1}^{(i)} \\ x_{j+1}^{(i)} \\ \vdots \\ x_p^{(i)} \end{pmatrix}.$$

After deleting the j^{th} equation from (9), the relationship (9) becomes

$$\widetilde{r}^{(i)} = \widetilde{b} - \widetilde{B}\widetilde{x}^{(i)} - B_j x_j^{(i)}. \tag{10}$$

Note that $x_j^{(i)}$ is the only unknown in (10). Therefore, when B_j is full rank, $x_j^{(i)}$ can be recovered by solving (10) with $x_j^{(i)}$ as unknown.

After $x_j^{(i)}$ is recovered, $r_j^{(i)}$ can be calculated by

$$r_j^{(i)} \quad = b_j - (A_{j1}x_1^{(i)} + \ldots + A_{jp}x_p^{(i)}). \tag{11}$$

□

Like in Section 3, whether B_j is full rank or not depends both the structure of the matrix A and the parallel data partition scheme.

When the matrix A can not be partitioned into such a way that B_j is full rank for all j, an approximation of $x_j^{(i)}$ can be reconstructed using the lossy approach from [47]. After $x_j^{(i)}$ is recovered, $r_j^{(i)}$ can be calculated from (11).

In what follows, without loss of generality, we will assume that, after any single process failure, the lost part of $x^{(i)}$ and $r^{(i)}$ can be recovered from the inherent redundant information in the residual relationship.

5. FAULT TOLERANT ITERATIVE METHODS

In this section, we show how to reconstruct the lost data on the failed process in selected iterative methods using the inherent redundancies discussed in Section 3 and 4. Although these inherent redundancies can be used in many other places, in this paper, we will focus on parallel iterative methods for solving the linear equation

$$Ax = b, \tag{12}$$

where A is a large sparse matrix. Assume the parallel data partition scheme in Section 3 is used.

5.1 Stationary Iterative Methods

In this subsection, we show how to recover the lost data in four representative stationary iterative methods: (1). the *Jacobi method*; (2). the *Gauss-Seidel method*; (3). the *Successive Overrelaxation (SOR) method*; and (4). the *Symmetric Successive Overrelaxation (SSOR) method.*

These four stationary iterative methods can all be expressed in the simple form [11]

$$x^{(k)} = Cx^{(k-1)} + c, \tag{13}$$

where neither the matrix C nor the vector c changes during the iteration.

For the stationary iterative methods represented in (13), when a process failure occurs during the execution of the i^{th} iteration, the computation can be restarted from the same iteration (i.e. the i^{th} iteration) if, after the failure, the following four values can be recovered: i, C, c, and $x^{(i-1)}$.

During the execution of the i^{th} iteration, if processes do not discard the messages they received in the last iteration and the update of the approximate solution vector x is not in place (i.e. $x^{(k-1)}$ is not overwritten in the i^{th} iteration), then a consistent $x^{(k-1)}$ with enough redundancy to recover itself is automatically maintained during the computation.

In the i^{th} iteration, when there is a process failure, the four values i, C, c, and $x^{(i-1)}$ that are necessary for the restart of the i^{th} iteration can be recovered from the following steps:

1. i: The iteration loop variable i can be recovered by getting it from any surviving neighbor process.

2. C: The constant matrix C can be reconstructed using the same way as they were constructed originally.

3. c: The constant vector c can be reconstructed using the same way as they were constructed originally.

4. $x^{(i-1)}$: The lost part of the vector $x^{(i-1)}$ can be accurately recovered using approaches from Section 3.

Now that i, C, c, and $x^{(i-1)}$ have all been successfully recovered, the algorithm can be restarted from the computation of $x^{(k)}$, which is the very iteration where the process failure occurs.

5.2 The Preconditioned Conjugate Gradient Method

The *Preconditioned Conjugate Gradient (CG)* method is one of the most commonly used iterative methods to solve the sparse linear system $Ax = b$ when the coefficient matrix A is symmetric positive definite. The method computes successive approximations to the solution, residuals corresponding to the approximate solutions, and search directions used to update both the approximate solutions and the residuals. The length of these vector sequences can be large, but only a small number of the vectors need to be maintained in memory. It involves one sparse matrix vector multiplication three vector updates, and two vector inner products in every iteration of the method. For more details of the algorithm, we refer readers to [11].

When a process failure occurs during the execution of the i^{th} iteration, if the following seven values can be recovered: i, A, M, $r^{(i-1)}$, ρ_{i-2}, $p^{(i-1)}$, and $x^{(i-1)}$, then the computation can be restarted from the i^{th} iteration.

In the i^{th} iteration, if processes do not discard: (1). the messages they received in the last two iterations; (2). the vector $p^{(i-1)}$ and $p^{(i-2)}$; and (3). the scalar β_{i-2}, then all seven values that are necessary for the restart of the i^{th} iteration can be recovered from the following steps:

```
Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
    if ( (recover) && ($i > 1$) )
      recover: $i$, $A$, $M$, $r^{(i-1)}$,
               $\rho_{i-2}$, $p^{(i-1)}$, and $x^{(i-1)}$.
    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 1: Fault tolerant preconditioned conjugate gradient algorithm

1. i: The iteration loop variable i can be recovered by getting it from any surviving neighbor process.

2. A: The constant matrix A can be reconstructed either through using the same way as it is initially constructed or through saving it into disks at the beginning.

3. M: The constant matrix M can be reconstructed either through using the same way as it is initially constructed or through saving it into disks at the beginning.

4. ρ_{i-2}: ρ_{i-2} can be recovered by getting it from any surviving neighbor process.

5. $p^{(i-1)}$: The lost part of the vector $p^{(i-1)}$ can be accurately recovered using the approach from Section 3.

6. $r^{(i-1)}$: In order to recover $r^{(i-1)}$, we first recover $r^{(i-2)}$. The scalar β_{i-2} can be recovered by getting it from any surviving neighbor process. The lost part of the vector $p^{(i-2)}$ can be accurately recovered using the approach from Section 3. The lost part of $z^{(i-2)}$ can be recovered from the relationship $p^{(i-1)} = z^{(i-2)} + \beta_{i-2} p^{(i-2)}$, where $p^{(i-1)}$ has been recovered in step 5, β_{i-2} and $p^{(i-2)}$ have just been recovered in this step. The lost part of $r^{(i-2)}$ can be recovered from the relationship $Mz^{(i-2)} = r^{(i-2)}$, where $z^{(i-2)}$ has just been recovered in this step. $q^{(i-1)}$ can be recalculated from $q^{(i-1)} = Ap^{(i-1)}$. α_{i-1} can be re-computed from $\alpha_{i-1} = \rho_{i-2}/p^{(i-1)^T} q^{(i-1)}$. Finally, $r^{(i-1)}$ can be re-computed by $r^{(i-1)} = r^{(i-2)} - \alpha_{i-1} q^{(i-1)}$.

7. $x^{(i-1)}$: Note that, in the algorithm, the residual vector $r^{(i-1)}$ satisfies $r^{(i-1)} = b - Ax^{(i-1)}$. $r^{(i-1)}$ have been recovered in step 6. The right hand side vector b can be reconstructed using the same way as they are constructed originally. The constant matrix A has be reconstructed in step 2. Therefore, the lost part of $x^{(i-1)}$ can be recovered by solving the equation $r^{(i-1)} = b - Ax^{(i-1)}$ with the **lost part** of the $x^{(i-1)}$ as unknown. Because A is nonsingular, therefore all columns of A are linear independent. Hence, the columns corresponding to the lost part of $x^{(i-1)}$ is also linear independent. Therefore, the equation will have a unique solution, which is the lost part of the $x^{(i-1)}$.

By now, the seven values i, A, M, $r^{(i-1)}$, ρ_{i-2}, $p^{(i-1)}$, and $x^{(i-1)}$ have all been successfully recovered, therefore, the computation can now be restart from the i^{th} iteration. Figure 3 shows the fault tolerant version of the PCG algorithm.

5.3 The Preconditioned Bi-Conjugate Gradient Stabilized Method

The *Preconditioned Bi-Conjugate Gradient Stabilized (Bi-CGSTAB)* method is often used to solve the sparse linear system $Ax = b$ when the coefficient matrix A is nonsymmetric. The method avoids the often irregular convergence patterns of the *Conjugate Gradient Squared (CGS) method*. Bi-CGSTAB needs two matrix-vector multiplication and four vector inner products. For more details of the algorithm, we refer the reader to [11].

```
Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r} = r^{(0)}$
for $i = 1, 2, \ldots$
    if ( (recover) && ($i > 1$) )
      recover: $i$, $A$, $M$, $\tilde{r}$, $r^{(i-1)}$, $\rho_{i-2}$, $\omega_{i-1}$,
               $p^{(i-1)}$, $v^{(i-1)}$, $\alpha_{i-1}$, and $x^{(i-1)}$.
    $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
    if $\rho_{i-1} = 0$, then the method fails
    if $i = 1$
       $p^{(i)} = r^{(i-1)}$
    else
       $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
       $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1} v^{(i-1)})$
    endif
    solve $M\hat{p} = p^{(i)}$
    $v^{(i)} = A\hat{p}$
    $\alpha_i = \rho_{i-1}/\tilde{r}^T v^{(i)}$
    $s = r^{(i-1)} - \alpha_i v^{(i)}$
    check norm of s;
    if small enough:  set $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p}$; stop
    solve $M\hat{s} = s$
    $t = A\hat{s}$
    $\omega_i = t^T s / t^T t$
    $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
    $r^{(i)} = s - \omega_i t$
    check convergence; continue if necessary
end
```

Figure 2: Fault tolerant preconditioned Bi-Conjugate Gradient Stabilized algorithm

If a process failure occurs in the i^{th} iteration, in order to restart the computation from the same iteration, the following eleven values have to be recovered: i, A, M, $\tilde{r}$, $r^{(i-1)}$, ρ_{i-2}, ω_{i-1}, $p^{(i-1)}$, $v^{(i-1)}$, α_{i-1}, and $x^{(i-1)}$.

During the execution of the i^{th} iteration, if processes do not discard: (1). the messages they received in the last iteration; and (2). the vector $\hat{p}$ and $\hat{s}$ in the last iteration, then the eleven values that are necessary for the restart of the i^{th} iteration can all be recovered from the following steps:

1. i: The iteration loop variable i can be recovered by getting it from any surviving neighbor process.
2. A: The constant matrix A can be reconstructed either through using the same way as it is initially constructed or through saving it into disks at the beginning.
3. M: The constant matrix M can be reconstructed either through using the same way as it is initially constructed or through saving it into disks at the beginning.
4. $\tilde{r}$: The constant vector $\tilde{r}$ can be reconstructed using the same way as they were constructed originally.
5. $r^{(i-1)}$: In order to recover $r^{(i-1)}$, we first recover $\hat{s}$ for the $(i-1)^{th}$ iteration using the approach discussed in Section 3. Then, the s for the $(i-1)^{th}$ iteration can be recomputed from the relationship $M\hat{s} = s$. The t for the $(i-1)^{th}$ iteration can be recomputed from the relationship $t = A\hat{s}$. ω_{i-1} can be recomputed from the relationship $\omega_{i-1} = t^T s / t^T t$. Finally, $r^{(i-1)}$ can be recovered from the relationship $r^{(i-1)} = s - \omega_{i-1} t$.
6. ρ_{i-2}: ρ_{i-2} can be recovered by getting it from any surviving neighbor process.
7. ω_{i-1}: ω_{i-1} have been recovered during the process of recovering $r^{(i-1)}$ in step 5.
8. $p^{(i-1)}$: In order to recover $p^{(i-1)}$, we first recover $\hat{p}$ for the $(i-1)^{th}$ iteration using the approach discussed in Section 3. Then, the $p^{(i-1)}$ can be recomputed from the relationship $M\hat{p} = p^{(i-1)}$.
9. $v^{(i-1)}$: $v^{(i-1)}$ can be recomputed from the relationship $v^{(i-1)} = A\hat{p}$, where the $\hat{p}$ is the $\hat{p}$ for the $(i-1)^{th}$ iteration recovered in step 8.
10. α_{i-1}: α_{i-1} can be recomputed from the relationship $\alpha_{i-1} = \rho_{i-2} / \tilde{r}^T v^{(i-1)}$, where ρ_{i-2} has been recovered in step 6, $\tilde{r}$ has been recovered in step 4, $v^{(i-1)}$ has been recovered in step 9.
11. $x^{(i-1)}$: Note that, in the algorithm, the residual vector $r^{(i-1)}$ satisfies $r^{(i-1)} = b - Ax^{(i-1)}$. $r^{(i-1)}$ have been recovered in step 5. The right hand side vector b can be reconstructed using the same way as they are constructed originally. The constant matrix A has be reconstructed in step 2. Therefore, the lost part of $x^{(i-1)}$ can be recovered by solving the equation $r^{(i-1)} = b - Ax^{(i-1)}$ with the **lost part** of the $x^{(i-1)}$ as unknown. Because A is nonsingular, therefore all columns of A are linear independent. Hence, the columns corresponding to the lost part of $x^{(i-1)}$ are also linear independent. Therefore, the equation has a unique solution, which is the lost part of the $x^{(i-1)}$.

After the eleven values i, A, M, $\tilde{r}$ $r^{(i-1)}$, ρ_{i-2}, α_{i-1}, ω_{i-1}, $p^{(i-1)}$, $v^{(i-1)}$, and $x^{(i-1)}$ have all been recovered, the computation can now be restart from the i^{th} iteration. Figure 4 shows the fault tolerant version of the Bi-CGSTAB algorithm.

6. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed algorithm-based recovery scheme and compare it with checkpointing. For the demonstration purpose, consider solving the following sparse linear system arising from discretizing a 3D Poisson's equation using finite-difference-method:

$$A_{n^3\times n^3} \;\; u_{n^3\times 1} \;\; = \;\; b_{n^3\times 1}, \tag{14}$$

where

$$A_{n^3\times n^3} = \begin{pmatrix} M_{n^2\times n^2} & I_{n^2\times n^2} & & & \\ I_{n^2\times n^2} & M_{n^2\times n^2} & I_{n^2\times n^2} & & \\ & \ddots & \ddots & \ddots & \\ & & I_{n^2\times n^2} & M_{n^2\times n^2} & I_{n^2\times n^2} \\ & & & I_{n^2\times n^2} & M_{n^2\times n^2} \end{pmatrix},$$

$$M_{n^2\times n^2} = \begin{pmatrix} T_{n\times n} & I_{n\times n} & & & \\ I_{n\times n} & T_{n\times n} & I_{n\times n} & & \\ & \ddots & \ddots & \ddots & \\ & & I_{n\times n} & T_{n\times n} & I_{n\times n} \\ & & & I_{n\times n} & T_{n\times n} \end{pmatrix},$$

$$T_{n\times n} = \begin{pmatrix} -6 & 1 & & & \\ 1 & -6 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -6 & 1 \\ & & & 1 & -6 \end{pmatrix}.$$

When the block row data partitioning scheme is used to partition the sparse matrix into n MPI processes, the corresponding matrix B in the formula (6) of Section 3 becomes

$$B = \begin{pmatrix} I_{n^2\times n^2} & I_{n^2\times n^2} & & & \\ & I_{n^2\times n^2} & I_{n^2\times n^2} & & \\ & & \ddots & \ddots & \\ & & & I_{n^2\times n^2} & I_{n^2\times n^2} \end{pmatrix}. \tag{15}$$

Note that no column of B in equals to the zero vector and B_j (i.e. the j^{th} column block of B) is full rank for all j, therefore, this partition satisfies the condition in Theorem 3 of Section 3 for recovery without checkpointing.

Note that all stationary iterative methods are very similar. Therefore, without loss of generality, in this paper, we will limit our fault tolerant experiments for stationary iterative methods to the Jacobi method.

For non-stationary iterative methods, note that the sparse matrix $A_{n^3\times n^3}$ in equation (14) is symmetric and positive definite, therefore, the equation (14) can be used to test our fault tolerant conjugate gradient method.

6.1 Algorithm-Based Recovery vs. Parallel-I/O-Based Checkpointing

In this subsection, we compare the overhead of the proposed algorithm-based recovery with the stable-storage-based checkpointing on the supercomputing cluster ra.mines.edu [3] at Colorado School of Mines.

Ra.mines.edu is a supercomputing cluster dedicated to energy sciences at Colorado School of Mines. The peak performance of the cluster is 23 teraflops. The achieved LINPACK benchmark [1] performance is about 17 teraflops. The parallel file system used on ra.mines.edu is Lustre [4]. The MPI implementation on the cluster is Open MPI version 1.3.4.

Table 1: Fault Tolerant Jacobi on Ra.

num. of proc	ckp size per proc (bytes)	time per ckp (secs)	time per ckp rcv (secs)	rollback per rcv (secs)	time per alg rcv (secs)
100	80004	0.60	0.49	298	0.0026
200	320004	1.02	0.89	297	0.0039
300	720004	1.51	1.04	301	0.0055
400	1280004	2.12	1.18	300	0.0073
500	2000004	5.43	1.26	299	0.0081
600	2880004	7.60	1.48	301	0.0088
700	3920004	11.1	1.51	300	0.0102
800	5120004	20.6	1.56	297	0.0112
900	6480004	27.1	1.92	302	0.0125
1000	8000004	35.2	1.99	301	0.0135

Table 2: Fault Tolerant CG on Ra.

num. of proc	ckp size per proc (bytes)	time per ckp (secs)	time per ckp rcv (secs)	rollback per rcv (secs)	time per alg rcv (secs)
100	240012	1.89	1.50	301	0.0074
200	960012	2.97	2.66	302	0.0104
300	2160012	4.59	3.11	296	0.0166
400	3840012	6.24	3.54	300	0.0220
500	6000012	16.2	3.78	299	0.0248
600	8640012	22.8	4.45	298	0.0269
700	11760012	33.3	4.66	296	0.0305
800	15360012	61.8	4.91	298	0.0346
900	19440012	81.3	5.92	296	0.0380
1000	24000012	106	6.09	295	0.0401

For stable-storage-based checkpointing, checkpoints are written in parallel by `MPI_File_write_at_all()`. The checkpoint frequency is six checkpoints per hour. During each hour, one recovery is simulated approximately in the middle of two consecutive checkpoints by reconstructing all the data necessary to continue the computation from the stable storage using the parallel I/O function `MPI_File_read_at_all()`. The recovery frequency is one recovery per hour.

For the algorithm-based recovery scheme, a single process failure recovery is simulated by reconstructing all the data necessary to continue the computation on the MPI process rank 0 using algorithms from Section 5. The recovery frequency is also one recovery per hour.

Table 1 and 2 report the size of checkpoint per process, the time for each checkpoint, the time for each checkpoint-based recovery, the wasted computation time due to rollback, and the time for each algorithm-based recovery. Figure 3 and 4 compare the failure-free overhead. Figure 5 and 6 compare the total fault tolerance overhead with recovery.

Figure 3 and 4 indicate that, when no failure occurs during the computation, the algorithm-based recovery scheme introduces much less overhead than the stable-storage-based checkpointing. This is because, for the algorithm-based recovery scheme, neither additional computations nor additional communications are introduced. But, for the stable-

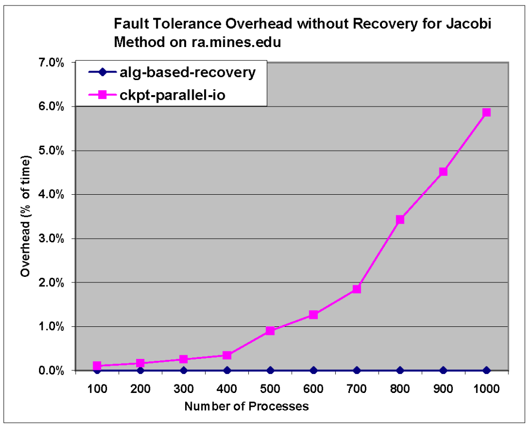

Figure 3: Fault tolerance overhead for Jacobi method without failures: checkpointing with parallel I/O vs. algorithm-based recovery.

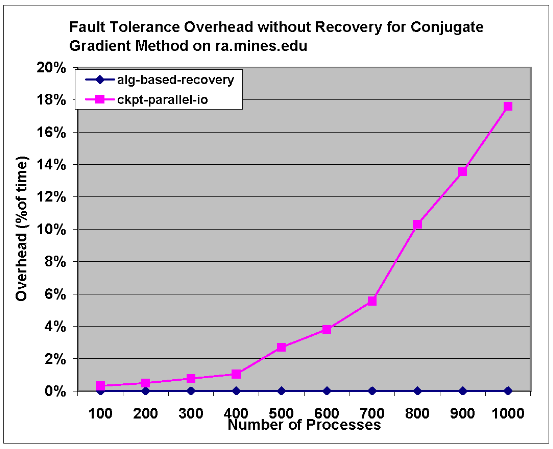

Figure 4: Fault tolerance overhead for Conjugate Gradient method without failures: checkpointing with parallel I/O vs. algorithm-based recovery.

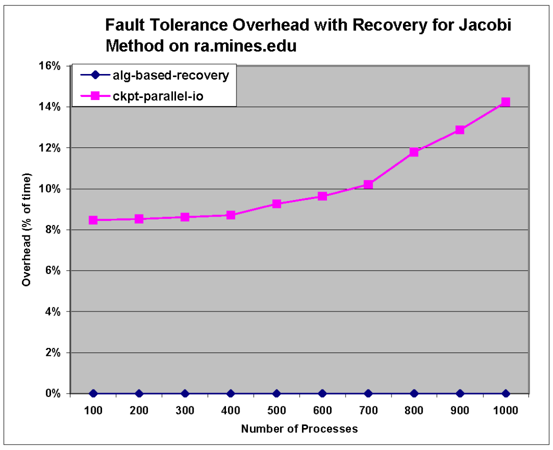

Figure 5: Fault tolerance overhead for Jacobi method with failures: checkpointing with parallel I/O vs. algorithm-based recovery.

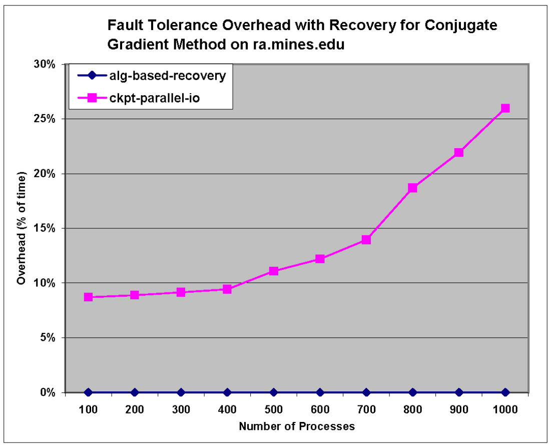

Figure 6: Fault tolerance overhead for Conjugate Gradient method with failures: checkpointing with parallel I/O vs. algorithm-based recovery.

storage-based checkpointing, it takes a fair amount time to write the checkpointing data into the stable storage.

Figure 5 and 6 demonstrate that, when there is a recovery, the algorithm-based recovery also introduces much less overhead than the stable-storage-based checkpointing. The main overheads for the stable-storage-based checkpointing scheme are: (1). the time for writting checkpoints periodically; (2). the time for re-computing due to rollback, which is approximately 5 minutes per rollback. Note that, in our experiments, the checkpoint frequency is six checkpoints per hour. Hence, a rollback of 5 minutes is the average rollback time after a failure. But, for the algorithm-based recovery, the computation is restarted from where the failure occurs. No rollback is involved. Therefore, overall, it introduces much less overhead than the stable-storage-based checkpointing.

Table 3: Fault Tolerant Jacobi on Kraken.

num. of proc	ckp size per proc (bytes)	time per ckp (secs)	time per ckp rcv (secs)	rollback per rcv (secs)	time per alg rcv (secs)
1000	8000004	0.07	0.08	300	0.006
2000	32000004	0.30	0.33	299	0.019
3000	72000004	0.68	0.70	301	0.048
4000	128000004	1.26	1.31	300	0.085
5000	200000004	1.97	2.05	299	0.132

6.2 Algorithm-Based Recovery vs. Diskless Checkpointing

In this section, we compare the overhead of the proposed algorithm-based recovery with diskless checkpointing on the current (as of January 10, 2011) world's eighth-fastest [1] supercomputer Kraken [2] at the National Institute for Computational Sciences (NICS).

For diskless checkpointing experiments, the checkpoints are first stored locally in memory, and then an XOR of these local checkpoints are computed using `MPI_Reduce()` and saved into a dedicated checkpoint process. The check-

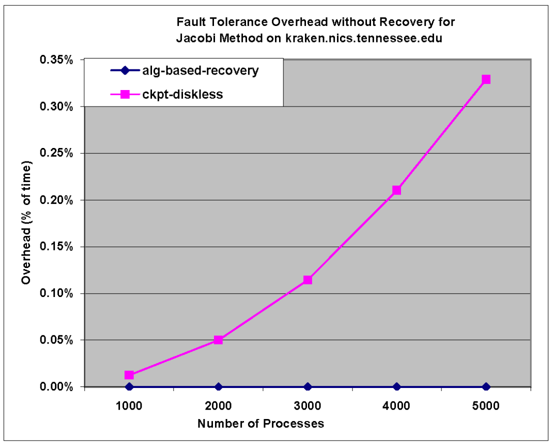

Figure 7: Fault tolerance overhead for Jacobi method without failures: diskless checkpointing vs. algorithm-based recovery.

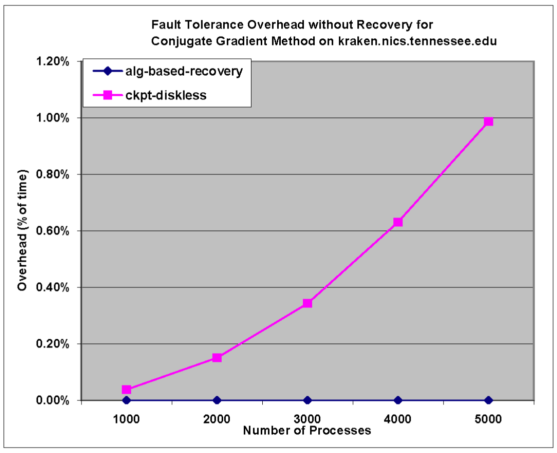

Figure 8: Fault tolerance overhead for Conjugate Gradient method without failures: diskless checkpointing vs. algorithm-based recovery.

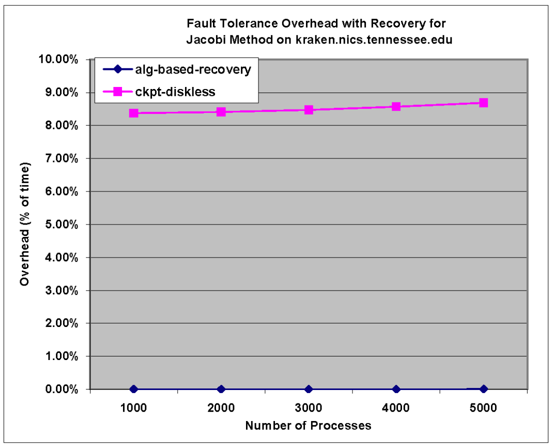

Figure 9: Fault tolerance overhead for Jacobi method with failures: diskless checkpointing vs. algorithm-based recovery.

Table 4: Fault Tolerant CG on Kraken.

num. of proc	ckp size per proc (bytes)	time per ckp (secs)	time per ckp rcv (secs)	rollback per rcv (secs)	time per alg rcv (secs)
1000	24000012	0.22	0.24	301	0.015
2000	96000012	0.90	0.95	300	0.054
3000	216000012	2.06	2.42	299	0.138
4000	384000012	3.87	3.98	298	0.246
5000	600000012	5.89	6.00	296	0.387

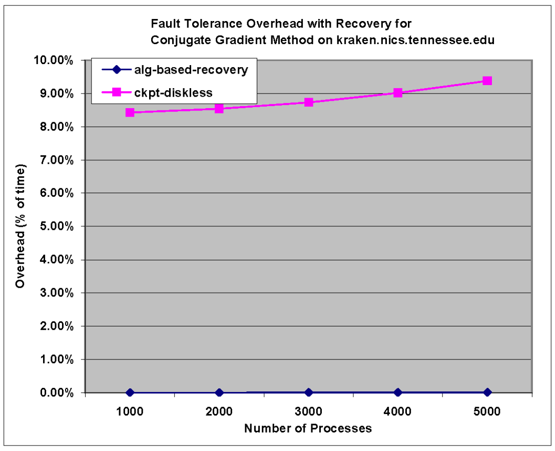

Figure 10: Fault tolerance overhead for Conjugate Gradient method with failures: diskless checkpointing vs. algorithm-based recovery.

point frequency is six checkpoints per hour. During each hour, a single process failure recovery is simulated approximately in the middle of two consecutive checkpoints. The recovery frequency is one recovery per hour.

The algorithm-based recovery experiments are configured the same way as in Subsection 6.1.

Table 3 and 4 report the size of checkpoint per process, the time per checkpoint, the time for each checkpoint-based recovery, the re-computation time due to each rollback, and the time for each algorithm-based recovery. Figure 7 and 8 compare the failure-free overhead. Figure 9 and 10 compare the total fault tolerance overhead with recovery.

Figure 7 and 8 demonstrate that, when no failure occurs during the computation, the algorithm-based recovery scheme introduces much less overhead than diskless checkpointing. This is because, for the algorithm-based recovery scheme, neither additional computations nor additional communications are introduced. But, for diskless checkpointing, it takes some time to perform the XOR encoding of the local checkpoints.

Figure 9 and 10 demonstrate that, when there is a recovery, the algorithm-based recovery also introduces much less overhead than diskless checkpointing. The main overhead for diskless checkpointing recovery is the time it takes to redo the wasted computation due to rollback, which is about 5 minutes. For algorithm-based recovery, the computation is restarted from where the failure occurs. No rollback is involved. The data recovery itself also takes slightly less time than diskless checkpointing. Therefore, overall, it introduces much less overhead than diskless checkpointing.

7. CONCLUSION

This paper presents an algorithm-based recovery scheme for iterative methods. It demonstrates that, for many iterative methods, if the parallel data partition scheme satisfies certain conditions, the iterative methods themselves can maintain enough inherent redundant information to tolerate failures in the computation. Neither checkpoint nor roll-back is necessary. The computation can be restarted from where the failure occurs. Experimental results demonstrate that the proposed recovery scheme introduces much less overhead than checkpointing. In the future, we would like to extend the technique to more iterative methods.

Acknowledgment

This research is partly supported by National Science Foundation (under grant #OCI-0905019) and Microsoft Corporation.

We would like thank the following organizations for the use of their computing resources:

- The National Institute for Computational Sciences: Kraken;
- The Golden Energy Computing Organization: Ra.

8. REFERENCES

[1] http://www.top500.org.
[2] http://www.nics.tennessee.edu.
[3] http://geco.mines.edu/hardware.shtml.
[4] http://wiki.lustre.org.
[5] The International Exascale Software Project. http://www.exascale.org.
[6] Coordinated Infrastructure for Fault Tolerant Systems. http://www.mcs.anl.gov/research/cifts.
[7] MPICH-V. http://mpich-v.lri.fr.
[8] D. C. Arnold. Reliable, Scalable Tree-based Overlay Networks. *Ph.D. dissertation*, University of Wisconsin-Madison, 2008.
[9] J. Anfinson and F. T. Luk. A Linear Algebraic Model of Algorithm-Based Fault Tolerance. *IEEE Transactions on Computers*, v.37 n.12, p.1599-1604, December 1988.
[10] P. Banerjee, J. T. Rahmeh, C. B. Stunkel, V. S. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, vol. C-39:1132–1145, 1990.
[11] 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.
[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. *Modern Software Tools in Scientific Computing*, pp 163–202, Birkhäuser Press, 1997.

[13] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk. Algorithmic fault tolerance using the lanczos method. *SIAM Journal on Matrix Analysis and Applications*, 13:312–332, 1992.

[14] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP03)*, San Diego, California, June 11-13, 2003.

[15] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C3: A System for Automating Application-level Checkpointing of MPI Programs. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC03)*, College Station, Texas, October 2-4, 2003.

[16] G. Bronevetsky, K. Pingali, P. Stodghill. Application-level Checkpointing for OpenMP Programs. In *Proceedings of the 20th International Conference on Supercomputing (ICS06)*, Queensland, Australia, June 28-July 1, 2006.

[17] G. Burns, R. Daoud, and J. Vaig. LAM: An Open Cluster Environment for MPI. *Proceedings of Supercomputing Symposium*, 1994.

[18] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. *proceedings of The IEEE/ACM SC2002 Conference*, Baltimore USA, November 2002.

[19] F. Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *International Journal of High Performance Computing Applications*, Vol. 23, No. 3, Page 212-226, 2009.

[20] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, Vol. 23, No. 4, Page 374-388, 2009.

[21] F. Cappello, H. Casanova, Y. Robert. Checkpointing vs. Migration for Post-Petascale Machines. CoRR abs/0911.5593, 2009.

[22] Z. Chen and J. Dongarra. Numerically stable real number codes based on random matrices. In *Proceeding of the 5th International Conference on Computational Science (ICCS2005)*, Atlanta, Georgia, USA, May 22-25, 2005. LNCS 3514, Springer-Verlag.

[23] Z. Chen and J. Dongarra. Condition Numbers of Gaussian Random Matrices. *SIAM Journal on Matrix Analysis and Applications*, Volume 27, Number 3, Page 603-620, 2005.

[24] Z. Chen, and J. Dongarra. Algorithm-Based Checkpoint-Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources. *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 25-29, 2006.

[25] Z. Chen, and J. Dongarra. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 19, No. 12, December, 2008.

[26] Z. Chen, and J. Dongarra. Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing. *IEEE Transactions on Computers*, July, 2009.

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

[28] Z. Chen. *Scalable techniques for fault tolerant high performance computing*. Ph.D. thesis, University of Tennessee, Knoxville, TN, USA, 2006.

[29] Z. Chen. Extending Algorithm-based Fault Tolerance to Tolerate Fail-stop Failures in High Performance Distributed Environments. *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, DPDNS'08 Workshop*, Miami, FL, USA, April 14-18, 2008.

[30] Z. Chen. Optimal Real Number Codes for Fault Tolerant Matrix Operations. *Proceedings of the ACM/IEEE SC09 Conference*, Portland, OR, November 14-20, 2009.

[31] T. Davies and Z. Chen. Fault Tolerant Linear Algebra: Recovering from Fail-Stop Failures without Checkpointing. *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium, PhD Forum*, Atlanta, GA, USA, April 19-23, 2010.

[32] D. Hakkarinen and Z. Chen. Algorithmic Cholesky Factorization Fault Recovery. *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010)*, Atlanta, GA, USA, April 19-23, 2010.

[33] J. Dongarra, P. Beckman, et al. The International Exascale Software Project Roadmap. *University of Tennessee EECS Technical Report*, UT-CS-10-652, January 6, 2010.

[34] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. *Berkeley Lab Technical Report*, LBNL-54941, December 2002.

[35] E. N. Elnozahy, L. Alvisi,Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, Volume 34, Issue 3, Page 375-408, 2002.

[36] R.D. Falgout, J.E. Jones, and U.M. Yang. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. *Numerical Solution of Partial Differential Equations on Parallel Computers*, pp 267–294, Springer-Verlag, 51, 2006.

[37] 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. *International Journal of High Performance Computing Applications*, 2004.

[38] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall Open MPI: Goals, Concept, and Design of a Next Generation MPI

Implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting* , Budapest, Hungary, September, 2004.

[39] G. A. Gibson, B. Schroeder, and J. Digney. Failure Tolerance in Petascale Computers. *CTWatchQuarterly*, Volume 3, Number 4, November 2007.

[40] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The John Hopkins University Press, , 1989.

[41] J. Gunnels, R. van de Geijn, D. Katz, E. Quintana-Ort. Fault-Tolerant High-Performance Matrix Multiplication: Theory and Practice *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)* , Washington, DC, USA, 2001.

[42] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine and J. Dongarra. CIFTS: A Coordinated infrastructure for Fault-Tolerant Systems. In *Proceedings of the 20th International Conference on Parallel Processing (ICPP09)*, Vienna, Austria, September 22-25, 2009.

[43] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3), 2005.

[44] P. Hough and V. Howle. Fault Tolerance in Large-Scale Scientific Computing. *Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds., SIAM Press, 2006.

[45] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, vol. C-33:518–528, 1984.

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

[47] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery Patterns for Iterative Methods in a Parallel Unstable Environment *SIAM Journal on Scientific Computing*, 30(1):102-116, 2007.

[48] F. T. Luk and H. Park An analysis of algorithm-based fault tolerance techniques. *SPIE Adv. Alg. and Arch. for Signal Proc.*, vol. 696, 1986, pp. 222-228.

[49] H. Ltaief, E. Gabriel, and M. Garbey. Fault tolerant algorithms for heat transfer problems. *Journal of Parallel and Distributed Computing*, Volume 68 , Issue 5, Pages 663-677, 2008.

[50] J. S. Plank, Y. Kim, and J. Dongarra. Fault Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing. *IEEE Journal of Parallel and Distributed Computing*, 43, 125-138, 1997.

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

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

[53] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, Volume 1 , Issue 3, pp 222-238, 1983.

[54] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, Philadelphia, PA, USA, June 25-28, 2006.

[55] G. Stellner. CoCheck: Checkpointing and process migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, Hawaii, April, 1996.

[56] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[57] J. L. Sung and G. R. Redinbo. Algorithm-Based Fault Tolerant Synthesis for Linear Operations. *IEEE Transactions on Computers*, Volume 45 , Issue 4, April 1996.

[58] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, page 1-15, Baltimore, Maryland, 2002.

[59] C. Wang, F. Mueller, C. Engelmann, and S. Scot. Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, March, 2007, Long Beach, CA, USA.*