Algorithm-Based Checkpoint-Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources

Zizhong Chen[1] and Jack Dongarra[1,2]

[1]The University of Tennessee, Knoxville
Department of Computer Science
Knoxville, TN 37996-3450 USA
{zchen, dongarra}@cs.utk.edu

[2]Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831-6367

Abstract

As the size of today's high performance computers increases from hundreds, to thousands, and even tens of thousands of processors, node failures in these computers are becoming frequent events. Although checkpoint/rollback-recovery is the typical technique to tolerate such failures, it often introduces a considerable overhead. Algorithm-based fault tolerance is a very cost-effective method to incorporate fault tolerance into matrix computations. However, previous algorithm-based fault tolerance methods for matrix computations are often derived using algorithms that are seldomly used in the practice of today's high performance matrix computations and have mostly focused on platforms where failed processors produce incorrect calculations.

To fill this gap, this paper extends the existing algorithm-based fault tolerance to the volatile computing platform where the failied processor stops working and applies it to scalable high performance matrix computations with two dimensional block cyclic data distribution. We show the practicality of this technique by applying it to the ScaLAPACK/PBLAS matrix-matrix multiplication kernel. Experimental results demonstrate that the proposed approach is able to survive process failures with a very low performance overhead.

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

1. Introduction

Today's long running scientific applications typically deal with faults by checkpoint/restart approaches in which all process states of an application are saved into stable storage periodically. The advantage of this approach is that it is able to tolerate the failure of the whole system. However, in this approach, if one process fails, usually all surviving processes are aborted and the whole application is restarted from the last checkpoint. The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints into stable storage [1]. The checkpoint of an application on a, say, ten-thousand-processor computer implies that all critical data for the application on all ten thousand processors have to be written into stable storage periodically, which may introduce an unacceptable amount of overhead into the checkpointing system. The restart of such an application implies that all processes have to be recreated and all data for each process have to be re-read from stable storage into memory or re-generated by computation, which often brings a large amount of overhead into restart. It may also be very expensive or unrealistic for many large systems such as grids to provide the large amount of stable storage necessary to hold all process state of an application of thousands of processes.

In order to tolerate partial failures with reduced overhead, diskless checkpointing [1, 2] has been proposed by Plank et. al. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing removes the main source of overhead in checkpointing [1]. Diskless checkpointing has been shown to achieve a decent performance to tolerate single process failure in [3]. For applications which modify a small amount of memory between checkpoints, it is shown in [4] that ,even to tolerate multiple simultaneous process failures, the overhead introduced by diskless checkpointing is still negligible.

However, for applications, such as matrix-matrix multiplication, which modify a large mount of memory between checkpoints, due to the large checkpoint size, even diskless checkpointing still introduces a considerable overhead into applications. Firstly, a local in memory checkpoint has to be maintained in diskless checkpointing, which introduces a large amount of memory overhead and hurts the efficiency of applications. Secondly, the local checkpoint in diskless checkpointing has to be taken and encoded periodically, which introduce a considerable performance overhead into applications. Despite the checksum and reverse computation technique in [3] has reduced the memory overhead, the overhead to calculate the checkpoint encodings periodically does not change. Furthermore after failures, this technique increase the recovery overhead by reversing the computation.

Inspired by the existing algorithm-based fault tolerance idea in [5], in this paper, we present an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoints periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Although this approach is not as generally applicable as typical checkpoint approaches, in parallel matrix computations where it usually works, fault tolerance for partial node failures can often be achieved with a surprisingly low overhead.

Despite the fact that there has been much research on algorithm-based fault tolerance [5, 6, 7] in which applications are modified to operate on encoded data to determine the correctness of some mathematical calculations on parallel platforms where failed processors produce incorrect calculations, to the best of our knowledge, this is the first time that applications are modified to operate on encoded data to maintain a global consistent state on parallel and distributed systems where failed processors stop working.

We show the practicality of this technique by applying it to the ScaLAPACK/PBLAS [8] matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK library to achieve high performance and scalability. Experimental results for matrix-matrix multiplication demonstrate that the proposed approach is able to survive a small number of process failures with a very low performance overhead.

The rest of this paper is organized as follows. Section 2 specifies the type of failures we focus on. Section 3 presents the basic idea of algorithm-based checkpoint-free fault tolerance. In Section 4, we present an example to demonstrate how algorithm-based checkpoint-free fault tolerance works in practice by applying this technique to the ScaLAPACK/PBLAS matrix-matrix multiplication kernel. In Section 5, we evaluate the performance overhead of applying this technique to the ScaLAPACK/PBLAS matrix-matrix multiplication kernel. Section 6 concludes the paper and discusses future work.

2 Failure Model

To define the problem we are targeting and clarify the differences with traditional algorithm-based fault tolerance, in this section, we specify the type of failures we are focusing on.

Assume the computing system consists of many nodes connected by network connections. Each node has its own memory and local disk. The communication between processes are assumed to be message passing. Assume the target application is optimized to run on a fixed number of processes.

We assume nodes in the computing system are volatile, which means a node may leave the computing system due to failure, or join the computing system after being repaired. Unlike in traditional algorithm-based fault tolerance which assumes a failed processor continues to work but produce incorrect results, in this paper, we assume a *fail-stop* failure model. That is the failure of a node will cause all processes on the failed nodes stop working. All data of the processes on the failed node is lost. The processes on survival nodes could not either send or receive any message from the processes on the failed node. Although there are many other type of failures exist, in this paper, we only consider this type of failures. This type of failure is common in today's large computing systems such as high-end clusters with thousands of nodes and computational grids with dynamic resources.

3 Algorithm-Based Checkpoint-Free Fault Tolerance

In this section, we present the basic idea of algorithm-based checkpoint-free fault tolerance. We restrict our scope to the long running numerical computing applications only. As indicated in Section 4, this approach can mainly be applied to linear algebra computations on parallel and distributed systems.

3.1 Failure Detection and Location

It is assumed that fail-stop failures can be detected and located with the aid of the programming environment. Many current programming environments such as PVM [9], Globus [10], FT-MPI [11], and Open MPI [13] do provide this kind of failure detection and location capability. We assume the lost of partial processes in the message passing system does not cause the aborting of the survival processes and it is possible to replace the failed processes in the message passing system and continue the communication after the replacement. FT-MPI [11] is one such programming environments that support all these functionalities. In the rest of this section, we will mainly focus on how to recover the application.

3.2 Failure Recovery

Today's long running scientific programs typically deal with faults by checkpoint and rollback recovery in which all process states of an application are saved into certain storage periodically. If one process fails, the data on *all* processes has to be recovered from the last checkpoint. The checkpoint and rollback 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 saved into and recovered from some storage periodically, which may introduce an unacceptable amount of overhead (both time and storage) into the checkpointing system. Considering that all data on all survival processes are still effective, it is interesting to ask: *is it possible to recover only the lost data on the failed process?*

Consider the simple case where there will be only one process failure. Before the failure actually occurs, we do not know which process will fail, therefore, a scheme to recover only the lost data on the failed process actually need to be able to recover data on *any* process. It seems difficult to be able to recover data on any process without saving all data on all processes somewhere. However, if we assume, at any time during the computation, the data on the i^{th} process P_i satisfies

$$P_1 + P_2 + \cdots + P_{n-1} = P_n, \tag{1}$$

where n is the total number of process used for the computation. Then the lost data on *any* failed process would be able to be recovered from (1). Assume the j^{th} process failed, then the lost data P_j can be recovered from

$$P_j = P_n - (P_1 + \cdots + P_{j-1} + P_{j+1} + \cdots + P_{n-1})$$

In this very special case, we are lucky enough to be able to recover the lost data on *any* failed process without checkpoint due to the special *checksum relationship* (1). In practice, this kind of special relationship is by no means natural. However, it is natural to ask: *is it possible to design an application to maintain such a special checksum relationship on purpose?*

Assume the original application is designed to run on n processes. Let P_i denotes the data on the i^{th} computation process. The special checksum relationship above can actually be designed on purpose as follows

- Add another encoding process into the application. Assume the data on this encoding process is C. For numerical computations, P_i is often an array of floating-point numbers, therefore, at the beginning of the computation, we can create a checksum relationship among the data of all processes by initializing the data C on the encoding process as
$$P_1 + P_2 + \cdots + P_n = C \tag{2}$$
- During the executing of the application, redesign the algorithm to operate both on the data of computation processes and on the data of encoding process in such a way that the checksum relationship (2) is always maintained.

The specially designed checksum relationship (2) actually establishes an equality between the data P_i on computation processes and the encoding data C on the encoding process. If any processor fails then the equality (2) becomes an equation with one unknown. Therefore, the data in the failed processor can be reconstructed through solving this equation.

4 Checkpoint-Free Fault Tolerance for Matrix Multiplication

As an example to demonstrate how the algorithm-based checkpoint-free fault tolerance works in practice, in this section, we apply this technique to the ScaLAPACK/PBLAS matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK to achieve high performance and scalability.

Actually, it is also possible to incorporate fault tolerance into many other ScaLAPACK routines through this approach. However, in this section, we will restrict our presentation to the matrix-matrix multiplication kernel.

4.1 Two-Dimensional Block-Cyclic Distribution

It is well-known [8] that the layout of an application's data within the hierarchical memory of a concurrent computer is critical in determining the performance and scalability of the parallel code. By using two-dimensional block-cyclic data distribution [8], ScaLAPACK seeks to maintain load balance and reduce the frequency with which data must be transferred between processes.

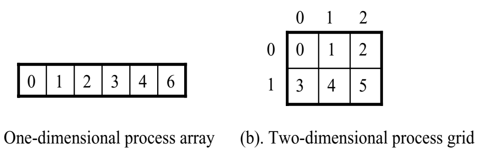

0 1 2 3 4 6
0 1 2
0 1 2
3 4 5
(a). One-dimensional process array
(b). Two-dimensional process grid

Figure 1. Process grid in ScaLAPACK

For reasons described above, ScaLAPACK organizes the one-dimensional process array representation of an abstract parallel computer into a two-dimensional rectangular process grid. Therefore, a process in ScaLAPACK can be referenced by its row and column coordinates within the grid. An example of such an organization is shown in Figure 1.

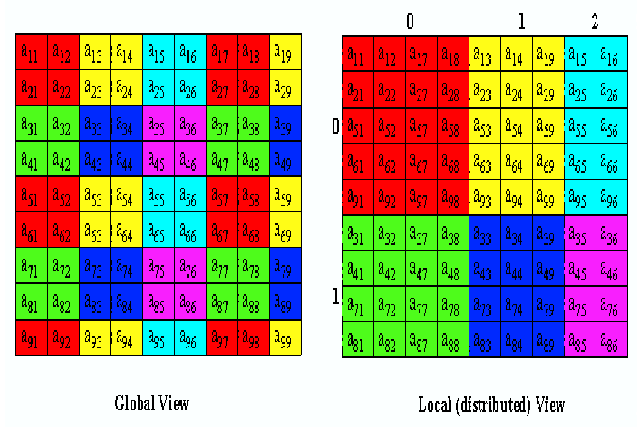


Figure 2. Two-dimensional block-cyclic matrix distribution

The two-dimensional block-cyclic data distribution scheme is a mapping of the global matrix onto the rectangular process grid. There are two pairs of parameters associated with the mapping. The first pair of parameters is (mb, nb), where mb is the row block size and nb is the column block size. The second pair of parameters is (P, Q), where P is the number of process rows in the process grid and Q is the number of process columns in the process grid. Given an element a_{ij} in the global matrix A, the process coordinate (p_i, q_i) that a_{ij} resides can be calculated by

$$\begin{cases} p_i = \lfloor \frac{i}{mb} \rfloor \bmod P, \\ q_i = \lfloor \frac{j}{nb} \rfloor \bmod Q, \end{cases}$$

The local coordinate (i_{p_i}, j_{q_j}) which a_{ij} resides in the process (p_i, q_i) can be calculated according to the following formula

$$\begin{cases} i_{p_i} = \lfloor \frac{\lfloor \frac{i}{mb} \rfloor}{P} \rfloor . mb + i \mod mb, \\ j_{q_j} = \lfloor \frac{\lfloor \frac{i}{nb} \rfloor}{Q} \rfloor . nb + i \mod nb, \end{cases}$$

Figure 2 is an example of mapping a 9 by 9 matrix onto a 2 by 3 process grid according two-dimensional block-cyclic data distribution with $mb = nb = 2$.

4.2 Encoding Two-Dimensional Block Cyclic Matrices

In this section, we will construct different encoding schemes which can be used to design checkpoint-free fault tolerant matrix computation algorithms in ScaLAPACK.

Assume a matrix M is originally distributed in a P by Q process grid according to the two dimensional block cyclic data distribution. For the convenience of presentation, assume the size of the local matrices in each process is the same. We will explain different coding schemes for the matrix M with the help of the example matrix in Figure 3. Figure 3 (a) shows the global view of an example matrix. After the matrix is mapped onto a 2 by 2 process grid with $mb = nb = 1$, the distributed view of this matrix is shown in Figure 3 (b).

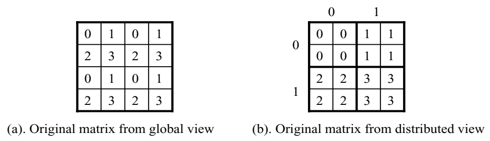
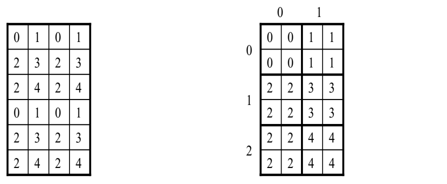

(a). Original matrix from global view
(b). Original matrix from distributed view

Figure 3. An example matrix

Suppose we want to tolerate a single process failure. We dedicate another $P + Q + 1$ additional processes and organize the total $PQ + P + Q + 1$ process as a $P + 1$ by $Q + 1$ process grid with the original matrix M distributed onto the first P rows and Q columns of the process grid.

The *distributed column checksum matrix* M^c of the matrix M is the original matrix M plus the part of data on the $(P+1)^{th}$ process row which can be obtained by adding all local matrices on the first P process rows. Figure 4 (b) shows the distributed view of the column checksum matrix of the example matrix from Figure 1. Figure 4 (a) is the global view of the column checksum matrix.



0	1	0	1
2	3	2	3
2	4	2	4
0	1	0	1
2	3	2	3
2	4	2	4

(a). Column checksum matrix from global view

	0		1	
0	0	0	1	1
	0	0	1	1
1	2	2	3	3
	2	2	3	3
2	2	2	4	4
	2	2	4	4

(b). Column checksum matrix from distributed view

Figure 4. Distributed column checksum matrix of the example matrix

The *distributed row checksum matrix* M^r of the matrix M is the original matrix M plus the part of data on the $(Q+1)^{th}$ process columns which can be obtained by adding all local matrices on the first Q process columns. Figure 5 (b) shows the distributed view of the row checksum matrix of the example matrix from Figure 1. Figure 5 (a) is the global view of the row checksum matrix.

0	1	1	0	1	1
2	3	5	2	3	5
0	1	1	0	1	1
2	3	5	2	3	5

(a). Row checksum matrix from global view

	0		1		2	
0	0	0	1	1	1	1
	0	0	1	1	1	1
1	2	2	3	3	5	5
	2	2	3	3	5	5

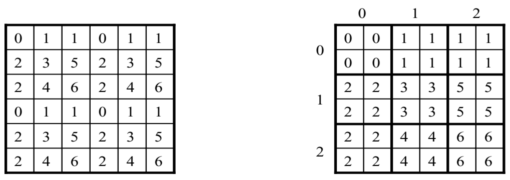
(b). Row checksum matrix from distributed view

Figure 5. Distributed row checksum matrix of the original matrix

The *distributed full checksum matrix* M^f of the matrix M is the original matrix M, plus the part of data on the $(P+1)^{th}$ process row which can be obtained by adding all local matrices on the first P process rows, plus the part of data on the $(Q+1)^{th}$ process column which can be obtained by adding all local matrices on the first Q process columns. Figure 6 (b) shows the distributed view of the full checksum matrix of the example matrix from Figure 3. Figure 6 (a) is the global view of the full checksum matrix.

4.3 Maintaining Global Consistent States by Computation

Assume A, B and C are distributed matrices on a P by Q process grid with the first element of each matrix



0	1	1	0	1	1
2	3	5	2	3	5
2	4	6	2	4	6
0	1	1	0	1	1
2	3	5	2	3	5
2	4	6	2	4	6

(a). Full checksum matrix from global view

	0		1		2	
0	0	0	1	1	1	1
	0	0	1	1	1	1
1	2	2	3	3	5	5
	2	2	3	3	5	5
2	2	2	4	4	6	6
	2	2	4	4	6	6

(b). Full checksum matrix from distributed view

Figure 6. Distributed full checksum matrix of the original matrix

on process (0,0). Let A^c, B^r and C^f denote the corresponding distributed checksum matrix. Let A^c_j denote the j^{th} column block of the matrix A^c and $B^{r\,T}_j$ denote the j^{th} row block of the matrix B^r. We first prove the following fundamental theorem for matrix matrix multiplication with checksum matrices.

Theorem 1 *Let $S_j = C^f + \sum_{k=0}^{j-1} A^c_k * B^{r\,T}_k$, then S_j is a distributed full checksum matrix.*

It is straightforward that $A^c_k * B^{r\,T}_k$ is a distributed full checksum matrix and the sum of two distributed full checksum matrices is a distributed checksum matrix. S_j is the sum of j distributed full checksum matrices, therefore is a distributed full checksum matrix.

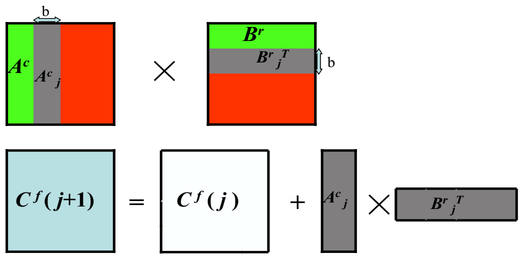


Figure 7. The j^{th} step of the fault tolerant matrix-matrix multiplication algorithm

Theorem 1 tells us that at the end of each iteration of the matrix matrix multiplication algorithm with checksum matrices, the checksum relationship of all checksum matrices are still maintained. This tells us that a coded global consistent state of the critical application data is maintained in memory at the end of each iteration of the matrix matrix multiplication algorithm if we perform the computation with related checksum matrices.

However, in a distributed environment, different process may update there local data asynchronously. Therefore, if when some process has updated their local matrix and some process is still in the communication stage, a failure happens, then the relationship of the data in the distributed matrix will no be maintained and the data on all processes would not form a consistent state. But this could be solved by simply performing a synchronization before performing local update. Therefore, in the following algorithm in Figure 8, there will always be a coded global consistent state (i.e. the checksum relationship) of the matrix A^c, B^r and C^f in memory. Hence, a single process failure at any time during the matrix matrix multiplication would be able to recovered from the checksum relationship.

Despite in this algorithm, the only modification to the library routine is to perform a synchronization before local update. However the amount of modification necessary to maintain a consistent state is highly dependent on the characteristic of an algorithm. For example, in LU factorization, due to the damage of the linear relationship by the global row pivoting, one also needs to adjust the encodings appropriately when performing pivoting to maintain a consistent encoded state in memory.

```
construct checksum matrices $A^c, B^r$, and $C^f$;
for $j = 0, 1, \ldots$
    row broadcast $A^c_j$;
    column broadcast ${B^r_j}^T$;
    synchronize;
    $C^f = C^f + A^c_j * {B^r_j}^T$;
end
```

Figure 8. A fault tolerant matrix-matrix multiplication algorithm

4.4 Overhead and Scalability Analysis

In this section, we analysis the overhead introduced by the algorithm-based checkpoint-free fault tolerance for matrix matrix multiplication.

For the simplicity of presentation, we assume all three matrices A, B, and C are square. Assume all three matrices are distributed onto a P by P process grid with m by m local matrices on each process. The size of the global matrices is Pm by Pm. Assume all elements in matrices are 8-byte double precision floating-point numbers. Assume every process has the same speed and disjoint pairs of processes can communicate without interfering each other. Assume it takes $\alpha + \beta k$ seconds to transfer a message of k bytes regardless which processes are involved, where α is the latency of the communication and $\frac{1}{\beta}$ is the bandwidth of the communication. Assume a process can concurrently send a message to one partner and receive a message from a possibly different partner. Let γ denote the time it takes for a process to perform one floating-point arithmetic operation.

4.4.1 Time Complexity for Parallel Matrix Matrix Multiplication

Note that the sizes of all three global matrices A, B, and C are all Pm, therefore, the total number of floating-point arithmetic operations in the matrix matrix multiplication is $2P^3m^3$. There are P^2 process with each process execute the same number of floating-point arithmetic operations. Hence, the total number of floating-point arithmetic operations on each process is $2Pm^3$. Therefore, the time T_{matrix_comp} for the computation in matrix matrix multiplication is

$$T_{matrix\_comp} = 2Pm^3\gamma.$$

In the parallel matrix matrix multiplication algorithm in Figure 8, the columns of A and the rows of B also need to broadcast to other column and row processes respectively. To broadcast one block columns of A using a simple *binary tree* broadcast algorithm, it takes $2(\alpha + 8bm\beta)\log_2 P$, where b is the row block size in the two dimensional block cyclic distribution. Therefore, the time T_{matrix_comm} for the communication in matrix matrix multiplication is

$$T_{matrix\_comm} = 2\alpha\frac{Pm}{b}\log_2 P + 16\beta Pm^2\log_2 P.$$

Therefore, the total time to perform parallel matrix matrix multiplication is

$$\begin{aligned} T_{matrix\_mult} &= T_{matrix\_comp} + T_{matrix\_comm} \\ &= 2Pm^3\gamma + 2\alpha\frac{Pm}{b}\log_2 P \\ &\quad +16\beta Pm^2\log_2 P. \qquad (3) \end{aligned}$$

4.4.2 Overhead for Calculating Encoding

To make matrix matrix multiplication fault tolerant, the first type of overhead introduced by the algorithm-based checkpoint-free fault tolerance technique is (1) constructing the distributed column checksum matrix A^c from A; (2) constructing the distributed row checksum matrix B^r from B; (3) constructing the distributed full checksum matrix C^f from C;

The distributed checksum operation involved in constructing all these checksum matrices performs the summation of P local matrices from P processes and saves the result into the $(P + 1)^{th}$ process. Let

T_{each_encode} denote the time for one checksum operation and T_{total_encode} denote the time for constructing all three checksum matrices A^c, B^r, and C^f, then

$$T_{total\_encode} = 4T_{each\_encode}$$

By using a *fractional tree* reduce style algorithm [14], the time complexity for one checksum operation can be expressed as

$$\begin{aligned} T_{each\_encode} &= 8m^2\beta\left(1+O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) \\ &\quad +O(\alpha\log_2 P)+O(m^2\gamma) \end{aligned}$$

Therefore, the time complexity for constructing all three checksum matrices is

$$\begin{aligned} T_{total\_encode} &= 32m^2\beta\left(1+O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) \\ &\quad +O(\alpha\log_2 P)+O(m^2\gamma). \qquad (4) \end{aligned}$$

In practice, unless the size of the local matrices m is very small or the size of the process grid P is extremely large, the total time for constructing all three checksum matrices is almost independent of the size of the process grid P.

The overhead (%) R_{total_encode} for constructing checksum matrices for matrix matrix multiplication is

$$\begin{aligned} R_{total\_encode} &= \frac{T_{total\_encode}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \qquad (5) \end{aligned}$$

From (5), we can conclude

1. If the size of the data on each process is fixed (m is fixed), then as the number of processes increases to infinite (that is $P \to \infty$), the overhead (%) for constructing the checksum matrices decreases to zero with a speed of $O(\frac{1}{P})$
2. If the number of processes is fixed (P is fixed), then as the size of the data on each process increases to infinite (that is $m \to \infty$) the overhead (%) for constructing the checksum matrices decreases to zero with a speed of $O(\frac{1}{m})$

4.4.3 Overhead for Performing Computations on Encoded Matrices

The fault tolerant matrix matrix multiplication algorithm in Figure 8 performs computations using checksum matrices which have larger size than the original matrices. However, the total number of processes devoted to computation also increases. A more careful analysis of the algorithm in Figure 8 indicates that the number of floating-point arithmetic operations on each process in the fault tolerant algorithm (Figure 8) is actually the same as that of the original non-fault tolerant algorithm.

As far as the communication is concerned, in the original algorithm, the column (and row) blocks are broadcast to P processes. In the fault tolerant algorithms (in Figure 8), the column (and row) blocks now have to be broadcast to $P+1$ processes.

Therefore, the total time to perform matrix matrix multiplication with checksum matrices is

$$\begin{aligned} T_{matrix\_mult\_checksum} &= 2Pm^3\gamma + 2\alpha\frac{Pm}{b}\log_2(P+1) \\ &\quad +16\beta Pm^2\log_2(P+1). \end{aligned}$$

Therefore, the overhead (time) to perform computations with checksum matrices is

$$\begin{aligned} T_{overhead\_matrix\_mult} &= T_{matrix\_mult\_checksum} - T_{matrix\_mult} \\ &= (2\alpha\frac{Pm}{b} + 16\beta Pm^2)\log_2(1+\frac{1}{P}). \end{aligned} \qquad (6)$$

The overhead (%) $R_{overhead\_matrix\_mult}$ for performing computations with checksum matrices in fault tolerant matrix matrix multiplication is

$$\begin{aligned} R_{overhead\_matrix\_mult} &= \frac{T_{overhead\_matrix\_mult}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \qquad (7) \end{aligned}$$

From (7), we can conclude that

1. If the size of the data on each process is fixed (m is fixed), then as the number of processes increases to infinite (that is $P \to \infty$), the overhead (%) for performing computations with checksum matrices decreases to zero with a speed of $O(\frac{1}{P})$
2. If the number of processes is fixed (P is fixed), then as the size of the data on each process increases to infinite (that is $m \to \infty$) the overhead (%) for performing computations with checksum matrices decrease to zero with a speed of $O(\frac{1}{m})$

4.4.4 Overhead for Recovery

The failure recovery contains two steps: (1) recover the programming environment; (2) recover the application data.

The overhead for recovering the programming environment depends on the specific programming environment. For FT-MPI [11] which we perform all our experiment on, it introduce a negligible overhead (refer Section 5).

The procedure to recover the three matrices A, B, and C is similar to calculating the checksum matrices. Except for matrix C, it can be recovered from either the row checksum or the column checksum relationship. Therefore, the overhead to recover data is

$$\begin{aligned} T_{recover\_data} &= 24m^2\beta \left(1 + O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) \\ &\quad + O(\alpha \log_2 P) + O(m^2\gamma) \end{aligned} \tag{8}$$

In practice, unless the size of the local matrices m is very small or the size of the process grid P is extremely large, the total time for recover all three checksum matrices is almost independent of the size of the process grid P.

The overhead (%) $R_{recover\_data}$ for constructing checksum matrices for matrix matrix multiplication is

$$\begin{aligned} R_{recover\_data} &= \frac{T_{recover\_data}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \end{aligned} \tag{9}$$

5 Experimental Evaluation

In this section, we experimentally evaluate the performance overhead of applying the algorithm-based checkpoint-free fault tolerance technique to the ScaLAPACK matrix-matrix multiplication kernel. We performed four sets of experiments to answer the following four questions:

1. What is the performance overhead of constructing checksum matrices?

2. What is the performance overhead of performing computations with checksum matrices?

3. What is the performance overhead of recovering FT-MPI programming environments?

4. What is the performance overhead of recovering checksum matrices ?

For each set of experiments, the size of the problems and the number of computation processes used are listed in Table 1.

All experiments were performed on a cluster of 32 Pentium IV Xeon 2.4 GHz dual-processor nodes. Each node of the cluster has 2 GB of memory and runs the

Table 1. Experiment Configurations

Size of original matrix	12,800	19,200	25,600
Size of checksum matrix	19,200	25,600	32,000
Process grid without FT	2 by 2	3 by 3	4 by 4
Process grid with FT	3 by 3	4 by 4	5 by 5

Linux operating system. The nodes are connected with a Gigabit Ethernet. The timer we used in all measurements is MPI_Wtime.

The programming environment we used is FT-MPI [11]. FT-MPI is a fault tolerant version of MPI that is able to provide basic system services to support fault survivable applications. FT-MPI implements the complete MPI-1.2 specification, some parts of the MPI-2 document and extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI can survive the failure of n-1 processes in a n-process job, and, if required, can respawn the failed processes. However, the application is still responsible for recovering the data structures and the data of the failed processes.

Although FT-MPI provides basic system services to support fault survivable applications, prevailing benchmarks show that the performance of FT-MPI is comparable [12] to the current state-of-the-art MPI implementations.

5.1 Overhead for Constructing Checksum Matrices

The first set of experiments is designed to evaluate the performance overhead of constructing checksum matrices. We keep the amount of data in each process fixed (that is the size of local matrices m fixed), and increase the size of the test matrices (hence the size of process grid).

Table 2. Time and overhead (%) for constructing checksum matrices

Size of original matrix	12,800	19,200	25,600
Time for original matrix	442.9	695.0	989.8
Time for encoding	38.0	40.8	43.2
Overhead (%) of encoding	8.6%	5.9%	4.4%

Table 2 reports the time for performing computations on original matrices and the time for constructing the three checksum matrices A^c, B^r, and C^f.

From Table 2, we can see that, as the size of the global matrices increases, the time for constructing checksum matrices increases only slightly. This is because, in the formula (4), when the size of process grid P is small, $32m^2\beta$ is the dominate factor in the time to constructing checksum matrices. Table 2 also indicates that the overhead (%) for constructing checksum matrices decreases as size of matrices increases, which is consistent with our theoretical formula (5) about the overhead for constructing checksum matrices.

5.2 Overhead for Performing Computations on Encoded Matrices

The algorithm-based checkpoint-free fault tolerance technique involve performing computations with checksum matrices, which introduces some overhead into the fault tolerance scheme. The purpose of this experiment is to evaluate the performance overhead of performing computations with checksum matrices.

Table 3. Time and overhead (%) for performing computations on encoded matrices

Size of original matrix	12,800	19,200	25,600
Size of checksum matrix	19,200	25,600	32,000
Time for original matrix	442.9	695.0	989.8
Time for encoded matrix	462.6	716.4	1013.3
Increased time	19.7	21.4	23.5
Overhead (%)	4.4%	3.1%	2.4%

Table 3 reports the execution time for performing computations on original matrices and the execution time for performing computations on checksum matrices for different size of matrices.

Table 3 indicates the amount time increased for performing computations on checksum matrices increases slightly as the size of matrices increases. The reason for this increase is that, when perform computations with checksum matrices, column blocks of A^c (and row blocks of B^r) have to be broadcast to one more process. The dominate time for parallel matrix matrix multiplication is the time for computation which is the same for both fault tolerant algorithm and non-fault tolerant algorithm. Therefore, the amount time increased for fault tolerant algorithm increases only slightly as the size of matrices increases.

5.3 Overhead for Recovering FT-MPI Environment

The overhead for recovering programming environments depends on the specific programming environments. In this section, we evaluate the performance overhead of recovering FT-MPI environment.

Table 4. Time and overhead (%) for recovering FT-MPI environment

Size of original matrix	12,800	19,200	25,600
Time for original matrix	442.9	695.0	989.8
Time for recover FTMPI	0.6	1.1	1.6
Overhead (%)	0.14%	0.16%	0.16%

Table 4 reports the time for recovering FT-MPI communication environment with single process failure. Table 4 indicates that the overhead for recovering FT-MPI is less than 0.2% which is negligible in practice.

5.4 Overhead for Recovering Application Data

The purpose of this set of experiments is to evaluate the performance overhead of recovering application data from single process failure.

Table 5. Time and overhead (%) for recovering rpplication data

Size of original matrix	12,800	19,200	25,600
Time for original matrix	442.9	695.0	989.8
Time for recovery data	28.5	30.6	32.4
Overhead (%)	6.4%	4.4%	3.3%

Table 5 reports the time for recovering the three checksum matrices A^c, B^r, and C^f in the case of single process failure. Table 5 indicates that ,as the size of the matrices increases, the time for recovering checksum matrices increases slightly and the overhead for recovering checksum matrices decreases, which again confirmed the theoretical results in Section 4.4.4.

6 Conclusion and Future Work

In this paper, we presented an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoint periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Although the applicability of this approach is not so general as the typical checkpoint/rollback-recovery approach, in parallel matrix computations where it usually works, process failures can often be tolerated with a surprisingly low overhead.

We showed the practicality of this technique by applying it to the ScaLAPACK/PBLAS matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK library to achieve high performance and scalability. Experimental results demonstrated that the proposed checkpoint-free approach is able to survive process failures with a very low performance overhead.

For the future, we plan to incorporate this fault tolerance technique into more ScaLAPACK library routines and more high performance computing applications. We would also like to evaluate this technique on systems with larger number of processors.

References

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

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

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

[4] 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.

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

[6] 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.

[7] V. Balasubramanian and P. Banerjee Compiler-Assisted Synthesis of Algorithm-Based Checkingin Multiprocessors. *IEEE Transactions on Computers*, vol. C-39:436-446, 1990.

[8] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 5, 1996.

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

[10] I. Foster and C. Kesselman. The globus toolkit. *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999.

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

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

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, 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. In *PVM/MPI*, pages 97–104, 2004.

[14] P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Inf. Process. Lett.*, 86(1):33–38, 2003.