Algorithm-Based Recovery for Newton's Method without Checkpointing

Hui Liu, Teresa Davies, Chong Ding, Christer Karlsson, and Zizhong Chen
Department of Mathematical and Computer Sciences
Colorado School of Mines
Golden, CO 80401, USA
{*huliu,tdavies,cding, ckarlsso, zchen*}@*mines.edu*

***Abstract*—Checkpointing is the most popular fault tolerance method used in high-performance computing (HPC) systems. However, increasing failure rates requires more frequent checkpoints, thus makes checkpointing more expensive. We present a checkpoint-free fault tolerance technique. It takes advantage of both data dependencies and communication-induced redundancies of parallel applications to tolerate fail-stop failures. Under the specified conditions, our technique introduces no additional overhead when there is no actual failure in the computation and recover the lost data with low overhead. We add fault-tolerant capacity to Newton's method by using our scheme and diskless checkpointing. Numerical simulations indicate that our scheme introduces much less overhead than diskless checkpointing does.**

***Keywords*-Dense Matrix Computations; Newton's method; Fault Tolerance; Algorithm-Based Recovery; Nonlinear Systems**

I. Introduction

Today's high performance computing (HPC) applications take from few hours to few months to complete. Since the mean-time-to-interrupt (MTTI) for many recent HPC systems varies from about half a month to less than half a day [1], HPC applications need to be able to tolerate failures and avoid restarting the computation from the beginning. Due to the large number of CPU cores in extreme scale systems, the probability that a failure occurs during an application execution is expected to be much higher than today's systems.

Fail-stop failure [2], [3] is a such failure where a failed process ceases to operate without sending malicious messages and destroys all associated data. HPC applications typically tolerate fail-stop failures by check-pointing with rollback-recovery. The application stores periodically checkpoints of the programs state to stable storage or in memory. If a failure occurs that causes the application to be terminated prematurely, the application can restart from the most recent state by reading saved checkpoints. Checkpointing loses at most an interval computation time in the event of a failure. Currently, there exist several commercial systems and publicly available libraries [4] supporting various flavors of checkpointing at a low performance overhead.

As the size of current parallel computer systems increases, failure rates increase. The necessary number of checkpoints increases as failure rates increases, this fact makes checkpointing cost more. We present an algorithm-based recovery scheme which tolerates faults without checkpointing. Our scheme uses the parallel application's data dependencies and communication-induced redundancies to recover the lost data on the failed process.

1) Data Dependencies: Dependency relationships are very common among the program variables and can be used to recover the lost data. For example, matrices A , B and C have different values in different iterations. Processes often achieve fault tolerance by saving them in checkpoints periodically during failure-free execution. In fact, if A and B are available, then C can be recovered from the dependency relationship $C = A + B$. Thus A and B are needed to be saved for the recovery, reducing the size of saved data.

2) Communication-induced Redundancies: A distributed memory programming model is often used in parallel implementations for many applications. In distributed memory machines, large data is partitioned among processes. Non-local data is accessed through inter-process communications, which is implemented via MPI [5]. These inter-process communications automatically duplicate the same data from sender to receiver. We define the received messages as communication-induced redundancies.

If a failed process's communication-induced redundancies contain its necessary recovery information, its lost data can be accurately recovered by communicating with its neighbors. Otherwise, we modifies the message-passing pattern to force the process into sending additional messages to its neighbor processes. Sometimes this modification introduces no additional time overhead.

Whenever a fail-stop failure occurs, our scheme uses the redundant information to recover the lost data on the failed process, losing at most an iteration computation time. Neither checkpoint nor roll-back is necessary. Our technique introduces less overhead compared to checkpointing techniques. Under the specified conditions, our technique introduces no overhead when there is no actual failure in the execution.

Nonlinear systems are always inevitable in many science and engineering disciplines such as geophysics, chemistry and physics. Newton's method is the most popular numerical technique for solving nonlinear systems. We add fault-tolerant ability to Newton's method by using our scheme and diskless checkpointing [6], respectively. Comparisons show that our scheme introduces much less overhead than diskless checkpointing does.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces our communication-induced checkpoint-free fault tolerance technique. Section 4 introduces the basic Newton's method and parallel Newton's method for nonlinear systems. In Section 5, we develop fault tolerant versions of parallel Newton's method so that fail-stop failures in these computations can be tolerated without checkpointing. This approach can be efficiently implemented by using MPI [5] and ScaLAPACK [15]. In Section 6, we analyze the fault tolerance overhead and scalability of the proposed method. In Section 7, we experimentally compare our method with diskless checkpointing on ra.mines.edu [17]. Section 8 concludes the paper and discusses the future work.

II. RELATED WORK

Checkpointing is a widely used mechanism in modern parallel systems to allow applications to protect themselves against failure. In [14] and [16], Bronevetsky et el. implements an application-level, coordinated, nonblocking checkpointing system for MPI programs using compiler technologies. Lipckpt [10] is a portable checkpointing tool for uniprocessor programs.

Plank et el. develops the diskless checkpointing technique [6] that stores efficiently checkpoints in processor memories. The technique removes the I/O bottleneck which is the main source of overhead in checkpointing. Multi-level checkpointing [7]–[9] allows applications to store lower-overhead, less-resilient checkpoints to stable storage and write the slowest but most resilient checkpoints to the parallel file system. Therefore, it benefits existing systems with better efficiency, reduced load and the ability to tolerate multiple failures.

However, applications such as dense linear algebra computations often modify a large mount of memory between checkpoints and check-pointing usually introduces considerable overhead in extreme scale systems. The algorithm-based fault tolerance (ABFT) [11], [12] schemes provides a low-cost error protection in single processor system. ABFT can detect, locate, and correct certain processor miscalculations in matrix computations by using checksum approach. Chen and Dongarra [13] propose a highly scalable checkpoint-free technique to tolerate single fail-stop failure in high performance matrix operations on large scale HPC systems. They also demonstrate that single fail-stop failure in ScaLAPACK [15] matrix multiplication can be tolerated without checkpointing at a decreasing overhead rate of $1/\sqrt{p}$, where p is the number of processors used for computation.

III. THE PROPOSED FAULT TOLERANCE METHOD

Many parallel simulations for applications are modeled as a message-passing distributed system. To allow a successful recovery, the fault tolerance scheme must contain enough amount of redundancy. Our fault tolerance implementations require the programmer to chooses sent messages within the execution of the application such that the collections of messages sent by all processes will yield the necessary recovery information. This method depends on data dependencies and communication-induced redundancies of the application. Furthermore, the programmer needs to modify the message passing pattern and implement recovery code, including the decision of which data structures and message passing patterns need to be constructed in order to improve the efficiency of communication and recovery. The recovery code reconstructs the connections among the processes and recover the lost data in the failed process by using the stored redundant information on the other processes.

A. Failure Model

Part of the distributed system can fail while the rest continues to work. Our scheme can take advantage of the redundancy offered by distributed systems to make applications more robust. The fail-stop failure model [2], [3] appears frequently in the distributed systems.

The fail-stop model makes some assumptions about the behavior of processes: the failed processes stop working and can not communicate with the others, the survival processes can continue working when some of processes die in the message passing system and it is possible to replace the failed processes in the message passing system and continue the communication after the replacement.

B. Failure Detection and Location

Generally, there are the following three steps to handle fault-tolerance [13] :

1) fault detection
2) fault location
3) fault recovery

Many programming environments such as FT-MPI [21] and Open MPI [20] can help the program to detect and locate fail-stop process failures.

C. Recovery

1) Data Dependencies: Generally, checkpointing techniques store the running programm's changed variables to stable storage or in memory periodically. Upon failure, checkpointing requires the entire parallel job rollback and restart from the most recent checkpoint. For some applications, storing all the changed variables is non-necessary and wasteful. The non-necessary information increases the amount of checkpoints. Storing large checkpoints (up to hundreds of megabytes per processor) becomes the main component that contributes to the time overhead.

The programmer exploits knowledge about the application to figure out the inherent dependencies among programm's variables. Dependency relationships between different variables are very common in applications. Newton's method is the most popular numerical technique for solving the nonlinear system $F(\vec{x}) = \mathbf{0}$. An inherent dependency of Newton's method is that the Jacobian matrix of F depends on $\vec{x}$. Generally, checkpointing stores Jacobian matrix and the variable $\vec{x}$ to tolerate fault. In fact, when $\vec{x}$ is recovered on the failed process, the relative Jacobian matrix can be recovered directly. Thus, we can recover Jacobian matrix without storing it. In other words, $\vec{x}$ is the data needed to be stored while Jacobian matrix is recovered by using data dependencies. Based on the inherent data dependencies, we reduce the amount of data to be saved. In some applications, we can significantly reduce checkpoint size for checkpointing by using data dependencies with less overhead.

2) Communication-induced Redundancies: A process of the message-passing distributed system receives some messages from the others and then performs some computations. If there is a message on its way, the process should wait for it and stop starting the relative computation. Those received messages are the inherent redundant information for the system during failure-free operation. If the receiver records the inherent redundant information, it is possible to recover the lost data on the failed processors without neither checkpoint nor roll-back. The computation can be restarted from where the failure occurs. If there is no failed processor in the total computation, the fault tolerate overhead (time) sometimes is zero.

Assume that there is p available processes. Let the process i send the message $Send_{ij}$ to the process j, where $i \neq j, 1 \leq i, j \leq p$. $Recoever_i$ denotes the process i's data needed to be stored on other processes according to the data dependencies analysis.

If $Recoever_i \leq \bigcup_{i \neq j, j=1}^{p} Send_{ij}$, the sent messages are enough for a successful recovery. Our scheme does not introduce overhead during failure-free execution.

Otherwise, we modify the message-passing pattern in order to increase the amount of redundancy. A potential way is forcing processes to send more messages which are the unsent necessary recovery information in the previous message-passing pattern. The new message-passing pattern introduces the additional communication overhead. The main challenge is minimizing the additional communication overhead.

If process i loses all associated data, it asks its neighbors to send back its necessary recovery information. Thus, some variables of process i is directly recovered through communication. Our scheme use the inherent dependencies among the variables to recover the rest variables via calculating. Once the rest variables are calculated, all data on process i is recovered and the application continues from the failure point.

IV. NEWTON'S METHOD

Consider the nonlinear system

$$F(\vec{x}) = \mathbf{0}, \quad \vec{x} = (x_1, x_2, \cdots, x_n) \tag{1}$$

where $\vec{x} \in D \in \Re^n$ are the variables and $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))$. We denote the Jacobian matrix of F at $\vec{x}$ by $J(\vec{x})$.

$$J(\vec{x}) = \begin{pmatrix} \frac{\partial f_1(\vec{x})}{\partial x_1} & \frac{\partial f_1(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\vec{x})}{\partial x_n} \\ \frac{\partial f_2(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\vec{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_n(\vec{x})}{\partial x_n} \end{pmatrix} \tag{2}$$

$$g = \begin{pmatrix} f_1(\vec{x}) \\ f_2(\vec{x}) \\ \vdots \\ f_n(\vec{x}) \end{pmatrix} \tag{3}$$

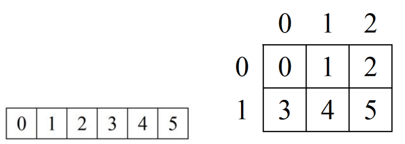
Assume that there is $\vec{x}^* \in D$ such that $F(\vec{x}^*) = \mathbf{0}$ with $J(\vec{x}^*)$ nonsingular.

Nonlinear systems are very hard to solve explicitly. In some instances, direct methods are computationally prohibitive. Generally, Newton's method [18] linearizes the nonlinear system about the current approximation. This will result in a linear system and thus can be solved for the next approximation. The procedure is iteratively executed until some certain stop criteria is met. The main advantages of Newton's method are the fast rate of convergence and easy implementation. Due to those advantages, it is one of the most powerful techniques for the nonlinear system. It is well known that Newtons method has local quadratic convergence if the Jacobian matrix is Lipschitz continuous and nonsingular at a solution $\vec{x}^*$ of the system [18].

Algorithm 1 Basic Newton's method

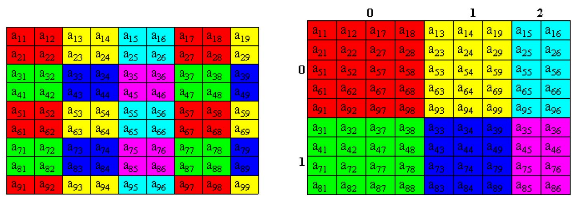
```
Require: $n > 0 \vee \vec{x_0} \vee \varepsilon$;
  $\vec{x} = \vec{x_0}$;
  $g = F(\vec{x})$ ;
  $J = J(\vec{x})$;
  while $\|g\| > \varepsilon$ do
    Solve $Jv = -g$;
    $x = x + v$;
    $g = F(\vec{x})$;
    $J = J(\vec{x})$;
  end while
```

Many applied problems can be reduced to nonlinear systems. However, the dimension of modern problems (millions



(a) One-dimensional process array (b) Two-dimensional process grid

Figure 1. Process grid in ScaLAPACK



(a) Global view (b) Local (distributed) view

Figure 2. The two-dimensional block-cyclic data distribution

of variables and hundreds of thousands of nonlinear equations) may be too large to be solved by sequential Newton's method. Therefore, the parallel version of Newton's method is needed for solving such large-scale scientific computing problems on parallel computer systems. TOUGH2-MP [26] is a general-purpose numerical parallel simulation program for multi-phase fluid and heat flow in porous and fractured media, which employs Newton's method to solve nonliear algebraic equations.

Large-scale scientific simulation has driven the size of high performance computers from thousands, to tens of thousands, and even to hundreds of thousands of processors. Linear system solver is the key computational kernel of the parallel Newton's method. Very efficient implementation of this operation for parallel computers with distributed architectures exists within ScaLAPACK [15]. Thus, we avoid the parallelization difficulties in Newton's method.

A. The Two-dimensional Block-Cyclic Distribution

Data decomposition is the core operation in constructing parallel algorithms. By using the two-dimensional block-cyclic data distribution [15], ScaLAPACK makes dense matrix computations as efficient as possible. The array descriptor $DESC\_$ is used for the ScaLAPACK routines solving dense linear systems and eigenvalue problems, which contains the details on block cyclic data distribution.

ScaLAPACK organizes this one-dimensional array of processes into a two-dimensional process grid. Therefore, we can identify a process by using its row and column coordinates. An example of such a map is shown in Figure 1.

The block-cyclic distribution scheme maps the global matrix onto the rectangular process grid. In this distribution, the global matrix is first divided into several blocks and each process owns a collection of blocks. Given an element a_{ij} of the global matrix A, the process coordinate (p_i, q_j) that a_{ij} resides can be calculated by

$$\begin{aligned} p_i &= \lfloor \frac{i}{mb} \rfloor \% P \\ q_j &= \lfloor \frac{j}{nb} \rfloor \% Q \end{aligned} \tag{4}$$

where mb is the row block size, nb is the column block size, P is the number of process rows in the process grid and Q is the number of process columns in the process grid.

Figure 2 is an example of mapping a $9-by-9$ matrix A onto a $2-by-3$ process grid according two-dimensional block-cyclic data distribution with $mb = nb = 2$. A vector is distributed, considering the vector as a column of the matrix.

B. Parellel Newton's Method

The parallel implementation of Newton's method requires that the operation for $Jv = -g$ be parallelized. In this paper, we use ScaLAPACK to solve $Jv = -g$. Thus J is distributed onto the process grid by using the two-dimensional block-cyclic data distribution. In this case, vectors $\vec{x}$, $\vec{v}$ and g are distributed onto one column of process grid.

In order to construct Hessian matrix J, the parallel method requires data exchange for $\vec{x}$. Assume all elements in matrices and vectors 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.

For example, each process has a copy of global $\vec{x}$ by employing MPI's all-gather reduction. In this implementation, the all-gather reduction requires each process to send m messages. The total number of $\vec{x}$ elements transmitted during the all-gather is $n(2^m - 1)/2^m$. Hence the expected execution time for the all-gather step is $\alpha m + 8n\beta(2^m - 1)/2^m$. To broadcast the global $\vec{x}$ using a simple binary tree broadcast algorithm, it takes $2(\alpha + 8n\beta)log_2Q$. Therefore, the time T_{vector_comm} for the communication in the Newton's step is:

$$\begin{aligned} T_{vector\_comm} &= \alpha m + 8n\beta(2^m - 1)/2^m + 2(\alpha + 8n\beta)log_2Q \\ &= \alpha(\frac{n}{P} + 2log_2Q) + 8n\beta[(2^{\frac{n}{P}} - 1)/2^{\frac{n}{P}} + 2log_2Q] \end{aligned}$$

V. Fault Tolerant Parallel Newton's Method

In this section, we show how to reconstruct the lost data on the failed process in parallel Newton's method to solve the nonlinear system. From now on, we represent fault tolerant parallel Newton's method by FT-Newton. FT-Newton with our scheme (checkpointfree-FT-Newton, for short) is a communication-induced checkpoint-free fault tolerance technique.

A. Analysis

In parallel Newton's method, the dense vector $\vec{x}$, g and the symmetric positive definite matrix J are often partitioned into p sub-vectors and sub-matrixes, x_i, g_i and J_i, respectively, where p is the number of processes used for the computation, x_i, g_i and J_i are assigned to the i^{th} process, and i = 1; 2; $\cdots$; p.

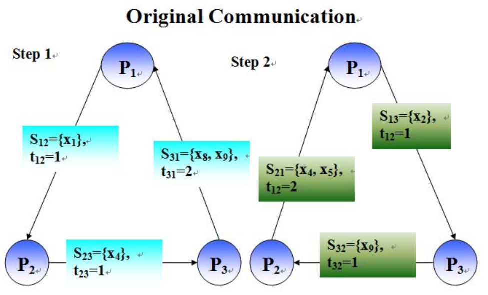
Calculating J is the the most important computation in Newton's method. In order to calculate $J_{ij} = \frac{\partial f_i(\vec{x})}{\partial x_j}$ on the i^{th} process, the sub-vector x_j , where j $\neq$ i, need to be sent from the j^{th} process to the i^{th} process. Hence, both the j^{th} and i^{th} process hold a copy of x_j. Therefore, in parallel Newton's method, it is possible to maintain multiple copies of the same subvector of $\vec{x}$ in different processes without additional time overhead. If a sub-vector of $\vec{x}$ on the j^{th} process is lost, it is possible to recover x_j by getting it from the i^{th} process. When x_j is available, the relative J can be recovered through calculating.

In many applications, Jacobian matrix is a sparse matrix, for example, J is a block diagonal matrix. Under this case, some processors' local data is enough for calculating their J. Thus communication-induced redundancies can not maintain enough amount of redundancy for the successful recovery. We should modify the message-passing pattern to force processes to send more information out, introducing additional overhead. In some special cases, additional overhead may be zero.

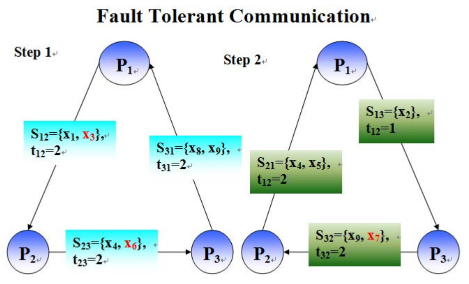
For example, assume that S_i denotes the local data on the process P_i, where $S_1 = \{x_1, x_2, x_3\}$, $S_2 = \{x_4, x_5, x_6\}$, $S_3 = \{x_7, x_8, x_9\}$. Each process need to exchange some elements with the other two processes. In the original communication, we need 4 units time to send all needed elements, but one element of each process does not have a copy on other processes. In the fault tolerant communication, we also need 4 units time. However, we ensure that each element has a copy on other processes. Therefore, all elements can be accurately reconstructed. Obviously, it is possible to maintain the copies for each sub-vector of $\vec{x}$ in different processes without additional time overhead.

B. Failure Recovery

The main benefit of our scheme is the ability to tolerate multiple failed processes as long as at least one of the survival processes that stores the necessary recover information. For example, we discuss the case where there is only one process failure. When a process failure occurs during the execution of the i^{th} iteration, if the following five values and two expression can be recovered: i, $\vec{x}^{(i-1)}$, J_descrip,



(a) Original Communication



(b) Fault Tolerant Communication

Figure 3. Message-passing pattern, S_{ij} denotes the messages from the i^{th} process to j^{th} process, t_{ij} is the communication time between the i^{th} process and j^{th} process.

g_descrip and x_descrip and g, H, then the computation can be restarted from the i^{th} iteration.

In the i^{th} iteration, if processes do not discard the messages they received in the last iteration, then all five values and two expressions that are necessary for restarting the i^{th} iteration can be recovered from the following steps:
1. The iteration loop variable i can be recovered by getting it from any surviving neighbor process.
2. J_descrip, g_descrip and x_descrip are the elements of the array descriptor $DESC\_$ and can be recovered by getting them from any surviving neighbor process. Then we can rebuild the missing process grid and data distribution information, because of the characteristic of ScaLAPACK.
3. $\vec{x}^{(i-1)}$ can be recovered by communicating with any surviving neighbor process.
4. g and J can be reconstructed by using data dependencies.

VI. Overhead and Scalability Analysis

In this section, we analyze the overhead introduced by the algorithm-based checkpoint-free fault tolerance and compare it with checkpointing.

Assume a program takes T to finish in a failure-free environment. The failure rate is a constant λ. Assume the time to failure of all processes follows an independent and identically-distributed exponential distribution. The probability that a failure will occur at τ hours into the execution

Algorithm 2 FT-Newton with our scheme

```
Require: $n > 0 \vee \vec{x} \vee \varepsilon$;
  Construct process grid;
  Distribute $H$, $\vec{x}$, $\vec{v}$ and $g$ onto the processes;
  if recover then
    Detect and locate fail-stop process failures with the aid of programming environment;
    The replacement of the failed process ask its neighbors to send i, $\vec{x}^{(i-1)}$, J_descrip, g_descrip and x_descrip;
    Calculate $g$ and $J$;
  end if
  $\vec{x} = \vec{x_0}$;
  With the need of computation, each process exchanges data $\vec{x}$ with neighbors.
  Each process records the received messages which are the necessary recovery information for its neighbors.
  $g = F(\vec{x})$ and $J = J(\vec{x})$;
  while $\|g\| > \varepsilon$ do
    Solve $Jv = -g$;
    With the need of computation, each process exchanges data $\vec{v}$ with neighbors.
    Each process records the received messages which are the necessary recovery information for its neighbors.
    $x = x + v$;
    $g = F(\vec{x})$ and $J = J(\vec{x})$;
  end while
```

which has no fault tolerance scheme is $\lambda e^{-\lambda\tau} d\tau$.

A. Overhead without Recovery

Let E_{non-ft} denote the expected program execution time without checkpoint.

$$\begin{aligned} E_{non-ft} &= e^{-\lambda\tau} \times T + \int_0^T (\lambda e^{-\lambda\tau} d\tau) \times (\tau + E) \\ &= \frac{e^{\lambda T} - 1}{\lambda} \end{aligned} \tag{5}$$

The expression (4) indicates that the expected program execution time will increase almost exponentially as the failure rate increases. As the complexity of a computer system increases, its failure rate is drastically increased. To improve the efficiency of parallel computing, our scheme should achieve fast self recovery through the computation period.

B. Overhead for Checkpointing

Extreme scale systems need more frequent checkpointing to fault tolerance. Many researchers [22]–[24] have proposed several models to approximate the optimal checkpoint interval. Suppose the time (i.e., overhead) for one checkpoint is t_c. It takes t_r to recover from a failure. Let I_{opt} denote the optimal checkpoint interval and E_{opt} denote the optimal expected program execution time with checkpoint. When a typical periodical checkpointing scheme is used in the application, we have

$$\begin{aligned} E_{interval} &= e^{-\lambda\tau} \times I + \int_0^I (\lambda e^{-\lambda\tau} d\tau) \times (\tau + E_{interval} + t_r) \\ &= (e^{\lambda I} - 1)(\frac{1}{\lambda} + t_r) \\ I &= \frac{T}{N} + t_c \end{aligned} \tag{6}$$

where N is the number of checkpoints.

$$\begin{aligned} \min_{\mathrm{N}}\{E\} &= \min_{\mathrm{N}}\{N \times E_{interval}\} \\ &= \min_{\mathrm{N}}\{N \times (e^{\lambda \frac{T}{N} + \lambda t_c} - 1)(\frac{1}{\lambda} + t_r)\} \end{aligned} \tag{7}$$

Let $\frac{\partial(E)}{\partial(N)} = (t_r + \frac{1}{\lambda}) \times [e^{\lambda(\frac{T}{N}+t_c)}(1 - \frac{\lambda T}{N}) - 1] = 0$, then we can find the minimum N.

$$\begin{aligned} & e^{-\lambda t_c} = e^{\lambda \frac{T}{N}}(1 - \frac{\lambda T}{N}) \\ & e^{-\lambda t_c} = (1 + \frac{1}{1!}\frac{\lambda T}{N} + \frac{1}{2!}(\frac{\lambda T}{N})^2 + \frac{1}{3!}(\frac{\lambda T}{N})^3 + \cdots)(1 - \frac{\lambda T}{N}) \\ & e^{-\lambda t_c} \approx 1 - (\frac{\lambda T}{N})^2 \frac{1}{2} \\ & N_{opt} \approx \frac{\lambda T}{\sqrt{2(1 - e^{-\lambda t_c})}} \end{aligned} \tag{8}$$

So $I_{opt} = \frac{\sqrt{2(1-e^{-\lambda t_c})}}{\lambda} + t_c$. Therefore, the optimal expected program execution time becomes:

$$E_{opt} = \frac{\lambda T}{\sqrt{2(1 - e^{-\lambda t_c})}}(\frac{1}{\lambda} + t_r)(e^{\sqrt{2(1-e^{-\lambda t_c})} + \lambda t_c} - 1) \tag{9}$$

If λt_c is small, then $e^{\lambda \frac{T}{N} + \lambda t_c} = \frac{1}{1 - \frac{\lambda T}{N}} = \frac{N}{N - \lambda T}$. In this case, $E_{opt} = \frac{\lambda T}{1 - \sqrt{2(1-e^{-\lambda t_c})}}(\frac{1}{\lambda} + t_r)$.

C. Overhead for Our Scheme

In this paper, the proposed fault tolerance scheme is a checkpoint-free technique. When a process failure occurs, the overhead to recover from the failure is t_{rr}. Let $E_{checkpoint-free}$ denote the expected program execution time by using the proposed fault tolerance scheme. We assume no failure occurs during the recovery. The expected number of failures during the whole program execution is consequently λT. The total recovering time is $\lambda T t_{rr}$. Therefore, the total expected program execution time $E_{checkpoint-free}$ can be estimated by

$$\begin{aligned} E_{checkpoint-free} &= T + \lambda T t_{rr} \\ &= T(1 + \lambda t_{rr}) \end{aligned} \tag{10}$$

The traditional checkpointing and our scheme are used in parallel applications, respectively. $P_{overhead}$ denotes the additional percentage of overhead introduced by checkpointing, where

$$P_{overhead} = \frac{E_{opt} - E_{checkpoint-free}}{E_{checkpoint-free}} = \frac{\lambda}{1 - \sqrt{2(1 - e^{-\lambda t_c})}} \frac{1 + \lambda t_r}{1 + \lambda t_{rr}} - 1 \quad (11)$$

When $t_r \approx t_{rr}$, $P_{overhead} \approx \frac{\lambda}{1-\sqrt{2(1-e^{-\lambda t_c})}} - 1$. If $\lambda t_c \ll 1$, we obtain $1-e^{-\lambda t_c} \approx \lambda t_c$. Therefore, $P_{overhead} \approx \frac{1}{1-\sqrt{2\lambda t_c}} - 1$, which indicates that our scheme reduced the fault tolerance overhead significantly especially when $t_c \ll \lambda$ is not satisfied.

Suppose that the system suffers two failures each day, a parallel application takes several days to complete and the time overhead for one checkpoint is 5 minutes. Under this assumption, $\lambda t_c = 0.00694$ and $P_{overhead} \approx 13.35\%$. When $t_c = 10$ minutes, $\lambda t_c = 0.01389$ and $P_{overhead} \approx 20\%$.

VII. Experimental Evaluation

In this section, we experimentally evaluate the proposed algorithm-based recovery scheme and compare it with diskless checkpointing.

The problem ARGTRIG from the CUTEr test set [25] is a nonlinear problem involving coupled trigonometric equations. For demonstration purpose, consider solving the following problem ARGTRIG using parallel Newton's method:

$$F(\vec{x}) = \vec{0}, \quad \vec{x} = (x_1, x_2, \cdots, x_n)$$

where $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))$ with

$$f_i(\vec{x}) = n - \sum_{j=1}^{n} cos(x_j) + i(1 - cos(x_i)) - sin(x_i) \quad (12)$$

The initial guess is $\vec{x_0} = (1/n, \cdots, 1/n)$. Note that this system has a full Jacobian.

A. Our Scheme vs. Diskless Checkpointing

In this subsection, we compare the overhead of our scheme with the diskless checkpointing on the supercomputing cluster ra.mines.edu [17] 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 MPI implementation on the cluster is Open MPI version 1.3.4.

Base on the characteristic of Newton's method, each generation has different $\vec{x}$, g and J. For diskless checkpointing experiments, the checkpoints are stored locally in memory. The simple implementation is to let process $2i$ and $2i+1$ backup data ($\vec{x}$, g and J) each other. If process $2i$ dies, it can ask process $2i+1$ to send the backup data to recover its own lost data. For our scheme, the failed process only requires its neighbor process to send $\vec{x}$ and then it can recover the related lost data.

Table I
Memory Overhead (bytes)

num. of proc	n	Diskless Checkpointing	Our Scheme
16	16	204	16
64	800	10508	800
256	256	1260	256

$$Overhead_{Diskless} = \frac{T_{Diskless} - T_{failure-free}}{T_{failure-free}}$$
$$Overhead_{Our\_scheme} = \frac{T_{Our\_scheme} - T_{failure-free}}{T_{failure-free}} \quad (13)$$

The total generations of FT-Newton is 100 and n is the size

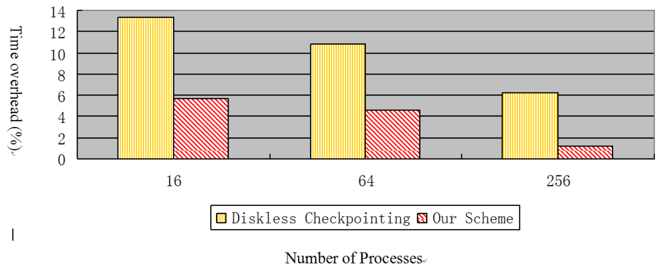


Figure 4. Time Overhead

of variables. The checkpoint frequency is one checkpoint per ten iterations. During each simulation, a single process failure recovery is simulated approximately in the middle of two consecutive checkpoints. Diskless Checkpointing and our scheme complement a recovery during each simulation.

Figure 4 reports the introduced time overhead of Diskless Checkpointing and our scheme, which are calculated by expression (13), respectively. As shown in Figure 4, our scheme introduces much less time overhead than diskless checkpointing. Diskless checkpointing recovery takes much time to redo the computations which are done before the failure occurs. For our scheme, 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. Table 1 reports the memory overhead and shows that our scheme introduces much less memory overhead than diskless checkpointing does.

VIII. Conclusion

This paper provides a new algorithm-based recovery scheme to deal with failures within MPI application for Exascale computing. Our scheme uses data dependencies and communication-induced redundancies to tolerant fail-stop failures with low overhead. The parallel Newton's method maintains enough inherent redundant information for the accurate recovery of the lost data on the failed process. The fault tolerant Newton's method with our scheme can restart the computation from where the failure occurs. Experimental results show that our scheme introduces much less overhead than diskless checkpointing does. They also demonstrate that our scheme is often highly scalable and have a good potential to scale to extreme scale computing and beyond. In the future, we will extend the technique to handle sparse matrix computation and integrate it with TOUGH2-MP to tolerate failures.

References

[1] B. Schroeder and G. A. Gibson, A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages:249-258, June 2006.

[2] 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.

[3] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. B. Zdonik. High-availability algorithms for distributed stream processing. *In Proc. ICDE*, pages 779-790, 2005.

[4] Elmootazbellah N. Elnozahy, and James S. Plank, Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 1(2):97-108, APRIL-JUNE 2004.

[5] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, MPI-The Complete Reference, Volume 1: The MPI Core, 2nd. (Revised) edition, *MIT Press*, 1998.

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

[7] E. Gelenbe. A Model of Roll-back Recovery with Multiple Checkpoints. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE 76)*, pages: 251C255, 1976.

[8] N. H. Vaidya. A Case for Multi-Level Distributed Recovery Schemes. *Texas A&M University, Tech. Rep.*, pages: 94-043, May 1994.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceesings of the 2010 ACM/IEEE conference on Supercomputing (SC'10)*, November 2010.

[10] J.S. Plank, M. Beck, G. Kingsley, and K. Li, Lipckpt: Transparent Checkpointing under UNIX, *Proc. USENIX Winter 1995 Technical Conf.*, pages: 213-223, Jan. 1995.

[11] KUANG-HUA Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518-528, 1984.

[12] CYNTHIA J. Anfinson and FRANKLIN T.Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computers*, 37(12):1599-1604, December 1988.

[13] Z. Chen and J. Dongarra, Algorithm-Based Fault Tolerance for Fail-Stop Failures, *IEEE Transactions on Parallel and Distributed Systems*. 19(12), pages: 1628-1641, December 2008.

[14] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, Automated Application-level Checkpointing of MPI Programs, *the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages: 84-94, June 2003.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley, ScaLAPACK Users' Guide, *SIAM*, Philadelphia, PA, 1997.

[16] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, C3: A System for Automating Application-level Checkpointing of MPI Programs, *the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC*, pages: 357-373, October 2003.

[17] http://geco.mines.edu/hardware.shtml.

[18] Dennis Jr., J. E., and Schnabel, R. B. (1996). Numerical methods for unconstrained optimization and nonlinear equations. SIAM Classics in Applied Mathematics.

[19] 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.

[20] 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 97C104, 2004.

[21] 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.

[22] John W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM*, 17(9):530-531, Sept. 1974.

[23] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303-312, 2004.

[24] Mohamed Slim Bouguerra, Denis Trystram, and *Frédéric* Wagner. An optimal algorithm for scheduling checkpoints with variable costs. Research report, INRIA, 2010.

[25] N. I. M. Gould, D. Orban, and P. L. Toint, CUTEr and SifDec: A constrained and unconstrained testing environment, revisited, *ACM Trans. Math. Software*, 29 (2003), pp. 373-394.

[26] http://tough2.com/index2.html