2011 IEEE International Parallel & Distributed Processing Symposium

Algorithm-Based Recovery Scheme for Extreme Scale Computing

Hui Liu
Department of Mathematical and Computer Sciences
Colorado School of Mines
Golden, CO 80401, USA
huliu@mines.edu

***Abstract*—We present an algorithm-based recovery scheme for Exascale computing, which uses both data dependencies and communication-induced redundancies of parallel codes to tolerate fault with low overhead. For some applications, our scheme significantly reduces checkpoint size and introduces no overhead when there is no actual failure in the computation. Fault tolerance Newton's method by tailoring our scheme to the algorithm is performed. Numerical simulations indicate that our scheme introduces much less overhead than diskless checkpointing does.**

***Keywords*-Fault Tolerance; Algorithm-Based Recovery; Fail-stop Failure; ScaLAPACK; Newton's Method**

I. Introduction

Since the mean-time-to-interrupt (MTTI) for many recent long running high performance computing (HPC) systems varies from about half a month to less than half a day, 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.

HPC applications typically tolerate fail-stop failures by check-pointing with rollback-recovery. If a failure occurs that causes the application to be terminated prematurely, the non-failed processors roll themselves back to their stored checkpoints and the failed processors uses the saved checkpoints to restart the application, wasting at most an interval's worth of computation. Bronevetsky [3], [5] et el. implemented an application-level, coordinated, nonblocking checkpointing system for MPI programs using compiler technologies. Plank et el. developed the diskless checkpointing technique [1] that stores the checkpoints locally in processor memories. However, applications such as dense linear algebra computations often modify a large mount of memory between checkpoints and check-pointing usually introduces considerable overhead when the number of processors used for computation is large. Chen and Dongarra [2] proposed a highly scalable checkpoint-free technique to tolerate single fail-stop failure in high performance matrix operations on large scale HPC systems.

A. *An Algorithm-based Recovery Scheme*

In this paper, fail-stop failure [9] is a such failure where a failed process ceases to operate without sending malicious messages and destroys all associated data. We present an algorithm-based recovery scheme to tolerate fail-stop failure.

1) Data Dependencies: Checkpointing techniques often store variables of running programs in stable storages or memories periodically. Upon failure, checkpointing requires the entire parallel job to rollback and to restart from the most recent checkpoint. In some applications, storing all variables is wasteful. The non-necessary information increases the sizes of checkpoints. Storing large checkpoints (up to hundreds of megabytes per processor) becomes the main overhead.

The prerequisite of using our scheme is the understanding of inherent data dependencies. 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: Many parallel simulations are modeled as a message-passing distributed system. Large data is partitioned among all available processors. In order to access non-local data, the interprocessor data exchange is performed via MPI [6]. One process of the system communicates with other processes to fetch messages and then performs some certain computations. If there is a message that is on its way, the process should wait for it and stop starting the relative computation. These communications lead to dependencies among processors during failure-free operation. If the receiver records received messages, 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, losing only one iteration's worth of computation in the event of a failure. If there is no failed processor in the total computation, the fault tolerate time overhead is zero. Assume that there is p

available processes, $Recoever_i$ is the necessary information of process i and also assume this process sends $Send_{ij}$ to process j, where $i \neq j; 1 \leq i,j \leq p$. There are two conditions to identify how to use our method to fault tolerance for different applications.

If $Recoever_i \leq \bigcup_{i \neq j, j=1}^{p} Send_{ij}$, the sent messages are enough for a successful recovery. Our scheme introduces no overhead when there is no actual failure in the execution. For example, If process i loses all associated data, it asks its neighbors to send back its necessary recovery information. Thus, some variables of process i are directly recovered. Process i uses the inherent dependencies among the variables to calculate the rest variables. Once the rest variables are calculated, all data on process i is recovered and the application continues from the failure point.

Otherwise, we modify the message-passing pattern to increase the amount of redundancies. A potential way is to force processes to send more previous unsent messages. The new message-passing pattern introduces the additional communication overhead. The main challenge is to minimize the additional communication overhead.

We give Newton's method the fault-tolerant ability by using our scheme. Comparisons show that our scheme introduces much *less* overhead than diskless checkpointing does.

II. NEWTON'S METHOD

Consider the nonlinear system $F(\vec{x}) = \mathbf{0}, \vec{x} = (x_1, x_2, \cdots, x_n)$, 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 $g = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))^T$ and the Jacobian matrix of F at $\vec{x}$ by $J(\vec{x})$. This system is always inevitable in many science and engineering disciplines such as geophysics, chemistry and physics.

$$J(\vec{x}) = \begin{pmatrix} \frac{\partial f_1(\vec{x})}{\partial x_1} & \frac{\partial f_1(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\vec{x})}{\partial x_n} \\ \frac{\partial f_2(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\vec{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_n(\vec{x})}{\partial x_n} \end{pmatrix} \quad (1)$$

Assume that there is $\vec{x}^* \in D$ such that $F(\vec{x}^*) = \mathbf{0}$ with $J(\vec{x}^*)$ nonsingular. Generally, Newton's method [7] 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. Linear system solver is the key computational kernel of the parallel Newton's method. Very efficient implementation of solving $Jv = -g$ for parallel computers with distributed architectures exists within ScaLAPACK [4]. Thus, we avoid the parallelization difficulties in Newton's method.

The data decomposition is the core operation in constructing parallel algorithms. By using the two-dimensional block-cyclic data distribution [4], ScaLAPACK makes dense matrix computations as efficient as possible. 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. A vector is distributed, considering it as a column of the matrix.

III. FAULT TOLERANT PARALLEL NEWTON'S METHOD

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. Generally,

Algorithm 1 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
```

there are the following three steps to handle fault-tolerance [2] : fault detection, fault location and fault recovery. Many programming environments such as FT-MPI [14] and Open MPI [13] can help the program to detect and locate fail-stop process failures.

If a process dies, it stops working and can not communicate with other processes. We assume that 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.

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 are assigned to the i^{th} process, and $i = 1; 2; \cdots; p$. 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 need to be sent from the j^{th} process to the i^{th} process, where j $\neq$ i. 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. When a sub-vector of $\vec{x}$ on one process is lost, it is possible to recover that lost sub-vector of $\vec{x}$ by getting it from another process. With the recovered $\vec{x}$, g and J can be reconstructed using the same way as they were constructed originally.

IV. OVERHEAD AND SCALABILITY ANALYSIS

Assume a program takes time T to finish in a failure-free environment and the failure rate λ is a constant. 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 which has no fault tolerance scheme is $\lambda e^{-\lambda\tau} d\tau$. Let E_{non-ft} denote the expected program execution time without checkpoint, where $E_{non-ft} = \frac{e^{\lambda T}-1}{\lambda}$. As the complexity of a computer system increases, its failure rate is drastically increased.

A. Overhead for Checkpointing

Extreme scale systems need more frequent checkpointing to fault tolerance. Many researchers [10]–[12] 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 $E_{interval} = (e^{\lambda I} - 1)(\frac{1}{\lambda} + t_r)$ and $I = \frac{T}{N} + t_c$, where N is the number of checkpoints.

$$\min_{\text{N}}\{E\} = \min_{\text{N}}\{N \times (e^{\lambda \frac{T}{N} + \lambda t_c} - 1)(\frac{1}{\lambda} + t_r)\} \qquad (2)$$

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

B. Overhead for Our Scheme

In order to obtain J and g, processes receive messages form another in each iteration of compute-intensive loops. This communication is a sufficient and necessary operation during failure-free execution. Our scheme uses these messages as recovery information to achieve fault tolerance. Obviously, our scheme introduces no overhead when there is no actual failure in the total computation.

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 $E_{checkpoint-free} = T + \lambda T t_{rr} = T(1 + \lambda t_{rr})$.

The traditional checkpointing and our scheme are used in parallel applications, respectively. $P_{overhead}$ denotes the additional percentage of overhead introduced by checkpointing, where $P_{overhead} = \frac{E_{opt}-E_{checkpoint-free}}{E_{checkpoint-free}} = \frac{\lambda}{1-\sqrt{2(1-e^{-\lambda t_c})}} \frac{1+\lambda t_r}{1+\lambda t_{rr}} - 1$. 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 $P_{overhead} \approx 20\%$.

V. EXPERIMENTAL EVALUATION

The problem ARGTRIG from the CUTEr test set [15] is a nonlinear problem involving coupled trigonometric equations: $F(\vec{x}) = \vec{0}, \vec{x} = (x_1, x_2, \cdots, x_n)$, where $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))$ with $f_i(\vec{x}) = n - \sum_{j=1}^{n} cos(x_j) + i(1 - cos(x_i)) - sin(x_i)$. Note that this system has a full Jacobian.

In this section, we compare the overhead of our scheme with the diskless checkpointing on the supercomputing cluster ra.mines.edu (http://geco.mines.edu/hardware.shtml) at Colorado School of Mines. 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. The total generations of FT-Newton is 100 and n is the size 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.

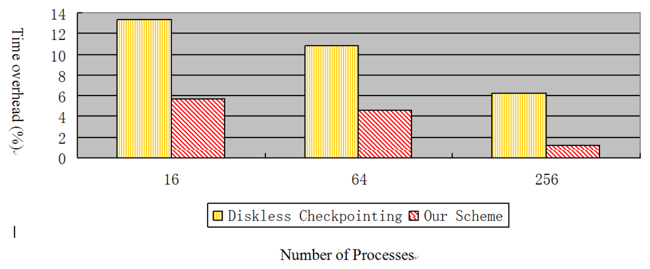

Figure 1. Time Overhead

Figure 1 reports the introduced time overhead of Diskless Checkpointing and our scheme, which are calculated by $Overhead_{Diskless} = \frac{T_{Diskless} - T_{failure-free}}{T_{failure-free}}$ and $Overhead_{Our\_scheme} = \frac{T_{Our\_scheme} - T_{failure-free}}{T_{failure-free}}$, respectively. As shown in Figure 1, 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.

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

VI. Conclusion

This paper provides an algorithm-based recovery scheme for Exascale computing. Based on data dependencies and communication-induced redundancies of parallel codes, our scheme can achieve fault tolerance with low overhead. Comparisons show that our scheme introduces much less overhead than diskless checkpointing does. In the future, we will analyze the inherent redundant information of other parallel algorithms and add fault-tolerant ability to them by using those inherent redundant information.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] Mohamed Slim Bouguerra, Denis Trystram, and $Fr\acute{e}d\acute{e}ric$ Wagner. An optimal algorithm for scheduling checkpoints with variable costs. Research report, INRIA, 2010.

[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, pp 97-104, 2004.

[14] 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.

[15] 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.