Fault Tolerant Matrix Operations for Networks of Workstations Using Multiple Checkpointing

Youngbae Kim[†]
[†]NERSC, Lawrence Berkeley National Laboratory
University of California
Berkeley, CA 94720, USA
youngbae@nersc.gov

James S. Plank[‡]
[‡§]Department of Computer Science
The University of Tennessee
Knoxville, TN 37996-1301, USA
plank@cs.utk.edu

Jack J. Dongarra[‡§]
[§]Mathematical Science Section
Oak Ridge National Laboratory
Oak Ridge, TN 37821-6367, USA
dongarra@[cs.utk.edu, msr.epm.ornl.gov]

Abstract

Recently, an algorithm-based approach using diskless checkpointing *has been developed to provide fault tolerance for high-performance matrix operations. With this approach, fault tolerance is incorporated into the matrix operations, making them resilient to any single process failure with low overhead. In this paper, we present a technique called* multiple checkpointing *that enables the matrix operations to tolerate a certain set of multiple processor failures by adding multiple checkpointing processors. Results of implementing this technique on a network of workstations show improvement in both the reliability of the computation and the performance of checkpointing.* [*]

1. Introduction

Due to the price and performance of uniprocessor workstations and off-the-shelf networking, networks of workstations (NOWs) have become a cost-effective parallel processing platform that is competitive with supercomputers. The popularity of NOW programming environments like PVM [10] and MPI [30] and the availability of high-performance numerical libraries like ScaLAPACK (Scalable Linear Algebra PACKage) [6] for scientific computing on NOWs show that networks of workstations are already in heavy use for scientific programming.

The major problem with programming on a NOW is the fact that it is prone to change. Idle workstations may be available for computation at one moment, but gone the next due to failure, load, or availability. We term any such event a *failure*. Thus, on the wish list of scientific programmers is a way to perform computation efficiently on a NOW whose components are tolerant to failure.

Recently, a fault-tolerant computing paradigm based on *diskless checkpointing* [25] has been developed in the papers [16, 17, 23, 24]. The paradigm is based on checkpointing and rollback recovery using processor and memory redundancy without any reliance on disk. Its underlying idea is to adopt the N+1 parity, used by Gibson to provide reliability in RAID (Redundant Array of Inexpensive Disks) [12]. The paradigm is an algorithm-based approach in which fault tolerance is especially tailored to the applications.

In this paradigm, application processors store checkpoints in local memory, and a separate checkpointing processor stores an encoding of the application processors' checkpoints. When a processor failure occurs, an extra idle processor replaces the failed processor and recovers its data from remaining application processors and the checkpoint encoding. Recently, checkpointing techniques based

[*] Youngbae Kim is partially supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number 76SF00098. James Plank is supported by National Science Foundation grants CCR-9409496 and CDA-95-29459, and the ORAU Junior Faculty Enhancement Award. Jack Dongarra is supported by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400 by the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615.

on parity [23, 24] or checksum and reverse computation [17], have been used to incorporate fault tolerance into high-performance matrix operations. Throughout this paper, we call these techniques *single checkpointing* because they employ only one checkpointing processor.

In this paper, we present a new technique called *multiple checkpointing*. In multiple checkpointing, we extend single checkpointing to tolerate a certain set of multiple processor failures by adding multiple checkpointing processors. The general idea of multiple checkpointing is to maintain coding information in m extra processors so that if one or more (up to m) application processors fail in the middle of computation, then they can be replaced instantly by one or more of the extra processors.

In our implementations, we have added multiple checkpointing processors to the fault-tolerant matrix operations using checksum and reverse computation developed in [17]. The analytic and experimental results show that using multiple checkpointing processors improves not only the reliability of the computation but also the performance of checkpointing. In particular, our technique reaps significant benefits from multiple checkpointing with relatively less memory by checkpointing at a finer-grain interval.

In Section 2, we review first the basic concept of single checkpointing and then introduce multiple checkpointing. In Section 3, we analyze the overhead of a multiple checkpointing technique and compare to the corresponding single checkpointing technique. In Section 4, we give a short description of how a multiple checkpointing technique can be incorporated in well-known algorithms in numerical linear algebra. In Section 5, we describe implementations in detail and show the performance of the implementations on a cluster of 20 Sun Sparc-5 workstations connected by a fast, switched Ethernet. In the subsequent sections, we discuss some issues raised by our technique, compare related work, draw conclusions, and suggest avenues for future work.

2. Checkpointing and Recovery

2.1. Basic Concept

Our technique for checkpointing and rollback recovery adopts the idea of *algorithm-based diskless checkpointing* [23] and enables a system with fail-stop failures [32] to tolerate these failures by periodically saving the entire state and rolling back to the saved state if a failure occurs.

We assume that the program is executing on a collection of N processors called *application processors*, and that there are m extra, idle processors in the system. At all points in time, a consistent checkpoint is held in the N processors in memory. A checksum (floating-point addition) of the N checkpoints is held in one of the m idle processors. This processor is called *checkpointing processor*, and the checksum is called the *checkpoint encoding*. If any processor fails, all live processors, including the checkpointing processor, cooperate in reversing the computations performed since the last checkpoint. Thus, the data is restored at the last checkpoint for rollback, and the failed processor's state can be reconstructed on the checkpointing processor as the checksum of the global checkpoint and the remaining $N-1$ processors' local checkpoints.

In the following two subsections, two recovery models are described—one for tolerating any single processor failure and the other for tolerating multiple processor failures.

2.2. Single-Failure Recovery Model

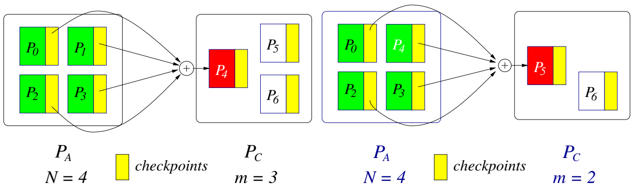

Figure 1. A single-failure recovery model: before/after a failure (P_1 fails)

In this model, after the system recovers from a failure, one of the remaining idle processors becomes the checkpointing processor and the system resumes. Thus, if failures are permanent, the system can tolerate m single failures during the lifetime of the application. Figure 1 depicts how checkpoints are encoded, and how recovery works in the presence of a single processor failure.

2.3. Multiple-Failure Recovery Model

The multiple-failure recovery model consists of $N + m$ processors that can tolerate up to m failures at once. Instead of having one dedicated processor for checkpointing, the entire set of application processors is divided into m groups, and one checkpointing processor is dedicated to each group. When one failure occurs in a group, the checkpointing processor in the group replaces the failed one, and the application rolls back and resumes from the last checkpoint. Figure 2 shows the application processors logically configured into a two-dimensional mesh, with a checkpointing processor dedicated to each row of processors. This model enables the algorithm to tolerate a certain set of multiple failures simultaneously, one failure for each group (e.g. each row or column of processors). This is called the *one-dimensional parity scheme* [13].

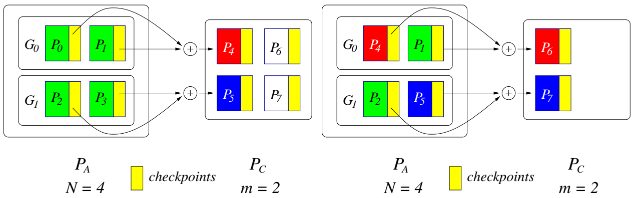

Figure 2. A multiple-failure recovery model: before/after two failures (P_0 and P_3 fail)

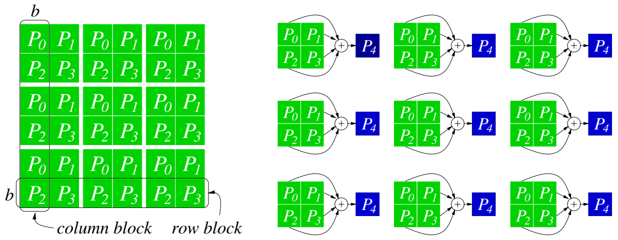

Figure 3. Data distribution and checkpointing of a matrix with 6×6 blocks over a 2×2 mesh of 4 processors (using a single checkpointing technique)

2.4. Multiple Checkpointing in Matrix Operations

In most high-performance implementations of distributed matrix operations, the processors are logically configured on a $P \times Q$ grid, and the matrix is decomposed onto that grid. To implement multiple checkpointing, we allocate one checkpointing processor per column of the grid. Note that such a scheme allows the program to tolerate Q simultaneous failures as long as failures occur in different groups, for example, columns of processors. With multiple checkpointing processors, each column of processors, including its dedicated checkpointing processor, cooperates to checkpoint its part of the matrix independently from the other columns of processors. Since the checkpointing and recovery can be distributed into groups of processors (i.e., columns of processors), the overhead of both checkpointing and recovery can be reduced. In addition, when the checksum is used, it reduces the possibility of overflow, underflow, and cancellation because fewer processors are involved in each checksum. Details of this technique can be found in [16].

3. Analysis of Checkpointing

In this section, the time complexity of checkpointing matrices is analyzed. This analysis will provide a basic formula for computing the overhead of checkpointing and recovery in each fault-tolerant matrix operation.

Throughout this paper, a matrix A is partitioned into square "blocks" of a user-specified block size b. Then A is distributed among the processors P_0 through P_{N-1}, logically reconfigured as a $P \times Q$ mesh, as in Figure 3. A row of blocks is called a "row block" and a column of blocks a "column block." If there are N processors and A is an $n \times n$ matrix, each processor holds $\frac{n}{Pb}$ row blocks and $\frac{n}{Qb}$ column blocks, where it is assumed that b, P, and Q divide n.

3.1. Analysis of Single Checkpointing

The basic checkpointing operation works on a panel of blocks, where each block consists of X floating-point numbers, and the processors are logically configured in a $P \times Q$ mesh (see Figure 3). The processors take the checkpoint with a global addition. This works in a spanning-tree fashion in three parts. The checkpoint is first taken column-wise, then taken row-wise, and then sent to the checkpointing processor P_C. The first part therefore takes $\lceil \log P \rceil$ steps, and the second part takes $\lceil \log Q \rceil$ steps. Each step consists of sending and then performing addition on X floating-point numbers. The third part consists of sending the X numbers to P_C. We define the following terms: γ is the time for performing a floating-point addition, α is the startup time for sending a floating-point number, and β is the time for transferring a floating-point number.

Details of this analysis can also be found in [16, 17]. The first part takes $\lceil \log P \rceil (\alpha + X(\beta + \gamma))$, the second part takes $\lceil \log Q \rceil (\alpha + X(\beta + \gamma))$, and the third takes $\alpha + X\beta$. Thus, the total time to checkpoint a panel is the following: $T_{panelckpt}(X, P, Q) = (\lceil \log P \rceil + \lceil \log Q \rceil)(\alpha + X(\beta + \gamma)) + (\alpha + X\beta)$. If we assume that X is large, the α terms disappear, and $T_{panelckpt}$ can be approximated by the following equation: $T_{panelckpt}(X, P, Q) \approx X(\beta + (\lceil \log P \rceil + \lceil \log Q \rceil)(\beta + \gamma))$.

If we define the function

$$T_{ckpt}(P, Q) = \frac{\beta + (\lceil \log P \rceil + \lceil \log Q \rceil)(\beta + \gamma)}{PQ}, \quad (1)$$

then $T_{panelckpt}(X, P, Q) \approx PQXT_{ckpt}(P, Q)$. For constant values of P and Q, $T_{ckpt}(P, Q)$ is a constant. Thus, $T_{panelckpt}(X, P, Q)$ is directly proportional to X. When an entire $n \times n$ matrix needs to be checkpointed, if we assume that n is large, the time complexity of checkpointing an entire $n \times n$ matrix is

$$T_{matckpt}(n, n, P, Q) = n^2 T_{ckpt}(P, Q). \quad (2)$$

We define the *checkpointing rate* R to be the rate of sending a message and performing addition on the message, measured in bytes per second. In 64-bit floating-point arithmetic, we approximate the relationship between R and $T_{ckpt}(P, Q)$ as follows:

$$T_{ckpt}(P, Q) \approx \frac{\lceil \log P \rceil + \lceil \log Q \rceil}{PQ} \frac{8}{R}. \quad (3)$$

3.2. Analysis of Multiple Checkpointing

In analyzing multiple checkpointing, we assume that one checkpointing processor is dedicated to checkpointing the data over a column of processors (i.e., a $P \times 1$ processor grid). One checkpoint is taken over each column of processors and is then sent to the corresponding checkpointing processor (see Figure 2). The first part takes $\lceil \log P \rceil (\alpha + X(\beta + \gamma))$ time, and the second takes $\alpha + X\beta$ time.

Thus, as discussed before, the time overhead of checkpointing a panel of X floating-point numbers can be approximated by the following equation: $T_{panelckpt}(X, P, Q) = X(\beta + \lceil \log P \rceil (\beta + \gamma))$. If we also define the function

$$T_{ckpt}(P, Q) = \frac{\beta + \lceil \log P \rceil (\beta + \gamma)}{PQ} \approx \frac{\lceil \log P \rceil}{PQ} \frac{8}{R}, \qquad (4)$$

the time overhead of checkpointing an $n \times n$ matrix is then given as in Eq. 2.

4. Fault-Tolerant Matrix Operations

We focus on three classes of matrix operations: matrix multiplication; direct, dense factorizations; and Hessenberg reduction. These matrix operations are at the heart of scientific computations and thus have been implemented in ScaLAPACK. The factorizations (Cholesky, LU, and QR) are operations for solving systems of simultaneous linear equations and finding least squares solutions of linear systems. Hessenberg reduction is an operation for solving a nonsymmetric eigenvalue problem. Note that we choose the highest performance, right-looking algorithms for the factorizations. Their fault-tolerant implementations using the single checkpointing technique based on checksum and reverse computation can be found in [17]. The results in this paper also include multiple checkpointing as described in Section 2.4.

5. Implementation Results

We implemented and executed these programs on a network of Sparc-5 workstations running PVM [10]. This network consists of 24 workstations, each with 96 Mbytes of RAM, connected by a switched 100 megabit Ethernet. The peak measured bandwidth in this configuration is 40 megabits per second between two random workstations. We ran our experiments on these machines when we could allocate them exclusively for our own use.

Each implementation was run on 20 processors, with 16 application processors logically configured into a 4×4 processor grid and 4 checkpointing processors, one for each processor column. The block size for all implementations was set at 50, and all implementations were developed for double-precision floating-point arithmetic.

We ran two sets of tests for each instance of each problem. In the first, there is no checkpointing. In the second, the program checkpoints, but there are no failures.

Experimental results of the implementations for matrix multiplication, LU factorization, and Hessenberg reduction are given in Figures 4 through 6, respectively. For comparison, each figure includes experimental results of the single and multiple checkpointing schemes. Each figure contains a table of experimental results and graphs of running times, percentage checkpoint overhead, and checkpointing rate experimentally determined. In each table, the fifth columns represent the average checkpointing interval in seconds, and the eighth columns represent the average time overhead of each checkpoint. Note that T_C includes the initial checkpointing overhead T_{init} and T_A represents the total running time of the algorithm without checkpointing. K represents the checkpointing interval in iterations and is chosen differently for each implementation to keep the checkpointing overhead small. Note that we use the same value of K for the multiple checkpointing technique as the single checkpointing technique.

n	T_A (sec)	N_C +3	T (sec)	$\frac{\Delta T}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)
		With Single Checkpointing						
1000	29	2+3	43	19.1	14	48.3	8	2.7
2000	197	3+3	261	74.6	64	32.5	32	9.1
3000	644	4+3	816	171.8	172	26.7	73	20.8
4000	1547	6+3	1941	323.5	394	25.5	131	43.8
5000	2951	7+3	3682	507.9	731	24.8	205	72.6
6000	5036	8+3	6170	725.9	1134	22.5	295	98.7
7000	7920	9+3	9546	979.1	1626	20.5	406	125.1
		With Multiple Checkpointing						
1000	29	2+3	37	16.4	8	27.6	5	1.3
2000	197	3+3	238	68.0	41	20.8	21	5.7
3000	644	4+3	756	159.2	112	17.4	48	13.5
4000	1547	6+3	1797	299.5	250	16.2	85	27.5
5000	2951	7+3	3418	471.4	467	15.8	149	43.9
6000	5036	8+3	5855	688.8	819	16.3	191	73.9
7000	7920	9+3	9108	934.2	1188	15.0	263	94.9

$K = 16, \Delta T_C = T_C - T_{init}, T = T_A + T_C, \Delta T = T - Tinit.$

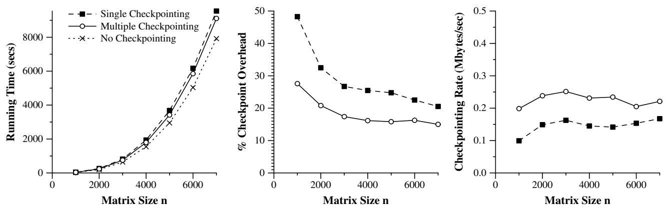

Figure 4. Matrix Multiplication: Timing Results

6. Discussion

The performance results and analyses presented in the preceding sections confirm that using multiple checkpoint-

n	T_A (sec)	N_C $+1$	T (sec)	$\frac{\Delta T}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)
		With Single Checkpointing						
1000	45	2	52	23.1	7	15.6	3	1.8
2000	153	3	180	51.4	27	17.6	11	4.6
3000	364	4	436	91.8	72	19.8	25	9.9
4000	745	6	884	147.3	139	18.7	43	16.0
5000	1293	7	1525	210.3	232	17.9	69	22.5
6000	2144	8	2525	297.1	381	17.8	98	33.3
7000	3211	9	3760	385.6	549	17.1	134	42.6
8000	4774	11	5590	508.2	816	17.1	175	58.3
9000	6268	12	7555	616.7	1287	20.5	229	86.4
10000	8651	13	10447	773.9	1796	20.8	282	112.1
		With Multiple Checkpointing						
1000	45	2	50	22.2	5	11.1	2	1.3
2000	153	3	170	48.6	17	11.1	7	2.9
3000	364	4	404	85.1	40	11.0	16	5.1
4000	745	6	825	137.5	80	10.7	29	8.5
5000	1293	7	1427	196.8	134	10.4	45	12.3
6000	2144	8	2368	278.6	224	10.4	65	18.7
7000	3211	9	3561	365.2	350	10.9	90	26.7
8000	4774	11	5271	479.2	497	10.4	115	34.7
9000	6268	12	7042	574.9	774	12.3	153	50.7
10000	8651	13	9732	720.9	1081	12.5	181	66.7

$K = 16, \Delta T_C = T_C - T_{init}, T = T_A + T_C, \Delta T = T - Tinit.$

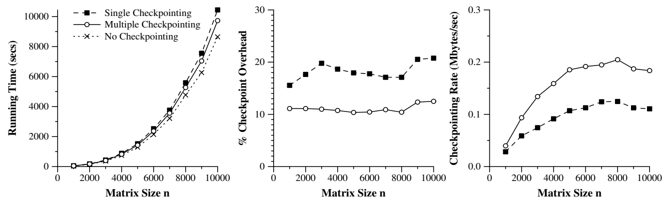

Figure 5. Right-looking LU: Timing results

ing processors improves the performance of checkpointing and recovery considerable for all of the fault-tolerant implementations. Thus, multiple checkpointing is more efficient and reliable by not only distributing the process of checkpointing and rollback recovery over groups of processors, but also by tolerating multiple failures in one of each group of processors.

In particular, when multiple checkpointing is combined with the implementations based on checksum and reverse computation, it reduces the checkpointing and recovery overhead without using more memory. As the performance is improved with multiple checkpointing, it can reduce checkpointing interval and hence use less memory for checkpointing. In addition, the probability of overflow, underflow, and cancellation error can be reduced.

7. Related Work

Considerable research has been carried out on algorithm-based fault tolerance for matrix operations on parallel platforms where (unlike the above platform) the computing nodes are not responsible for storage of the input and output elements [14, 20, 28]. These methods concentrate mainly on fault-detection and, in some cases, correction.

n	T_A (sec)	N_C $+1$	T (sec)	$\frac{\Delta T}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)
		With Single Checkpointing						
3000	1524	16	1607	100.4	83	5.4	24	3.7
4000	3158	21	3491	166.2	333	10.5	44	13.8
5000	5744	26	6423	247.0	679	11.8	68	23.5
6000	9323	31	10653	343.6	1330	14.3	98	39.7
7000	14258	36	16478	457.7	2220	15.6	135	57.9
		With Multiple Checkpointing						
3000	1524	16	1572	98.2	48	3.1	16	2.0
4000	3158	21	3340	159.0	182	5.8	28	7.3
5000	5744	26	6122	235.5	378	6.6	45	12.8
6000	9323	31	10033	323.6	710	7.6	64	20.8
7000	14258	36	15585	432.9	1327	9.3	88	34.4

$K = 4, \Delta T_C = T_C - T_{init}, T = T_A + T_C, \Delta T = T - T_{init}.$

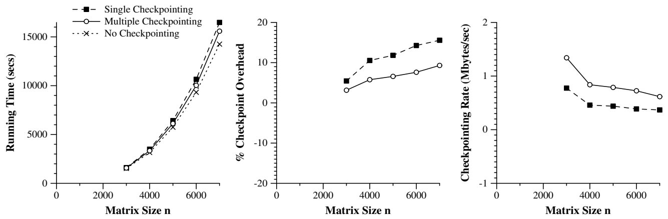

Figure 6. Hessenberg reduction: Timing results

Checkpointing on parallel and distributed systems has been studied and implemented in many research projects [4, 7, 8, 9, 15, 18, 19, 26, 29, 32]. All of this work, however, focuses on either checkpointing to disk or on process replication.

Some efforts are underway to provide programming platforms for heterogeneous computing that can adapt to changing load. These efforts can be divided into two groups: those presenting new paradigms for parallel programming that facilitate fault tolerance/migration [1, 2, 8, 11], and migration tools based on consistent checkpointing [5, 27, 31]. They cannot handle processor failures or revocation due to availability, without checkpointing to a central disk.

8. Conclusions and Future Work

We have presented a new technique for executing certain scientific computations on a changing or faulty network of workstations (NOWs). This technique employs multiple checkpointing processors to adapt the algorithm-based diskless checkpointing to the matrix operations. It also enables a computation designed to execute on N processors to run on a NOW platform where individual processors may leave and enter the NOW because of failures or load. As long as the number of the processors in the NOW is greater than N, and as long as only one processor fails per group, the computation can proceed efficiently.

We have implemented this technique on the core matrix operations and shown performance results on a fast network of Sparc-5 workstations. This technique has been shown to improve not only the reliability of the computation but also the performance of the checkpointing and recovery. The results indicate that our technique can also obtain lower overhead with less amount of extra memory while checkpointing at smaller checkpointing intervals.

There are several more complicated schemes for configuring multiple checkpointing processors to tolerate more general sets of multiple failures. These schemes include *two-dimensional parity* and *multi-dimensional parity* [13], the *Reed-Solomon* coding scheme [21, 22], and *Evenodd parity* [3].

References

[1] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. DOME: Parallel programming in a distributed computing environment. April 1996.

[2] D. E. Bakken and R. D. Schilchting. Supporting fault-tolerant parallel programming in Linda. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. pages 245—254, April 1994.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[5] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[6] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, Vol. 5, pages 173–184, 1996.

[7] F. Cristian and F. Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.

[8] D. Cummings and L. Alkalaj. Checkpoint/rollback in a distributed system using coarse-grained dataflow. In *24th International Symposium on Fault-Tolerant Computing*, pages 424–433, June 1994.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[11] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage. pages 417–427, June 1992.

[12] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California, Berkeley, CA, December 1990.

[13] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson. Failure correction techniques for large disk arrays. pages 123–132, April 1989.

[14] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[15] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[16] Y. Kim. *Fault Tolerant Matrix Operations for Parallel and Distributed Systems*. PhD thesis, The University of Tennessee, Knoxville TN, August 1996.

[17] Y. Kim, J. S. Plank, and J. J. Dongarra. Fault tolerant matrix operations using checksum and reverse computation. In *The 6th Symposium of The Frontiers of Massively Parallel Computation*, pages 70–77, Annapolis MD, October 1996.

[18] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[19] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[20] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5:172–184, 1988.

[21] W. Peterson and E. J. Weldon. *Error-Correcting Codes*. MIT Press, Cambridge MA, second edition edition, 1972.

[22] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Technical Report UT-CS-96-332, University of Tennessee to appear in *Software Practice & Experience*, 1996.

[23] J. S. Plank, Y. Kim, and J. J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *The 25th International Symposium on Fault-Tolerant Computing*, pages 351–360, Pasadena, CA, June 1995.

[24] J. S. Plank, Y. Kim, and J. J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing (To appear)*, June 1997.

[25] J. S. Plank and K. Li. Faster checkpointing with $N + 1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[26] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[27] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. April 1995.

[28] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations. In *24th International Symposium on Fault-Tolerant Computing*, pages 38–47, Austin, TX, June 1994.

[29] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. *Portable Checkpointing and Recovery*. 1995.

[30] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, MA, 1996.

[31] G. Stellber. CoCheck: Checkpointing and process migration for MPI. April 1996.

[32] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.