Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations

James S. Plank[†] Youngbae Kim[†] Jack J. Dongarra[†‡]

[†]Department of Computer Science, University of Tennessee
[‡]Mathematical Science Section, Oak Ridge National Laboratory

Abstract

This paper is an exploration of diskless checkpointing for distributed scientific computations. With the widespread use of the "Network Of Workstation" (NOW) platform for distributed computing, long-running scientific computations need to tolerate the changing and often faulty nature of NOW environments. We present high-performance implementations of several algorithms for distributed scientific computing, including Cholesky factorization, LU factorization, QR factorization, and preconditioned conjugate gradient. These implementations are able to run on PVM networks of at least N processors, and can complete with low overhead as long as any N processors remain functional. We discuss the details of how the algorithms are tuned for fault-tolerance, and present the performance results on a PVM network of SUN workstations, and on the IBM SP2.

1 Introduction

For decades, scientific computation has been a driving force behind parallel and distributed computing. Traditionally such computations have been performed on the largest and most expensive supercomputers: the Cray C90, Intel Paragon, and Maspar MP-2. Recently the price and performance of uniprocessor workstations and off-the-shelf networking has improved to the point that networks of workstations (NOWs) provide a parallel processing platform that is competitive with the supercomputers. In fact, many new supercomputers like the Thinking Machines CM5, the IBM SP2, and the new SHRIMP multicomputer [BLA+94] bear a closer resemblance to NOWs than they do to their supercomputer ancestors. The popularity of NOW programming environments like PVM [GBD+94], and the availability of high-performance libraries for scientific computing on NOWs like ScaLAPACK [CDPW92] 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, and gone the next. Thus, on the wish list of scientific programmers is a way to perform computation on a NOW whose components may change over time due to failure or availability.

This paper provides a solution to this problem, especially tailored to the needs of scientific programmers. The solution is based on *diskless checkpointing*, a means of providing fault-tolerance without any dependence on disk. The end result is that as long as there are N processors available in the NOW (where N is defined by the user), and as long as processor failures come singly, the computation can progress reliably.

We describe our approach of incorporating diskless checkpointing in four subroutines from ScaLAPACK: Cholesky factorization, LU decomposition, QR factorization and Preconditioned Conjugate Gradient (PCG). It is subroutines like these that are the heart of scientific computations. We show the performance of these subroutines on two PVM networks of seventeen workstations in the absence of failures, and in the face of single processor failures.

The importance of this work is that it demonstrates a novel technique of executing high-performance scientific computations on a changing pool of resources.

2 Supercomputers vs. NOWs

The two main differences between supercomputers and NOWs are non-uniformity and transience. These mandate a different approach to fault tolerance on each platform. This is important because fault-tolerance on supercomputers is essentially a solved problem. In this section, we outline the differences between supercomputers and NOWs and their impact on fault-tolerance.

A supercomputer is a single computing resource. Each processor in a supercomputer is of the same type, and each runs a special operating system so that every node is a uniform part of the whole. A supercomputer is usually allocated exclusively for a single user, or, if it can be partitioned, then each partition is allocated exclusively. The file system is often implemented using special disks and processors at the periphery of the supercomputer so that files are uniformly available, regardless of which partition is being used. If one processor or part of the network fails, the whole computational platform is rendered useless until the faulty part is fixed.

For this reason, fault-tolerance in supercomputers is straightforward. *Consistent checkpointing* can be used to save the state of a parallel program. In consistent checkpointing, all processors cooperate to take a global checkpoint. This checkpoint is composed of uniprocessor checkpoints for every processor in the system, and a log of messages that are in transit during checkpointing. Many algorithms exist for taking consistent checkpoints [CL85, KT87, LNP92] and implementations have shown that the simplest of these, a two-phase commit called "Sync-and-stop" yields performance on par with the most complex [PL94b]. Checkpointing performance is dependent on the size of the individual checkpoints, the speed of the file system, and the amount of physical memory available for buffering [EJZ92, PL94b]. These conclusions are not likely to change as new supercomputers are released unless the model of exclusive node partitioning and wholesale partition failures is changed.

In contrast, a NOW is a distributed resource that is highly shared. Processors can be of different architectures, and are usually running a time-sharing operating system like Unix[1]. As such, the processors are never allocated exclusively, and more often than not are running numerous user-level processes, most of which are very short-lived or I/O bound (such as editors, shells window systems, etc). Thus, the processors have the capacity to execute CPU-bound programs, but in a less tightly-controlled manner than in supercomputers.

Each processor generally has its own local disk and access to other disks via a network file system like NFS. However, file availability and access speed varies from processor to processor. If processors are in different domains, then they may share no files; this requires binary and data files to be replicated across domains. In short, the consistent view to files provided by supercomputer file systems is lost.

Programs for NOWs are generally written using some NOW programming environment that provides process control, message passing, etc. The fault-tolerance of the program depends on the failure model of the programming environment. In early NOW programming environments, the user had to configure a "virtual machine" before running his or her program. Once the collection of processors comprising the machine was selected, processors could not be added or deleted, and individual processor failures rendered the entire system useless, as in a supercomputer. In recent NOW programming environments, like PVM version 3 [GBD+94] and Isis [BM89], the "virtual machine" can tolerate change. Processors can be added to or removed from the collection and failures are tolerated.

In such systems, consistent checkpointing is overkill. If one processor fails, the whole collection of processors must restart themselves from disk. Moreover, if the failed processor cannot be brought back online, then its checkpoint file will be unavailable unless it has been saved on a central file server which will then be a source of contention during checkpointing [LFS93]. Instead of consistent checkpointing, a more relaxed model of checkpointing is needed — one that is tailored to the dynamic nature of NOWs. We describe such a model in the next section.

3 A Model for Scientific Programs that Live on a NOW

Ideally, a scientific program executing on a NOW should be able to "live" on whatever pool of processors is currently available to that NOW. Processors should be able to leave the NOW whenever they fail or become too heavily loaded, and they should be able to join the NOW when they become functional and idle. We describe a model of scientific computation that approaches this ideal.

We are running a high-performance scientific program, like electromagnetic scattering or atomic structure calculation. The bulk of the work in such programs is composed of well known subproblems: solving partial differential equations and linear systems. These subproblems are typically solved using high-performance libraries, such as ScaLAPACK [CDPW92], which are designed to get maximum performance out of the computing platform. An important performance consideration is *domain decomposition*, which is how the problem is partitioned among the available processors to achieve optimal locality for minimizing cache

[1]Unix is a trademark of Unix Systems Laboratories

misses, and to minimize the effects of message transmission. To perform domain decomposition properly, the number of processors is usually fixed at some N, often a perfect square or power of two.

To retain high-performance, we assume that the program is optimized to run on exactly N processors. Our computing platform is assumed to be a NOW, which can contain any number of processors at any one time. Our model of computation for fault-tolerance is as follows.

Whenever the NOW contains at least N processors, the computation should be running on N of the processors. Whenever the NOW contains fewer than N processors, the computation is *swapped off* the NOW. This can be done by some sort of consistent checkpointing scheme that saves a global checkpoint to a central file server at very coarse intervals (for example once every hour). Such checkpointing schemes are straightforward and have been discussed and implemented elsewhere [KT87, EJZ92, LFS93, EZ94, PL94b].

Whenever the NOW contains *more than* N processors, then the computation should be running in such a manner that if any processor that is running the computation drops out of the NOW, due to failure or load, then it can be replaced *quickly* by another processor in the NOW. This is the important part of the computing model, because it means that as long as the pool of processors in the NOW numbers greater than N elements, then even if the pool itself changes, the computation should be progressing *efficiently*, while still maintaining fault-tolerance to wholesale failures.

4 The Checkpointing Algorithm

The algorithm is based on *diskless checkpointing* [PL94a]. If the program is executing on N processors, then there is a $N+1$-st processor called the *parity processor.* At all points in time, a consistent checkpoint is held in the N processors in memory. Moreover, the bitwise exclusive-or $\oplus$ of the N checkpoints is held in the parity processor. This is called the *parity checkpoint.* If any processor fails, then its state can be reconstructed on the parity processor as the exclusive-or of the parity checkpoint and the remaining $N-1$ processors' checkpoints.

Diskless checkpointing has been shown to be effective at providing fault-tolerance for single processor failures as long as there is enough memory to hold single checkpoints in memory. To reduce the memory requirements, incremental checkpointing can be used, and compression can be helpful in reducing the load on network bandwidth [PL94a].

To make checkpointing as efficient as possible, we implement algorithm-based checkpointing. In other words, rather than implement checkpointing transparently as in Fail-Safe PVM [LFS93], we hard-wire it into the program. This is beneficial for several reasons. First, the checkpointing can be placed at synchronization points in the program, which means that checkpoint consistency is not a worry. Second, the checkpointed state can be minimized because the checkpointer knows exactly what to save and how to reconstruct state. This is as opposed to a transparent checkpointer that must save all program state because it knows nothing about the program. Third, with transparent checkpointing, checkpoints are binary memory dumps, which rules out a heterogeneous recovery. With algorithm-based checkpointing, the recovery routines can plan for recovery by a different processor. In short, algorithm-based checkpointing is good because it enables the checkpointing to be as efficient as possible [LMJ93]. Its major drawback is that it requires the programmer to worry about and program for fault-tolerance. However, if the algorithms being checkpointed can be put into frequently-used library calls, then the extra work is justifiable [SVS94].

5 Checkpointing High-Performance Distributed Matrix Operations

We focus on two classes of matrix operations: direct, dense factorizations and an iterative equation solver. The matrix factorizations (Cholesky, LU and QR) are operations for solving systems of simultaneous linear equations and finding least squares solutions of linear systems. The iterative equation solver called Preconditioned Conjugate Gradient (PCG) is used to solve sparse systems of linear equations [Bar94]. All have been implemented in LAPACK [ABB+92] and ScaLAPACK [CDPW92], which are public-domain libraries providing high-performance implementations of linear algebra operations for uniprocessors and all kinds of parallel processing platforms.

We have implemented fault-tolerant versions of Cholesky, LU, QR, and PCG in ScaLAPACK. In the sections that follow, we provide an overview of how each operation works, and how we make it fault-tolerant. Further details on the ScaLAPACK implementations may be found in [GV89], [DDSv91], and [CDPW92].

5.1 Cholesky Factorization

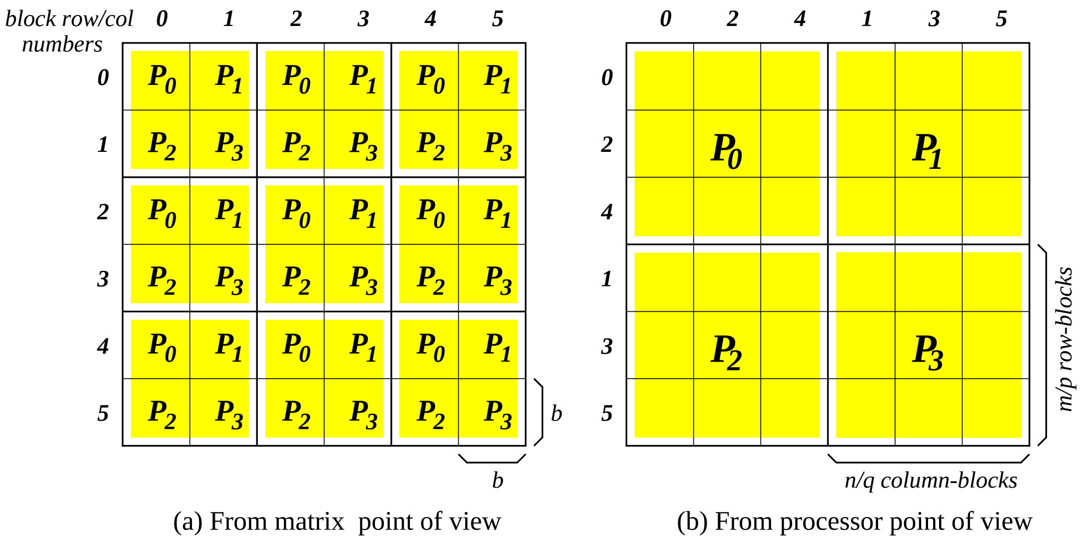
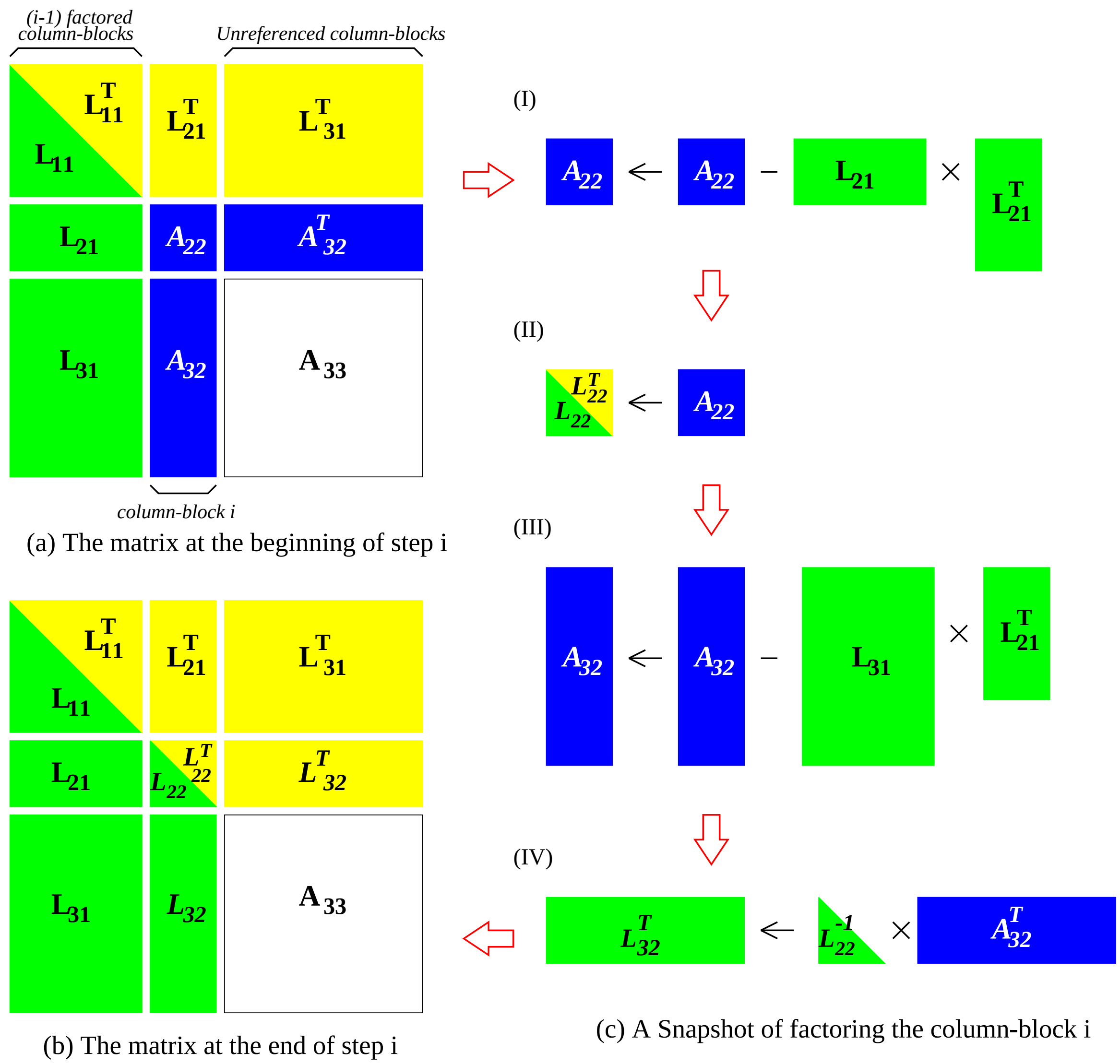


Figure 1: Data distribution of a matrix with 6×6 blocks over a 2×2 mesh of processors

Of the three factorizations implemented here, Cholesky is the simplest, and thus we describe it first. In Cholesky factorization, a dense, symmetric, positive definite matrix A is factored into two matrices L and L^T (i.e. $A = LL^T$) such that L is lower triangular. The algorithm for performing Cholesky factorization in ScaLAPACK is called "top-looking", and works as follows.

First, the 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 1. For obvious reasons, a row of blocks is called a "row-block" and a column of blocks is called a "column-block." If there are N processors, and A is an $m \times n$ matrix, then each processor holds m/p row-blocks and n/q column-blocks, where it is assumed that p and q divide m and n, respectively.

The factorization of A is performed in place. In other words, the matrix A is given as input to the factoring subroutine, and when the subroutine is finished, A has been replaced by L in its lower triangular half, and L^T in its upper triangular half.



Blocks to be updated at step i   Blocks to be referenced at step i

Figure 2: A snapshot of step i for Cholesky Factorization

The factorization is performed in steps, one for each column-block of the matrix. At the beginning of step i, the leftmost $i-1$ column-blocks are assumed to be factored, and the remaining column-blocks are unchanged. In step i, the i-th column-block gets factored. Thus, each step looks as in Figure 2. Inherent in this picture is the communication — for example, to perform $A_{22} \leftarrow A_{22} - L_{21} \times L_{21}^T$, all the involved blocks must be sent to the processor holding A_{22}. Note also that Figure 2 is also a logical representation of the system. Since A is symmetric, only half of the matrix needs to be stored.

The key fact to notice from Figure 2 is that at step i, only A_{22}, A_{32}, and A_{32}^T get modified. The rest of the blocks in the factorization remain the same. Since A_{32}^T is not stored by the system, this means that only column-block i is modified during step i.

To make the Cholesky factorization fault-tolerant, we first allocate a processor P_N. For each panel of N blocks in the matrix, there is one block in P_N containing the bitwise exclusive-or of each block in the panel. This is depicted in Figure 3 for the example system of Figure 1.

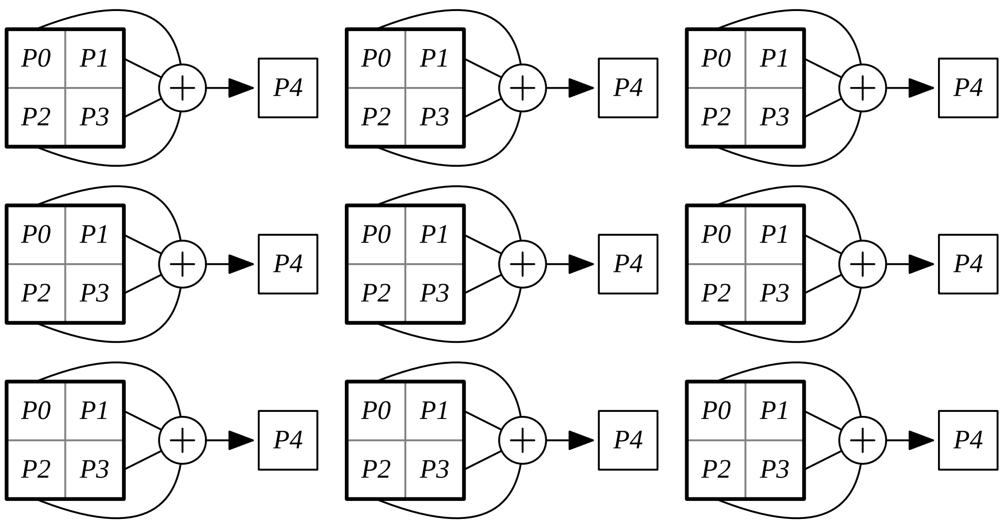
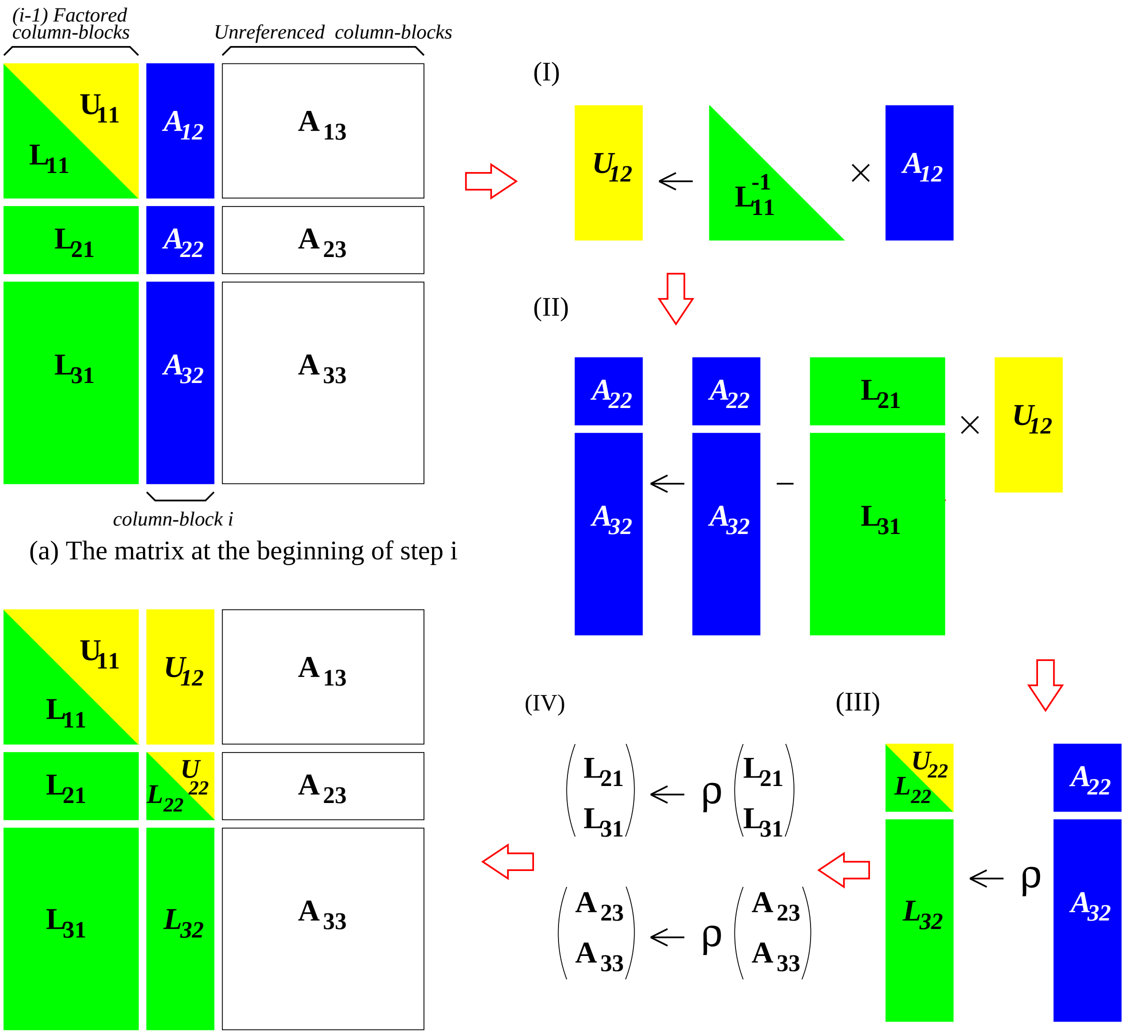


Figure 3: Partitioning the matrix for checkpointing

Moreover, each processor P_j (including P_N) allocates room for an extra column-block called CB_j. Now, the algorithm for performing fault-tolerant Cholesky Factorization is as follows:

- Initialize the global state of the system (including P_N).
- For each step i:
  - Let P_j be a processor with blocks in column-block i. P_j copies these blocks to CB_j.
  - P_N also copies its blocks corresponding to blocks in column-block i to CB_N.
  - The processors perform step i.
  - The processors P_j $(0 \le j < N)$ cooperate with P_N to update the exclusive-or for newly-modified blocks in column-block i.
  - The processors synchronize, and go to the next iteration.



(a) The matrix at the beginning of step i

(b) The matrix at the end of step i

(c) A snapshot of factoring the i-th column-block

Figure 4: A snapshot of step i for LU Factorization

Thus, at the beginning of each step, the processors hold the state of the factorization as depicted in Figure 3. If any one processor P_j fails, then it can be replaced by P_N, or by a new processor. This new processor calculates P_j's state from the bitwise exclusive-or of the remaining processors. Obviously, P_N can be replaced in a similar manner.

If any one processor P_j fails in the middle of a step, then the remaining processors can roll back to the beginning of the step by copying CB back to column-block i. Then P_j can be recovered as described in the preceding paragraph.

It is assumed here that failure detection is provided by the computing platform.

5.2 LU Factorization

In LU factorization, a dense matrix A is factored using a sequence of elementary eliminations with pivoting such that $\rho A = LU$, where L is a lower triangular matrix with 1's on the diagonal, U is an upper triangular matrix, and ρ is a permutation matrix. ρ is necessary for numerical stability — i.e. permuting rows of A to minimize the growth of roundoff error during the elimination. LU factorization involves a general non-symmetric matrix; and thus is computationally more complex than Cholesky factorization.

There are many different algorithm variants for implementing LU factorization on parallel machines. (See [DDSv91] for details.) These variants can be viewed as differing in which regions of data are being accessed and computed during each step. Below, we describe the "left-looking" variant and how it is checkpointed.

Like Cholesky factorization, LU factorization is performed in place, replacing A with L and U. Moreover, there is a permutation matrix ρ which is included as output to the subroutine. Since a permutation matrix is simply the identity matrix I with rows permuted, it may be represented by a one dimensional array, where the i^{th} entry contains the index of the non-zero element in row i of ρ. Like A, ρ is distributed among the processors. Each processor P_j contains its portion of ρ in ρ_j.

As before, the matrix is partitioned into blocks and distributed among the processors and the factorization proceeds in steps, one for each column-block in A. In step i, the i-th column-block is factored into its L and U components, and ρ is updated to reflect the pivoting. Each step looks as in Figure 4.

The key fact from Figure 4 is that at each step, only column-block i and the permutation matrix ρ, represented as a linear array, are modified. Therefore, the fault-tolerant version of LU factorization is much like that of Cholesky factorization, except there is this additional array ρ that must be checkpointed. To be specific, P_N starts as in Cholesky factorization, with blocks containing the exclusive-or of panels of blocks of A. Moreover, P_N has some memory ρ_N, which contains the bitwise exclusive-or of each ρ_j. Each processor P_j (including P_N) allocates room for an extra column-block, CB_j, and for a cache of ρ_j called ρ'_j. Now, the algorithm for performing fault-tolerant LU Factorization is as follows:

- Initialize the global state of the system (including P_N).
- For each step i:
  - Let P_j be a processor with blocks in column-block i. P_j copies these blocks to CB_j.
  - P_N copies its blocks corresponding to blocks in column-block i to CB_N.
  - All P_j $(0 \leq j \leq N)$ copy ρ_j to ρ'_j.
  - The processors perform step i.
  - The processors P_j $(0 \leq j < N)$ cooperate with P_N to update the exclusive-or for newly-modified blocks in column-block i, and for the newly modified ρ.
  - The processors synchronize, and go to the next iteration.

The fault-tolerant operation of the LU factorization is much like the fault-tolerant operation of Cholesky factorization. The only difference is that ρ must be copied and restored along with column-block i.

5.3 QR Factorization

In QR factorization a real $m \times n$ matrix A is factored such that

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where Q is an $m \times n$ orthogonal matrix and R an $n \times n$ upper triangular matrix. In the ScaLAPACK implementation of the QR factorization, the matrix Q is not generated explicitly since it would require too much extra memory. Instead, Q can be applied or manipulated through the identity $Q = I - VTV^T$, where V is a lower triangular matrix of "Householder" vectors, and T is an upper triangular matrix constructed from information in V. In the ScaLAPACK implementation, when the factorization is complete, the matrix A is transformed into V, T and R, where V is in the lower triangle of the two-dimensional array used to hold the original matrix A, R is in the upper triangle of the array used to hold the original matrix A, and T is stored in a one-dimensional array.

Complete details of the implementation of this algorithm are described in [GV89, DDSv91]. A high-level picture is provided in Figure 5.

It should be clear from Figure 5 that only column-block i of matrix A is changed during factoring step i. Therefore, the fault-tolerant version of QR works exactly like the fault-tolerant version of Cholesky — each processor P_j allocates an extra column-block CB_j to hold the initial value of column-block i during step i, so that the computation can be rolled back to the beginning of step i if there is a failure.

5.4 Iterative equation solver (PCG)

Iterative equation solvers are used for the following problem: Given a large sparse matrix A and a vector b, find the vector x such that $Ax = b$. Iterative equation solvers work by providing an initial approximation to x, and then iteratively refining this approximation until $Ax = b$ to within some error tolerance. Unfortunately, no single iterative method is robust enought to solve all sparse linear systems accurately and efficiently. Therefore, we limit our scope to one such method, known as "Preconditioned Conjugate Gradient" (PCG).

If A is positive definite symmetric, then PCG can be used to solve the system $Ax = b$ by projecting A onto a "Krylov subspace" and then solving the system in this subspace. The details of the algorithm are beyond the scope of this paper [GV89, DDSv91, Bar94]. However its mechanics as they impact fault-tolerance are simple. First, the sparse matrix A is represented in a dense form, and is then distributed along with b among the processors P_0 through P_{N-1}. After this point, A and b are not altered.

Now, the vectors p_0, r_0, w_0 and ξ_0 are calculated from A and b. These intermediate vectors are used to calculate the vector x_0, which is the first approximation to x. The algorithm then iterates as follows: The values of A, B, x_{i-1}, p_{i-1}, r_{i-1}, w_{i-1} and ξ_{i-1} are used to calculate p_i, r_i, w_i and ξ_i. These are then used to calculate x_i, the i-th approximation to x. The iterations continue until $Ax_i = b$ to within a given error tolerance.

Adding fault-tolerance to the PCG algorithm is straightforward. First, the processors distribute A, b, and allocate memory for x_i, p_i, r_i, w_i, and ξ_i. The extra processor P_N is

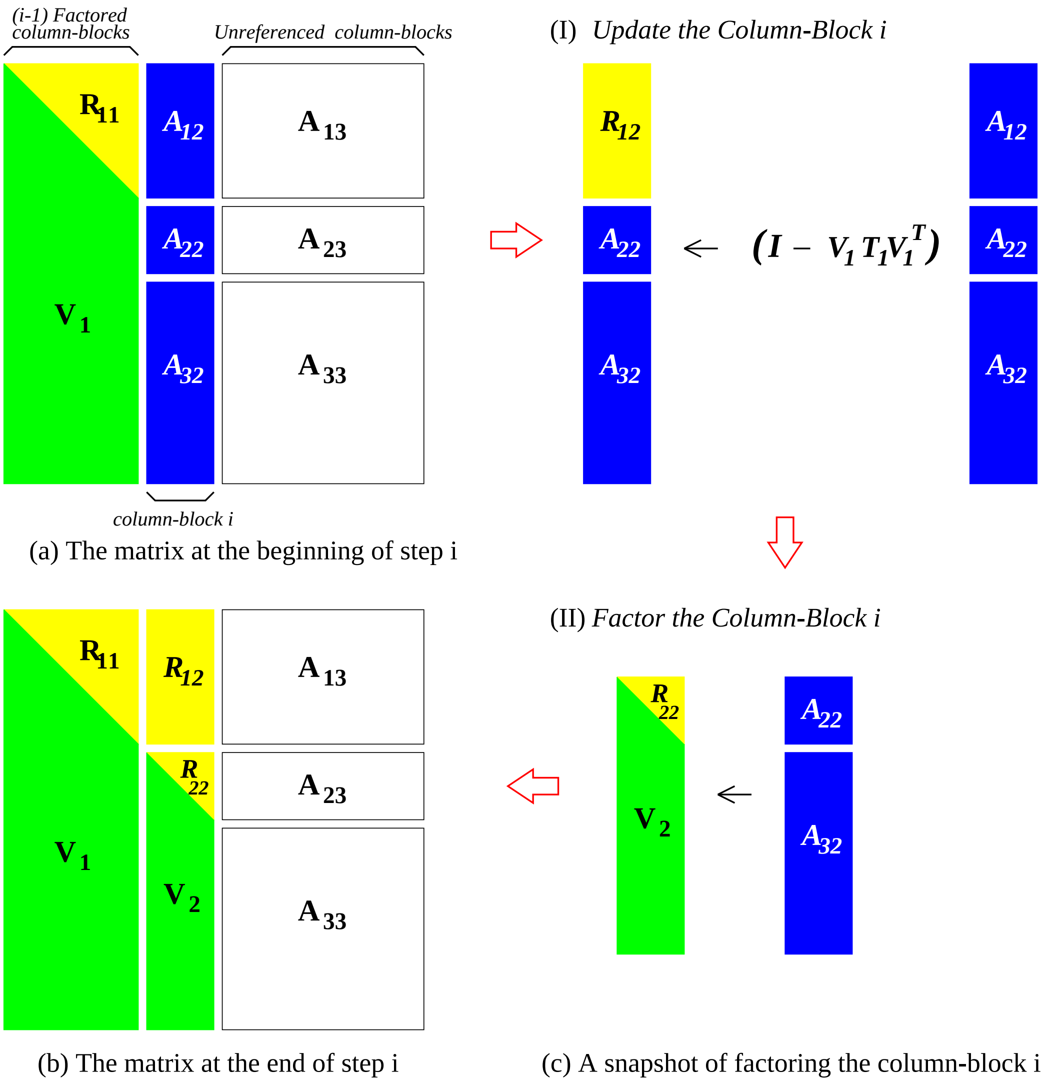


Figure 5: A snapshot of step i for QR Factorization

initialized to contain the bitwise exclusive-or of all these variables. Now, each processor (including P_N) must include extra vectors for each of x, p, r, w and ξ. These extra vectors are maintained like *CB* in the factorization examples. They hold the values of x_{i-1}, p_{i-1}, r_{i-1}, w_{i-1} and ξ_{i-1} during step i so that the step can be rolled back following a failure.

Note that in PCG, we can checkpoint every k steps, by copying x_i, p_i, r_i, w_i and ξ_i to the extra vectors and computing the bitwise exclusive-or of x, p, r, w and ξ only when i is a multiple of k. The result is that processors may roll back up to k steps upon a failure. However, since checkpoints are only taken every k steps, the overhead of checkpointing will be reduced by a factor of k.

Network of Sparc-2's:										
Matrix Size		Running Time	With Checkpointing							Recovery Time
n	Total Size (MBytes)	(sec)	# of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head		(sec)
1000	8	36	40	0.9	40	4	0.10	11.1		9
2000	32	171	80	2.1	188	17	0.21	9.9		33
3000	72	453	120	3.8	487	34	0.28	7.5		72
4000	128	934	160	5.8	994	60	0.38	6.4		130
5000	200	1666	200	8.3	1756	90	0.45	5.4		201

IBM SP2:									
Matrix Size		Running Time	With Checkpointing						Recovery Time
n	Total Size (MBytes)	(sec)	# of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head	(sec)
1000	8	4.5	40	0.1	6	1.5	0.04	33.3	1
2000	32	18.0	80	0.2	25	7.0	0.09	38.9	4
3000	72	40.8	120	0.3	56	15.2	0.13	37.3	8
4000	128	70.9	160	0.4	99	28.1	0.18	39.6	14
5000	200	119.6	200	0.6	157	37.4	0.19	31.3	22

Table 1: Results for Cholesky Factorization on a 17 processor system.

6 Implementation Results

We ran these programs on two computing platforms: a network of Sparc-2 workstations, and the IBM SP2 supercomputer. Both platforms support PVM [GBD+94].

The network of Sparc-2 workstations is a true NOW platform, consisting of 30 workstations connected via an ethernet in a terminal room. These workstations are generally allocated for undergraduate classwork, and thus are usually idle during the evening and busy executing I/O-bound and short CPU-bound jobs during the day. We ran our experiments on these machines at night when we could allocate them exclusively for our own use.

The IBM SP2 is a supercomputer that looks like a network of RS6000 processors. It supports the typical failure model of supercomputers — if one processor fails the supercomputer shuts down. However, in the absence of failures, its performance mirrors that of a high-performance NOW (i.e. a NOW with a faster network than an ethernet).

The results presented here are for a network of 17 processors, where 16 are running the program and one is calculating the parity. We ran three 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, and in the third, a processor failure is injected randomly to one of the processors, and the program completes with 16 processors. In the results that follow, we present only the time to perform the recovery, since there is no checkpointing after recovery.

6.1 Cholesky Factorization

We ran five different instances of the Cholesky Factorization, one for each of five matrix sizes. In each run, the block size was 25. The data for this experiment is in Table 1.

Network of Sparc-2's:										
Matrix Size		Running	With Checkpointing							Recovery
n	Total Size (MBytes)	Time (sec)	# of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head		Time (sec)
1000	8	332	20	16.6	340	8	0.40	2.4		24
2000	32	1085	40	27.1	1112	27	0.68	2.5		72
3000	72	2295	60	38.2	2363	68	1.13	3.0		148
4000	128	3992	80	49.9	4135	143	1.79	3.6		255
5000	200	6672	100	66.7	7209	537	5.37	8.0		420

IBM SP2:									
Matrix Size		Running	With Checkpointing						Recovery
n	Total Size (MBytes)	Time (sec)	# of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head	Time (sec)
1000	8	100	20	5.0	102	2.0	0.10	2.0	5
2000	32	291	40	7.3	296	5.0	0.12	1.7	9
3000	72	558	60	9.3	568	10.0	0.17	1.8	15
4000	128	923	80	11.5	941	18.0	0.23	2.0	26
5000	200	1378	100	13.8	1409	31.0	0.31	2.2	43

Table 2: Results for LU Factorization on a 17 processor system.

Cholesky Factorization is an $O(n^3)$ algorithm. Since each checkpoint consists of a column-block's worth of data `XOR`'d together, checkpointing is $O(n \log N)$. For a fixed number of processors, this is simply $O(n)$. The overhead of checkpointing depends on the overhead of individual checkpoints multiplied by the total number of checkpoints. The total number of checkpoints is n/b (where b is the block size), which is $O(n)$. This means that the total overhead of checkpointing is $O(n^2)$. Thus we expect the percentage overhead of checkpointing to decrease as n increases — this is corroborated in our experiment.

Recovery consists of taking the bitwise exclusive-or of every processor's matrix A. Thus, recovery is $O(n^2 \log N) = O(n^2)$. This too is reflected in the data. Notice that the time it takes to recover is irrespective of the location of the failure.

An interesting thing to notice is that the block size has little impact on the overhead of checkpointing. This is because the same total number of bytes ($O(n^2)$) get checkpointed during the lifetime of the computation.

6.2 LU and QR Factorization

The results from the LU and QR factorizations are in Tables 2 and 3. They are very similar to the results from the Cholesky factorizations; however, more computation is performed, meaning that individual factoring iterations are longer, and thus checkpointing is a smaller percentage of the overall runtime. Note that the recovery times for LU and QR are roughly twice those for Cholesky. This is because the matrix A is symmetric in Cholesky factorization, and therefore only half of it needs to be checkpointed.

Network of Sparc-2's:										
Matrix Size		Running	With Checkpointing							Recovery
	Total	Time	# of	Ckp	Running	Total	Overhead	Over-		Time
	Size		Check-	Interval	Time	Overhead	per ckp	head		
n	(MBytes)	(sec)	points	(sec)	(sec)	(sec)	(sec)			(sec)
1000	8	845	20	42.2	894	49	2.45	5.8		20
2000	32	3492	40	87.3	3566	74	1.85	2.1		64
3000	72	9479	60	158.0	9578	99	1.65	1.0		156
4000	128	20201	80	252.5	20357	156	1.95	0.8		259
5000	200	37278	100	372.8	37666	388	3.88	1.0		439

IBM SP2:									
Matrix Size		Running	With Checkpointing						Recovery
	Total	Time	# of	Ckp	Running	Total	Overhead	Over-	Time
	Size		Check-	Interval	Time	Overhead	per ckp	head	
n	(MBytes)	(sec)	points	(sec)	(sec)	(sec)	(sec)		(sec)
1000	8	231	20	11.6	241	10.0	0.50	4.3	2
2000	32	642	40	16.1	657	15.0	0.38	2.3	6
3000	72	1362	60	22.7	1386	24.0	0.40	1.8	15
4000	128	2512	80	31.4	2546	34.0	0.42	1.4	25
5000	200	4261	100	42.6	4302	41.0	0.41	1.0	37

Table 3: Results for QR Factorization on a 17 processor system.

6.3 Preconditioned Conjugate Gradient

We executed two instances of PCG: one with a 262144×262144 matrix A for 2000 iterations on the network of Sparc-2's, and one with a 1048576×1048576 matrix A for 5000 iterations on the SP2. These calculated x to within a tolerance of 10^{-7} and 10^{-10} respectively. The results of these instances with varying values of k is in Table 4.

As stated in Section 5.4, the saved state of the PCG program consists of two parts — the unchanging part (A and b), and the part that is checkpointed every k iterations (x, p, r, w and ξ). The dense representation of A is a $5 \times n$ matrix, where $n = 262144$ or 1048576. Therefore, we expect the initial checkpointing step and each subsequent checkpointing step to be roughly equal in duration. Table 4 corroborates this expectation — from columns three and six, we calculate that the initial checkpointing of A and b takes approximately 38 seconds on the Sparc-2's and 30 seconds on the SP2, and each checkpoint of x, p, r, w and ξ takes approximately 25 seconds on the Sparc-2's and 51 seconds on the SP2.

7 Discussion

7.1 The Results

The results presented in the previous section show that on current NOWs, the performance of this method for fault-tolerant computation is very good. In the matrix factorizations on the Network of Sparc-2's, the overhead of checkpointing is with only one exception under 10%. The lone case with overhead above 10% is the $n = 1000$ instance of Cholesky factorization, whose short running time (36 seconds) would preclude the necessity for fault tolerance.

On the SP2, the fault-tolerant Cholesky factorizations exhibit high overhead (40%). This can be attributed to the fact that the fast processors and networking of the SP2 allow each

Network of Sparc-2's:								
k	Running Time (sec)	With Checkpointing # of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head %	Recovery Time (sec)
10	2269	200	12	7309	5040	25.2	222	51
50	2269	40	58	3316	1047	26.2	46.1	53
100	2269	20	116	2807	538	26.9	23.7	53
500	2269	4	581	2406	137	34.3	6.0	54
1000	2269	2	1162	2355	86	43	3.8	55
2000	2269	1	2323	2330	61	61	2.7	56

IBM SP2:								
k	Running Time (sec)	With Checkpointing # of Check-points	Ckp Interval (sec)	Running Time (sec)	Total Overhead (sec)	Overhead per ckp (sec)	Over-head %	Recovery Time (sec)
50	2362	100	24	7548	5186	51.9	220	29
100	2362	50	47	4967	2605	52.1	110	29
500	2362	10	236	2899	523	52.3	22.1	30
1000	2362	5	472	2640	278	55.6	11.7	30
5000	2362	1	2362	2447	85	85	3.5	30

Table 4: Results for PCG on on a 17 processor system.

iteration of the factorization to run very fast with respect to the taking of each checkpoint. The SP2 executes the factorizations so quickly (under two minutes for $n = 5000$) that fault-tolerance is not really necessary. As n grows larger still and the running times get into the range where fault-tolerance is desirable, the overhead of checkpointing will decrease (see Section 6.1) to more reasonable values. The other factorizations on the SP2 have a greater ratio of computation to checkpointing, and therefore exhibit very nice checkpointing behavior.

In the PCG implementation, whenever k is greater than 265 on the Sparc-2's, and 1250 on the SP2, the overhead of checkpointing should be less than 10% (this value of k is obtained using the checkpoint times stated in Section 6.3). This means that checkpoints can be taken every five minutes on the network of Sparc-2's, and every ten minutes on the SP-2. Both are reasonable checkpointing intervals.

7.2 Extra Parity Processors

The choice of one parity processor P_N was made simply to present the concept of diskless checkpointing. If the NOW executing the computation contains $N + m$ processors, then there is no reason why $m - 1$ of them should be idle. Instead of having all N processors checkpoint to P_N, we can partition the N processors into m groups $G_0, \ldots G_{m-1}$, and have P_{N+j} be responsible for checkpointing the processors in G_j, for $0 \leq j < m$. This is basically a 1-dimensional parity scheme, which can tolerate up to m simultaneous processor failures, as long as each failure occurs in a different group [GHK+89].

The extreme we have presented is $m = 1$. At the other extreme are systems like Isis [BJ89, BM89] or Targon [BBG+89] where $m = N$, and every processor has a backup processor to which it sends checkpoints. As m grows, the overhead of checkpointing and recovery will decrease, since the recovery group size is smaller, which necessitates less information to be combined at each parity processor.

To reliably tolerate *any* combination of multiple processor failures, extra parity processors must be combined with more sophisticated error-correction techniques [BM93, BBBM94, PFL94]. This means that every processor's checkpoint must be sent to multiple parity processors. In the absence of broadcast hardware, this kind of fault-tolerance will likely impose too great an overhead.

8 Related Work

Checkpointing supercomputers and distributed systems has been studied and implemented by many people [BBG83, TS84, CL85, SY85, LY87, BBG+89, BJ89, SW89, JZ90, CJ91, LRG91, EZ92, KMBT92, WF92, LFS93, LMJ93, PK93, VJ93, WF93, XN93, CA94, EZ94, LNP94, PL94b, SVS94, Vai94]. All of this work however focuses on either checkpointing to disk, or process replication. The technique of using a collection of extra processors to provide fault-tolerance with no reliance on disk comes from Plank and Li [PL94a] and is unique to this work.

There are efforts to provide programming platforms for heterogeneous computing which can adapt to changing load [Gel86, GK92, BDG+93, BSS94, CA94]. In all of these however, the programmer must make his or her program conform to the programming model of the platform. None are garden variety message-passing environments like PVM.

9 Conclusions

We have given a method for executing certain scientific computations on a changing Network of Workstations. This method enables a computation designed to execute on N processors to run on a NOW platform, where individual processors may leave and enter the NOW due to failures or load. As long as the number of processors in the NOW numbers greater than N, and as long as processors leave the NOW singly, the computation can proceed efficiently.

We have implemented this method on four scientific calculations and shown performance results on two parallel testbeds: a network of Sparc-2 workstations connected by an ethernet, and the IBM SP2. The results show that our methods exhibit low overhead while checkpointing at a relatively fine-grained intervals (in most cases less than 5 minutes).

Our future goals for this work are threefold. First, we are adding the ability for processors to join the NOW in the middle of a calculation, and participate in the fault-tolerant operation of the program. Currently, once a processor quits, the system merely completes with exactly N processors and no checkpointing. Second, we are adding the capacity for multiple parity processors as outlined in Section 7.2. This should improve both the reliability of the computation and the performance of checkpointing.

Finally, we would like to integrate this scheme with general load-balancing. In other words, if a few processors are added to or deleted from the NOW, then the system continues running using the mechanisms outlined in this paper. However, if the size of the processor pool changes by an order of magnitude, then it makes sense to reconfigure the system with a different value of N. Such an integration would represent a truly adaptive, high-performance methodology for scientific computations on NOWs.

Acknowledgements

The authors thank the following people for their helpful comments and suggestions: Richard Barrett, Micah Beck, Jaeyoung Choi, Kai Li, Bob Manchek and Clint Whaley. We also thank the system administrators in Department of Computer Science at University of Tennessee.

This research was performed in part using the SP2 operated by the Maui High Performance Computing Center. We thank Lon Walters for his help allowing us to access their machine.

James Plank is supported by National Science Foundation grant CCR-9409496. 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 No. CCR-8809615.

References

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 1992.

[Bar94] R. Barrett *et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Press, Philadelphia, Pennsylvania, 1994.

[BBBM94] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual International Symposium on Computer Architecture*, pages 245—254, Chicago, IL, April 1994.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, GA, October 1983.

[BBG+89] 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.

[BDG+93] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. Heterogeneous network computing. In *Sixth SIAM Conference on Parallel Processing*, 1993.

[BJ89] K. P. Birman and T. A. Joseph. Exploiting replication in distributed systems. In S. Mullender, editor, *Distributed Systems*. ACM Press, Addison-Wesley, New York, 1989.

[BLA+94] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *The 21st Annual International Symposium on Computer Architecture*, Apr 1994.

[BM89] K. P. Birman and K. Marzullo. ISIS and the meta project. *Sun Technology*, Summer 1989.

[BM93] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, Toulouse, France, June 1993.

[BSS94] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, May 1994.

[CA94] 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, Austin, TX, June 1994.

[CDPW92] J. Choi, J. Dongarra, R. Pozo, and D. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Publishers, 1992.

[CJ91] 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.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):3–75, February 1985.

[DDSv91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, Pennsylvania, 1991.

[EJZ92] 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.

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in Manetho. In *22nd Annual International Symposium on Fault-Tolerant Computing*, 1992.

[EZ94] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, Austin, TX, June 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. *PVM — A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Boston, 1994.

[Gel86] D. Gelernter. Domesticating parallelism. *IEEE Computer*, pages 12–16, Aug 1986.

[GHK+89] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, April 1989.

[GK92] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *International Conference on Supercomputing*, pages 417–427, Washington, D.C., June 1992. ACM.

[GV89] G. H. Golub and C. V. Van Loan. *Matrix Computations, 2nd ed.* The Johns Hopkins University Press, Baltimore, Mayland, 1989.

[JZ90] 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.

[KMBT92] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[LFS93] J. León, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[LMJ93] L. A. Laranjeira, M. Malek, and R. M. Jenevein. Space/time overhead analysis and experiments with techniques for fault tolerance. *Dependable Computing and Fault-Tolerant Systems*, 8(3):303–318, 1993.

[LNP92] K. Li, J. F. Naughton, and J. S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Processing*, 20(3):159–180, June 1992.

[LNP94] 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.

[LRG91] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic failure recovery for very large networks. In *10th Symposium on Reliable Distributed Systems*, pages 66–75, October 1991.

[LY87] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[PFL94] J. S. Plank, J. Friedman, and K. Li. A failure correction technique for parallel storage devices with minimal device overhead. Technical Report CS-94-243, University of Tennessee, August 1994.

[PK93] S. L. Peterson and P. Kearns. Rollback based on vector time. In *12th Symposium on Reliable Distributed Systems*, October 1993.

[PL94a] 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.

[PL94b] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[SVS94] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High Performance Computing Conference*, pages 694–701, Knoxville, TN, May 1994.

[SW89] A. P. Sistla and J. Welch. Efficient distributed recovery message logging. In *Principles of Distributed Computing*, 1989.

[SY85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[TS84] Y. Tamir and C. H. Séquin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, 1984.

[Vai94] N. H. Vaidya. Consistent logical checkpointing. Technical Report 94-051, Texas A&M University, July 1994.

[VJ93] S. Venkatesan and T. Juang. Optimistic crash recovery without rolling back non-faulty processors. *Information Sciences*, 1993.

[WF92] Y. M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *11th Symposium on Reliable Distributed Systems*, pages 147–154, October 1992.

[WF93] Y. M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *12th Symposium on Reliable Distributed Systems*, pages 78–85, October 1993.

[XN93] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1993.