JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **43,** 125–138 (1997)
ARTICLE NO. PC971336

Fault-Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing

James S. Plank,* Youngbae Kim,† and Jack J. Dongarra*,‡

**Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996; †National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, University of California, Berkeley, California 94720; and ‡Mathematical Science Section, Oak Ridge National Laboratory, P.O. Box 2008, Building 6012, Oak Ridge, Tennessee 37821-6367*

Networks of workstations (NOWs) offer a cost-effective platform for high-performance, long-running parallel computations. However, these computations must be able to tolerate the changing and often faulty nature of NOW environments. We present high-performance implementations of several fault-tolerant algorithms for distributed scientific computing. The fault-tolerance is based on diskless checkpointing, a paradigm that uses processor redundancy rather than stable storage as the fault-tolerant medium. These algorithms are able to run on clusters of workstations that change over time due to failure, load, or availability. As long as there are at least *n* processors in the cluster, and failures occur singly, the computation will complete in an efficient manner. 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 connected by a fast, switched ethernet. © 1997 Academic Press

1. INTRODUCTION

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. The popularity of NOW programming environments such as PVM [19] and MPI [34, 42] and the availability of high-performance libraries for scientific computing on NOWs like ScaLAPACK [13] 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 due to failure, load, or ownership. We term any such event a failure. Thus, on the wish list of scientific programmers is a way to perform computation on a NOW whose components may change over time.

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 failures come singly, the computation can progress reliably.

We describe our approach of incorporating diskless checkpointing into four well-known algorithms in linear algebra: Cholesky factorization, LU factorization, QR factorization, and Preconditioned Conjugate Gradient (PCG) [4, 16]. Subroutines such as these at are the heart of scientific computation. We show the performance of these subroutines on a cluster of 17 Sun Sparc5 workstations connected by a fast (100 megabit) switched ethernet.

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

2. SUPERCOMPUTERS VS NOWS

A supercomputer is a single computing resource. We usually think of each processor in a supercomputer as being identical—every node is a uniform part of the whole. Typically, a supercomputer is allocated exclusively for a single application, such as a grand challenge. 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 the partition 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 to stable storage. In consistent checkpointing, all processors cooperate to save 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 [12, 26, 30] and implementations have shown that the simplest of these, a two-phase commit called “Sync-and-stop,” yields performance on a par with the most complex [38]. 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 [17, 38]. These conclusions are not likely to change as new machines are released unless the model of exclusive node partitioning and wholesale partition failures is changed.

In contrast, a NOW is a distributed computing resource that is highly shared. Processors usually run a general-purpose time-sharing operating system, and each is often owned by a different user. Although the processing capacity of the NOW as a whole may be consistently large, individual processors can run the gamut from idle to unavailable (e.g., in use by the owner) back to idle in a relatively small time frame [35]. Programs for NOWs are generally written using some NOW programming environment such as PVM or MPI that provides convenient primitives for message passing. Such programming environments allow individual processors to enter or leave the NOW dynamically due to availability, load, or failure. We term all such events failures. Thus, NOWs present a far more flexible failure model than supercomputers.

In such systems, consistent checkpointing to disk is overkill. If one processor becomes unavailable, the whole collection of processors must restart themselves from stable storage. 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 [36]. Therefore, a more relaxed model of checkpointing is needed—one that is tailored to the dynamic nature of NOWs.

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. Processors should be able to leave the NOW whenever they fail, and they should be able to join the NOW when they become functional. We describe a model of scientific computation that approaches this ideal.

We assume that we are running a high-performance scientific program, such as 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 [13], 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 minimize cache misses and 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 2.

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 a consistent checkpointing scheme that saves a global checkpoint to a central file server at very coarse intervals (for example, once every hour or day). Such checkpointing schemes are straightforward and have been discussed and implemented elsewhere [11, 17, 18, 26, 29, 36, 38, 43].

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, load, or ownership, 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 more than n members, then even if the pool itself changes, the computation should be progressing efficiently, while still maintaining fault-tolerance to wholesale failures.

If a processor fails but is still available in a limited capacity (for example, due to high load or some forms of ownership revocation), then its process should be migrated to a free processor. Migration systems are efficient and straightforward and have been implemented for popular programming environments like PVM and MPI [11, 43]. However, if a processor fails completely and its resources are totally unavailable, then migration strategies do not work.

In their survey of internet host reliability, Long *et al.* measured a mean time to failure of 12.99 days for an average workstation [32]. Assuming independent failures, this means that the MTTF of a collection of 16 workstations is 19.49 h, which is significantly small. The algorithm described in this paper focuses on this failure scenario. It is designed to recover quickly from single processor failures where the state of the processor is unavailable to the network following a failure.

Note that failure identification may be provided by monitoring tools such as CARMI [39], which can classify failures into the proper category for efficient recovery.

4. THE CHECKPOINTING ALGORITHM

The algorithm is based on *diskless checkpointing* [37]. If the program is executing on n processors, then there is an $(n + 1)$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 [37].

To make checkpointing as efficient as possible, we implement algorithm-based checkpointing. In other words, rather than implement checkpointing transparently as in MIST [11], Fail-Safe PVM [29], or CoCheck [43], we hardwire 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 (defining network state [12]) 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 kind of processor. In short, algorithm-based checkpointing is good because it enables the checkpointing to be as efficient as possible [28]. Its major drawback is programmer effort, since the fault-tolerance must be incorporated carefully into the program. However, if the algorithms being checkpointed can be put into frequently used library calls, then the extra work is justifiable [41].

It should be noted that this checkpointing algorithm can be viewed as a highly optimized application of consistent checkpointing that tailors the checkpointing to tolerate single-processor failures with low overhead. This performance optimization is achieved by a combination of application-based incremental checkpointing, parity redundancy, and no reliance on stable storage.

5. CHECKPOINTING HIGH-PERFORMANCE DISTRIBUTED MATRIX OPERATIONS

We focus on two classes of matrix operations: direct, dense factorizations and an iterative equation solver. The factorizations (Cholesky, LU, and QR) are operations for solving systems of simultaneous linear equations and finding least squares solutions of linear systems. All have been implemented in LAPACK [1] and ScaLAPACK [13], which are public-domain libraries providing high-performance implementations of linear algebra operations for uniprocessors and all kinds of parallel processing platforms. The iterative equation solver called Preconditioned Conjugate Gradient (PCG) is a well-known technique for solving sparse systems of linear equations [4].

We have implemented fault-tolerant versions of Cholesky, LU, QR, and PCG. 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 books by Dongarra [16] and Golub [22].

5.1. Cholesky Factorization and the Basic Checkpointing Scheme

Of the three factorizations, Cholesky is the simplest. 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 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 Fig. 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 $N \times N$ matrix, then each processor holds N/bp row-blocks and N/bq column-blocks, where it is assumed that b, p, and q divide N.

The factorization of A is performed in place, and proceeds in N/b 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 ith column-block gets factored using a multiplication, subtraction, and factorization.

Thus, each step appears as in Fig. 2. Inherent in this picture is communication—for example, to perform $\hat{A}_{22} \leftarrow A_{22} - L_{21}L_{12}^T$, all the involved blocks must be sent to the processor holding A_{22}. Note also that Fig. 2 is a logical representation of the system. Since A is symmetric and L^T is the transpose of L, only half of A and none of L^T need be stored.

The key fact to notice from Fig. 2 is that at step i, only A_{22} and A_{32} get modified. The rest of the blocks in the factorization remain the same. Thus, only blocks from column-block i are modified during step i.

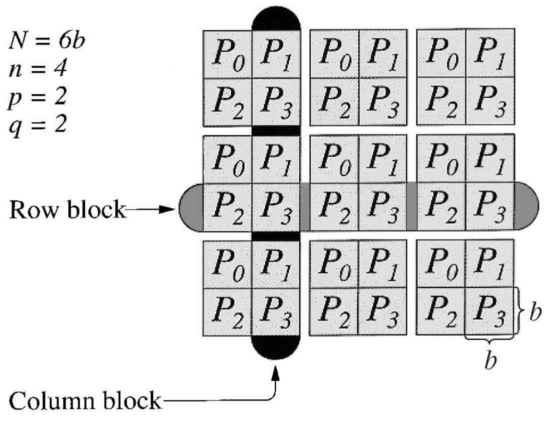
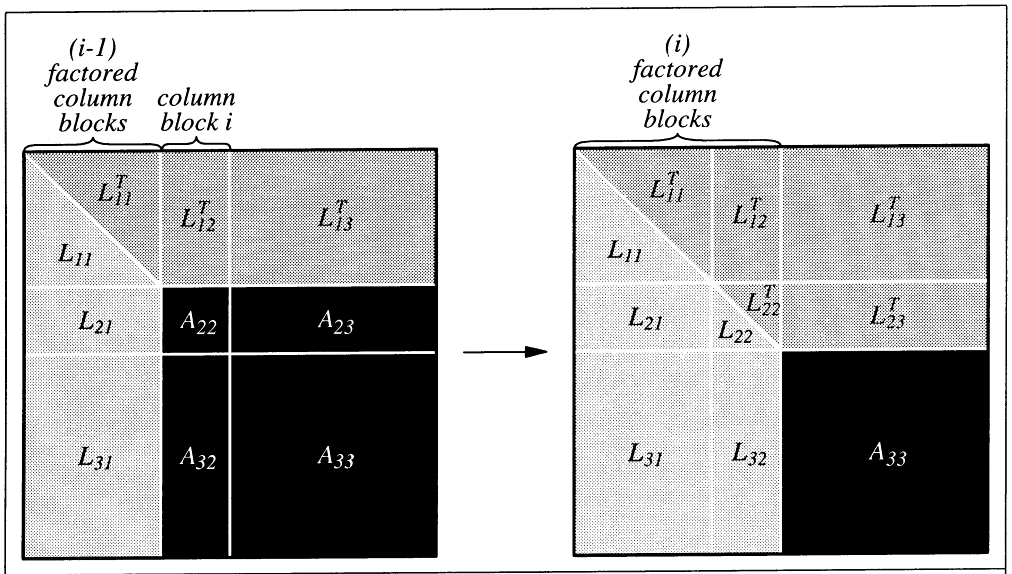


FIG. 1. Data distribution of a matrix with 6 × 6 blocks over a 2 × 2 mesh of processors.



(I – Multiplication/Subtraction)

$$\begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} L_{12}^T$$

(II – Factorization)

$$\begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} L_{22}^T$$

FIG. 2. Step *i* for Cholesky factorization.

To make the Cholesky factorization fault-tolerant, we first allocate a parity 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 Fig. 3 for the example system of Fig. 1.

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.
- For each step i:

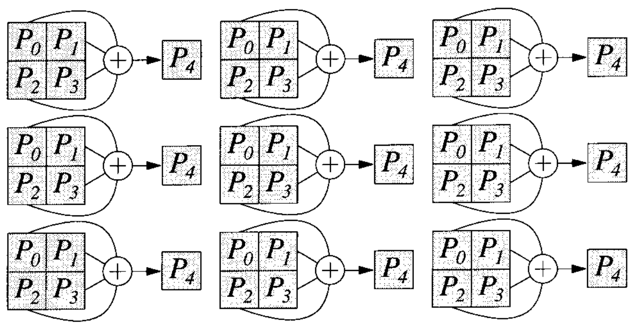


FIG. 3. Configuring the system for checkpointing.

— 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 $(0 \leq j < n)$ cooperate with P_n to update the exclusive-or for the newly modified blocks in column-block i.

— The processors synchronize, and go to step $i + 1$.

Thus, at the beginning of each step, the processors hold the state of the factorization as depicted in Fig. 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. For example, PVM detects processor and certain network failures, and a resource manager like CARMI [39] can be added to PVM to detect failures due to load and ownership.

5.2. *LU Factorization*

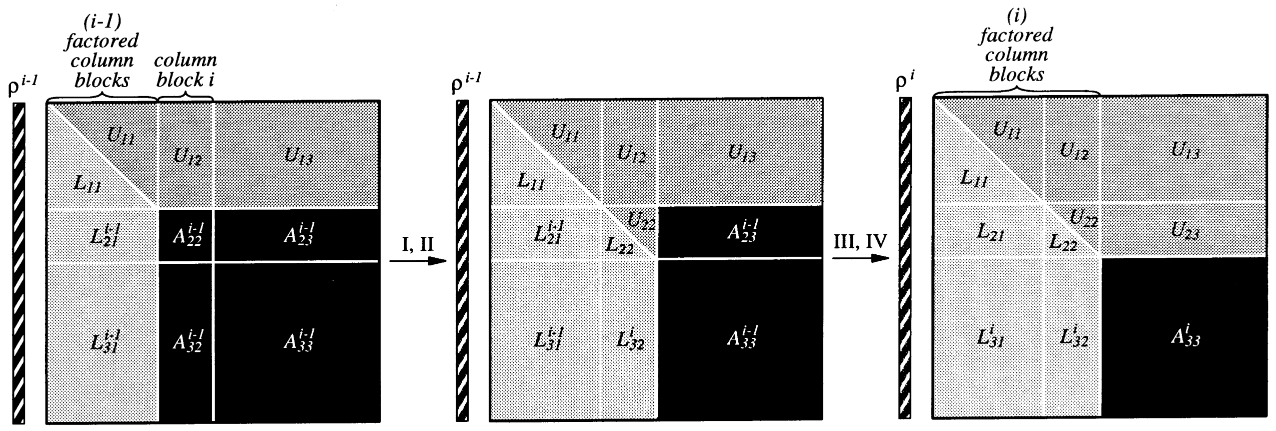
In LU factorization, a dense matrix A is factored using a sequence of elementary eliminations with pivoting such that $\rho A = LU$, where L is a lower triangular matrix with ones on the diagonal and U is an upper triangular matrix. ρ is a permutation matrix necessary for numerical stability: a proper permutation of the rows of A minimizes the growth of roundoff error during the elimination. LU factorization involves a general non-symmetric matrix, and is computationally more complex than Cholesky factorization.

There are three well-known algorithm variants for implementing LU factorization on parallel machines: left-looking, right-looking, and Crout. These variants differ in the regions of data that are accessed and computed during each step (see [16] for details). Below, we describe the Crout variant and how it is checkpointed. We discuss the ramifications of algorithm selection and checkpointing performance in Section 7.3.

Like Cholesky factorization, LU factorization is performed in place, replacing A with L and U. Moreover, the permutation matrix ρ is generated as output from 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 ith entry contains the index of the nonzero 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. The factorization proceeds in steps, one for each column/row block in A. In step i, the ith column-block is factored, and the result of this factoring is used to factor the ith row-block. The details are in Fig. 4.

The memory update patterns in Crout LU are more complex than in Cholesky factorization. In step i, both column-block i and row-block i are modified. Moreover, the permutation matrix ρ is altered, and at most b rows in L_{31} and A_{33} are swapped with rows in row-block i due to pivoting. Thus, the



(I – Multiplication/Subtraction)
$$\begin{pmatrix} A^i_{22} \\ A^i_{32} \end{pmatrix} \leftarrow \begin{pmatrix} A^{i-1}_{22} \\ A^{i-1}_{32} \end{pmatrix} - \begin{pmatrix} L^{i-1}_{21} \\ L^{i-1}_{31} \end{pmatrix} U_{12}$$

(II – Factorization / Definition of $\hat{\rho}$)
$$\hat{\rho} \begin{pmatrix} A^i_{22} \\ A^i_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L^i_{32} \end{pmatrix} U_{22}$$

(III – Pivoting)
$$\begin{pmatrix} L^i_{21} \\ L^i_{31} \end{pmatrix} \leftarrow \hat{\rho} \begin{pmatrix} L^{i-1}_{21} \\ L^{i-1}_{31} \end{pmatrix}, \quad \begin{pmatrix} A^i_{23} \\ A^i_{33} \end{pmatrix} \leftarrow \hat{\rho} \begin{pmatrix} A^{i-1}_{23} \\ A^{i-1}_{33} \end{pmatrix}, \quad \rho^i \leftarrow \hat{\rho}\rho^{i-1}$$

(IV – Solve/Multiplication/Subtraction)
$$U_{23} \leftarrow \left(L^i_{32}\right)^{-1} A^i_{23} - L_{21} U_{12}$$

FIG. 4. Step i for Crout LU factorization.

algorithm to make Crout LU fault-tolerant, though similar to Cholesky factorization, is necessarily more complex.

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 processor's ρ_j. Each processor P_j (including P_n) allocates room for an extra column-block, CB_j, an extra row-block, RB_j, and a cache of ρ_j called ρ'_j. Finally, each processor P_j (including P_n) allocates room for a row-block's worth of pivoting rows PR_j. The fault-tolerant LU factorization proceeds 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.
  - — Let P_j be a processor with blocks in row-block i. P_j copies these blocks to RB_j.
  - — P_n copies its blocks corresponding to blocks in column-block i and row-block i to CB_n and RB_n.
  - — All P_j $(0 \le j \le n)$ copy ρ_j to ρ'_j.
  - — The processors perform substeps I and II of step i.
  - — In substep III, b rows of the matrix are swapped with rows in row-block i. Before doing so, the processors P_j that own these rows copy them to PR_j. P_n copies its rows corresponding to these rows to PR_n.
  - — Now the processors perform substeps III and IV.
  - — The processors P_j $(0 \le j < n)$ cooperate with P_n to update the exclusive-or for the newly modified blocks in column-block i, row-block i, the swapped pivot rows, and ρ.
  - — The processors synchronize and go to step $i + 1$.

As in Cholesky factorization, if a processor P_j fails during step i of the computation it can be replaced by P_n, or by a new processor. The replacement proceeds as follows:

- For all remaining processors P_k (this includes P_n), if P_k had started substep III, then it copies any rows back from PR_k to their original position.
- All P_k copy their data from CB_k, RB_k, and ρ'_k back to column-block i, row-block i, and ρ_j, respectively.
- P_j's state is reconstructed from the bitwise exclusive-or of the blocks in the other P_k.
- The computation proceeds from the beginning of step i.

5.3. *QR Factorization*

In QR factorization a real $M \times N$ matrix A is factored so 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 QR factorization, the matrix Q is not generated explicitly since it would require too much extra storage. Instead, Q can be applied or manipulated through the identity $Q = I - VTV^{\mathrm{T}}$, where V is a lower triangular matrix of "Householder" vectors and T is an upper triangular matrix constructed from information in V. When the factorization is complete, the matrix A is transformed into V, T, and R, where V is in the lower triangle of the original matrix A, R is in the upper triangle, and T is stored in a one-dimensional array.

Like LU factorization, there are multiple algorithms for QR factorization. We focus on the left-looking algorithm. Complete details of the implementation of this algorithm are beyond the scope of this paper but may be found in Dongarra's book [16]. A high-level picture is provided in Fig. 5.

It should be clear from Fig. 5 that only column-block i of matrix A is changed during factoring step i. Therefore

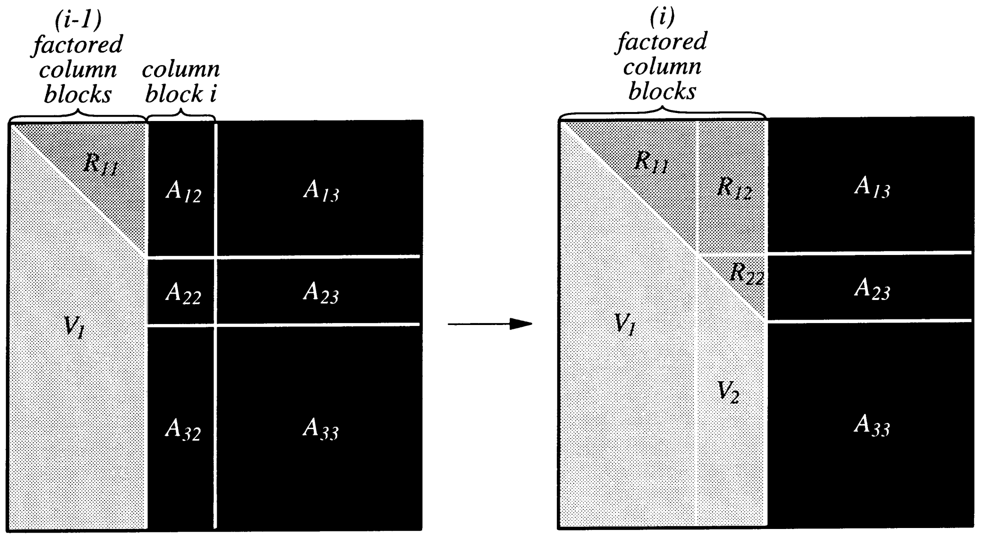


FIG. 5. Step i of QR factorization.

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 as follows. Given an initial approximation to x, the method iteratively refines this approximation until $Ax = b$ to within some error tolerance. Unfortunately, no single iterative method is robust enough 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 [4, 16, 22]. 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 and two preconditioners M_1 and M_2 to the processors P_0 through P_{n-1}. M_1 and M_2 are diagonal matrices and thus may be represented by linear arrays. After this point, A, b, M_1, and M_2 are not altered.

Now, the vectors p_0, r_0, w_0, and ξ_0 are calculated from A, b, M_1, and M_2. 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, M_1, M_2, 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 ith approximation to x. The iterations continue until $Ax_i = b$ to within a given error tolerance.

Adding fault-tolerance to the PCG algorithm is straightforard. First, the processors distribute A, b, M_1, and M_2 and allocate memory for x_i, p_i, r_i, w_i, and ξ_i. The extra processor P_n is 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 to the previous checkpoint upon a failure. However, since checkpoints are only taken every k steps, the overhead of checkpointing will be reduced by a factor of k.

6. IMPLEMENTATION RESULTS

We implemented and executed these programs on a network of Sparc-5 workstations running PVM [19]. 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. 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 when we could allocate them exclusively for our own use.

The results presented here are for a network of 17 processors, where 16 are running the program ($n = 16$, $p = q = 4$) 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 ten different instances of the Cholesky factorization, one for each of ten matrix sizes from $N = 1{,}000$ to $N = 10{,}000$. In each run, the block size was 50. The data for this experiment is in Fig. 6.

As displayed in the leftmost graph of Fig. 6, Cholesky factorization has a running time of $O(N^3)$. The total overhead of checkpointing consists of the following two components:

• T_{init}: *The time to take the initial checkpoint of matrix A.* In our calculations below, we assume that message bandwidth dominates the overhead of message-passing enough that we can ignore message latency. Each entry of A is a double precision floating point number (8 bytes). As stated in Section 5.1 above, since A is symmetric, only half of it needs to be stored. Therefore the total amount of storage needed for A is $4N^2$ bytes. These bytes are distributed evenly among the n processors, which perform the XOR using a binary tree algorithm. Thus, the first checkpoint takes $[4N^2(\log n)]/nR$ seconds, where R is the rate of sending a message and XOR-ing it, expressed in bytes per second.

• T_{rest}: *The time to take the column-block checkpoints.* There are N/b of these checkpoints, with an average size of $4bN$. Since p processors cooperate for each of these checkpoints, the total overhead of these checkpoints is N/b $(4bN[(\log p)/pR]) = [4N^2(\log p)]/pR$.

Thus, the total overhead of checkpoint is

$$\mathrm{OV}_{\text{Cholesky}} = \frac{4N^2}{R}\left(\frac{\log n}{n} + \frac{\log p}{p}\right). \qquad (1)$$

Matrix Size		Running Time	With Checkpointing							Recovery Time
N	Total Size (Mbyte)	(sec)	# of Check-points	Checkpoint Interval (sec)	Running Time (sec)	Total Overhead (sec)	T_{init} (sec)	T_{rest}: Avg. Per Ckp. (sec)	Overhead %	(sec)
1000	4	12	20(+1)	0.7	15	3	2	0.1	25.0	2
2000	16	58	40(+1)	1.6	70	12	6	0.1	20.7	6
3000	36	163	60(+1)	2.9	189	26	14	0.2	16.0	13
4000	64	331	80(+1)	4.4	379	48	26	0.3	14.5	23
5000	100	602	100(+1)	6.4	682	80	38	0.4	13.3	36
6000	144	992	120(+1)	8.8	1112	120	52	0.6	12.1	51
7000	196	1516	140(+1)	11.5	1681	165	70	0.7	10.9	69
8000	256	2244	160(+1)	14.9	2494	250	102	0.9	11.1	98
9000	324	3057	180(+1)	18.0	3367	310	134	1.0	10.1	132
10000	400	4112	200(+1)	21.7	4489	377	152	1.1	9.2	148

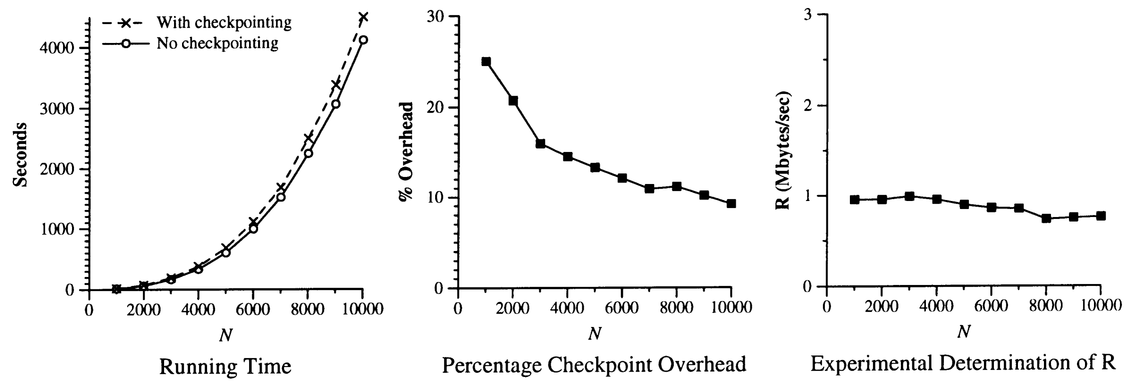


FIG. 6. Results for Cholesky factorization.

Since OV_{Cholesky} is $O(N^2)$ and Cholesky factorization is $O(N^3)$, we expect the percentage overhead of checkpointing to decrease as N increases. This is plotted in the middle graph of Fig. 6.

The rightmost graph of Fig. 6 plots R as determined by Eq. (1) for each value of N. Since the peak observed network performance is 40 megabits per second, we expect that R will be somewhat lower than 5 Mbytes/s to take account of synchronization and the XOR time. This is shown to be the case.

Recovery consists of taking the bitwise exclusive-or of every processor's matrix A. Thus, the overhead of recovery should equal T_{init}, which is reflected in the last column of Fig. 6. Notice that the time it takes to recover is irrespective of the location of the failure.

6.2. LU Factorization

The results from the LU factorization are in Fig. 7. Again, the block size was 50. The results are very similar to the results from the Cholesky factorizations. Like Cholesky, Crout LU consumes $O(N^3)$ floating point operations but its constants are greater (by a factor of 2), resulting in longer running times.

Since the matrix is not symmetric, the first checkpoint of A takes twice as long as in Cholesky factorization. Moreover, the calculation of T_{rest} is more complex. Each checkpoint consists of a portion of a column block, two row-blocks, and the pivot vector ρ. The overhead of sending the row-blocks is $[16Nb(\log q)]/qR$, and the overhead of sending ρ is $4b/R$, because only b elements of ρ are altered per iteration, and these b elements are contained entirely in one processor. The average size of L^i_{32} is $N/2b$. Therefore the average overhead of sending the column block is $[4N(\log p)]/bpR$. This yields the following equation for the overhead of checkpointing the Crout LU factorization:

$$OV_{\text{Crout}} = \frac{4N}{R} + \frac{4N^2}{R}\left(\frac{(\log p)}{p} + \frac{4(\log q)}{q} + \frac{2(\log n)}{n}\right). \qquad (2)$$

The rightmost graph of Fig. 7 shows that Eq. (2) yields values of R similar to those for Cholesky factorization. The recovery time once again is roughly equal to T_{init}.

6.3. QR Factorization

The results from the QR factorization are in Fig. 8. Once more, the block size was 50. QR factorization is another $O(N^3)$ algorithm whose constants are greater than the LU factorizations. As such, only five values of N were tested because of the large running times.

Matrix Size		Running	With Checkpointing							Recovery
	Total	Time	# of	Checkpoint	Running	Total	T_{init}	T_{rest}: Avg.	Overhead	Time
	Size		Check-	Interval	Time	Overhead		Per Ckp.	%	
N	(Mbyte)	(sec)	points	(sec)	(sec)	(sec)	(sec)	(sec)		(sec)
1000	8	56	20(+1)	3.3	66	10	3	0.3	17.9	3
2000	32	190	40(+1)	5.5	221	31	12	0.5	16.3	11
3000	72	451	60(+1)	8.7	519	68	25	0.7	15.1	24
4000	128	853	80(+1)	12.2	978	125	50	0.9	14.7	44
5000	200	1509	100(+1)	17.1	1715	206	79	1.3	13.7	69
6000	288	2294	120(+1)	21.7	2605	311	104	1.7	13.6	99
7000	392	3545	140(+1)	28.4	3978	433	148	2.0	12.2	136
8000	512	4959	160(+1)	34.5	5521	562	195	2.3	11.3	178
9000	648	6877	180(+1)	42.4	7627	750	253	2.8	10.9	248
10000	800	8850	200(+1)	49.0	9793	943	295	3.2	10.7	283

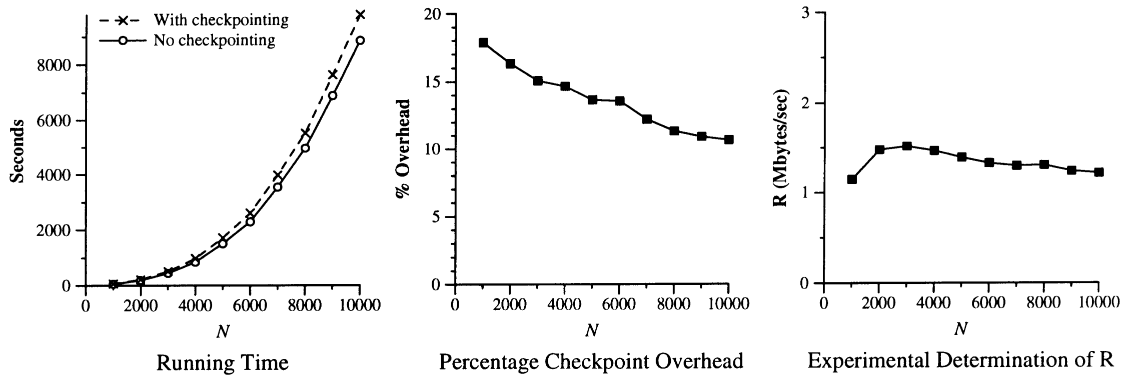


FIG. 7. Results for LU factorization.

Matrix Size		Running	With Checkpointing							Recovery
	Total	Time	# of	Checkpoint	Running	Total	T_{init}	T_{rest}: Avg.	Overhead	Time
	Size		Check-	Interval	Time	Overhead		Per Ckp.	%	
N	(Mbyte)	(sec)	points	(sec)	(sec)	(sec)	(sec)	(sec)		(sec)
1000	8	376	20(+1)	19.4	388	12	5	0.3	3.2	4
2000	32	1210	40(+1)	31.1	1246	36	14	0.6	3.0	12
3000	72	2909	60(+1)	49.8	2988	79	32	0.8	2.7	29
4000	128	6052	80(+1)	77.5	6197	145	55	1.1	2.4	52
5000	200	10934	100(+1)	111.6	11162	228	74	1.5	2.1	70

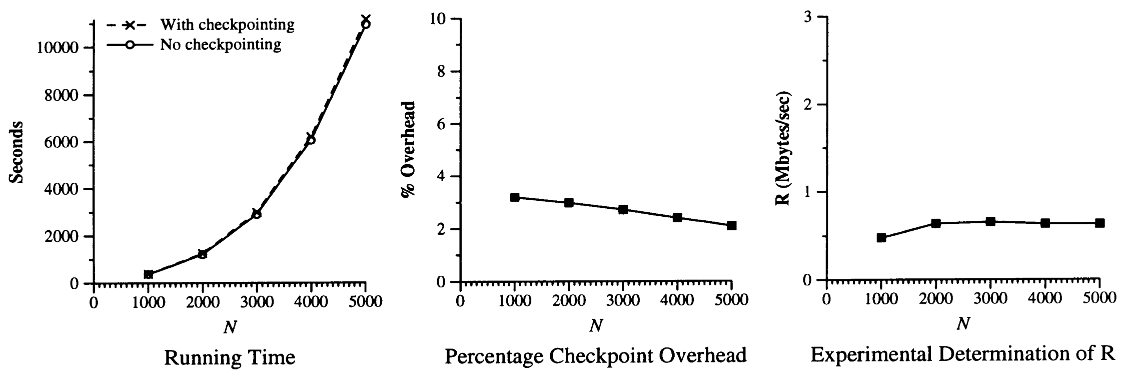


FIG. 8. Results for QR factorization.

QR checkpointing is exactly like Cholesky checkpointing, except that all of A is checkpointed initially, and every column-block is checkpointed in its entirety. Therefore, the overhead of QR checkpointing is exactly twice that of Cholesky checkpointing:

$$\mathrm{OV}_{\mathrm{QR}} = \frac{8N^2}{R}\left(\frac{\log n}{n} + \frac{\log p}{p}\right). \tag{3}$$

QR factorization has the lowest checkpointing overhead percentage of all the factorizations.

6.4. Preconditioned Conjugate Gradient

We executed an instance of PCG with an $N \times N$ matrix A for I iterations, where N = 1,048,576 and I = 5,000. This calculated x to within a tolerance of 10^{-8}. The results of this instance with varying values of k (iterations per checkpoint) are in Fig. 9.

As in the factorizations, the overhead of checkpointing is broken into two parts: T_{init}, which is the time to checkpoint A, b, M_1, and M_2, and T_{rest}, which accounts for all of the checkpoints of x, p, r, w, and ξ. The dense representation of A is a $5 \times N$ matrix, yielding a value of $[64N(\log n)]/nR$ for T_{init}. The remaining checkpoints comprise five vectors of length N. These should take $[40N(\log n)]/nR$ each. Since there are I/k of these checkpoints, the overhead of checkpointing PCG is

$$\mathrm{OV}_{\mathrm{PCG}} = \frac{8N(\log n)}{nR}\left(8 + \frac{5I}{k}\right). \tag{4}$$

To recover, matrix A and all eight vectors need to be reconstructed. Thus, the overhead of recovery time should be $[104N(\log n)]/nR$, which is the sum of the T_{init} and T_{rest} columns of Fig. 9.

7. DISCUSSION

7.1. Checkpointing Overhead and Interval

The results presented in the previous section show that on current NOWs, the performance of this method for fault-tolerant computation is surprisingly good. In the Cholesky and LU factorizations, checkpoints are taken less than a minute apart, yet the overhead is low. In the long-running instances, the total checkpointing overhead is under 15%. In the QR factorizations, the overhead of checkpointing is under two percent in all instances, while the checkpointing interval is less than two minutes. In all the factorizations, the overhead of checkpointing is $O(N^2)$, while the running time complexity is $O(N^3)$. Thus, the percentage of checkpointing overhead decreases as the problem size increases.

One interesting thing to notice is that there is no term for the block size b in Eqs. (1), (2), and (3). This means that given

k	Running Time	With Checkpointing							Recovery Time
		# of Check-points	Checkpoint Interval (sec)	Running Time (sec)	Total Overhead (sec)	T_{init} (sec)	T_{rest}: Avg. Per Ckp. (sec)	Overhead %	
	(sec)								(sec)
25	5789	200(+1)	47	9436	3647	28.9	18.1	63.0	41
50	5789	100(+1)	76	7607	1818	28.6	17.9	31.4	40
100	5789	50(+1)	134	6746	957	29.7	18.5	16.5	41
500	5789	10(+1)	600	6004	215	29.7	18.5	3.7	40
1000	5789	5(+1)	1181	5909	120	29.1	18.2	2.1	41
2500	5789	2(+1)	2926	5853	64	28.4	17.8	1.1	41
5000	5789	1(+1)	5835	5835	46	28.3	17.7	0.8	42

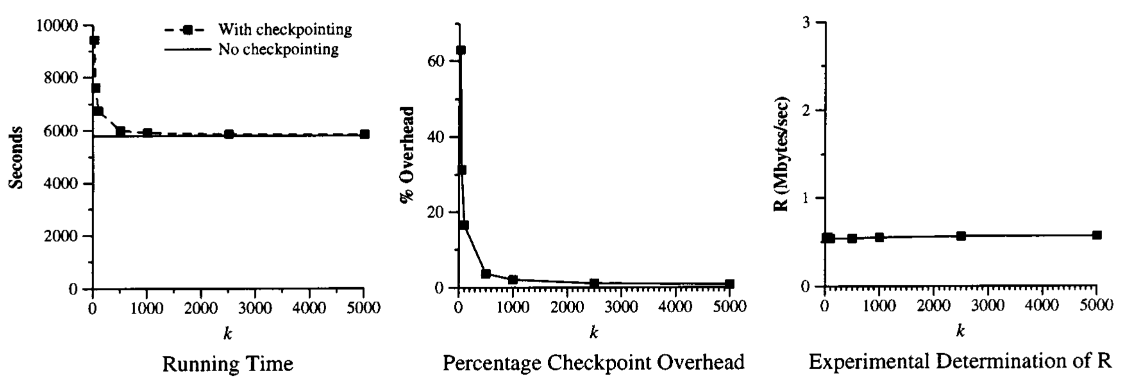


FIG. 9. Results for PCG.

the assumptions of these equations, the block size has little impact on the overhead of checkpointing. This in turn means that the checkpointing interval should have little impact on the overhead of checkpointing. One key assumption made by Eqs. (1), (2), and (3) is that message latency can be ignored. While this is true for larger block sizes, message latency becomes more significant as b decreases. Therefore, there is an extra penalty for small block sizes that is not reflected in the equations.

Also of interest is the fact that there is no term for q in Eqs. (1) and (3). Thus, the best checkpointing performances in Cholesky and QR factorization should be realized when $p = n$ and $q = 1$. Of course, the selection of p and q also impacts the performance of the factorization [13].

In the PCG implementation, there is a tradeoff between the checkpointing interval and the overhead of checkpointing. This tradeoff is controlled by the variable k. In our example, the total overhead of checkpointing is roughly (29 + 18 (5000/k)) s. Therefore, one can choose a value of k to achieve a desired checkpointing overhead or interval. For example, to achieve a 10% checkpointing overhead, one can choose k to be 164. This will yield roughly 578 s of checkpointing overhead and checkpoints every 207 s.

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 that $m - 1$ of them should be idle. Instead of having all n processors checkpoint to P_n, we can partition the n processors into m groups $G_0, \cdots, G_{m-1}$, 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 [21].

The extreme we have presented as $m = 1$. At the other extreme are systems like Isis [5] or Targon [8] where $m = n$, and every processor has a backup processor to which it sends checkpoints. As m grows, the overhead of checkpointing and recovery decreases because there is less contention for the parity processors and there are fewer XOR operations.

To tolerate *any* combination of m processor failures, m parity processors must be combined with more sophisticated error-correction techniques [6, 9, 36]. 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.

7.3. *Choice of Factorization Algorithm*

As a final remark, the choice of the algorithms for the factorization has an impact on both the performance of the factorization and the performance of checkpointing. Specifically, for Cholesky factorization, there are top-looking and right-looking algorithm variants, and for LU factorization, there are left-looking, right-looking, and Crout variants [16]. In general, the right-looking algorithms perform the best because they minimize communication overhead. However, when factoring column-block i, they modify all blocks in column-block j and row-block k such that $j \geq i$ and $k \geq i$. Thus, the average iteration modifies $\sum_{i=1}^{N/b} (bi)^2 \approx N^3/3b$ matrix elements that would have to be checkpointed. This would lead to prohibitively high overheads in terms of both time and memory.

To assess the impact of our algorithm selection, we implemented all algorithm variants of Cholesky and LU factorization without checkpointing. Figure 10 plots the running times of these variants for all problem sizes and includes the results of checkpointing. The lower row of graphs plot the overhead of checkpointing compared to the right-looking factorization variants.

Figure 10 shows that the checkpointing overhead of both Cholesky and LU factorizations is low even compared to the right-looking variants. They too exhibit the trend of decreasing percentage of overhead as the problem size increases.

8. RELATED WORK

There has been much research 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 [23, 33, 40]. These methods concentrate mainly on fault-detection, and in some cases correction. It is future research to see whether these techniques or a combination of these techniques with backward error assertions [7] can be used to further improve diskless checkpointing.

Checkpointing on parallel and distributed systems has been studied and implemented by many people [8, 14, 15, 17, 24, 27, 31, 38, 41, 44–46]. 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 [37] and is unique to this work.

An interesting comparison of this work to disk-based checkpointing can be obtained using the results of Elnozahy *et al.* [17]. In this paper, they checkpoint a program `gauss`, which performs an LU factorization with partial pivoting on a 1024 $\times$ 1024 matrix using 16 diskless Sun 3/60 processors. Checkpoints are taken to two central file servers every 2 min and two optimizations are employed: copy-on-write and incremental checkpointing. The checkpointing performance is excellent. Checkpoints take about 14 s to commit, and with the copy-on-write optimization, the overhead is approximately 0.42 s per checkpoint. By comparison, our method checkpoints the same instance every 3.3 s with an average commit time and overhead of 0.43 s per checkpoint. This is without the benefit of operating system modification or use of the memory management system. The employment of asynchronous message sending similar to the asynchronous checkpoint writing of copy-on-write checkpointing could decrease the overhead of our scheme even further.

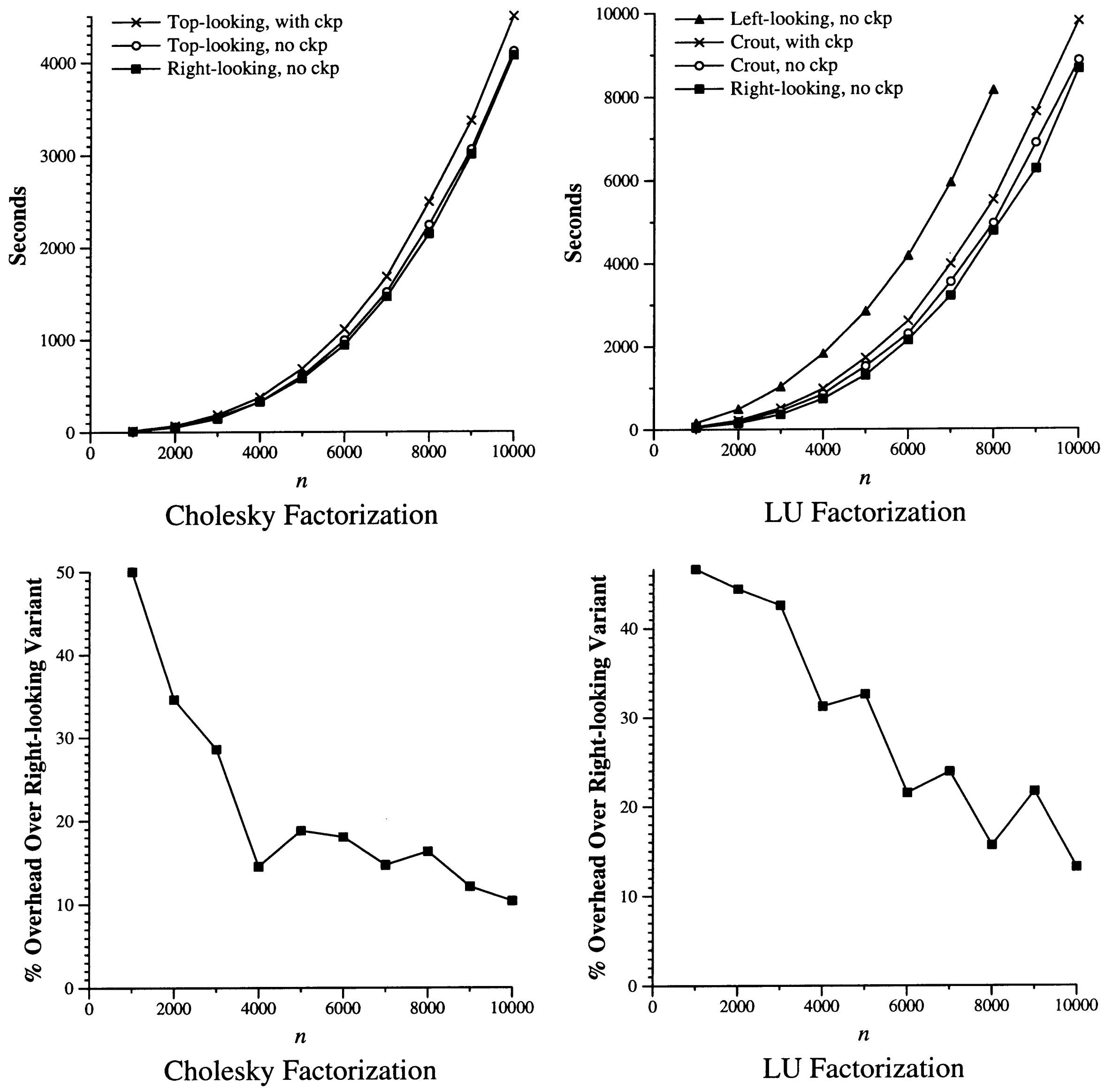


FIG. 10. Performance of checkpointing compared to all factorization variants.

There are efforts to provide programming platforms for heterogeneous computing that can adapt to changing load. These can be divided into two groups—those presenting new paradigms for parallel programming that facilitate fault-tolerance/migration [2, 3, 15, 20], and migration tools based on consistent checkpointing [10, 39, 43]. In the former group, the programmer must make his or her program conform to the programming model of the platform. None are garden variety message-passing environments such as PVM or MPI. Those in the latter group achieve transparency, but cannot migrate a process without that process's participation. Thus, they cannot handle processor failures or revocation due to ownership without checkpointing to a central disk.

9. CONCLUSIONS

We have given a method for executing certain scientific computations on a changing or faulty 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 is 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 a fast network of Sparc-5 workstations. The results show that our methods exhibit low overhead while checkpointing at a fine-grained interval (in most cases less than 5 min).

Our continuing progress with this work has been in three directions. 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 have added the capacity for multiple parity processors as outlined in Section 7.2. Preliminary results have shown that this improves both the reliability of the computation and the performance of checkpointing. Third, we are designing a technique to checkpoint the right-looking factorizations using a checksum approach and reverse computation to restore the bulk of processor state upon failure [25]. The result is a mechanism that checkpoints at somewhat larger intervals, but with lower overhead than the algorithms described in this paper.

For the future, we would like to integrate our 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.

ACKNOWLEDGMENTS

The authors thank the following people for their help concerning this research: Richard Barrett, Micah Beck, Randy Bond, Jaeyoung Choi, Brian Davis, Chris Jepeway, Mark Jones, Kai Li, Bob Manchek, Nitin Vaidya, Clint Whaley, and the anonymous referees.

James Plank was supported by National Science Foundation Grant CCR-9409496 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, and by the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615.

REFERENCES

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. *Lapack User's Guide.* SIAM, Philadelphia, 1992.
2. Arabe, A. N. C., Beguelin, A., Lowekamp, B., Seligman, E., Starkey, M., and Stephan, P. Dome: Parallel programming in a distributed computing environment. *Proc. 10th International Parallel Processing Symposium.* IEEE Comput. Soc., 1996.
3. Bakken, D. E., and Schilchting, R. D. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. Parallel Distrib. Systems* **6,** 3 (Mar. 1995), 287–302.
4. Barrett, R., *et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.
5. Birman, K. P., and Marzullo, K. ISIS and the meta project. *Sun Technol.* (1989).
6. Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. *Proc. 21st Annual International Symposium on Computer Architecture.* 1994, 245–254.
7. Boley, D., Golub, G. H., Makar, S., Saxena, N., and McCluskey, E. J. Floating point fault tolerance with backward error assertions. *IEEE Trans. Comput.* **44,** 2 (Feb. 1995).
8. Borg, A., Blau, W., Graetsch, W., Herrman, F., and Oberle, W. Fault tolerance under UNIX. *ACM Trans. Comput. Systems* **7,** 1 (Feb. 1989), 1–24.
9. Burkhard, W. A., and Menon, J. Disk array storage system reliability. *Proc. 23rd International Symposium on Fault-Tolerant Computing.* IEEE Compt. Soc., 1993, pp. 432–441.
10. Casas, J., Clark, D. L., Konuru, R., Otto, S. W., Prouty, R. M., and Walpole, J. MPVM: A migration transparent version of PVM. *Compt. Systems* **8,** 2 (Spring 1995), 171–216.
11. Casas, J., Clark, D. L., Galbiati, P. S., Konuru, R., Otto, S. W., Prouty, R. M., and Walpole, J. MIST: PVM with transparent migration and checkpointing. *3rd Annual PVM Users' Group Meeting.* 1995.
12. Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Systems* **3,** 1 (Feb. 1985), 3–75.
13. Choi, J., Dongarra, J., Pozo, R., and Walker, D. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. *Proc. 4th Symposium on the Frontiers of Massively Parallel Computation.* IEEE Compt. Soc., 1992, pp. 120–127.
14. Cristian, F., and Jahanain, F. A timestamp-based checkpointing protocol for long-lived distributed computations. *Proc. 10th Symposium on Reliable Distributed Systems.* 1991, pp. 12–20.
15. Cummings, D., and Alkalaj, L. Checkpoint/rollback in a distributed system using coarse-grained dataflow. *Proc. 24th International Symposium on Fault-Tolerant Computing.* IEEE Compt. Soc. 1994, pp. 424–433.
16. Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, 1991.
17. Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. The performance of consistent checkpointing. *Proc. 11th Symposium on Reliable Distributed Systems.* 1992, pp. 39–47.
18. Elnozahy, E. N., and Zwaenepoel, W. On the use and implementation of message logging. *Proc. 24th International Symposium on Fault-Tolerant Computing.* 1994, pp. 298–307.
19. Geist, A., Beguelin, A., Dongarra, J., Manchek, R., Jaing, W., and Sunderam, V. *PVM—A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Boston, 1994.
20. Gelernter, D., and Kaminsky, D. Supercomputing out of recycled garbage: Preliminary experience with piranha. *Proc. International Conference on Supercomputing.* ACM, 1992, pp. 417–427.
21. Gibson, G. A., Hellerstein, L., Karp, R. M., Katz, R. H., and Patterson, D. A. Failure correction techniques for large disk arrays. *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 1989, pp. 123–132.
22. Golub, G. H., and Van Loan, C. V. *Matrix Computations,* 2nd ed. Johns Hopkins Univ. Press, Baltimore, MD, 1989.
23. Huang, K-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.* **C-33,** 6 (June 1984), 518–528.
24. Johnson, D. B., and Zwaenepoel, W. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms* **11,** 3 (Sep. 1990), 462–491.
25. Kim, Y., Plank, J. S., and Dongarra, J. Fault tolerant matrix operations using checksum and reverse computation. *Proc. 6th Symposium on the Frontiers of Massively Parallel Computation.* 1996.

26. Koo, R., and Toueg, S. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Engrg.* **SE-13,** 1 (Jan. 1987), 23–31.
27. Lai, T. H., and Yang, T. H. On distributed snapshots. *Inform. Process. Lett.* **25** (May 1987), 153–158.
28. Laranjeira, L. A., Malek, M., and Jenevein, R. M. Space/time overhead analysis and experiments with techniques for fault tolerance. *Dependable Comput. Fault-Tolerant Systems* **8,** 3 (1993), 303–318.
29. León, J., Fisher, A. L., and Steenkiste, P. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Tech. Report CMU-CS-93-124, Carnegie Mellon University, Feb. 1993.
30. Li, K., Naughton, J. F., and Plank, J. S. An efficient checkpointing method for multicomputers with wormhole routing. *Int. J. Parallel Process.* **20,** 3 (June 1992), 159–180.
31. Li, K., Naughton, J. F., and Plank, J. S. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Systems* **5,** 8 (Aug. 1994), 874–879.
32. Long, D., Muir, A., and Golding, R. A longitudinal survey of internet host reliability. *Proc. 14th Symposium on Reliable Distributed Systems.* 1995, pp. 2–9.
33. Luk, F. T., and Park, H. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.* **5** (1988), 172–184.
34. Message Passing Interface Forum. MPI: A message-passing interface standard. *Int. J. Supercomputer Appl.* **8,** 3/4 (1994).
35. Mutka, M. W., and Livny, M. The available capacity of a privately owned workstation environment. *Performance Evaluation* (1991).
36. Plank, J. S. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. *Proc. 15th Symposium on Reliable Distributed Systems.* 1996, pp. 76–85.
37. Plank, J. S., and Li, K. Faster checkpointing with $N + 1$ parity. *Proc. 24th International Symposium on Fault-Tolerant Computing.* 1994, pp. 288–297.
38. Plank, J. S., and Li, K. Ickp—a consistent checkpointer for multicomputers. *IEEE Parallel Distrib. Technol.* **2,** 2 (Summer 1994), 62–67.
39. Pruyne, J., and Livny, M. Parallel processing on dynamic resources with CARMI. *Proc. First IPPS Workshop on Job Scheduling Strategies for Parallel Processing.* 1995.
40. Roy-Chowdhury, A., and Banerjee, P. Algorithm-based fault location and recovery for matrix computations. *Proc. 24th International Symposium on Fault-Tolerant Computing.* 1994, pp. 38–47.
41. Silva, L. M., Silva, J. G., Chapple, S., and Clarke, L. Portable checkpointing and recovery. *Proc. HPDC-4, High-Performance Distributed Computing.* 1995, pp. 188–195.
42. Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. J. *MPI: The Complete Reference.* MIT Press, Boston, 1996.
43. Stellner, G. CoCheck: Checkpointing and process migration for MPI. *Proc. 10th International Parallel Processing Symposium.* 1996.
44. Strom, R. E., and Yemini, S. Optimistic recovery in distributed systems. *ACM Trans. Comput. Systems* **3,** 3 (Aug. 1985), 204–226.
45. Sure, G., Janssens, R., and Fuchs, W. K. Reduced overhead logging for rollback recovery in distributed shared memory. *Proc. 25th International Symposium on Fault-Tolerant Computing.* 1995, pp. 279–288.
46. Wang, Y. M., and Fuchs, W. K. Lazy checkpoint coordination for bounding rollback propagation. *Proc. 12th Symposium on Reliable Distributed Systems.* 1993, pp. 78–85.

Received March 1, 1996; revised April 1, 1997; accepted April 30, 1997

JAMES PLANK received his B.S. from Yale in 1988, his M.A. from Princeton in 1990, and his Ph.D. from Princeton in 1993. He is currently an assistant professor in the Computer Science Department at the University of Tennessee. His research interests are in fault tolerance, specifically fast checkpointing and rollback recovery of sequential and parallel computations.

YOUNGBAE KIM is currently a member of the Scientific Computing Group in the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory (LBL), University of California, Berkeley. His research interests include parallel and distributed computing focusing on scientific computing and numerical linear algebra, network computing, and fault tolerance. He received his B.S. and M.S. in electronics engineering in 1982 and 1984, respectively, from Seoul National University at Seoul, Korea. He also earned a M.S. in electrical and computing engineering in 1990 from the University of Colorado at Boulder, and his Ph.D. in Computer Science in 1996 from the University of Tennessee at Knoxville.

JACK DONGARRA holds a joint appointment as Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee (UT) and as Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory (ORNL) under the UT/ORNL Science Alliance Program. He received a B.S. in mathematics from Chicago State University in 1972, a M.S. in computer science from the Illinois Institute of Technology in 1973, and a Ph.D. in applied mathematics from the University of New Mexico in 1980. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. Other current research involves the development, testing, and documentation of high quality mathematical software. He was involved in the design and implementation of the software packages EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib/XNetlib, PVM/HeNCE, and MPI and the National High-Performance Software Exchange, and is currently involved in the design of algorithms and techniques for high-performance computer architectures.