Fault Tolerant Matrix Operations Using Checksum and Reverse Computation

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

†Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301

‡Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37821-6367

Abstract

In this paper, we present a technique, based on checksum and reverse computation, that enables high-performance matrix operations to be fault-tolerant with low overhead. We have implemented this technique on five matrix operations: matrix multiplication, Cholesky factorization, LU factorization, QR factorization and Hessenberg reduction. The overhead of checkpointing and recovery is analyzed both theoretically and experimentally. These analyses confirm that our technique can provide fault tolerance for these high-performance matrix operations with low overhead.

1 Introduction

The price and performance of uniprocessor workstations and off-the-shelf networking have made networks of workstations (NOWs) a cost-effective parallel processing platform that is competitive with supercomputers. The popularity of NOW programming environments like PVM [14] and MPI [17, 30] and the availability of high-performance numerical libraries like ScaLAPACK (Scalable Linear Algebra PACKage) [7] for scientific computing on NOWs show that networks of workstations are already in heavy use for scientific programming.

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

Recently, the papers [23, 24] have developed such a fault-tolerant computing paradigm. The paradigm is based on checkpointing and rollback recovery using processor and memory redundancy. It is called *diskless checkpointing* as it provides fault tolerance without any reliance on disk. For this paradigm, a parity-based checkpointing technique is used to incorporate fault tolerance into high-performance matrix operations. Thus, the paradigm is an algorithm-based approach in which fault tolerance is especially tailored to the applications.

As discussed in paper [24], however, when the parity-based technique is mixed with the right-looking variants of the general factorizations, extra memory is required to store the entire matrix every iteration. If only a small amount of extra memory is available in each processor, we need to devise an alternative that recovers the state of the last checkpoint, rather than saving it.

One solution is to reverse the computations performed since the last checkpoint. A drawback of this solution is that it may introduce floating-point roundoff errors due to reverse computation. This roundoff error may change any bit in binary representation of a floating-point number. Thus, incorrect data would be generated when the bitwise exclusive-or operation is performed. Such data is totally unpredictable and may be neither recovered nor corrected.

Therefore, with the reverse computation technique, we propose to use a checksum, rather than parity, to encode the data. Since the checksum is a floating-point addition, roundoff errors are minimized, rather than magnified as they are by parity. Admittedly, the possibility exists for overflow, underflow, and roundoff errors due to cancellation when the checksum is computed [16, 32]. The effect of such problems is considered to be negligible, however, because the checksum involves the addition of only as many floating-point numbers as the total number of the application processors.

We use our new technique to checkpoint the right-looking factorizations and other matrix operations and to restore the bulk of processor state upon failure. The target matrix operations include matrix multiplication, the right-looking variants of Cholesky, LU, and QR factorizations, and Hessenberg reduction [1, 7, 8, 11], which are at the heart of scientific computations. Our technique results in checkpointing at somewhat larger intervals, but with lower overhead than the algorithms described in [24]. 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 Checkpointing and Rollback Recovery

2.1 Basic Scheme

Checkpointing and rollback recovery enables a system with fail-stop failures [33] to tolerate failures by periodically saving the entire state and rolling back to the saved state if a failure occurs. Our technique for checkpointing and rollback recovery adopts the idea of *algorithm-based diskless checkpointing* [23]. If the program is executing on N processors, there is a $N+1$-st processor called the *checkpointing processor*. At all points in time, a consistent checkpoint is held in the N processors in memory. A checksum (floating-point addition) of the N checkpoints is held in the checkpointing processor. This is called the *global checkpoint*. If any processor fails, all live processors, including the checkpointing processor, cooperate in reversing the computations performed since the last checkpoint. Thus, the data is restored at the last checkpoint for rollback, and the failed processor's state can be reconstructed on the checkpointing processor as the checksum of the global checkpoint and the remaining $N-1$ processors' local checkpoints.

2.2 Analysis of Basic Checkpointing

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

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

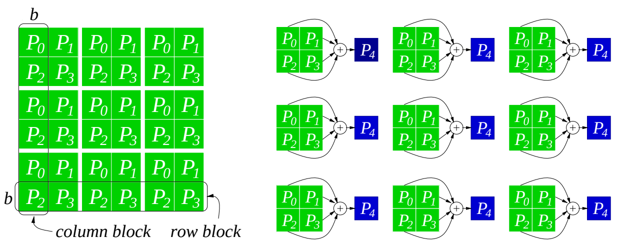


Figure 1: Data distribution and checkpointing of a matrix with 6×6 blocks over a 2×2 mesh of 4 processors

The basic checkpointing operation works on a panel of blocks, where each block consists of $X = b^2$ floating-point numbers, and the processors are logically configured in a $P \times Q$ mesh (see Figure 1). The processors take the checkpoint with a global addition. This works in a spanning-tree fashion in three parts. The checkpoint is first taken rowwise, then taken columnwise, and then sent to the checkpointing processor P_C. The first part therefore takes $\lceil \log P \rceil$ steps, and the second part takes $\lceil \log Q \rceil$ steps. Each step consists of sending and then performing addition on X floating-point numbers. The third part consists of sending the X numbers to P_C. We define the following terms: γ is the time for performing a floating-point addition, α is the startup time for sending a floating-point number, and β is the time for transferring a floating-point number.

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

To simplify our equations in the subsequent chapters, we define the function

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

Note that $T_{panelckpt}(X, P, Q) \approx PQXT_{ckpt}(P, Q)$. For constant values of P and Q, $T_{ckpt}(P, Q)$ is a constant. Thus, $T_{panelckpt}(X, P, Q)$ is directly proportional to X.

Sometimes, an entire $m \times n$ matrix needs to be checkpointed. If we assume that m and n are large, the time complexity of this operation is

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

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

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

3 Fault-Tolerant Matrix Operations

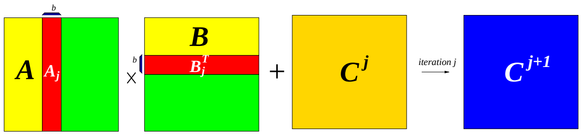
We focus on three classes of matrix operations: matrix multiplication, direct, dense factorizations, and Hessenberg reduction. In this paper, due to the similar nature of such algorithms, we cover only matrix multiplication, right-looking LU factorization and Hessenberg reduction. The right-looking Cholesky and QR factorizations can be explained as special cases of the right-looking LU factorization and Hessenberg reduction, respectively. 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 the literature by Dongarra [11] and Choi [7, 8].

3.1 Matrix Multiplication

Let an $m \times k$ matrix A be multiplied by a $k \times n$ matrix B to produce the $m \times n$ matrix C. Matrix-matrix multiplication can be formulated as a sequence of rank-one updates by

$$C = C + \sum_{j=0}^{k} A_j B_j^T,$$

where A_j is the jth column vector of A and B_j^T is the jth row vector of B. Let us assume that the matrices are partitioned into blocks of size b. Its corresponding block algorithm is depicted in Figure 2. The block algorithm performs the rank-b updates of a column block A_j and a row block B_j^T. A rank-b update is an operation that multiplies an $m \times b$ matrix by a $b \times n$ matrix and then updates an $m \times n$ matrix by adding the result matrix to it.



$$A_j \times B_j^T + C^j \quad \rightarrow \quad C^{j+1}$$

Figure 2: Iteration j of the matrix multiplication algorithm. C^{j+1} is the result matrix after iteration j

A parallel algorithm can be obtained by parallelizing the rank-b update at each iteration [1]. Each iteration j in the parallel algorithm must broadcast a column block A_j of A within the column, broadcast a row block B_j^T of B within the row, and then do the rank-b update.

3.1.1 Checkpointing

It is straightforward to incorporate fault tolerance into the algorithm. Since the result matrix C is modified by one rank b update at each iteration, a fault-tolerant algorithm needs to checkpoint only the matrix C periodically at the end of an iteration. In our implementation, the matrix C is checkpointed at the end of every K of iterations (we call it a *sweep*), where K is chosen by the programmer. Specifically, the fault-tolerant algorithm works as follows:

1. Checkpoint the matrices A, B, and C initially.
2. Perform K rank-b updates.
3. Checkpoint the matrix C, where this checkpoint is done with addition.
4. Synchronize, and go to Step 2.

No extra memory is required for checkpointing. The time overhead of checkpointing consists of the total time for performing Steps 1 and 3, which are equivalent to $3T_{matckpt}(n,n,P,Q)$ and $\frac{n}{Kb}T_{matckpt}(n,n,P,Q)$, respectively. Thus, the total time overhead of checkpointing, T_C, is

$$\begin{aligned} T_C &= 3T_{matckpt}(n,n,P,Q) + \frac{n}{Kb}T_{matckpt}(n,n,P,Q) \\ &\approx (3+\frac{n}{Kb})n^2 T_{ckpt}(P,Q). \end{aligned} \tag{4}$$

3.1.2 Recovery

Throughout the following sections, for the description of recovery we assume that each checkpoint is taken every K of iterations (from j_1 to j_K) and that a failure occurs at iteration j_f in the lth sweep. Then, the following operations are performed by all live processors including the checkpointing processor.

1. Reconfigure the processor grid by replacing the failed processor with an extra processor or the checkpointing processor.
2. Recover the failed processor's data of A and B by using the checksum and the live processors' data.
3. Reverse the computations of the live processors from the current iteration j_f to the first iteration j_1 of the current sweep l by performing the following computations: $C^{l-1} = C^l - \sum_{j=j_f}^{j_1} A_j B_j^T$, where C^{l-1} and C^l are the matrix C at the end of the $(l-1)$st sweep and lth sweep, respectively.
4. Recover the failed processor's data of C by using the checksum of P_C.
5. Resume the computation from the beginning of iteration j_1.

As stated above, recovery involves both reversing the computation and recovering three matrices by using the checksum. The time overhead of recovery consists of the time of performing Steps 3 and 4. The time of performing Step 3 is equivalent to $KT_{rank}(n,n,P,Q)$, where $T_{rank}(n,n,P,Q)$ is defined as the time of performing one rank-b update and is given by $\frac{2n^2b}{PQ}$ [1]. Thus, T_R is approximated by

$$\begin{aligned} T_R &= T_{rank}(n,n,P,Q) + 3T_{matrecv}(n,n,P,Q) \\ &\approx n^2\frac{2Kb}{PQ}\gamma + 3n^2 T_{ckpt}(P,Q). \end{aligned} \tag{5}$$

3.2 Right-looking LU Factorization

In the right-looking LU algorithm, at the beginning of iteration j the leftmost $j-1$ column blocks and uppermost $j-1$ row blocks have been factored. At iteration j, the current column block is factored first, and then the remaining matrix is updated by one rank-b update using the current column block and row block. Pivoting is also done before the rank-b update. A typical iteration is given in Figure 3.

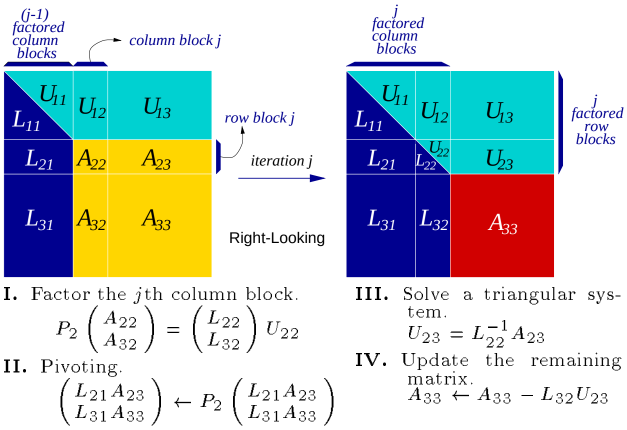


Figure 3: Iteration j of the right-looking LU algorithm

3.2.1 Checkpointing

First, extra memory for W_C, W_R, W_P, and W_ρ is allocated for checkpointing the right-looking LU algorithm. The $\lceil \frac{K}{Q} \rceil$ row and column blocks to be factored during the current sweep are saved in $W_R$and W_C, respectively. The pivoting rows and indices to be generated during the current sweep are saved in W_P and W_ρ, respectively.

At iteration j, the factored L_j and U_j are to be checkpointed as well as the pivoting rows and indices.

At the end of every sweep, the newly modified remaining matrix is checkpointed. The entire algorithm of checkpoint is described as follows.

1. All the processors checkpoint the matrix A initially.
2. For every K of iterations,
   (a) For each iteration j $(j_1, \ldots, j_K)$,
      i. The processors owning the jth column save their corresponding column blocks into W_C.
      ii. The processors owning the jth row save their corresponding row blocks into W_R.
      iii. All the processors save the jth block of the pivoting indices into W_ρ.
      iv. P_C saves the jth row and column blocks corresponding to the jth row and column into W_R and W_C, respectively, and save the jth block of pivoting indices into W_ρ.
      v. All P_A's perform factorization on the jth column block.
      vi. The processors owning any pivoting row save the pivoting row into W_P.
      vii. P_C saves the rows corresponding to the pivot rows into W_P.
      viii. Update the remaining matrix by the rank-b update: $A^l = A^{l-1} - L_j U_j$.
      ix. Checkpoint the jth row and column blocks and the pivoting rows and indices. At this point, P_C maintains the checksums of all L_j's, U_j's, the pivoting rows, and indices for the current sweep.
   (b) Checkpoint the remaining matrix at the last iteration of the current sweep.

Memory requirement for W_C, W_R, W_P, and W_ρ: Extra memory M_C is required for storing the jth column blocks for $j = j_1, \ldots, j_K$, which are distributed over Q columns of processors. Therefore, $M_C = \lceil \frac{K}{Q} \rceil \left(\frac{n}{P} + 2\frac{n}{Q} + \frac{1}{2} \right) b$.

Time complexity for checkpointing, T_C: In addition to the initial checkpointing, the jth row and column blocks are to be saved, and all L_j's, U_j's, and the jth pivoting rows and indices are to be checkpointed for $j = j_1, \ldots, j_K$. Finally, at the end of a sweep of K iterations, the remaining matrix is to be checkpointed. Similarly, for large n the time of performing Step (b) dominates the time overhead of checkpointing. Then, T_C is approximated by

$$\begin{aligned} T_C &\approx \sum_{j=1}^{\frac{n}{Kb}} T_{matckpt}(n - jKb, n - jKb, P, Q) \\ &= \sum_{j=1}^{\frac{n}{Kb}} (n - jKb)^2 T_{ckpt}(P, Q) \\ &\approx n^3 \frac{1}{3Kb} T_{ckpt}(P, Q). \end{aligned} \tag{6}$$

3.2.2 Recovery

Similarly, the following operations are performed for recovery. Let L_j and U_j represent the jth column and row blocks at iteration j after being updated during the current sweep, respectively.

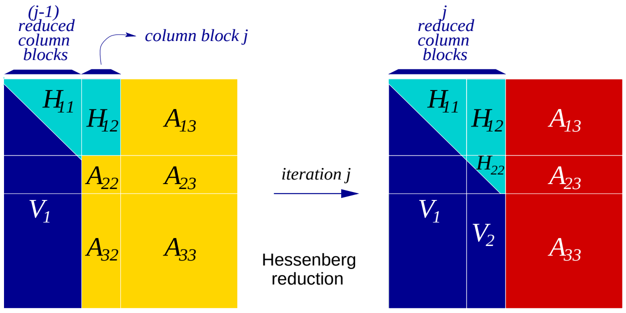
1. Reconfigure the processor grid.
2. Recover L_j's and U_j's for $j = j_1, \ldots, j_f$ from their corresponding checksum.
3. Recover their pivoting rows and indices for iteration $j = j_1, \ldots, j_f$ from their corresponding checksum.
4. Reverse the computation for the remaining matrix from iteration j_f to j_1 as follows:
   For $j = j_f$ to j_1
   (a) $A \leftarrow A + L_j U_j$
   (b) Restore the jth pivoting rows and indices from their corresponding W_P and W_ρ.
5. Restore their column and row blocks from W_C and W_R.
6. Recover the failed processor's portion of the matrix from the checksum.
7. Resume the computation from the beginning of iteration j_1.

Time complexity for recovery T_R: The time overhead of recovery varies depending on the iteration at which a failure occurs. In addition to the time for recovering the column and row blocks and the pivoting rows and indices modified since the last checkpoint, time is required for recovering the failed processor's L_j and U_j and for performing up to K rank-b updates by reverse computation. Since the time overhead of recovery is dominated by the time for performing Steps 4 and 6, T_R is approximated as follows.

$$T_R \approx n^2 \frac{2Kb}{PQ}\gamma + n^2 T_{ckpt}(P, Q). \tag{7}$$

3.3 Hessenberg Reduction

Hessenberg reduction reduces an $m \times n$ matrix A to an upper-Hessenberg matrix H by using similarity transformation based on Householder reflectors. The algorithm can be formulated by $A = Q^T H Q$, where Q is an $m \times m$ orthogonal matrix and H an $m \times n$ upper-Hessenberg matrix.



I. Reduce the j column block. **II.** Update the trailing matrix.

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = Q_2 \begin{pmatrix} H_{22} \\ 0 \end{pmatrix}, \qquad \begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} \leftarrow Q_2^T \begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} Q_2.$$

where $Q_2 = I - V_2 T_2 V_2^T$

Figure 4: Iteration j of the Hessenberg reduction algorithm

The block algorithm proceeds by reducing each column block in each iteration. At each iteration j, the current column of processors reduces the jth column block to upper-Hessenberg form by using Householder reflectors. Then, a sequence of the Householder reflectors is applied to the remaining matrix by both pre- and post-multiplication (Figure 4). In Step II, we may write the update to A performed at an iteration of the block algorithm as $A \leftarrow (I - VTV^T)^T A(I - VTV^T) = (I - VT^TV^T)(A - YV^T)$, where $Y = AVT$. Let $A \leftarrow A - YV^T$. Then, it may be written by $A \leftarrow (I - VTV^T)^T A = A - VW^T$, where $W^T = T^TV^TA$. Based on this representation, Step II proceeds in the following five phases.

1. compute T.
2. compute $Y = AVT$.
3. do the rank-b update, $A \leftarrow A - YV^T$.
4. compute $W^T = T^TV^TA$.
5. do the rank-b update, $A \leftarrow A - VW^T$.

3.3.1 Checkpointing

Updating the remaining matrix by pre- and post-multiplication requires the intermediate computations $Y = AVT$ and $W^T = T^TV^TA$ as described in the preceding section. Thus, for recovery, those intermediate results Y and W^T must be saved and checkpointed at each iteration.

1. All the processors checkpoint the matrix A.
2. For each sweep l of K iterations,
   (a) For each step j ($j_1, \ldots, j_K$),
      i. The processors owning the jth column save their corresponding column blocks into W_C.
      ii. Perform factorization for the jth column-block.
      iii. Update the remaining matrix by performing $A^l = A^{l-1} - Y_jV_j^T - V_jW_j^T$.
      iv. Checkpoint Y_j and W_j^T.
   (b) Checkpoint the remaining matrix at the last iteration of the current sweep.

Memory requirement for W_C, W_Y, and W_W: Extra memory is required for storing the column blocks to be factored within a sweep and for Y_j and W_j^T. Thus, $M_C = \lceil \frac{K}{Q} \rceil \left(2\frac{n}{P} + \frac{n}{Q}\right) b$.

Time complexity for checkpointing, T_C:

$$\begin{aligned} T_C &\approx \sum_{j=1}^{\frac{n}{Kb}} T_{matckpt}(n, n - jKb, P, Q) \\ &\approx \sum_{j=1}^{\frac{n}{Kb}} n(n - jKb)T_{ckpt}(P, Q) \\ &\approx n^3 \frac{1}{2Kb} T_{ckpt}(P, Q). \end{aligned} \tag{8}$$

3.3.2 Recovery

1. Reconfigure the processor grid.
2. Recover the failed processor's V_j, Y_j, and W_j^T from their checksum, if necessary. At this point, all V_j's, Y_j's, and W_j^T's for $j = j_1, \ldots, j_f$ have been recovered.
3. Perform $A^{l-1} = A^l + \sum_{j=j_1}^{j_f} \left(V_jW_j^T + Y_jV_j^T\right)$. for the updated part of the matrix.
4. Restore the jth column blocks from W_C for $j = j_1, \ldots, j_K$.
5. Recover the failed processor's data from the checksum matrix.
6. Resume the computation from the beginning of iteration j_1.

Time complexity for recovery, T_R:

$$T_R \approx n^2 \frac{4Kb}{PQ}\gamma + n^2 T_{ckpt}(P, Q). \tag{9}$$

4 Implementation Results

We implemented and executed these programs on a network of Sparc-5 workstations running PVM [14]. 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.

Each implementation was run on 17 processors, with 16 application processors logically configured into a 4×4 processor grid and one checkpointing processor. The block size for all implementations was set at 50, and all implementations were developed for double-precision floating-point arithmetic.

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. 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. Note that the failures were forced to occur at the last iteration of the first checkpointing interval.

Experimental results of the implementations are given in Figures 5 through 7. Each figure includes a table of experimental results and graphs of running times, percentage checkpoint overhead, and checkpointing rate experimentally determined. Note that T_C includes the initial checkpointing overhead T_{init}, T_A represents the total running time of the algorithm without checkpointing, and M is the total memory size of each problem in bytes. K represents the checkpointing interval in iterations and is chosen differently for each implementation to keep the checkpointing overhead small.

5 Discussion

Based on the performance results and analyses of time complexity presented in the preceding section, we make the following observations.

1. The total time overhead of checkpointing can be kept small by expanding the checkpointing interval K or block size b. As either K or b increases, more memory is required, and hence the total time overhead of recovery increases.
2. The total time overhead of recovery depends partially upon the location of the failure.

n	M (MB)	T_A (sec)	With Checkpointing N_C +3	T (sec)	T_C (sec)	%	T_{init} (sec)	T_R (sec)
1000	8	29	2+3	43	14	48.3	8	33
2000	32	197	3+3	261	64	32.5	32	109
3000	72	644	4+3	816	172	26.7	73	244
4000	128	1547	6+3	1941	394	25.5	131	451
5000	200	2951	7+3	3682	731	24.8	205	638
6000	288	5036	8+3	6170	1134	22.5	295	938
7000	392	7920	9+3	9546	1626	20.5	406	1297

Checkpointing Interval $K = 16$, $T = T_A + T_C$

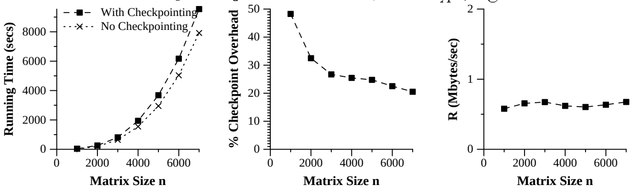


Figure 5: Matrix Multiplication: Timing Results

n	M (MB)	T_A (sec)	With Checkpointing N_C +1	T (sec)	T_C (sec)	%	T_{init} (sec)	T_R (sec)
1000	8	154	6	133	-21	-13.6	3	32
2000	32	601	11	569	-32	-5.3	11	74
3000	72	1524	16	1607	83	5.4	24	135
4000	128	3158	21	3491	333	10.5	44	211
5000	200	5744	26	6423	679	11.8	68	308
6000	288	9323	31	10653	1330	14.3	98	415
7000	392	14258	36	16478	2220	15.6	135	552

Checkpointing Interval $K = 4$, $T = T_A + T_C$

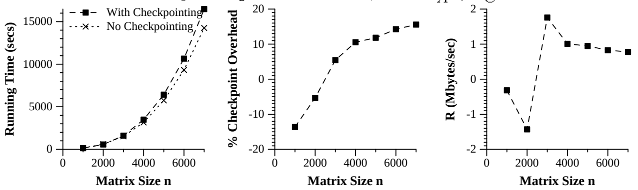


Figure 7: Hessenberg reduction: Timing results

n	M (MB)	T_A (sec)	With Checkpointing N_C +1	T (sec)	T_C (sec)	%	T_{init} (sec)	T_R (sec)
1000	8	45	2	52	7	15.6	3	47
2000	32	153	3	180	27	17.6	11	86
3000	72	364	4	436	72	19.8	25	144
4000	128	745	6	884	139	18.7	43	190
5000	200	1293	7	1525	232	17.9	69	279
6000	288	2144	8	2525	381	17.8	98	399
7000	392	3211	9	3760	549	17.1	134	528
8000	512	4774	11	5590	816	17.1	175	646
9000	648	6268	12	7555	1287	20.5	229	811
10000	800	8651	13	10447	1796	20.8	282	982

Checkpointing Interval $K = 16$, $T = T_A + T_C$

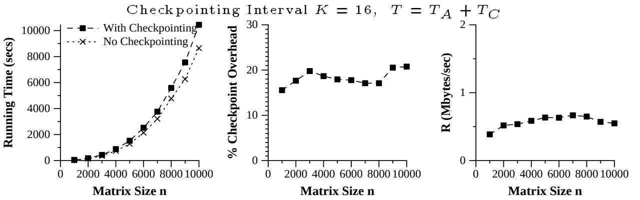


Figure 6: Right-looking LU: Timing results

n	M (MB)	T_A (sec)	With Checkpointing N_C +1	T (sec)	T_C (sec)	%	T_{init} (sec)	T_R (sec)
1000	8	10	2	14	4	40.0	2	11
2000	32	52	3	70	18	34.6	6	32
3000	72	147	4	178	31	21.1	12	61
4000	128	332	6	391	59	17.8	23	88
5000	200	574	7	697	123	21.4	34	131
6000	288	942	8	1140	198	21.0	50	184
7000	392	1466	9	1754	288	19.6	68	261
8000	512	2145	11	2551	406	18.9	88	304
9000	648	3004	12	3581	577	19.2	114	387
10000	800	4068	13	4833	765	18.8	141	478

Checkpointing Interval $K = 16$, $T = T_A + T_C$

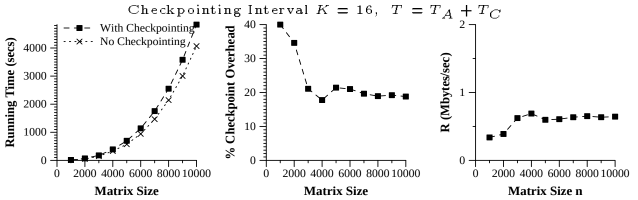


Figure 8: Right-looking Cholesky: Timing results

3. Good performance can be achieved with a moderate amount of extra memory for the right-looking QR algorithm and Hessenberg reduction.

4. For Hessenberg reduction (and small values of n), the extra memory that we use for checkpointing significantly improves the performance of the failure-free algorithm by saving some communication and computation. The values of checkpointing overhead and checkpointing rate reflect this fact.

5. Matrix multiplication is the simplest algorithm into which we incorporate the technique, and we obtain good performance without any memory overhead.

6. The checkpointing rate R for all the implementations is very close (less than 1 Mbyte per second), with the exception of Hessenberg reduction. Since the measured peak bandwidth of the network is 40 Mbits per second, we expect that the checkpointing rate should be somewhat lower than 5 Mbytes per second considering synchronization, copying, performing matrix addition, message latency, and network contention.

5.1 Checkpointing Overhead and Interval

As shown in the analytic models of various matrix operations, while the failure-free algorithms of matrix operations require $O(n^3)$ floating-point operations for a matrix of size n, the checkpointing steps require $O(\frac{n^3}{Kb})$ operations if checkpoints are taken every K of iterations. Thus, the technique provides the flexibility of selecting the checkpointing interval K to tune the overhead. The complexity analyses also show that a tradeoff exists between the extra memory requirement M_C and the checkpointing overhead T_C. In order to reduce the overhead, the checkpointing interval must get larger, and hence more memory is required to store the data (i.e., column blocks or row blocks) updated during each sweep.

5.2 Roundoff Errors

It is well known that in the realm of floating-point computations no computation is exact [36]. While the parity-based technique does not involve any floating-point computations for providing fault tolerance for numerical algorithms, the technique based on the checksum and reverse computation involves both floating-point additions and matrix computations. Thus, there is a possibility of numerical prob-

			With Checkpointing					
n	M (MB)	T_A (sec)	N_C +1	T (sec)	T_C (sec)	T_C %	T_{init} (sec)	T_R (sec)
1000	8	52	6	59	7	13.5	3	13
2000	32	203	11	248	45	22.2	11	33
3000	72	535	16	677	142	26.5	25	67
4000	128	1152	21	1473	321	27.9	44	107
5000	200	2147	26	2733	586	27.3	69	162
6000	288	3545	31	4488	943	26.6	99	225
7000	392	5482	36	6983	1501	27.4	133	298
8000	512	8002	41	10336	2334	29.2	174	385

Checkpointing Interval $K = 4$, $T = T_A + T_C$

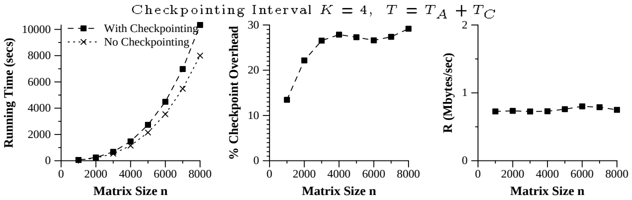


Figure 9: Right-looking QR: Timing results

lems in floating-point arithmetic subject to the checksum as well as reverse computation.

In our fault-tolerant algorithms, reverse computation usually requires rank-b updates. The rank-b updates could cause roundoff error. However, since the matrix operations we target are known to be *backward stable* [36], roundoff errors due to reverse computation are of little concern.

Some numerical problems are possible because of the checksum encoding. First, overflow and underflow can occur if the checksum is too large or small. However, since each checksum element is composed of just one floating-point number per processor, the possibility that overflow and underflow occur in computing the checksum is very low and is considered negligible unless an element of the matrix is too big or too small. The overflow and underflow in floating-point arithmetic can be avoided by organizing the computations differently, for example, normalizing each number by dividing it by the maximum among the numbers [13, 36]. Second, roundoff errors due to cancellation can be a serious numerical problem. Cancellation usually occurs when two numbers of approximately the same size are subtracted. The cancellation subject to the checksum can be avoided by performing the summation of the floating-point numbers in different order.

6 Related Work

Considerable research has been carried out on algorithm-based fault tolerance for matrix operations on parallel platforms where (unlike the above platform) the computing nodes are not responsible for storage of the input and output elements [18, 22, 28]. These methods concentrate mainly on fault-detection and, in some cases, correction. One open question is whether these techniques can be used to further improve our checksum and reverse computation-based technique.

Checkpointing on parallel and distributed systems has been studied and implemented by many people [5, 9, 10, 12, 19, 20, 21, 26, 29, 33, 34, 35]. All of this work, however, focuses on either checkpointing to disk or on 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 [25] and is unique to this work.

Some efforts are underway to provide programming platforms for heterogeneous computing that can adapt to changing load. These efforts can be divided into two groups: those presenting new paradigms for parallel programming that facilitate fault tolerance/migration [2, 3, 10, 15], and migration tools based on consistent checkpointing [6, 27, 31]. In the former group, the programmer must make a 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 availability, without checkpointing to a central disk.

7 Conclusions and Future Work

We have presented a new technique for executing certain scientific computations on a changing or faulty network of workstations (NOWs). This technique employs checksum and reverse computation to adapt the algorithm-based diskless checkpointing to the matrix operations. It also enables a computation designed to execute on N processors to run on a NOW platform where individual processors may leave and enter the NOW because of failures or load. As long as the number of 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 technique on the core matrix operations and shown performance results on a fast network of Sparc-5 workstations. The results indicate that our technique can obtain low overhead with reasonable amount of extra memory while checkpointing at a reasonably small interval (it may vary depending on the algorithms). The possibility of numerical problems such as overflow, underflow, and roundoff error due to cancellation exists, but is of little practical concern. To reduce the effect of roundoff errors, if any, we suggest an iterative refinement scheme [4, 37] for the solution if it does not meet the desired error bound of the algorithms [13, 36].

Our continuing progress with this work has been in the following 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 checkpointing processors as outlined in paper [24]. Preliminary results have shown that this improves both the reliability of the computation and the performance of checkpointing. In particular, our technique reaps significant benefits from such multiple checkpointing with relatively less memory by checkpointing at a finer-grain interval.

For the future, our scheme can be integrated with general load-balancing. In other words, if a few processors are added to or deleted from the NOW, the system would continue running, using the mechanisms outlined in this paper. However, if the size of the processor pool changes by an order of magnitude, 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: Randy Bond, Jaeyoung Choi, Chris Jepeway, Kai Li, Bob Manchek, and Clint Whaley.

James Plank is 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 by the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615.

References

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Systems Journal*, 38(6):673–681, November 1994.

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

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

[4] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey. Floating point fault tolerance with backward error assertions. *IEEE Transactions on Computers*, C-44(2):302–311, February 1995.

[5] 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.

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

[7] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming (to appear)*, 1996.

[8] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel, dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numerical Algorithms*, 10:379–399, 1995.

[9] 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.

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

[11] 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, PA, second edition, 1991.

[12] 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.

[13] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice Hall, Englewood Cliffs, NJ, 1977.

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

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

[16] G. Golub and C. Van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, second edition, 1989.

[17] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Psaaing Interface*. MIT Press, Cambridge, MA, 1994.

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

[19] 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.

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

[21] 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.

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

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

[24] J. S. Plank, Y. Kim, and J. J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Accepted for publication in* "Journal of Parallel Distributed Computing", June 1997.

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

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

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

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

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

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

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

[32] G. W. Stewart. *Introduction to Matrix Computations*. Academic Prcess, San Diego, CA, 1973.

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

[34] G. Sure, R. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. pages 279–288, June 1995.

[35] 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.

[36] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice Hall, Englewood Cliffs, NJ, 1963.

[37] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford, Clarendon Press, 1965.