UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Checksum-Based Fault Tolerance for LU Factorization

Permalink
https://escholarship.org/uc/item/7tk439n9

Author
Davies, Teresa

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Checksum-Based Fault Tolerance for LU Factorization

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Teresa Davies

December 2014

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Walid Najjar
Dr. Rajiv Gupta
Dr. Philip Brisk

Copyright by
Teresa Davies
2014

The Dissertation of Teresa Davies is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank my advisor for his valuable help throughout the process of developing this work.

I would also like to thank the National Center for Computational Sciences and the Golden Energy Computing Organization for the use of their computing resources.

This research was partly supported by the National Science Foundation under grants #OCI-0905019, #CNS-1304969, #CCF-1305622, and #OCI-1305624 and the Department of Energy under grant DE FE#0000988.

Parts of this work have been published in the following papers:

- T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In Proceedings of the International Conference on Supercomputing, 2011.

- T. Davies and Z. Chen. Correcting soft errors online in LU factorization. In Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing, 2013.

ABSTRACT OF THE DISSERTATION

Checksum-Based Fault Tolerance for LU Factorization

by

Teresa Davies

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2014
Dr. Zizhong Chen, Chairperson

In high-performance systems, the probability of failure is higher for larger systems. The probability that a failure will occur before the end of the computation increases as the number of processors used in a high performance computing application increases. For long running applications using a large number of processors, it is essential that fault tolerance be used to prevent a total loss of all finished computations after a failure. There are two main classes of errors that we are concerned with: errors involving loss of data, and errors involving corruption of data.

A fail-stop failure, where a process is lost along with its data, can be handled for any application with checkpointing. While checkpointing has been very useful to tolerate failures for a long time, it often introduces a considerable overhead especially when applications modify a large amount of memory between checkpoints and the number of processors is large. Therefore an application-specific approach to handling fail-stop failures is likely to allow fault tolerance with much lower overhead.

Errors in calculations may occur that cannot be detected by any other means. These are called soft errors, and usually are the errors in the data that cause the program to deliver the wrong result at the end, but do not have any other effect that would indicate an error occurred. An existing approach uses duplication to confirm the results of the calculation at the end, and it can be applied to any problem to protect against soft errors.

In this work, we propose an algorithm-based recovery scheme for the High Performance Linpack benchmark (which modifies a large amount of memory in each iteration) to tolerate fail-stop failures and soft errors. It has been proved by Huang and Abraham that,

if LU factorization is used to factor a checksum matrix, the checksum relationship will be preserved *at the end of the computation.* We first demonstrate that, for both the top and the left looking LU factorization algorithms, the checksum relationship in the input checksum matrix is *NOT maintained in the middle of the computation.* We then prove that in the right looking LU factorization algorithm, the checksum relationship will be maintained *at each step in the middle of the computation.* It was proved by Huang and Abraham that a checksum added to a matrix will be maintained after the matrix is factored. We demonstrate that, for the right-looking LU factorization algorithm, the checksum is maintained at each step of the computation. Based on this checksum relationship maintained at each step in the middle of the computation, we demonstrate that errors in High Performance Linpack can be tolerated.

Because of error propagation, the existing approach for soft errors has to repeat calculations. Our approach detects and corrects errors before they are propagated. The frequency of checking can be adjusted for the error rate, resulting in a flexible method of fault tolerance. Because no periodical checkpoint is necessary during computation and no roll-back is necessary during recovery, the proposed recovery scheme is highly scalable and has a good potential to scale to extreme scale computing and beyond. Experimental results on the supercomputer Jaguar demonstrate that the fault tolerance overhead introduced by the proposed recovery scheme is negligible.

These techniques are guaranteed to handle at least one error. In certain cases, multiple simultaneous errors can be handled, but it is not guaranteed. If multiple errors are likely, multiple checksums can be used to handle more than one simultaneous error. Multiple checksums can be created using coefficients to calculate different sums from the same set of elements. Since the calculations to recover from an error use real numbers, it is extremely important that the coefficients be chosen to minimize the error created by a recovery, and this is the main challenge for adding the ability to recover from multiple errors.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
- 1.1 Fail-stop failures 1
- 1.2 Soft errors 2

2 Related Work **5**
- 2.1 Algorithm-Based Fault Tolerance 5
- 2.2 Diskless Checkpoint 6
- 2.3 Applications of Checksum-Based Recovery 7
- 2.4 Soft Error Detection and Recovery 7
- 2.5 Multiple Simultaneous Errors 8

3 High Performance Linpack **9**
- 3.1 2D Block-Cyclic Data Distribution 9
- 3.2 Right-looking Algorithm 10
- 3.3 Partial pivoting 11

4 Fail-Stop Failures **14**
- 4.1 Failure Model 14
- 4.2 Checksum Setup 15
- 4.3 Checksum behavior during LU factorization 18
  - 4.3.1 Left-looking LU factorization 18
  - 4.3.2 Right-looking LU factorization 19
- 4.4 HPL and Distributed Checksum 21
- 4.5 Performance Analysis 23
  - 4.5.1 Overhead of constructing the checksum 23
  - 4.5.2 Overhead of computation 24
  - 4.5.3 Overhead of one recovery 27
  - 4.5.4 Comparison to checkpointing 28
  - 4.5.5 Expected time to complete a calculation 29

4.6 Experiments . . . 31
4.6.1 Platforms . . . 31
4.6.2 Overhead without recovery . . . 33
4.6.3 Overhead with recovery . . . 33
4.6.4 Algorithm-based recovery versus diskless checkpointing . . . 36

5 Soft Errors **40**
5.1 Failure Model . . . 40
5.2 Error Propagation in High Performance Linpack . . . 41
5.2.1 Right-looking LU factorization . . . 41
5.2.2 Error Propagation . . . 44
5.3 Soft Error Detection, Location, and Correction . . . 46
5.3.1 Checksum Setup . . . 46
5.3.2 Proof of Correctness for Checksum . . . 49
5.3.3 Local Checksum after Panel Factorization . . . 52
5.3.4 Error Detection and Location . . . 54
5.3.5 Soft Error Correction . . . 56
5.4 Performance Analysis . . . 58
5.4.1 Overhead of checksum setup and error detection . . . 58
5.4.2 Overhead of checking for errors in stored values . . . 59
5.4.3 Overhead of computation . . . 59
5.4.4 Time complexity of HPL . . . 60
5.4.5 Overhead of recovery . . . 62
5.4.6 Comparison to alternatives . . . 62
5.5 Experimental Results . . . 63
5.5.1 Platforms . . . 63
5.5.2 Overhead without failure . . . 63
5.5.3 Overhead with Failure: Our approach compared to ABFT . . . 65

6 Multiple error correction **68**
6.1 Existing approaches . . . 69
6.2 Condition number and supercondition number . . . 69
6.3 Simulated annealing approach . . . 70
6.4 Origin of moves . . . 71
6.5 Variations on add-and-delete . . . 74
6.6 Results . . . 75
6.7 Optimization . . . 78

7 Conclusion **80**

Bibliography **81**

List of Figures

2.1 A checkpoint is periodically saved to a checkpoint process. 8

3.1 Global and local matrices under the 2D block cyclic distribution [1]. 10
3.2 Three different algorithms for LU factorization [2]. The part of the matrix that changes color is the part that is modified by one iteration. Unlike right-looking, left- and top-looking variants change only a small part of the matrix in one iteration. 12

4.1 Global view of a matrix with checksum blocks and diagonal blocks marked. Because of the checksum blocks, the diagonal of the original matrix is displaced in the global matrix. 17
4.2 Global and distributed view of a matrix, with checksum added 24
4.3 Top-looking algorithm does not maintain a checksum 25
4.4 Right-looking algorithm maintains a checksum at each step 26
4.5 On Jaguar, when run with and without checksum fault tolerance, the times are very similar. In fact, variations in the runtime from other causes are greater than the time added by the fault tolerance, with all effects included. 35
4.6 Fault tolerance overhead without recovery: Algorithm-based recovery versus diskless checkpointing 37
4.7 On smaller runs on Ra, the difference between checksum and checkpoint can be easily seen. 37
4.8 Fault tolerance overhead with recovery: Algorithm-based recovery versus diskless checkpointing 38

5.1 The part of the matrix that is involved in the three parts of an iteration: (1) panel factorization, (2) row panel update, (3) trailing matrix update. 43
5.2 An error in the row panel is propagated down the column. If there is a second error in the matrix, it creates a situation where there are two errors in the same row, making recovery impossible. 45
5.3 A single error can be propagated to a very large section of the matrix when it occurs in the panel. 47
5.4 An example matrix, shown first in the original global view, then as it would be distributed on a 3×2 grid with a block size of 2. 48

5.5 The matrix after local sums are appended to the matrices in each process, and how this addition affects the global matrix. 48
5.6 The matrix after global sums are also added, with the global view of the matrix with all checksums. 49
5.7 Relevant sections of a matrix in the factorization of one panel. 53
5.8 The small boxes indicate the sections of the matrix where errors could occur at different points in an iteration. 55
5.9 The element -12 is replaced by 10. The sum verification shows that an error exists: $-1 + 10 \neq -13$. This is not enough to tell which element is incorrect. The reduce is done on the whole panel, and the result for this element is $1 - 16 - (-3) = 12$, restoring the correct element. 57
5.10 Assuming the times shown as the time between failures on average, the expected times using ABFT and our method are shown. 66

6.1 A $3 \times n$ matrix with one particular submatrix highlighted 71
6.2 Comparison of two add-and-delete approaches when 10 columns are added then deleted at a time 73
6.3 Comparison of two add-and-delete approaches when 1 column is added then deleted at a time 73
6.4 Three runs of 3×10 matrix with simulated annealing 76
6.5 A run of 2×100 matrix with simulated annealing 76
6.6 A run of 3×40 matrix with simulated annealing 77
6.7 A run of 3×100 matrix with simulated annealing 77
6.8 For a 2×1000 matrix, comparison in progress between adding 1 or 3 random columns per move 78

List of Tables

4.1 Jaguar: local matrix size 2000×2000, block size 64 32
4.2 Jaguar: local matrix size 4000×4000, block size 64 32
4.3 Jaguar: local matrix size 2000×2000, block size 128 32
4.4 Kraken: local matrix size 2000×2000, block size 64 34
4.5 Ra: local matrix size 4000×4000, block size 64 34
4.6 Jaguar: local matrix size 2000×2000, block size 64 34
4.7 Ra: local matrix size 4000×4000, block size 64 35
4.8 Jaguar: projected checkpoint overhead 39

5.1 Ra: $N \times N$ matrix on $P \times P$ process grid, T_p is the panel verification time, and T_t is the trailing matrix verification time, block size 256, time in seconds, performance in Gflops 64
5.2 Kraken: $N \times N$ matrix on $P \times P$ process grid, T_p is the panel verification time, T_t is the trailing matrix verification time, block size 256, time in seconds, performance in Gflops 64
5.3 With a $N \times N$ matrix, the ratio of trailing matrix verification to panel verification on Ra and Kraken 66

Chapter 1

Introduction

1.1 Fail-stop failures

Fault tolerance is becoming more important as the number of processors used for a single calculation increases [3]. When more processors are used, the probability that one will fail increases [4]. Therefore, it is necessary, especially for long-running calculations, that they be able to survive the failure of one or more processors. One critical part of recovery from failure is recovering the lost data. Depending on the application, which method has the least overhead varies. General methods for recovery exist, but for some applications specialized optimizations are possible. There is usually overhead associated with preparing for a failure, even during the runs when no failure occurs, so it is important to choose the method with the lowest possible overhead so as not to hurt the performance more than necessary. We have developed a technique using checksums that has lower overhead than any alternative for the LU factorization and other operations.

There are various approaches to the problem of recovering lost data involving saving the processor state periodically in different ways, either by saving the data directly [5, 2, 6, 7] or by maintaining some sort of checksum of the data [1, 8, 9, 10] from which it can be recovered. A method that can be used for any application is Plank's diskless checkpointing [6, 11, 12, 13, 14, 15, 16], where a copy of the data is saved in memory, and when a node is lost the data can be recovered from the other nodes. However, its performance degrades when there is a large amount of data changed between checkpoints [2], as in for instance matrix operations. Since matrix operations are an important part of most

large calculations, it is desirable to make them fault tolerant in a way that has lower overhead than diskless checkpointing.

Chen and Dongarra discovered that, for some algorithms that perform matrix multiplication, it is possible to add a checksum to the matrix and have it maintained at every step of the algorithm [17, 18]. If this is the case, then a checksum in the matrix can be used in place of a checkpoint to recover data that is lost in the event of a processor failure. In addition to matrix multiplication, this technique has been applied to the Cholesky factorization [9]. In this work, we extend the checksum technique to the LU factorization used by High Performance Linpack (HPL) [19].

LU is different from other matrix operations because of pivoting, which makes it more costly to maintain a column checksum. Maintaining a column checksum with pivoting would require additional communication. However, we can show that HPL has a feature that makes the column checksum unnecessary. We prove that the row checksum is maintained at each step of the algorithm used by HPL, and that it can be used to recover the required part of the matrix. Therefore, in this method we use only a row checksum, and it is enough to recover in the event of a failure. Additionally, we show that two other algorithms for calculating the LU factorization do not maintain a checksum.

The checksum-based approach that we have used to provide fault tolerance for the dense matrix operation LU factorization has a number of advantages over checkpointing. The operation to perform a checksum or a checkpoint is the same or similar, but the checksum is only done once and then maintained by the algorithm, while the checkpoint has to be redone periodically. The checkpoint approach requires that a copy of the data be kept for rolling back, whereas no copy is required in the checksum method, nor is rolling back required. The operation of recovery is the same for each method, but since the checksum method does not roll back its overhead is less. Because of all of these reasons, the overhead of fault tolerance using a checksum is significantly less than with checkpointing.

1.2 Soft errors

Modern computer systems are becoming more vulnerable to soft errors, both because of smaller, denser components and because they are becoming larger. The more components there are, the more likely a failure. Dealing with soft errors is becoming more

of a concern [20, 21, 22, 23, 24], with both hardware and software solutions proposed. Although methods exist to detect errors in stored values, the more calculations there are, the more likely it is that an error will go undetected. If a soft error changes the data only, it is possible that the error will be completely undetected until the calculation returns the wrong answer at the end, because no other way of detecting soft errors exists [25, 26]. For a large matrix operation, one error could be propagated to a large fraction of the matrix. Therefore, it is useful to be able to detect errors that are not found any other way soon after they occur, and to recover the correct value in order to continue. A method that is designed for a specific matrix operation will have lower overhead than a more general technique.

An existing technique is algorithm-based fault tolerance (ABFT) [27, 28, 29, 30], which puts a checksum onto a matrix, and the sum will stay correct through an operation on the matrix. If the sum is incorrect at the end of the calculation, it indicates that an error occurred. Depending on the approach and the type of error, the sum may be used to determine the correct values, or the calculation may be repeated. Using a sum to correct an error at the end of the calculation is limited in the number of errors it can handle, so it is more reasonable that, when ABFT is used, an error will cause the entire calculation to be repeated. With the LU factorization, error propagation happens frequently enough to make it impossible to recover from the checksums at the end of the calculation. Instead, we are able to recover as soon as errors occur.

In the LU factorization, a single error can be propagated to large sections of the matrix [31]. This can include affecting the checksums as well. Soft errors are not easily detected, so it is necessary to check the matrix for correctness often enough that the errors will not be propagated to the extent that they can no longer be recovered. We have found that errors in different sections of the matrix are more or less sensitive to errors, so that some sections do not have to be verified as frequently.

Previously, a global checksum of a matrix has been used to make the LU factorization able to recover from fail-stop failures [32]. This approach used a single checksum, and relied on getting the information about a failure from some other source. When a process dies, it is apparent that there has been a failure, and it should also be possible to obtain the information about which process has failed. In contrast, soft errors often do not result in any noticeable sign that an error has occurred. The only indication of a soft error might be that the result of a calculation is wrong. Therefore, in our approach it is necessary

to have two separate checksums of each matrix element - one that indicates that an error occurred, and one that can be used to recover. The matrix elements must be periodically verified using one set of checksums. If the check shows that an error has occurred, the first checksums are not enough to recover from it, and a second set of checksums is needed to allow recovery to take place.

We have taken the idea of a checksum on a matrix and extended it, finding the way to set up a checksum on a matrix so that it is correct throughout the calculation. For the LU factorization, we are able to provide fault tolerance using a row checksum, where the sums are of elements in the same row. With multiple checksums we can both detect and recover errors - with soft errors, there is no other way to detect them, so at least two checksums are needed.

This approach has low overhead. With ABFT the entire calculation has to be repeated when there are more errors than the number of checksums can recover. In our method, a single iteration - a small fraction - of the calculation is repeated when an error occurs. Therefore the expected amount of recalculation due to errors is much smaller than with ABFT. The overhead when no error occurs is $O(\frac{1}{N})$, where the matrix is $N \times N$, and can be adjusted to the error rate.

Chapter 2

Related Work

2.1 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance [33, 28, 27, 10, 34, 35, 36] is a technique that has frequently been used to detect miscalculations in matrix operations. This technique consists of adding a checksum row or column to the matrices being operated on. For many matrix operations, some sort of checksum can be shown to be correct at the end of the calculation, and can be used to find errors after the calculation is done. It can be used for the LU factorization with a single additional row and column added onto the matrix, which has the sum of all elements in its row or column.

The original algorithm-based fault tolerance uses a checksum to determine at the end whether the calculation had been successful or not. If the sums are not consistent with the final result, the calculation is repeated. If the probability of a failure is p, the expected number of runs is $\frac{1}{1-p}$. If p is small, then the overhead of using this method on average is low, but the cost of repeating is larger the higher the probability of failure. The probability of failure increases linearly with the number of processors.

The existence of a checksum that is correct at the end raises the question: is the checksum correct also in the middle? It turns out that it is not maintained for all algorithms [1], but there do exist some for which it is maintained. In other cases, it is possible to maintain a checksum with only minor modifications to the algorithm, while still keeping an advantage in overhead over a diskless checkpoint. It may also be possible to use a checksum to maintain redundancy of part of the data, while checkpointing the rest. Even a

reduction in the amount of data needing to be checkpointed can give a gain in performance. The checksum approach has been used for many matrix operations to detect errors and correct them at the end of the calculation, which indicates that it may be possible to use it for recovery in the middle of the calculation as well.

The idea of adding a checksum to a matrix is useful for multiple types of errors. A checksum of a matrix can be used to both locate and correct entries in the solution matrix that are incorrect. In some cases, one checksum must be used to detect the error, then another is required in order to correct it. For fail-stop failures, the location is determined by the message passing library in the case of the failure of a processor. In this case, only one checksum is required in order to correct one failure.

2.2 Diskless Checkpoint

In order to do a checkpoint, it is necessary to save the data so that it can be recovered in the event of a failure. One approach is to save the data to disk periodically [7]. In the event of a failure, all processes are rolled back to the point of the previous checkpoint, and their data is restored from the data saved on the disk. Unfortunately, this method does not scale well. For most scientific computations, all processes make a checkpoint at the same time, so that all of the processes will simultaneously attempt to write their data to the disk. Most systems are not optimized for a large amount of data to go to the disk at once, so this is a serious bottleneck, made worse by the fact that disk accesses are typically extremely slow.

In response to this issue diskless checkpointing [6] was introduced. Each processor saves its own checkpoint state in memory, thereby eliminating the need for a slow write to disk. Additionally, an extra processor is used just for redundancy, as shown in figure 2.1, which could be parity, checksum, or some other appropriate reduction. Typically there would be a number of such processors, each one for a different group of the worker processors. This way, upon the failure of a process in one group, all of the other processes can revert to their stored checkpoint, and the redundant data along with the data of all the other processes in the group is used to recover the data of the failed process. The lost process is recovered by reversing the operation that was used to compute the values stored in the checkpoint process. The checkpoint process can potentially be used to replace the

lost process, or some form of redundancy can be used to provide a replacement for the lost process [37].

Diskless checkpointing has several similarities to the checksum-based approach. When a checksum row is added to a processor grid, each checksum processor plays the role of the redundancy processor in a diskless checkpoint. The difference is that the redundancy of the checksum data is maintained naturally by the algorithm. Therefore there are two main benefits: the working processors do not have to use extra memory keeping their checkpoint data, and less overhead is introduced in the form of communication to the checkpoint processors when a checkpoint is made. A key factor in the performance of diskless checkpointing is the size of the checkpoint. The overhead is reduced when only data that has been changed since the last checkpoint is saved [38]. However, matrix operations are not susceptible to this optimization [2], since many elements, up to the entire matrix, could be changed at each step of the algorithm. When the checkpoint is large, the overhead is large [39].

2.3 Applications of Checksum-Based Recovery

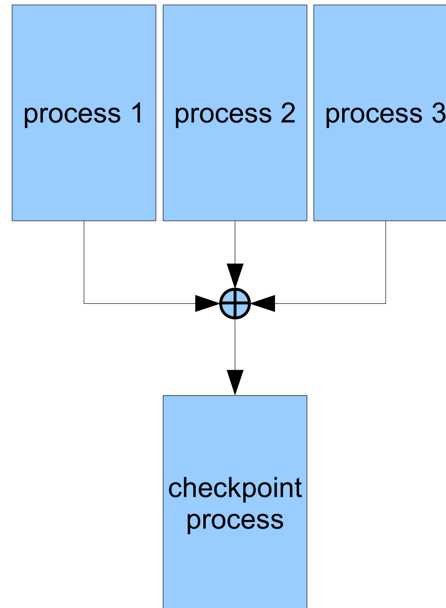
It has been shown in [1] how a checksum is maintained in the outer product matrix-matrix multiply. It is also made clear that not all algorithms will maintain the checksum in the middle of the calculation, even if the checksum can be shown to be correct at the end. So it is important to ensure that the algorithm being used is one for which a checksum is maintained before using a checksum for recovery. Another application of a checksum is described in [9]. Here a checksum is used to recover from a failure during the ScaLAPACK Cholesky factorization.

2.4 Soft Error Detection and Recovery

Although handling of soft errors has not been very important in the past, as systems and problems continue to increase in size, the importance of detecting and correcting soft errors has increased, leading to techniques to deal with them being created [40, 41]. Handling soft errors for matrix operations is an important problem [31, 30, 42].

To both detect and correct soft errors, multiple checksums are needed. One approach is to use differently weighted checksums of the same elements in the matrix [8, 43, 44].

Figure 2.1: A checkpoint is periodically saved to a checkpoint process.



This approach requires finding optimal weights for numerical stability, which is a difficult problem. Our approach uses only single checksums of different sets of elements to achieve the same effect as multiple checksums. One checksum is kept of the local matrix on each process, and another checksum is kept of the global matrix. The local checksum can be used to detect errors, and the global checksum to correct them. Any approach that uses a single checksum could be extended to use multiple weighted checksums to provide tolerance of more errors.

2.5 Multiple Simultaneous Errors

Algorithm-based fault tolerance can be set up with multiple checksums weighted with coefficients in order to handle multiple errors [45]. The same approach can be applied when the errors will be found in the middle of the calculation instead of at the end.

Chapter 3

High Performance Linpack

HPL performs a dense LU factorization using a right-looking partial pivoting algorithm. The matrix is stored in two-dimensional block-cyclic data distribution. These are the most important features of HPL to our technique.

3.1 2D Block-Cyclic Data Distribution

For many parallel matrix operations, the matrix involved is very large. The number of processors needed for the operation may be selected based on how many it takes to have enough memory to fit the entire matrix. It is common to divide the matrix up among the processors in some way, so that the section of the matrix on a particular processor is not duplicated anywhere else. Therefore an important fact about recovering from the failure of a processor is that a part of the partially finished matrix is lost, and it is necessary to recover the lost part in order to continue from the middle of the calculation rather than going back to the beginning.

Matrices are often stored in 2D block cyclic fashion [1, 9, 2]. Block cyclic distribution is used in HPL. This storage pattern creates a good load balance for many operations, since it is typical to go through the matrix row by row or column by column (or in blocks of rows or columns). With a block cyclic arrangement, matrix elements on a particular processor are accessed periodically throughout the calculation, instead of all at once. An example of 2D block cyclic distribution is shown in figure 3.1. As this figure indicates, the global matrix will not necessarily divide evenly among the processors. However, we currently are

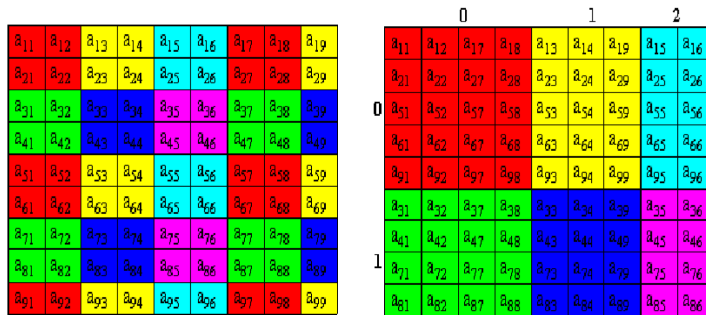


Figure 3.1: Global and local matrices under the 2D block cyclic distribution [1].

assuming matrices that divide evenly along both rows and columns of the processor grid to simplify the problem. The block size is how many contiguous elements of the matrix are put in a processor together. Blocks are mapped to processors cyclically along both rows and columns.

3.2 Right-looking Algorithm

In Gaussian elimination, the elements of L are found by dividing some elements of the original matrix by the element on the diagonal. If this element is zero the division is clearly not possible, but with floating point numbers a number that is very close to zero could be on the diagonal and not be obvious as a problem. In order to ensure that this does not happen, algorithms for LU factorization use pivoting, where the row with the largest element in a the current column is swapped with the current row. The swaps are not done in the L matrix, which enforces our idea that it is not necessary to be able to recover it. The equation $Ax = b$ can be rewritten as $LUx = b$ using the LU factorization, where $Ux = y$, so that $Ly = b$. The algorithm transforms b to y, so that L is not needed to find x at the end. The factorization is performed in place, which means that the original matrix is replaced by L and U.

HPL does the factorization by panels instead of one column at a time. A panel is a block of elements that extends from the current diagonal element to the last row and is the width of a block. The order of one iteration is panel factorization, panel broadcast, and trailing matrix update. This is why it makes sense to not apply pivots to L: it would mean

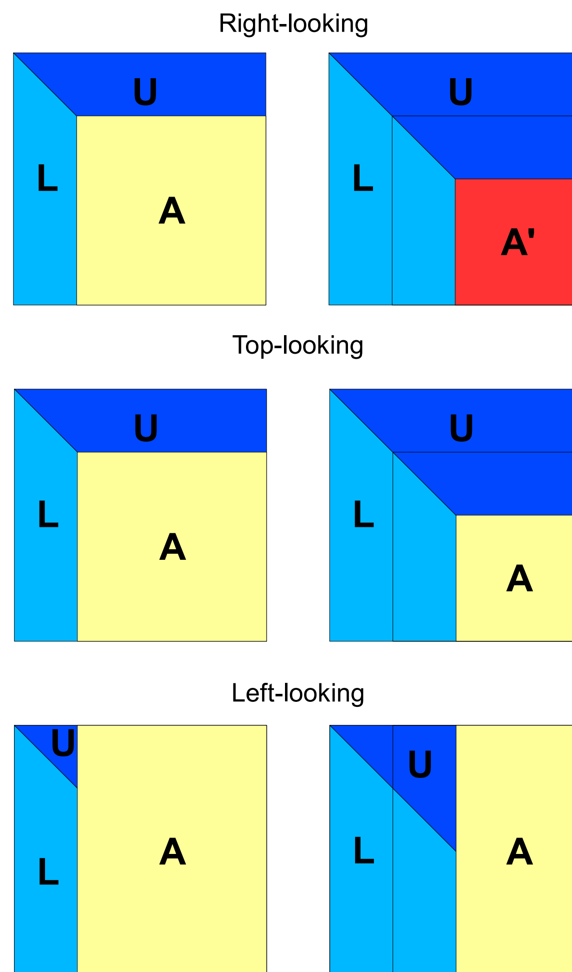
sending the information to the processors that hold parts of L, when the swap in L is not necessary for the solution at the end. By not applying the pivots to L, some communication time is saved each iteration.

The right-looking variation is the version of LU that updates the trailing matrix. When one row and column are factored in basic Gaussian elimination, the remaining piece of the original matrix is updated by subtracting the product of the row and column. It is possible to put off the update of a particular section of the matrix until that section is going to be factored. However, this approach means that large sections of the matrix are unchanged after each iteration. Figure 3.2 shows how the matrix is changed in each of the three variations. Because of the data distribution, the load balance is best when each processor does some work on its section of the matrix during each iteration. When the trailing matrix is not updated at each step, the work of updating it has to be done consecutively by a small set of processors when the factorization reaches each new part. When the update is done at every step, the work is done at the same time that other processors are doing factorization work, taking less total time.

3.3 Partial pivoting

In the LU factorization, some values in the matrix are divided by elements on the main diagonal. If a value on the diagonal is either zero or very small relative to the other values in the matrix, it will cause a roundoff error in the result of the factorization. Fortunately, it is possible to exchange rows of the matrix without changing the answer found by the algorithm. (If the matrix represents a system of equations, then exchanging rows is equivalent to putting the equations in a different order, which clearly does not change the solution to the system of equations.) It would be difficult for the algorithm to attempt to judge whether each main diagonal value is too small; instead, it simply searches down columns to find the largest available value for every spot in the diagonal and exchanges rows to put values in the correct location. The name partial pivoting refers to the fact that the entire matrix is not updated with the new row permutation. Instead, only rows to the right of the current column are swapped; the L matrix is not updated because it is not necessary to the solution at the end. This decision has two effects on the way the checksum is set up, both of which make using a row checksum only to be the better option.

Figure 3.2: Three different algorithms for LU factorization [2]. The part of the matrix that changes color is the part that is modified by one iteration. Unlike right-looking, left- and top-looking variants change only a small part of the matrix in one iteration.



First, the checksum down columns becomes a sum of only the elements in the L matrix after the factorization is completed. However, once the factorization of elements from the L matrix is completed, those elements are never used again. Therefore protecting them from being lost is not necessary. Second, the checksum value added along a column would sometimes be the largest value in a particular column, and would therefore be the value detected by the pivoting routine to be swapped into the main diagonal. The method of pivoting would have to be altered to skip checksum rows, requiring extra checking and slowing it down. Because the column fault tolerance is already unnecessary to the final answer, it seems that the column checksum is better left out of the LU factorization fault tolerance scheme when the goal is solving a system of equations. The application will have slightly better performance while still being able to find the correct answer at the end.

Chapter 4

Fail-Stop Failures

4.1 Failure Model

The type of failure that we focus on in this section is a fail-stop failure, which means that a process stops and loses all of its data. The main task that we are able to perform is to recover the lost data. Another type of failure that this technique could handle is a soft failure, where one or more numbers become incorrect without any outwardly detectable signs. A checksum can be used to detect and correct these types of errors as well. However, it has higher overhead than handling only fail-stop failures. Detecting a soft failure requires that all elements of the matrix be made redundant by two different checksums, where different elements of the matrix go into each checksum.

For example, having both row and column checksums allows detection. The matrix that will be factored into L and U starts with double redundancy but changes to single redundancy in L and U, so another way of making a checksum for L and U would be required for detecting soft failures. The other way in which the overhead of detecting and recovering from soft failures is greater than recovering from fail-stop failures is that detecting soft failures requires periodically checking to see if a failure has occurred. This does not have to be done every step, but it should be done often enough that a second failure is not likely to occur and make it impossible to recover.

Soft failures are another problem that can use the checksum technique, although the idea behind having multiple sums for the same set of elements is similar. Instead, we are focused on recovering from one hard failure, where the fact of the failure and the processor

that failed are provided by some other source, presumably the MPI library. The ability to continue after a process failure is not something that is currently provided by most MPI implementations. However, for testing purposes it is possible to simulate a failure in a reasonable manner. When one process fails, no communication is possible, but the surviving processes can still do work. Therefore, the general approach to recovery is to have the surviving processes write a copy of their state to the disk. This information can then be used to restart the computation. The state of the surviving processes is used to recover the failed process.

4.2 Checksum Setup

Adding checksums to a matrix stored in block cyclic fashion is most easily done by adding an extra row, column, or both of processors to the processor grid. The extra processors hold an element by element sum of the local matrices on the processors in the same row for a row checksum or the same column for a column checksum. This way processors hold either entirely checksum elements or entirely normal matrix elements. If this were not the case it might make recovery impossible.

The checksum is computed by a reduce across rows of the entire local matrix of each process. The result of the reduce is stored in the last process in the row, which is added after the processes holding the original matrix. Computing lost elements is the reverse process, subtracting all of the other local matrices in the row from the local matrix of the checksum process.

In the following matrix, a row checksum has been added. The last element in each row is the sum of the other elements.

$$\begin{pmatrix} 3 & 1 & 2 & 6 \\ 2 & 4 & 3 & 9 \\ 4 & 3 & 1 & 8 \end{pmatrix}$$

If one element is unknown, as represented by an x here, it can be calculated from the other elements.

$$\begin{pmatrix} 3 & 1 & 2 & 6 \\ 2 & \mathrm{x} & 3 & 9 \\ 4 & 3 & 1 & 8 \end{pmatrix}$$

Here $2 + x + 3 = 9$, so $x = 4$, which is the correct recovered value. This relationship works the same way if the individual elements are replaced by matrices. For a distributed matrix, the local matrices are added together to create a checksum matrix for that row, which is stored in another process in the same row of the process grid.

The block cyclic distribution makes it so that the checksum elements are interpreted as being spread throughout the global matrix, rather than all in the last rows or columns as they would be in a sequential matrix operation. Figure 4.1 shows the global view of a matrix with a row checksum. The width of each checksum block in the global matrix is the block size nb. The block size is also used as the width of a column block that is factored in one iteration. Periodically during the calculation checksum elements may need to be treated differently from normal elements. In the LU factorization, when a checksum block on the main diagonal is come to, that iteration can be skipped because the work done in it is not necessary to maintaining the checksum. When there is only a row checksum, as in this case, the elements on the diagonal of the original matrix are not all on the diagonal of the checksum matrix. Instead of considering every element $a_{i,i}$ of the matrix during Gaussian elimination, the element is $a_{i,i+c \cdot nb}$, where c is the number of checksum blocks to the left of the element under consideration.

When a column checksum is used it is necessary to skip over checksum elements when searching for a pivot. The largest element in a column is chosen to be switched with the element on the diagonal. If all the elements in a column were positive, then the checksum element would be the largest and would be selected as the pivot if it was treated as a normal part of the matrix. The checksum elements cannot be moved off of their processors because the method partly relies on processors containing either all checksum or all normal elements.

It would be possible to have a system of recovery where processors contain both checksum and normal elements, but only if only one of the elements going into a sum, or the sum itself, were on the failed processor. Clearly, if both a sum and an element of the sum are lost, it is not possible to recover the lost element. Requiring that processors contain strictly checksum or non-checksum elements removes the possibility that sums and their elements could be on the same processor and simplifies the process of maintaining the sums.

The checksum elements must be skipped when pivoting. If the matrix were not stored in block cyclic fashion, the checksum elements would all be at the end of the matrix.

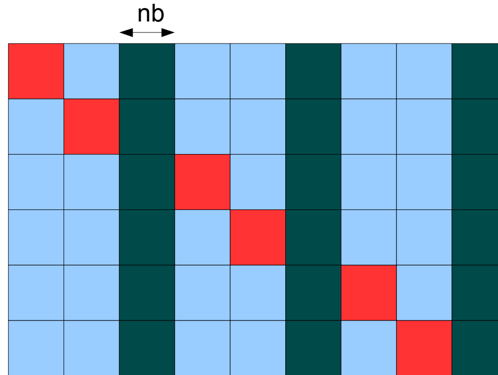


Figure 4.1: Global view of a matrix with checksum blocks and diagonal blocks marked. Because of the checksum blocks, the diagonal of the original matrix is displaced in the global matrix.

This would simplify the calculation slightly because there would be no need to skip over checksum elements or treat them differently for pivoting. However, the complication is very minor.

Another feature of the matrix storage for LU that is worth noting is that the factorization is done in place, so that L and U replace the original matrix one panel at a time. An implication of this fact is that the local matrix in a particular processor may have sections of all three matrices. This means that different checksum might be needed to recover different elements of the local matrix. This is one of the potential issues that are avoided by our choice of LU within HPL.

It is not guaranteed that the checksum can be maintained for any algorithm for calculating the LU factorization. The method relies on whole rows of the matrix being operated on in the same iteration.

The right-looking LU factorization can be used with our technique, but the left-looking LU factorization cannot. In the left-looking variation, columns to the left of the currently factored column are not updated. So the row checksum will not be maintained, and it will not be possible to recover U.

4.3 Checksum behavior during LU factorization

Like all matrix operations, there are different algorithms for the LU factorization, not all of which will necessarily maintain a checksum at each step. However, there is at least one algorithm for LU that maintains the checksum in a way that is useful for recovery. This is the right-looking algorithm with partial pivoting. If pivoting is not used at all, the right-looking algorithm also maintains a checksum, but without pivoting the outcome of the algorithm is not as numerically stable.

4.3.1 Left-looking LU factorization

Left-looking and top-looking LU factorization do not maintain a checksum because various parts of the matrix are not updated in a given step, shown in figure 3.2 by the sections that do not change color. When some sections of the matrix are changed and others are not, a sum that includes elements from both sections will not be maintained. Only the right-looking variant updates the entire matrix at every step, maintaining the checksum. Interestingly, this characteristic makes the right-looking variant the least favorable for diskless checkpointing.

The left-looking algorithm can be shown to not maintain the necessary checksums in the middle of the operation.

This matrix has row and column checksums added to it.

$$\begin{pmatrix} 3 & 1 & 4 \\ 2 & 4 & 6 \\ 5 & 5 & 10 \end{pmatrix}$$

Going through the left-looking LU factorization:

$$\begin{pmatrix} 3 & 1 & 4 \\ 2/3 & 4 & 6 \\ 5/3 & 5 & 10 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 1 & 4 \\ 2/3 & 10/3 & 6 \\ 5/3 & 1 & 10 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 1 & 4 \\ 2/3 & 10/3 & 10/3 \\ 5/3 & 1 & 0 \end{pmatrix}$$

The sum relationships in this matrix are: in the first row, 3+1=4; in the first column, 1+2/3=5/3 (1 on diagonal); in the second row, 10/3=10/3; and in the last row the 1 is the remaining sum of the 1 on the diagonal, and the 0 is the sum of the trailing matrix with no elements in it.

In the intermediate steps, there are checksum relationships along columns, but they do not exist for every row. Therefore the left-looking variation cannot be used with our checksum method. A similar problem exists with the top-looking variation, since the trailing matrix is not updated each step.

4.3.2 Right-looking LU factorization

The right-looking algorithm for LU maintains a checksum at each step as is required because each iteration operates on entire rows and columns. If an entire row is multiplied by a constant, or rows are added together, the checksum will still be correct. The same is true of columns. When an operation finishes factoring a part of the matrix into part of L or U, the checksums in that part will be sums on L or U only. Elements belonging to L will go into column checksums only, and elements belonging to U will go into row checksums only. The elements of L and U that are not stored in the matrix-ones on the diagonal of L and zeros above or below the diagonal-also go into the checksums.

Right-looking LU factorization maintains a checksum at each step, as shown below. This version is also faster than the others, left-looking and top-looking.

The right-looking algorithm is:

```
for i = 1 to n-1
    A(i+1:n,1) = A(i+1:n,1)/A(i,i)
    A(i+1:n,i+1:n) = A(i+1:n,i+1:n)
                   - A(i+1:n,i)*A(i,i+1:n)
```

In an iteration of the loop, the original matrix is:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & \sum_{j=1}^{n} a_{1j} \\ a_{21} & a_{22} & \dots & a_{2n} & \sum_{j=1}^{n} a_{2j} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & \sum_{j=1}^{n} a_{nj} \end{pmatrix}$$

Dividing it up by the sections that are relevant to the step, this matrix is

$$\begin{pmatrix} a_{11} & A_{12} & \sum A_{12} \\ A_{21} & A_{22} & \sum A_{22} \end{pmatrix}$$

where

$$A_{12} = \begin{pmatrix} a_{12} & \dots & a_{1n} \end{pmatrix}$$

$$\sum A_{12} = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}$$

$$A_{22} = \begin{pmatrix} a_{22} & \dots & a_{2n} \\ \vdots & & \vdots \\ a_{n2} & \dots & a_{nn} \end{pmatrix}$$

$$\sum A_{22} = \begin{pmatrix} \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{nj} \end{pmatrix}$$

The first part of the iteration makes the matrix into

$$\begin{pmatrix} a_{11} & A_{12} & \sum A_{12} \\ A_{21}/a_{11} & A_{22} & \sum A_{22} \end{pmatrix}$$

The second step modifies the trailing matrix as follows:

$$
\begin{aligned}
& \begin{pmatrix} A_{22} & \sum A_{22} \end{pmatrix} - \begin{pmatrix} A_{21}/a_{11} \end{pmatrix} \begin{pmatrix} A_{12} & \sum A_{12} \end{pmatrix} \\
= & \begin{pmatrix} A_{22} & \sum A_{22} \end{pmatrix} - \begin{pmatrix} A_{21}A_{12}/a_{11} & A_{21}/a_{11} \sum A_{12} \end{pmatrix} \\
= & \begin{pmatrix} A_{22} - A_{21}A_{12}/a_{11} & \sum A_{22} - A_{21}/a_{11} \sum A_{12} \end{pmatrix}
\end{aligned}
$$

Note that $A_{22} - A_{21}A_{12}/a_{11} = a_{ij} - a_{i1}a_{1j}/a_{11}$ for $i = 2, \dots n$ and $j = 2, \dots n$. The term representing the sums is

$$
\begin{aligned}
& \sum A_{22} - A_{21}/a_{11} \sum A_{12} \\
= & \begin{pmatrix} \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{nj} \end{pmatrix} - \begin{pmatrix} a_{21}/a_{11} \\ \vdots \\ a_{n1}/a_{11} \end{pmatrix} \begin{pmatrix} \sum_{j=1}^{n} a_{1j} \end{pmatrix} \\
= & \begin{pmatrix} \sum_{j=1}^{n} a_{2j} - a_{21}a_{1j}/a_{11} \\ \vdots \\ \sum_{j=1}^{n} a_{nj} - a_{n1}a_{1j}/a_{11} \end{pmatrix} \\
= & \begin{pmatrix} \sum_{j=2}^{n} a_{2j} - a_{21}a_{1j}/a_{11} \\ \vdots \\ \sum_{j=2}^{n} a_{nj} - a_{n1}a_{1j}/a_{11} \end{pmatrix}
\end{aligned}
$$

The first term of each sum is zero. Therefore they become sums of the elements in the trailing matrix only. The trailing matrix contains correct checksums at the end of the iteration. The row that became part of U has a checksum for itself. The column that is part of L no longer has a checksum that it is part of, but with the HPL algorithm it is no longer needed for the final result.

4.4 HPL and Distributed Checksum

In the HPL algorithm, the L matrix is applied to the right hand side of the equation as it is factored, and so it is not necessary for solving the equation at the end, and does not need to be recovered if part of it is lost in a failure. So the process of recovering is made simpler, and requires only one reduce operation across the row with the failure. However,

both U and the original matrix A are recovered using row checksums, so the elements of L in the row are simply set to zero so that they do not affect the outcome. The recovery is done by a reduce across the row containing the failed process. If recovery of both L and U was required, reduce operations along both the row and column of the failed processor would be necessary, with each reduce involving only elements from the matrix stored in that checksum.

The checksum is most useful when it is maintained at every step of the calculation. The shorter the period of time when the checksum is not correct or all processors do not have the information necessary to update the checksum, the less vulnerable the method is. This period of time is equivalent to the time it takes to perform a checkpoint when the checkpointing method is used. If a failure occurs during the checkpoint, it is likely that the data cannot be recovered. The same might be true for the checksum method.

Specifically for HPL, it is necessary to take some extra steps to reduce this vulnerability. One iteration of the factorization is broken down into two parts: the panel factorization and the trailing matrix update. During the panel factorization, only the panel is changed, and the rest of the matrix is not affected. This operation makes the checksum incorrect. Fortunately, simply keeping a copy of the panel before factorization starts is enough to eliminate this problem. The size of the panel is small compared to the total matrix. HPL already keeps copies of panels for some of the variations available for optimization.

Once the panel factorization is completed, the factored panel is broadcast to the other processors, and the panel is used to update the trailing matrix. If a failure occurs while the panel is being factored, recovery consists of replacing the partially factored panel with its stored earlier version, then using the checksums to recover the part of the matrix that was kept on the failed processor.

Figure 4.2 shows how a checksum is added to a matrix with 2×2 block size. The matrix is distributed on a 2×3 process grid. The checksum is calculated as the sum of the local matrices in the row, so that the last process column contains a matrix that is the sum of the matrices in the same row. When the expanded matrix is viewed as a global matrix, the checksum elements appear in stripes throughout the matrix. The elements of the original matrix are near the checksums they go into in the global view of the checksum matrix.

In figure 4.3, the first step of the top-looking algorithm is shown, and the checksum

is not maintained. The checksum elements marked in red no longer reflect the elements of the matrix because some elements going into those sums are changed, but the sums themselves are not. The top-looking algorithm, like the left-looking algorithm, changes only elements in one section at a time, and specifically only one column of processes at a time. Therefore a checksum that must be kept on a separate process in order to survive a failure can never be automatically updated by these algorithms.

In contrast to the top-looking version, the first step of the right-looking algorithm in figure 4.4 maintains the checksum. The trailing matrix update has the effect of making the checksum elements consistent with the rest of the matrix. Note that the sums are of only elements in the U and trailing matrices. This reflects our decision not to make L recoverable.

4.5 Performance Analysis

Even when there is no failure, the process of preparing for one has a cost. For this method, the cost is performing the checksum at the beginning, as well as the extra processors required to hold the checksums. The checksum is done by a reduce. The number of extra processors required is the number of rows in the processor grid, since an extra processor is added to each row. The number of iterations is not increased because the checksum rows are skipped. Performing the factorization on checksum blocks is not necessary for maintaining a correct checksum, so the work done in that step would be pointless.

4.5.1 Overhead of constructing the checksum

This method competes with diskless checkpointing for overhead. Because the extra processors do the same sort of tasks as the normal processors in the same row, the time to do an iteration is not significantly increased by adding the checksums. In contrast, in order to do a checkpoint it is necessary to do extra work periodically to update the checkpoint. If the checkpoint is done every iteration, then the cost of recovery is the same as with a checksum, but the cost during the calculation is clearly higher.

Figure 4.2: Global and distributed view of a matrix, with checksum added

19	3	1	12	1	16	1	3	11	0
-19	3	1	12	1	16	1	3	11	0
-19	-3	1	12	1	16	1	3	11	1
-19	-3	-1	12	1	16	1	3	11	0
-19	-3	-1	-12	1	16	1	3	11	0
-19	-3	-1	-12	-1	16	1	3	11	0
-19	-3	-1	-12	-1	-16	1	3	11	0
-19	-3	-1	-12	-1	-16	-1	3	11	0
-19	-3	-1	-12	-1	-16	-1	-3	11	0

19	3	1	3	1	12	11	0	1	16
-19	3	1	3	1	12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	-1	16
-19	-3	-1	-3	-1	-12	11	0	-1	-16
-19	-3	1	3	1	12	11	1	1	16
-19	-3	1	3	-1	12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	-1	-16
-19	-3	-1	3	-1	-12	11	0	-1	-16

4.5.2 Overhead of computation

The only parts of an iteration that are affected by there being more processors are the parts with communication. There are broadcasts in both rows and columns, but only broadcasting in rows is affected because there are no column checksums. If the original matrix dimension is P, then with a checksum added it is $P + 1$. So the overhead of each iteration is the difference between a broadcast among $P + 1$ processors and a broadcast among P processors. Depending on the implementation, the value varies. With a binomial tree, the overhead would be $\log(P + 1) - \log P$. Using pipelining, where the time for the broadcast is nearly proportional to the size of the message, the overhead is even smaller.

The total overhead of the checksum technique is

$$T_{P+1} + (T_{P+1} - T_P) \cdot \frac{N}{nb}$$

where T_P is the time for either a broadcast or a reduce on P processors, N is the matrix dimension, and nb is the block size. N/nb is the number of iterations. It seems reasonable to assume that $T_{P+1} - T_P$ is a very small quantity. Whether this term is significant depends on the exact value and the number of iterations, but for certain ranges of matrix size the overhead is essentially the time to do one reduce across rows. The total overhead of checkpointing is

$$T_{P+1} \cdot \frac{N}{nb} / I$$

where I is the number of iterations in the checkpointing interval. There is an interval for

Figure 4.3: Top-looking algorithm does not maintain a checksum

19	3	1	3	1	12	11	0	1	16	0	0	21	31	12	3
-19	3	1	3	1	12	11	0	1	16	0	0	-17	31	12	3
-19	-3	1	3	-1	-12	11	0	1	16	0	0	-19	1	12	3
-19	-3	1	3	-1	-12	11	0	-1	16	0	0	-21	1	12	3
-19	-3	-1	-3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	-3
-19	-3	1	3	1	12	11	1	1	16	0	0	-17	25	12	4
-19	-3	1	3	-1	12	11	0	1	16	0	0	-19	25	12	3
-19	-3	1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	12	3
-19	-3	-1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	3

19	3	1	12	1	16	21	31	1	3	11	0	0	0	12	3
-19	3	1	12	1	16	-17	31	1	3	11	0	0	0	12	3
-19	-3	1	12	1	16	-17	25	1	3	11	1	0	0	12	4
-19	-3	-1	12	1	16	-19	25	1	3	11	0	0	0	12	3
-19	-3	-1	-12	1	16	-19	1	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	16	-21	1	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	-16	-21	-31	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	-16	-21	-31	-1	3	11	0	0	0	10	3
-19	-3	-1	-12	-1	-16	-21	-31	-1	-3	11	0	0	0	10	-3

19	3	1	12	1	16	21	31	1	3	11	0	0	0	12	3
-1	6	2	24	2	32	4	62	2	6	22	0	0	0	24	6
-1	0	1	12	1	16	-17	25	1	3	11	1	0	0	12	4
-1	0	-1	12	1	16	-19	25	1	3	11	0	0	0	12	3
-1	0	-1	-12	1	16	-19	1	1	3	11	0	0	0	12	3
-1	0	-1	-12	-1	16	-21	1	1	3	11	0	0	0	12	3
-1	0	-1	-12	-1	-16	-21	-31	1	3	11	0	0	0	12	3
-1	0	-1	-12	-1	-16	-21	-31	-1	3	11	0	0	0	10	3
-1	0	-1	-12	-1	-16	-21	-31	-1	-3	11	0	0	0	10	-3

19	3	1	3	1	12	11	0	1	16	0	0	21	31	12	3
-1	6	2	6	2	24	22	0	2	32	0	0	4	62	24	6
-1	0	1	3	-1	-12	11	0	1	16	0	0	-19	1	12	3
-1	0	1	3	-1	-12	11	0	-1	16	0	0	-21	1	12	3
-1	0	-1	-3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	-3
-1	0	1	3	1	12	11	1	1	16	0	0	-17	25	12	4
-1	0	1	3	-1	12	11	0	1	16	0	0	-19	25	12	3
-1	0	1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	12	3
-1	0	-1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	3

Figure 4.4: Right-looking algorithm maintains a checksum at each step

19	3	1	3	1	12	11	0	1	16	0	0	21	31	12	3
-19	3	1	3	1	12	11	0	1	16	0	0	-17	31	12	3
-19	-3	1	3	-1	-12	11	0	1	16	0	0	-19	1	12	3
-19	-3	1	3	-1	-12	11	0	-1	16	0	0	-21	1	12	3
-19	-3	-1	-3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	-3
-19	-3	1	3	1	12	11	1	1	16	0	0	-17	25	12	4
-19	-3	1	3	-1	12	11	0	1	16	0	0	-19	25	12	3
-19	-3	1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	12	3
-19	-3	-1	3	-1	-12	11	0	-1	-16	0	0	-21	-31	10	3

19	3	1	12	1	16	21	31	1	3	11	0	0	0	12	3
-19	3	1	12	1	16	-17	31	1	3	11	0	0	0	12	3
-19	-3	1	12	1	16	-17	25	1	3	11	1	0	0	12	4
-19	-3	-1	12	1	16	-19	25	1	3	11	0	0	0	12	3
-19	-3	-1	-12	1	16	-19	1	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	16	-21	1	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	-16	-21	-31	1	3	11	0	0	0	12	3
-19	-3	-1	-12	-1	-16	-21	-31	-1	3	11	0	0	0	10	3
-19	-3	-1	-12	-1	-16	-21	-31	-1	-3	11	0	0	0	10	-3

19	3	1	12	1	16	21	31	1	3	11	0	0	0	12	3
-1	6	2	24	2	32	4	62	2	6	22	0	0	0	24	6
-1	0	2	24	2	32	4	56	2	6	22	1	0	0	24	7
-1	0	0	24	2	32	2	56	2	6	22	0	0	0	24	6
-1	0	0	0	2	32	2	32	2	6	22	0	0	0	24	6
-1	0	0	0	0	32	0	32	2	6	22	0	0	0	24	6
-1	0	0	0	0	0	0	0	2	6	22	0	0	0	24	6
-1	0	0	0	0	0	0	0	0	6	22	0	0	0	22	6
-1	0	0	0	0	0	0	0	0	0	22	0	0	0	22	0

19	3	1	3	1	12	11	0	1	16	0	0	21	31	12	3
-1	6	2	6	2	24	22	0	2	32	0	0	4	62	24	6
-1	0	2	6	0	0	22	0	2	32	0	0	2	32	24	6
-1	0	2	6	0	0	22	0	0	32	0	0	0	32	24	6
-1	0	0	0	0	0	22	0	0	0	0	0	0	0	22	0
-1	0	2	6	2	24	22	1	2	32	0	0	4	56	24	7
-1	0	2	6	0	24	22	0	2	32	0	0	2	56	24	6
-1	0	2	6	0	0	22	0	0	0	0	0	0	0	24	6
-1	0	0	6	0	0	22	0	0	0	0	0	0	0	22	6

which these overheads are the same:

$$\begin{aligned} T_{P+1} + (T_{P+1} - T_P) \cdot \frac{N}{nb} &= T_{P+1} \cdot \frac{N}{nb} / I \\ I &= \frac{T_{P+1} \cdot \frac{N}{nb}}{T_{P+1} + (T_{P+1} - T_P) \cdot \frac{N}{nb}} \end{aligned}$$

If $\frac{N}{nb}$ is large, the number of iterations required in the interval would be approximately $\frac{T_{P+1}}{T_{P+1} - T_P}$, which could be a very large number, depending on the implementation of broadcast and reduce.

In addition to the time overhead, both checksum and checkpoint techniques have the overhead of additional processors that are required to hold either the checksum or checkpoint. The overhead in number of processors for the checksum is approximately $\sqrt{P}$, where P is the number of processors, so the relative increase in processors is smaller the more processors there are.

4.5.3 Overhead of one recovery

Both checkpoint and checksum methods make it so that at any particular time, only a small part of the total execution is vulnerable to a failure. With either method, only the time spent in the most recent interval can be lost. With a checkpoint this interval depends on the failure rate of the system, but with a checksum the interval is always one iteration. For the checksum method, the checksums are consistent at the beginning of each iteration. To recover from a failure, it is necessary to go back to the beginning of the current iteration and restart from there.

Both checkpoint and checksum recoveries use a reduce of some sort to calculate the lost values, so that the recovery time t_r is comparable for the two methods.

The other aspect of the overhead of recovery, beside t_r, is the amount of calculation that has to be redone. Since the checkpoint interval can be varied while the checksum interval cannot, it seems that this overhead could favor one method or the other depending on the circumstances. However, the optimum checkpointing interval is determined partly by the time it takes to perform a checkpoint, and one of the main points emphasized in this paper is that the time to perform a checkpoint is very long for matrix operations because of the large amount of data that changes between checkpoints. Therefore it is reasonable to

assume that the interval required by the checkpoint will be larger than that of the checksum method, and the overhead introduced by the repeated work will be less with the checksum method.

There are two ways in which the overhead of the checksum method is less than that of checkpointing: the time added even when there is no failure, and the time lost when a failure occurs.

4.5.4 Comparison to checkpointing

One way to evaluate the relative merit of the checkpointing and checksum techniques is to compare their overhead. The most straightforward way of measuring overhead is to find how much longer a run on the same matrix size takes with fault tolerance than without. This way all of the effects of the additional work will be included.

A problem with this comparison is that the optimal rate of checkpointing depends on the expected rate of failure, among other factors. The time between checkpoints that gives the best performance is given in [46]. This means that it is impossible to absolutely state that checkpointing has higher overhead. However, it is possible to show that the interval would have to be extremely long, which is only possible when the failure rate is extremely low, for checkpointing to achieve overhead as low as that of the checksum method.

Making a checkpoint is the same operation as making the checksum, the difference being that a checksum is only done once while the checkpoint is done many times. Aside from that fact, the difference lies in the fact that the checksum needs to have some work done to keep it correct, while the checkpoint ends with doing the sum periodically. The extra work comes from the fact that the blocks with the sums in them are treated as part of the matrix. However, no extra iterations are added because it is possible to skip over the sum blocks and keep the sums correct. So the number of steps is the same; the only difference is how much longer each step takes when there are more processors in the grid.

Another consideration is how the overhead scales. If the checkpoint is done at the same interval regardless of the matrix size, then the overhead would remain nearly constant. However, when more processors are added, the expected rate of failure increases, so that in practice the checkpoint interval would likely have to be shorter when a larger matrix is used. In contrast, the fraction of total time that is overhead in the checksum technique should decrease as the size of the matrix increases. Since much of the overhead comes from

making the sum at the beginning of the calculation, the overhead as a fraction of the total time will decrease as the length of the calculation increases.

When no error occurs, the overhead of performing a checkpoint is the time it takes to do one checkpoint multiplied by the number of checkpoints done. For checkpointing, the optimum interval depends on the failure rate and the time it takes to do a checkpoint. The more frequently failures are likely to occur, the smaller the interval must be. The longer the checkpoint itself takes, the fewer checkpoints there should be in the total running time, so the interval is longer for a larger checkpoint. Whatever the optimum interval is, the additional overhead from the checkpoint when no failure occurs is Nt_c, where N is the total number of checkpoints and t_c is the time to perform one checkpoint.

The checksum technique, in contrast, does not take any extra time to keep the sum up to date. The only overhead when no failure occurs is the time to calculate the sum at the beginning. Since both the sum and the checkpoint operation will use some sort of reduce, the time to calculate the checksum is comparable to the time to perform one checkpoint.

Whether fault tolerance is used or not, a failure means that some amount of calculation time is lost and has to be repeated. When no fault tolerance is used, the time that has to be repeated is everything that has been done up to the point of the failure. The higher the probability of failure, the less likely it is that the computation will ever be able to finish.

4.5.5 Expected time to complete a calculation

Both the checkpoint and checksum techniques require some calculation to be repeated. The total computation can be divided into segments where, if an error occurs during the execution of the segment, the entire segment will have to be repeated. When a segment will have to be repeated an unknown number of times until it succeeds, the expected time to complete the segment can be defined as t/P, where t is the length of the segment and P is the probability of successfully completing it. The probability of succeeding, then, is $1 - t/M$, where M is the mean time to failure.

Therefore, the expected time is

$$\begin{aligned} \frac{t}{1 - t/M} &= \frac{t}{\frac{M-t}{M}} \\ &= \frac{tM}{M - t} \\ &= t\frac{M}{M - t} \end{aligned}$$

The factor by which the time increases over the error free time is $\frac{M}{M-t}$. This value shows how when there are very few errors, so that M is large compared to t, the expected time only increases by a small amount; usually no error will occur and the segment will complete successfully on the first try. On the other hand, if errors occur more frequently, the time to complete will be greater due to the greater number of segments repeated.

In order to compare the checksum and checkpoint methods, consider what the expected time will be for the total calculation. The total time when no errors occur is $T = t_1 * n_1$: the total time is divided into n_1 segments of length t_1 each. When errors occur, there are still n_1 segments, but they will each have an expected time determined by the equation above. Then the expected runtime is

$$n_1 \frac{t_1 M}{M - t_1} = \frac{TM}{M - t_1}$$

This equation can be used to compare the ratio of the two different methods on the same problem, where the error-free runtime can be assumed to be the same. In fact, each method adds some amount of overhead. However, since it has been shown that even without considering errors, the checkpoint method adds more overhead, the ratio resulting from assuming the error-free runtime is the same will be slightly too favorable to the checkpoint method.

Defining t_1 as the interval between checkpoints and t_2 as the time that has to be repeated in the case of an error in the checksum method, the ratio of the checkpoint time to the checksum time is

$$\frac{TM/(M - t_1)}{TM/(M - t_2)} = \frac{M - t_2}{M - t_1}$$

Since t_2 is a much smaller value than t_1, the value is greater than one and the checkpoint technique will clearly have a longer expected runtime than the checksum technique. Also, assuming the same calculation is done in environments of different error rates, more frequent

errors mean that M is a smaller number, making the ratio even more favorable to the checksum method.

The more frequently errors are expected to occur, the better the checksum method will perform compared to checkpointing.

4.6 Experiments

4.6.1 Platforms

We evaluate the proposed fault tolerance scheme on the following platforms:

Jaguar at Oak Ridge National Laboratory (ranks No. 2 in the current TOP500 Supercomputer List): 224,256 cores in 18,688 nodes. Each node has two Opteron 2435 "Istanbul" processors linked with dual HyperTransport connections. Each processor has six cores with a clock rate of 2600 MHz supporting 4 floating-point operations per clock period per core. Each node is a dual-socket, twelve-core node with 16 gigabytes of shared memory. Each processor has directly attached 8 gigabytes of DDR2-800 memory. Each node has a peak processing performance of 124.8 gigaflops. Each core has a peak processing performance of 10.4 gigaflops. The network is a 3D torus interconnection network. We used Cray MPI implementation MPT 3.1.02.

Kraken at the University of Tennessee (ranks No. 8 in the current TOP500 Supercomputer List): 99,072 cores in 8,256 nodes. Each node has two Opteron 2435 "Istanbul" processors linked with dual HyperTransport connections. Each processor has six cores with a clock rate of 2600 MHz supporting 4 floating-point operations per clock period per core. Each node is a dual-socket, twelve-core node with 16 gigabytes of shared memory. Each processor has directly attached 8 gigabytes of DDR2-800 memory. Each node has a peak processing performance of 124.8 gigaflops. Each core has a peak processing performance of 10.4 gigaflops. The network is a 3D torus interconnection network. We used Cray MPI implementation MPT 3.1.02.

Ra at Colorado School of Mines: 2,144 cores in 268 nodes. Each node has two 512 Clovertown E5355 quad-core processor at a clock rate of 2670 MHz supporting 4 floating-point operations per clock period per core. Each node has 16 GB memory. Each node has a peak processing performance of 85.44 gigaflops. The network uses a Cisco SFS 7024 IB Server Switch. We used Open MPI 1.4.

Table 4.1: Jaguar: local matrix size 2000 × 2000, block size 64

N	P	Total time (s)	Checksum time (s)	Overhead (%)	Performance (Gflops)
192000	9312	161.83	1.22	0.759	29160
216000	11772	186.24	1.24	0.670	36070
240000	14520	206.08	1.26	0.615	44720
264000	17556	238.56	1.29	0.541	51420

Table 4.2: Jaguar: local matrix size 4000 × 4000, block size 64

N	P	Total time (s)	Checksum time (s)	Overhead (%)	Performance (Gflops)
384000	9312	913.16	5.72	0.630	41340
432000	11772	995.98	5.70	0.576	53960
480000	14520	1137.26	5.69	0.503	64830
528000	17556	1254.81	5.91	0.473	78200

Table 4.3: Jaguar: local matrix size 2000 × 2000, block size 128

N	P	Total time (s)	Checksum time (s)	Overhead (%)	Performance (Gflops)
192000	9312	162.52	1.29	0.800	29030
216000	11772	184.92	1.28	0.697	36330
240000	14520	210.50	1.29	0.617	43780
264000	17556	244.63	1.33	0.547	50140

4.6.2 Overhead without recovery

We ran our code on both a larger scale (Jaguar and Kraken) and on a smaller scale (Ra). Since the time required to perform the checksum can be kept almost constant when the matrix size is increased, the larger scale shows lower overhead as a fraction of the total time.

Tables 4.1, 4.2, and 4.3 show the overhead of making a checksum at the beginning of the calculation for a matrix of size $N \times N$ on P processes. The processes are arranged in a a grid of size $p \times (p+1) = P$. The sum is kept on the extra processes in the last column of the processor grid. When the local matrix on each process is the same size, the time to perform the checksum is nearly the same for different total matrix sizes. The overhead of the checksum method consists almost entirely of the time taken to perform the checksum at the beginning, so it decreases as a fraction of the total time. Changing the block size has very little effect on the overhead. However, when the local matrix on each process is increased from 2000×2000 to 4000×4000, the overhead is less for the same number of processes, while the performance is greater.

Table 4.4 shows the results on a different large system. Here also the overhead is typically less than 1%. Table 4.5 shows the results for small matrices. Even with few processes the overhead is low, and it decreases as the size increases.

Figure 4.5 shows runtimes with and without fault tolerance. The difference in times between the two cases is smaller than the variation that can arise from other causes, as in the case of sizes 264000 and 288000, where the untouched code took longer for some reason.

4.6.3 Overhead with recovery

Tables 4.6 and 4.7 show simple recovery times for a single failure. Here the recovery is done at the end of an iteration, and requires only a reduce. Consequently, the time needed to recover is very similar to the time needed to perform the checksum in the beginning. In order to find the recovery time, we did the recovery operation to a copy of the local matrix of an arbitrary process, using this to both time the recovery operation and to check its correctness by comparing to the original local matrix.

By this measure, the recovery time is only the time needed for a reduce. In the case of a real failure it may be necessary to repeat at most one iteration. As an example

Table 4.4: Kraken: local matrix size 2000×2000, block size 64

N	P	Total time (s)	Checksum time (s)	Overhead (%)	Performance (Gflops)
144000	5256	214.71	1.16	0.543	9272
168000	7140	195.06	1.18	0.609	16210
192000	9312	256.91	1.17	0.457	18370
216000	11772	307.34	1.18	0.385	21860
240000	14520	342.28	1.18	0.346	26930

Table 4.5: Ra: local matrix size 4000×4000, block size 64

N	P	Total time (s)	Checksum time (s)	Overhead (%)	Performance (Gflops)
16000	20	36.51	2.16	6.29	74.81
20000	30	44.44	1.84	4.32	120.0
24000	42	54.98	1.97	3.72	167.7
28000	56	65.82	2.23	3.51	222.4
32000	72	77.20	2.43	3.25	283.0
36000	90	89.95	2.46	2.81	345.8
40000	110	81.44	2.27	2.87	523.9

Table 4.6: Jaguar: local matrix size 2000×2000, block size 64

N	P	Total time (s)	Recovery time (s)
192000	9312	161.83	1.19
216000	11772	186.24	1.24
240000	14520	206.08	1.24
264000	17556	238.56	1.25

Figure 4.5: On Jaguar, when run with and without checksum fault tolerance, the times are very similar. In fact, variations in the runtime from other causes are greater than the time added by the fault tolerance, with all effects included.

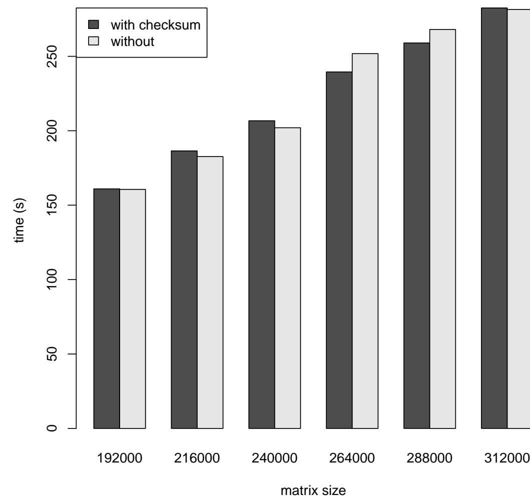


Table 4.7: Ra: local matrix size 4000×4000, block size 64

N	P	Total time (s)	Recovery time (s)
16000	20	36.51	1.52
20000	30	44.44	1.94
24000	42	54.98	2.60
28000	56	65.82	3.03
32000	72	77.20	3.41
36000	90	89.95	4.25

of the amount of work that is repeated, the first entry in table 4.6 did 3000 iterations, which means that each iteration took less than 0.05 seconds, which is not very significant compared to the other cost of recovery.

4.6.4 Algorithm-based recovery versus diskless checkpointing

According to [47], an approximation for the optimum checkpoint interval is

$$I = \sqrt{\frac{2t_c(P)M}{P}}$$

where $t_c(P)$ is the time to perform one checkpoint when there are P processes and M is the mean time to failure of one process, assuming that the process failures are independent so that, if the failure rate of one is $\frac{1}{M}$, then the failure rate for the entire system is $\frac{P}{M}$. This formula illustrates the balance between the two main factors that determine the optimum interval. The longer it takes to perform a checkpoint, the less often it should be done for the sake of overhead. The term M/P is the mean time to failure for the entire system. When the expected time until a failure is less, checkpoints need to be done more often for the optimum expected runtime. Since the time to perform a checkpoint only increases slightly as the number of processes increases, the significant factor is the number of processes, which makes failures more likely and decreases the length of the checkpoint interval.

As an example of possible checkpoint overhead, Figure 4.6 shows the overhead when the mean time to failure is 10000 hours for one process. Because both the operation to create the backup (checksum or checkpoint) and the operation to recover from a failure are essentially the same between the two different approaches, using the same values for both the checksum and the checkpoint approach gives an approximation for how the overheads compare that is fair to the checkpointing approach.

The checkpoint interval decreases as more processes are added because the probability of a failure increases. This means that the amount of work repeated because of one failure is less, but the expected number of failures during the run increases. On average, half of the checkpoint interval will have to be repeated. The running time increases as the problem size is larger, while the checkpoint interval decreases. So there will be an increasing number of checkpoints, and therefore the overhead increases as the number of processes increases. With the checksum method, on the other hand, the overhead decreases as the number of processes increases. Figure 4.7 shows the results of smaller runs, com-

Figure 4.6: Fault tolerance overhead without recovery: Algorithm-based recovery versus diskless checkpointing

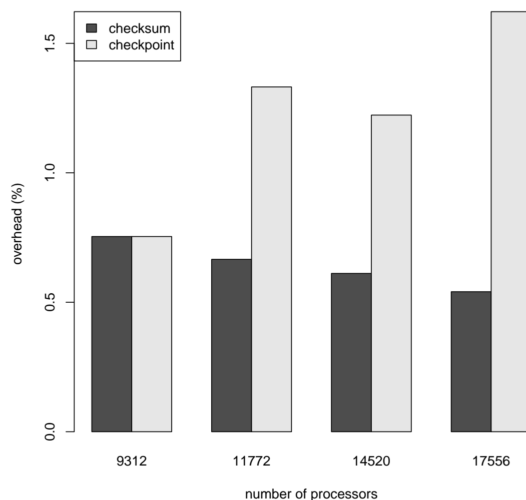


Figure 4.7: On smaller runs on Ra, the difference between checksum and checkpoint can be easily seen.

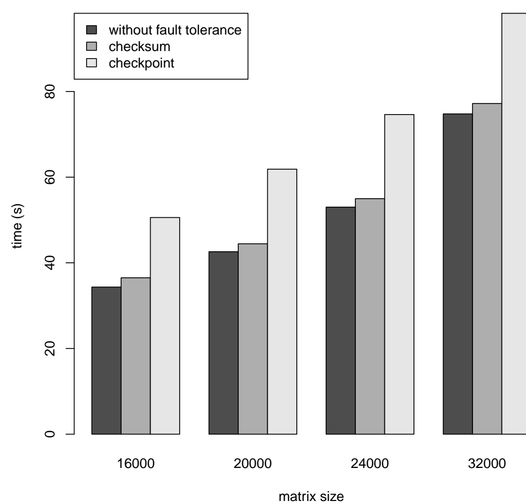
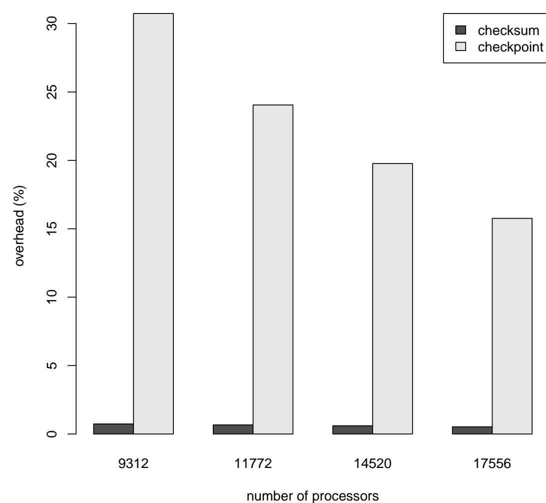


Figure 4.8: Fault tolerance overhead with recovery: Algorithm-based recovery versus diskless checkpointing



paring checksum and checkpoint overhead without a failure. Figure 4.8 shows the cost of one recovery with the algorithm-based recovery scheme. Here the overhead is less than one percent for each case, compared to at least 15 percent with a checkpoint. Table 4.8 also shows the comparison of overhead from checkpoint and the checksum method.

Table 4.8: Jaguar: projected checkpoint overhead

N	P	Total time (s)	Checksum time (s)	Recovery time (s)	Checkpoint interval (s)	Overhead: no failure (%)	Overhead: one failure (%)	ABR overhead (%)
192000	9312	161.83	1.22	1.19	97.1	0.75	30.7	0.73
216000	11772	186.24	1.24	1.24	87.1	1.33	24.0	0.67
240000	14520	206.08	1.26	1.24	79.0	1.22	19.8	0.60
264000	17556	238.56	1.29	1.25	72.7	1.62	15.8	0.52

Chapter 5

Soft Errors

5.1 Failure Model

The type of failure that we handle with this technique is a fail-continue failure, where the failure is not evident except by the fact that the application has the wrong data. We use a set of checksums to both detect and correct errors. We assume that errors can only occur during calculation, so that stored values will not become wrong. Matrix elements can only potentially be wrong when they are changed.

We assume that the failure is not detectable from outside the application. This is different from a larger hardware failure where it is obvious from other signs where the problem is located. Because it is most likely not possible to know if or where a soft error has occurred, an important part of our technique is verifying the results of the calculation periodically. An error is detected by a failure in verification, which indicates that recovery is necessary.

In any application, the biggest problem from soft errors is that they may be propagated soon after they occur [41]. The error will spread to other processors as soon as an incorrect value is included in some sort of message, which could happen immediately after the error occurs. Once the error is propagated, a single error becomes multiple incorrect values, which the fault tolerance scheme will most likely not be able to correct. In a general case, the best approach might be to have error correction on each message sent. Fortunately, in the case of HPL, there are specific points where errors can be detected before they are propagated to the rest of the matrix.

In HPL, error propagation can cause a single error to affect nearly the entire matrix. Therefore, the verification must be done often enough to catch errors before they can affect enough of the matrix to make recovery impossible. The results of a part of the calculation can be verified before they are broadcast to other processes, which is when propagation occurs. When an error is detected before it affects multiple processes in a row of the process grid, it is still possible to recover.

5.2 Error Propagation in High Performance Linpack

HPL performs a dense LU decomposition using a right-looking algorithm. The matrix is stored in a two-dimensional block-cyclic data distribution [32]. Matrices are often stored in 2D block cyclic fashion [1, 9, 2]. These are the most important features of HPL to this technique.

Generally matrix operations are done in parallel because the matrix involved is very large. The matrix will most likely not fit in the memory of one processor, and so it will be distributed so that each process has one part of the matrix. Therefore an important part of recovering from errors is the fact that the affected part of the matrix is not duplicated anywhere else, and has to be recovered.

5.2.1 Right-looking LU factorization

In Gaussian elimination, the elements of L are found by dividing some elements of the original matrix by the element on the diagonal. If this element is zero the division is clearly not possible, but with floating point numbers a number that is very close to zero could be on the diagonal and not be obvious as a problem. In order to ensure that this does not happen, algorithms for LU factorization use pivoting, where the row with the largest element in a the current column is swapped with the current row. Partial pivoting means that the swaps are not done in the L matrix. The equation $Ax = b$ can be rewritten as $LUx = b$ using the LU decomposition, where $Ux = y$, so that $Ly = b$. The algorithm transforms b to y, so that L is not needed to find x at the end. The factorization is performed in place, which means that the original matrix is replaced by L and U.

The right-looking algorithm is the version of LU factorization that updates the trailing matrix, unlike other versions. It is possible to put off the update of a particular

section of the matrix until just before that section is going to be factored. However, this approach means that large sections of the matrix are unchanged after each iteration. Only the processes holding a small section of the matrix are working at any particular time, making the efficiency of these versions lower. Our technique requires the matrix to be updated each step, which is what is done in the right-looking version. Figure 3.2 shows how the matrix is changed in each of the three variations. Because of the data distribution, the load balance is best when each processor does some work on its section of the matrix during each iteration. When the trailing matrix is not updated at each step, the work of updating it has to be done consecutively by a small set of processors when the factorization reaches each new part. When the update is done at every step, the work is done at the same time that other processors are doing factorization work, taking less total time.

The factorization in HPL is done by iterating over panels, which are sections of columns the width of the block size. An example panel is shown as section 1 in figure 5.1. Each iteration takes its panel from the trailing matrix of the previous iteration (section 3 in figure 5.1), so the height of the panel decreases by one block size each iteration.

Algorithm 1 An iteration has three main steps that are of interest for our technique.

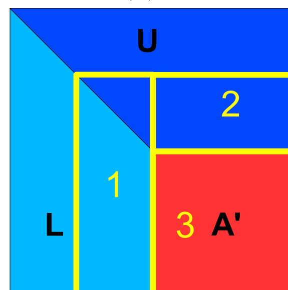
```
for i = 0 to N/nb do
    factor the panel at (i*nb, i*nb)
    broadcast the panel to the rest of the matrix
    update the matrix using the factored panel
end for
```

Each step - the panel factorization, the row panel update, and the trailing matrix update - changes only its particular section of the matrix. After the first two steps, matrix elements are broadcast to be used in the updates. The broadcast is where errors can be propagated outside of processes, which is when an error could potentially become unrecoverable. Since we use row checksums only, we are only concerned about error propagation along rows. It turns out that propagation along rows can only happen during the broadcast after the panel factorization. Therefore, it is sufficient to verify the checksums only after the panel factorization of each iteration in order to guarantee that all errors will be detected while it is still possible to recover successfully, as long as only one error occurs. It is necessary to verify the trailing matrix with a frequency depending on the failure rate.

Algorithm 2 Error checking is added to the main factorization loop as shown

for i = 0 to N/nb **do**
  factor the panel at (i*nb, i*nb)
  verify the panel
  if the panel contains an error **then**
    perform recovery
    return to the beginning of this loop iteration
  end if
  broadcast the panel to the rest of the matrix
  update the matrix using the factored panel
  if it is time to verify the entire matrix **then**
    verify the matrix
    if the matrix contains an error **then**
      perform recovery
    end if
  end if
end for

Figure 5.1: The part of the matrix that is involved in the three parts of an iteration: (1) panel factorization, (2) row panel update, (3) trailing matrix update.



In algorithm 2, the step "verify the panel" must be done to ensure that the values that are being broadcast are correct. It uses the local checksums to verify the result of the panel factorization. The step "verify the matrix" means checking the trailing matrix for correctness. This step takes longer than verifying just the panel, and it is less critical. An error outside of the current panel will only make recovery impossible if it is a second error that occurs before the first error can be corrected. Therefore, the frequency of verifying the trailing matrix depends on the expected error rate.

If a recovery is needed, it is done using the global checksums. A recovery just after the panel factorization will actually undo the result of the factorization: at this point in the loop iteration, only the panel has been factored, and the panel has just been shown to contain an error. Therefore, the entire panel must be replaced with the recovered version. The recovery is done using elements that are still in the state from the previous iteration, so the result will be to recover the panel from before it was factored.

However, in the case an error is found during the trailing matrix verification, this is the last step in the loop: the entire matrix has now been updated to the next step in the factorization. Therefore, the entire matrix is consistent with the end of an iteration, and the recovery of any part of the matrix will not require any work to be repeated.

The recovery depends on there not being uncorrected errors outside of the panel. This error check must be done periodically, with a frequency determined by the error rate. If the entire matrix is verified occasionally, then when an error is found in the panel we can assume that the other elements involved in the recovery will be correct. There is still a chance of an error occurring between the last full verification and the recovery, but it is very unlikely.

5.2.2 Error Propagation

Errors that occur in different sections of the matrix have different levels of risk to the correctness of the calculation. Section 1 of figure 5.1, the column panel, is the most important to verify. Errors that occur here will be propagated across the rows. This panel is used to update the rest of the matrix. An error will affect the elements in the same row to the right of the affected element. Because the global checksum is across rows, when more than one process in a row is affected, recovery becomes impossible. This panel is factored separately, then it is broadcast to the rest of the matrix. The verification must be done

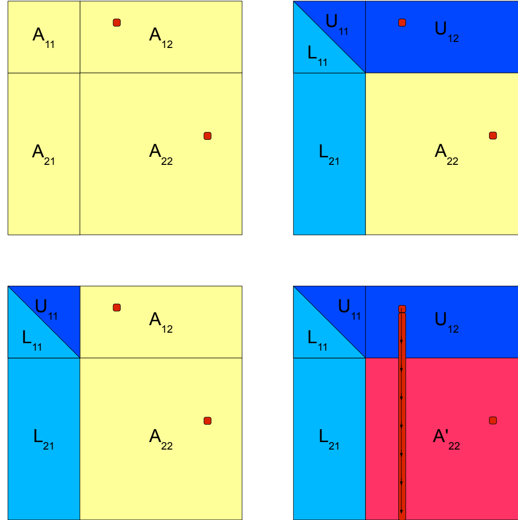


Figure 5.2: An error in the row panel is propagated down the column. If there is a second error in the matrix, it creates a situation where there are two errors in the same row, making recovery impossible.

before the broadcast to prevent errors from being propagated. Because of the way the local checksum is set up, the stored sums for most of this panel will be zeros as long as no error occurred. This check is the shortest, because only one block of sums need to be recalculated.

Section 2 is the row panel, and is also sensitive to errors. An error in this section is not immediately damaging. It is propagated down the column it occurs in, which does not make recovery impossible. However, since the entire column is affected, a second error in a different location would make recovery impossible unless it happened to occur in the same process column. Therefore it is important to check this section for correctness as well, although it is possible to lengthen the period between verifications depending on the error rate. The sums in this panel all need to be recalculated for the verification, so it takes longer than the column panel verification.

Section 3 is the trailing matrix, and is least sensitive to errors. An error in this section is not propagated until it becomes part of a panel in a later iteration, at which point it can be handled by the panel verification. There is still a possibility that a second error could make recovery impossible, but the probability that an error would do so is lower than for the row panel. It can be necessary to verify this section of the matrix as well, again

depending on the error rate, and less frequently than the row panel verification. Again this verification takes longer than the row panel verification, because every sum in the trailing matrix must be recalculated.

Figure 5.3 shows how much an error is propagated when it occurs in the U section of the panel. The error is propagated down the entire column that it occurs in during the panel factorization. This is the point when it would be detected by our method, before it can become impossible to recover. The corrupted elements are in one process column, so recovery is possible. If it were not recovered at this point, the error would be propagated to every row in the row panel that had an error in the column panel. Finally, the error is spread to the entire trailing matrix.

5.3 Soft Error Detection, Location, and Correction

5.3.1 Checksum Setup

We use two types of checksums, global and local. The local checksum is a sum of elements in the local matrix on each process, and is stored in the same local matrix. The global checksum is a sum of local matrices, and it is stored in additional processes. The local checksum is verified periodically; if the sum is not correct, then the global sum is used for recovery. No extra communication is needed unless an error occurs.

Any checksum of elements in the same row can be maintained at the end of an iteration. However, in this technique we verify the checksums after the panel factorization, where the panel is factored but the rest of the matrix is not updated. Therefore we use local checksums by blocks - the elements in the block are added up across the row and stored just after the block, and the block size is increased by one. After the panel factorization, the checksums within the panel are correct because they involve only elements of the panel.

If an error is detected by the check after the panel factorization, the recovery will take the matrix state back to the beginning of the iteration and the panel factorization will have to be repeated. At the point just after the panel factorization, the panel only is updated (section 1 in figure 5.1). The global sums, which are used for recovery, have not yet been changed from their state in the previous iteration. When an error is detected in the panel, it is necessary to replace the entire panel with a recovered version due to error propagation. The recovered panel is constructed using elements that are still in the state

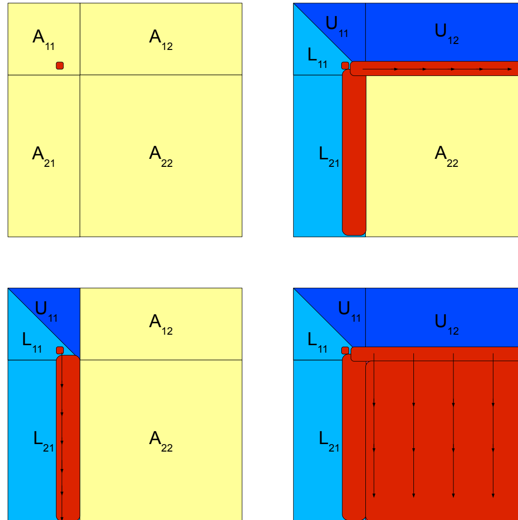


Figure 5.3: A single error can be propagated to a very large section of the matrix when it occurs in the panel.

of the previous iteration; therefore, the panel recovered will be the one from the previous iteration, and the panel factorization will have to be repeated. With this method there is no other way to recover: any error detected in the panel factorization will require a recovery back to just before the factorization. With only one local sum, even an incorrect sum will cause a recovery, since with a single sum it is impossible to tell which element of the sum is incorrect.

Figure 5.4 shows an example matrix that will have a checksum added. The matrix is distributed on the process grid according to the 2D block cyclic distribution. Figure 5.5 shows the same matrix with row sums added to each local matrix. The elements in one block of a row of a local matrix are summed, and the result is stored just after the block that it is the sum of. The block size is increased by one so that the sums are always included with their elements in operations. Maintaining checksums at a fine-grained level requires operating on both a sum and the elements that go into it at the same time. Figure 5.6 shows the matrix after global sums in additional processes are added. These sums are the sum of the local matrices, stored in extra processes. In the global view of the matrix, these checksums appear to be spread out periodically through the matrix. However, where the sums are located in the global matrix does not impact their correctness.

Figure 5.4: An example matrix, shown first in the original global view, then as it would be distributed on a 3×2 grid with a block size of 2.

19	3	1	12	1	16	1	3	11	0
-19	3	1	12	1	16	1	3	11	0
-19	-3	1	12	1	16	1	3	11	1
-19	-3	-1	12	1	16	1	3	11	0
-19	-3	-1	-12	1	16	1	3	11	0
-19	-3	-1	-12	-1	16	1	3	11	0
-19	-3	-1	-12	-1	-16	1	3	11	0
-19	-3	-1	-12	-1	-16	-1	3	11	0
-19	-3	-1	-12	-1	-16	-1	-3	11	0

19	3	1	3	1	12	11	0	1	16
-19	3	1	3	1	12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	-1	16
-19	-3	-1	-3	-1	-12	11	0	-1	-16
-19	-3	1	3	1	12	11	1	1	16
-19	-3	1	3	-1	12	11	0	1	16
-19	-3	1	3	-1	-12	11	0	-1	-16
-19	-3	-1	3	-1	-12	11	0	-1	-16

Figure 5.5: The matrix after local sums are appended to the matrices in each process, and how this addition affects the global matrix.

19	3	22	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0
-19	3	-16	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	1	16	17	0	0	0
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	-1	16	15	0	0	0
-19	-3	-22	-1	-3	-4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0
-19	-3	-22	1	3	4	1	12	13	11	1	12	1	16	17	0	0	0
-19	-3	-22	1	3	4	-1	12	11	11	0	11	1	16	17	0	0	0
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0
-19	-3	-22	-1	3	2	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0

19	3	22	1	12	13	1	16	17	1	3	4	11	0	11	0	0	0
-19	3	-16	1	12	13	1	16	17	1	3	4	11	0	11	0	0	0
-19	-3	-22	1	12	13	1	16	17	1	3	4	11	1	12	0	0	0
-19	-3	-22	-1	12	11	1	16	17	1	3	4	11	0	11	0	0	0
-19	-3	-22	-1	-12	-13	1	16	17	1	3	4	11	0	11	0	0	0
-19	-3	-22	-1	-12	-13	-1	16	15	1	3	4	11	0	11	0	0	0
-19	-3	-22	-1	-12	-13	-1	-16	-17	1	3	4	11	0	11	0	0	0
-19	-3	-22	-1	-12	-13	-1	-16	-17	-1	3	2	11	0	11	0	0	0
-19	-3	-22	-1	-12	-13	-1	-16	-17	-1	-3	-4	11	0	11	0	0	0

Figure 5.6: The matrix after global sums are also added, with the global view of the matrix with all checksums.

19	3	22	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0	21	31	52	12	3	15
-19	3	-16	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0	-17	31	14	12	3	15
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	1	16	17	0	0	0	-19	1	-18	12	3	15
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	-1	16	15	0	0	0	-21	1	-20	12	3	15
-19	-3	-22	-1	-3	-4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	10	-3	7
-19	-3	-22	1	3	4	1	12	13	11	1	12	1	16	17	0	0	0	-17	25	8	12	4	16
-19	-3	-22	1	3	4	-1	12	11	11	0	11	1	16	17	0	0	0	-19	25	6	12	3	15
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	12	3	15
-19	-3	-22	-1	3	2	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	10	3	13

19	3	22	1	12	13	1	16	17	21	31	52	1	3	4	11	0	11	0	0	0	12	3	15
-19	3	-16	1	12	13	1	16	17	-17	31	14	1	3	4	11	0	11	0	0	0	12	3	15
-19	-3	-22	1	12	13	1	16	17	-17	25	8	1	3	4	11	1	12	0	0	0	12	4	16
-19	-3	-22	-1	12	11	1	16	17	-19	25	6	1	3	4	11	0	11	0	0	0	12	3	15
-19	-3	-22	-1	-12	-13	1	16	17	-19	1	-18	1	3	4	11	0	11	0	0	0	12	3	15
-19	-3	-22	-1	-12	-13	-1	16	15	-21	1	-20	1	3	4	11	0	11	0	0	0	12	3	15
-19	-3	-22	-1	-12	-13	-1	-16	-17	-21	-31	-52	1	3	4	11	0	11	0	0	0	12	3	15
-19	-3	-22	-1	-12	-13	-1	-16	-17	-21	-31	-52	-1	3	2	11	0	11	0	0	0	10	3	13
-19	-3	-22	-1	-12	-13	-1	-16	-17	-21	-31	-52	-1	-3	-4	11	0	11	0	0	0	10	-3	7

5.3.2 Proof of Correctness for Checksum

Each iteration of the LU factorization operates on the trailing matrix from the previous step. Once factored, the sections of L and U are not changed again. Beginning with a matrix with row checksums, one factorization step results in panels of L and U, and a trailing matrix with correct row checksums. Since one step maintains the checksum, it is maintained through the entire calculation.

The panel factorization on a matrix indexed starting from 1, with a block size of nb, is

```
for i = 1 to nb
    A(i+1:n,i) = A(i+1:n,i)/A(i,i)
    A(i+1:n,i+1:nb) -= A(i+1:n,i)*A(i,i+1:nb)
```

The first step in this loop creates elements of L, which are not maintained with checksums in our approach. The second step uses the modified elements from the first step. Therefore, the elements at the end of the step in terms of the elements from the beginning of the step are

```
A(i+1:n,i+1:nb) - A(i+1:n,i)*A(i,i+1:nb)/A(i,i)
```

Checksums are included in this calculation at the end of each block, so sections of the matrix where the index goes to nb will have checksums appended.

Theorem 1 *After one iteration of the main loop in the HPL algorithm, if the matrix started with correct global and local checksums, then the resulting matrix sections of U and the trailing matrix will each have correct global and local checksums.*

Each iteration operates on the trailing matrix from the previous iteration. As we will show below, the trailing matrix begins the iteration with all checksums within the trailing matrix consisting only of elements from the trailing matrix. Therefore it can be treated as an entire matrix, independent of the sections that are already completely factored. This is why the indices start from 1 instead of i below. Before the panel factorization, the panel is set up with a checksum:

$$\left[A_{1:n,1:nb} \sum_{j=1}^{nb} A_{1:n,j} \right]$$

At each iteration for i from 1 to nb, only the matrix $A_{i:n,i:nb}$ is involved. So if the previous iteration resulted in correct checksum relationships, then row i in iteration i will have a correct checksum and will not be changed again in the panel factorization. Iteration i results in

$$\begin{aligned} & \left[A_{i+1:n,i+1:nb} \sum_{j=i}^{nb} A_{i+1:n,j} \right] \\ & -A_{i+1:n,i} \left[A_{i,i+1:nb} \sum_{j=i}^{nb} A_{i,j} \right] / A_{i,i} \\ = \; & \Big[A_{i+1:n,i+1:nb} - A_{i+1:n,i} \cdot A_{i,i+1:nb} / A_{i,i} \\ & \sum_{j=i}^{nb} A_{i+1:n,j} - A_{i+1:n,i} \sum_{j=i}^{nb} A_{i,j} / A_{i,i} \Big] \end{aligned}$$

$$
\begin{aligned}
&= \Big[A_{i+1:n,i+1:nb} - A_{i+1:n,i} \cdot A_{i,i+1:nb}/A_{i,i} \\
&\quad \sum_{j=i}^{nb} A_{i+1:n,j} - A_{i+1:n,i} \cdot A_{i,j}/A_{i,i}\Big] \\
&= \Big[A_{i+1:n,i+1:nb} - A_{i+1:n,i} \cdot A_{i,i+1:nb}/A_{i,i} \\
&\quad A_{i+1:n,i} - A_{i+1:n,i} \cdot A_{i,i}/A_{i,i} \\
&\quad + \sum_{j=i+1}^{nb} A_{i+1:n,j} - A_{i+1:n,i} \cdot A_{i,j}/A_{i,i}\Big] \\
&= \Big[A_{i+1:n,i+1:nb} - A_{i+1:n,i} \cdot A_{i,i+1:nb}/A_{i,i} \\
&\quad \sum_{j=i+1}^{nb} A_{i+1:n,j} - A_{i+1:n,i} \cdot A_{i,j}/A_{i,i}\Big]
\end{aligned}
$$

At the end of iteration i, the sum to the panel is the sum from $i+1$ to nb. So iteration $i+1$ will operate on a panel and a checksum that includes only elements that are involved in the iteration.

Adding a checksum to a matrix can be represented as multiplying the matrix by another matrix H_r, which is set up to produce the appropriate checksum, and is assumed below to have the correct dimensions for the matrix in question. For example, the H_r that adds a row checksum to a 3×3 matrix is

$$
\begin{pmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1
\end{pmatrix}
$$

After the panel factorization, the row panel is updated according to the equation $A = LU$, where $A = A_{1:nb,nb+1:n}$, L is the lower triangle of $A1 : nb, 1 : nb$ with the diagonal replaced with ones, and U is the new value of $A_{1:nb,nb+1:n}$. This step maintains the checksum because of the original ABFT idea: since A with a checksum can be represented as $A \cdot H_r$, $A \cdot H_r = (LU) \cdot H_r = L(U \cdot H_r)$, so that U has a checksum as well. The checksum is actually the sum across the entire row, while the part of the matrix being updated is the entire row except for the first nb columns. Therefore, the sums are correct for the entire block of the first nb rows of U. The first nb columns have already been calculated, so they are not included. Nevertheless, the sums will be correct.

Referring to the parts of the matrix shown in figure 5.7, the relationship is $[A_{11}A_{12}] = L_{11}[U_{11}U_{12}]$, so $A_{12} = L_{11}U_{12}$. With a checksum,

$$[A_{11}A_{12}]H_r = L_{11}[U_{11}U_{12}]H_r = L_{11}\left([U_{11}U_{12}]H_r\right)$$

The checksum on the right hand side is a checksum of U only.

5.3.3 Local Checksum after Panel Factorization

The checksum verification in general requires recalculating the sums and comparing to the stored values. The overhead involved could be significant because of the memory access required. However, for the most frequent verification, the operation is faster.

All checksums are of either the original matrix or of U. Once elements that went into a checksum have been replaced by elements of L, the checksum reflects the fact that the equivalent position in U has a zero. When the panel is factored, the first $nb \times nb$ block contains a part of U, but the rest is L. In this section, the checksums will all be zeros. Therefore, no computation is required, and the memory access is less, for the main part of the panel verification. Since this verification must be done every iteration, a lower overhead for it is useful.

Theorem 2 *The sums of L in the panel will be zeros immediately after the factorization of that panel if and only if no errors occurred at any point earlier in the calculation either in the panel or in the columns the panel belongs to.*

Starting with a checksum matrix, where local row sums are done for each block, after a panel is factored, the sums for the part of the panel that becomes L will be zeros. The reason for this fact is that the algorithms changes the sums from sums on the original matrix to sums on U only. In the place where the L matrix is stored, the U matrix has zeros. The sums become the sums of zeros.

If there are no errors, then the L sums are zero. As shown in the proof of the correctness of the checksum, the sums are correct only for U. This is achieved by the algorithm when the sum elements that are below the diagonal go to zero. These elements are not stored as part of U because they are zeros. Therefore, when the elements that formerly went into the sum are all storing part of L, the sum will be zero. The panel

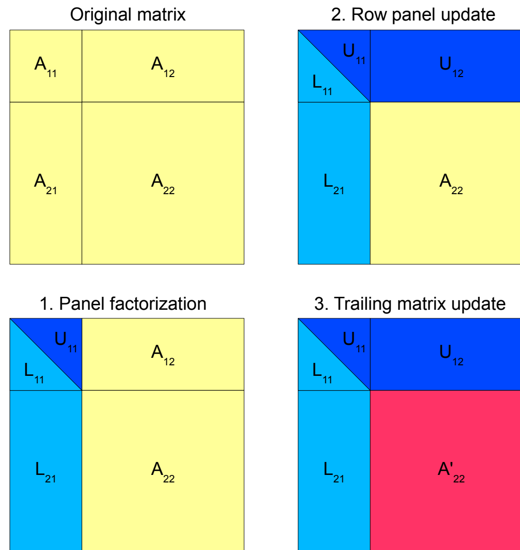


Figure 5.7: Relevant sections of a matrix in the factorization of one panel.

factorization is done in a loop from 1 to nb. After iteration $nb - 1$, the sum is from nb to nb, or just the last element. After iteration nb, no elements go into the sum, so it is zero.

If there are errors, then the L sums are not zero (which is the same as saying if the L sums are zero, then there are no errors). As stated above, the elements of the panel are updated in a loop:

```
for i = 1 to nb
    A(i+1:n,i) = A(i+1:n,i)/A(i,i)
    A(i+1:n,i+1:nb) -= A(i+1:n,i)*A(i,i+1:nb)
```

When an element in $A_{i+1:n,i}$ has an error during the first line of the loop, it will affect the second line as well. In the second line the checksums are updated. If an element of $A_{i+1:n,i}$ is incorrect, then the value subtracted from $A_{i+1:n,i+1:nb}$ will be incorrect. In the case of the checksums, they will be nonzero.

Figure 5.8 shows the different sections where errors could occur. The matrix shown is the trailing matrix from the previous iteration, which is the only area of the matrix where values are changed, and therefore is the only area where errors can occur with our assumption of errors only occurring from calculations. The error shown in case 1 is when an error occurs in L. In this case, since L is not included in the checksums, the error will remain, and no recovery will be done. In case 2, the error will be detected by the checksum

in the same block, and a recovery of the entire panel will be required. Case 3 is detected by the sums being nonzero. Errors shown in case 4 and case 5 can be detected with one verification, and recovered at the same time, even if both occur. This is possible because errors in different rows do not interact; multiple errors can always be recovered as long as they occur in different rows.

5.3.4 Error Detection and Location

In order to detect a single error, it is sufficient to verify the sums of the panel after each panel factorization. The reason is the way that errors are propagated. There are three phases of each iteration: panel factorization, broadcast and row panel update, and trailing matrix update. During the updates of the upper row panel and the trailing matrix, rows are combined. Any error in these sections will be spread down its column. Because global checksums are kept across rows, an error of this sort can be recovered. When it is propagated down a column, it amounts to one erroneous local matrix per row. As long as the error is kept to one process column, it can be recovered.

The panel factorization is the part of the iteration where there is a possibility of an unrecoverable error occurring. After the panel is factored, it is broadcast to the rest of the matrix, which is multiplied by the panel. In this case an error would affect entire rows, making recovery impossible. When multiple local matrices in a row are affected, the error cannot be recovered. Therefore, to guarantee recovery, the results of the panel factorization must be verified with local sums before there is any communication, so that a potential error is contained to the column containing the current panel.

Verifying the panel factorization consists of recalculating the sums for the U section of the panel, and of checking that the rest of the sums are zeros. If any sum is found to be incorrect, the entire panel must be recovered and the factorization repeated. The calculation has to be repeated because the global checksums, which the recovery is done from, have not been updated at the time of the panel factorization, so the recovered local matrices will be from the beginning of the iteration before the factorization. Although an error in the panel factorization could be propagated to the entire panel or not, the recovery can only restore the original unfactored panel, so the entire panel must be recovered in any case.

It is possible to detect all errors with only the check after the panel factorization. With this approach, the verification process requires only the calculation of sums in $nb \times nb$

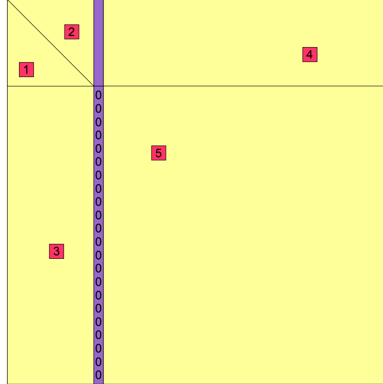


Figure 5.8: The small boxes indicate the sections of the matrix where errors could occur at different points in an iteration.

blocks, as well as checking if the sums in the rest of the panel are zeros. Therefore, it has extremely low overhead, significantly less than if even the trailing matrix is also verified. However, this method is vulnerable to multiple errors. The approach of only verifying the panel means that an error could potentially occur a long time before it can be detected. Any error is guaranteed not to leave its column until it becomes part of the panel to be factored, at which point it will be detected and corrected. However, this delay leaves the method vulnerable to a second error. If another error occurs before the first error is detected, recovery is impossible and the calculation must be repeated. What this means for the recovery process is that all of the local matrices must be verified with their sums before using them to recover the corrupted panel. Also, in cases where the error rate is high enough, this method will not be effective. In that case, more frequent checks can be used to detect errors immediately, making it much more likely that each error will be handled before another can occur.

In order to ensure that up to one error per iteration can be handled, and to remove the improbable but fatal possibility of two errors far apart in time causing an impossible recovery, we eventually decided to verify both the panel factorization and trailing matrix update. The panel factorization verification is necessary because errors that occur in this step will be propagated along rows, making recovery impossible. The verification after the trailing matrix update, which is at a time when the checksums in the entire matrix are

consistent, ensures that the matrix will be entirely correct going into the next iteration, and that no error will go undetected for a long period of time.

5.3.5 Soft Error Correction

An error on the diagonal within the panel will be propagated down the column and along all of the rows that are affected by the first propagation. This type of error would corrupt the entire trailing matrix if it were not caught immediately after the factorization, at which point it is still contained within one column of the process grid. An error above the diagonal will propagate down its column, which is harmless in this method. An error below the diagonal will propagate across its row, which is another type that must be caught before it can leave the panel.

An error that occurs in the row panel section of the matrix will be propagated down the column. It cannot cause an impossible recovery until part of the affected area is in the current panel. At this point, an entire column with erroneous values will exist in the panel, but the errors cannot propagate outside the column if the error is detected immediately after the panel factorization.

An error that occurs in the trailing matrix will not be propagated during the update. The error will have no effect until it is part of either the column or row panel, at which point it has the same effect as an error that occurred in one of those areas.

When an error is detected in the panel, it is necessary to recover, not just the entire panel, but the entire column it belongs to. This is because of the situation where an error occurs in place where it is propagated down a column but not immediately detected. This type of error will affect part of the matrix that has already been factored, and this part of the matrix will have to be recovered as well. The recovery is possible even though the error occurred earlier and is in an entire column. Errors cannot spread outside of a single column unless they are part of the panel, and up to one corrupted process per row can be recovered.

Figure 5.9 shows the idea of the detection and recovery technique with a simple example. With a real block size, more elements go into each local sum - perhaps 32 or 64. Even if multiple elements in the same row are affected, the checksum will indicate an error as long as the elements do not change in such a way that their sum remains the same - for instance, if two values are changed in a way that cancel each other out. The error

19	3	22	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0	21	31	52	12	3	15
-19	3	-16	1	3	4	1	12	13	11	0	11	1	16	17	0	0	0	-17	31	14	12	3	15
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	1	16	17	0	0	0	-19	1	-18	12	3	15
-19	-3	-22	1	3	4	-1	10	-13	11	0	11	-1	16	15	0	0	0	-21	1	-20	12	3	15
-19	-3	-22	-1	-3	-4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	10	-3	7
-19	-3	-22	1	3	4	1	12	13	11	1	12	1	16	17	0	0	0	-17	25	8	12	4	16
-19	-3	-22	1	3	4	-1	12	11	11	0	11	1	16	17	0	0	0	-19	25	6	12	3	15
-19	-3	-22	1	3	4	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	12	3	15
-19	-3	-22	-1	3	2	-1	-12	-13	11	0	11	-1	-16	-17	0	0	0	-21	-31	-52	10	3	13

Figure 5.9: The element -12 is replaced by 10. The sum verification shows that an error exists: $-1 + 10 \neq -13$. This is not enough to tell which element is incorrect. The reduce is done on the whole panel, and the result for this element is $1 - 16 - (-3) = 12$, restoring the correct element.

propagation that can still happen even with the checking that we do is only down columns. Therefore, for more than one error to occur among the elements that go into one local checksum, two soft errors would have to occur within a short time.

The situation where a single error has propagated down a column before being detected is easily handled by our method, although many elements might be affected. We are unable to locate the specific element within the local matrix without communication. Doing this would require a second local checksum. Instead, we recover the entire affected local matrix using the global checksum. Locating the error before recovering only the elements with errors can require as much communication as just recovering the entire local matrix, when an error is propagated down an entire column.

It is possible, as an exercise, to see how the exact location of the error can be found. In the example in figure 5.9, the sums need to be checked to find the failure. The elements in equivalent locations in the equivalent blocks of the local matrices are compared. This is done by doing the matrix addition and comparing the elements, but for this example we look at one element at a time to demonstrate. All of the sums need to be verified at the appropriate times, but looking just at the row where the error is, $-19+(-1)+(-1) = -21$, but $-3+10+16 = 23$, while the stored sum is 1. The last sum $-22+(-13)+15 = -20$ also checks as correct. The global sum shows that the second element in the block is incorrect, so it is replaced with a variable in the local sum and solved for: $-1+x = -13$, so $x = -12$.

5.4 Performance Analysis

5.4.1 Overhead of checksum setup and error detection

Calculating the checksum at the beginning of the calculation requires calculating the local checksum, followed by a reduce for the global checksum.

The overhead when no failure has occurred consists of the time to verify the local checksums in each iteration. The sums are verified twice per iteration, once for just the panel and once for the entire trailing matrix. The verification of the panel requires that nb numbers are summed, then the result is compared to the stored value. The verification of the trailing matrix requires each block in the local matrix to be summed and verified. If the maximum number of blocks in a local matrix is B, and the average time to sum nb numbers and do one comparison is t_{nb}, then the total time for both comparisons in one row is

$$t_{nb}(B+1)$$

To repeat it for every row in a local matrix, if the matrix and grid are both square, is $t_{nb}(B+1)Bnb$. This operation occurs every iteration. If the total matrix size is n, the number of iterations is $\frac{n}{nb}$. Therefore the total overhead for the calculation is

$$nt_{nb}(B+1)B$$

The value of t_{nb} depends on the time it takes to access the memory.

The panel verification, which takes $nt_{nb}B$, must be done every iteration. However, the trailing matrix verification, which takes $nt_{nb}B^2$, can be done less often. The frequency of checking the trailing matrix depends on the error rate. If two errors occur between trailing matrix verifications, it is likely to be impossible to recover. Therefore the interval for checking the trailing matrix can be adjusted so that the probability of two errors occurring in that time is acceptably low. If the interval is T_i, with a time between errors of T_f, the probability of an error in that interval is $\frac{T_i}{T_f}$, so the probability that two errors occur is $\left(\frac{T_i}{T_f}\right)^2$ Since the overhead of the trailing matrix check is B times that of the panel verification, doing the check less often is worthwhile.

5.4.2 Overhead of checking for errors in stored values

This technique mainly focuses on errors in the unfactored parts of the matrix, because these are the errors that can be propagated and make large parts of the matrix incorrect. However, errors could occur in the factored section of the matrix. Since these errors will not be propagated, multiple errors can be corrected in some cases. Only multiple errors in the same row can make it impossible to recover. Even if multiple errors occur in one row within one process, this will still be recoverable because the entire local matrix can be recovered from the other processes in the same row. So unless it is likely that errors will appear in multiple processes that are all in the same row in the process grid, errors can be corrected in the stored values with a high probability of success by doing a verification (and recovery if necessary) on the matrix after the factorization is complete. This verification would actually only apply to the U matrix. During the entire calculation, L is not maintained because it is not necessary to the solution to the problem at the end, so the same applies here.

The time to check the entire factored matrix for errors is similar to the time to verify the trailing matrix, $nt_{nb}B^2$, because it involves verifying up to the entire local matrix in some processes. No communication is required just for the verification, so the time to finish the verification is the time for the slowest process to finish.

5.4.3 Overhead of computation

The time for one iteration aside from the verification overhead is made of the panel factorization, the broadcast, and the trailing matrix update. For the panel factorization: repeated nb times, do B divisions followed by a matrix multiplication and subtraction. Some communication is also required to determine the pivots. After the factorization the panel is broadcast to the processes containing the trailing matrix. The trailing matrix update requires more communication. The row and column panels are multiplied, and the result is subtracted from the trailing matrix.

The block size is increased by one, and the entire matrix is increased by the size of the local matrices in one column of the process grid. The number of iterations is not increased because the global checksum elements can be skipped in the factorization. When a panel made of global checksum elements is reached, the sums in it are already zeros because the elements going into them are in the lower diagonal, which is zeros in the U matrix.

The only parts of an iteration that are affected by there being more processors are the parts with communication. There are broadcasts in both rows and columns, but only broadcasting in rows is affected because there are no column checksums. If the original matrix dimension is P, then with a checksum added it is $P + 1$. So the overhead of each iteration is the difference between a broadcast among $P + 1$ processors and a broadcast among P processors. Depending on the implementation, the value varies. With a binomial tree, the overhead would be $\log(P + 1) - \log P$. Using pipelining, where the time for the broadcast is nearly proportional to the size of the message, the overhead is even smaller.

The number of iterations is the same, and the length of each iteration is close to the same as when no fault tolerance is used. Any difference in the computation time comes from the increase in the block size. With typical block sizes, this increase might be between 0.5% and 2%.

5.4.4 Time complexity of HPL

According to [19], the runtime of the HPL algorithm is approximated as

$$\frac{2\gamma_3 N^3}{3P^2} + \frac{2\beta N^2}{P} + \frac{\alpha N((NB+1)\log P + P)}{NB}$$

where the time to send a message of length L is defined as $\alpha + \beta L$, γ_3 is the time to perform one floating point operation during a matrix-matrix operation, and there is a $N \times N$ matrix distributed on a $P \times P$ process grid with a block size of NB. This equation is an approximation with the highest order terms only.

When the checksum is added to the original matrix, the size of the matrix increases from N to $N + N/P + N/NB$. The increase of N/P comes from the global sum, which is additional elements equal to the number of elements in one process, and the increase of N/NB comes from the local sum, where one additional element is added for each block. To make a simpler comparison, this will be an upper bound on the runtime, since the following will not take into account the fact that additional processes are used. The use of additional processes reduces the impact of the larger matrix on the runtime. Then the new time with this larger matrix will be

$$\frac{2\gamma_3(N + N/P + N/NB)^3}{3P^2} + \frac{2\beta(N + N/P + N/NB)^2}{P} + \frac{\alpha(N + N/P + N/NB)((NB+1)\log P + P)}{NB}$$

An expression for the percentage overhead can be found by $1+(T_1-T_0)/T_0$, where T_0 is the original runtime and T_1 is the runtime with the larger matrix.

So

$$T_0 = \frac{2\gamma_3 N^3}{3P^2} + \frac{2\beta N^2}{P} + \frac{\alpha N((NB+1)\log P + P)}{NB}$$

and

$$\begin{aligned}
T_1 - T_0 &= \frac{2\gamma_3(N+N/P+N/NB)^3}{3P^2} + \frac{2\beta(N+N/P+N/NB)^2}{P} \\
&\quad + \frac{\alpha(N+N/P+N/NB)((NB+1)\log P + P)}{NB} \\
&\quad - \frac{2\gamma_3 N^3}{3P^2} + \frac{2\beta N^2}{P} \\
&\quad + \frac{\alpha N((NB+1)\log P + P)}{NB} \\
&= \frac{2\gamma_3(3N^2(N/P+N/NB) + 3N(N/P+N/NB)^2 + (N/P+N/NB)^3)}{3P^2} \\
&\quad + \frac{2\beta(2N(N/P+N/NB) + (N/P+N/NB)^2)}{P} \\
&\quad + \frac{\alpha(N/P+N/NB)((NB+1)\log P + P)}{NB}
\end{aligned}$$

Looking at the term involving γ_3, every component involves N^3 in the numerator and some combination of P and NB in the denominator. The values of $1/P$ and $1/NB$ are significantly larger than $1/P^2$, $1/NB^2$, $1/PNB$, and other combinations that arise. The same pattern exists in the rest of the expression. Therefore, to simplify this expression, look at the higher order terms only:

$$\frac{2\gamma_3(3(N^3/P+N^3/NB))}{3P^2} + \frac{2\beta(2(N^2/P+N^2/NB))}{P} + \frac{\alpha(N/P+N/NB)((NB+1)\log P + P)}{NB}$$

Comparing this to the expression for the original time, each term is multiplied by some constant and by $1/P + 1/NB$. So in a very rough approximation, the increase in the runtime depends on both the number of processes and the block size. Generally the block size will stay the same for different sizes of matrices on the same system. However, larger matrices will typically be run on larger numbers of processes. So the percentage overhead added to the runtime by the addition of checksums has a component that will stay constant as the matrix size is increased, but has another component that will decrease as the matrix size is increased. The overall overhead as a percentage of the total time decreases for larger matrices.

5.4.5 Overhead of recovery

The overhead of correcting an error is the time for a reduce, along with the calculation of the panel factorization that has to be repeated if the error is found in the panel check. The time to do a reduce is comparable but less than the time of a broadcast. Therefore the total cost of recovery is similar to the time one iteration takes, without the trailing matrix update. The overhead of one recovery as a fraction of the total time is approximately $\frac{nb}{n}$. Since the block size is much smaller than the total matrix size, the overhead fraction is small. Our technique should have low overhead because it does not use communication during an error-free run. However, it does access large parts of the matrix during some checks. We have tested the technique experimentally to see how much the memory access affects the overhead.

5.4.6 Comparison to alternatives

Generally, the alternative method to our technique is ABFT, using a checksum to verify the result at the end and repeating the entire calculation if there was an error. With multiple weighted checksums, it is possible to recover a certain number of errors - another checksum is needed for each expected error. Another option that is available with HPL is residual checking. The solution x to the system $Ax = b$ is multiplied by A and compared to b, which shows whether the solution is correct. The calculation can be repeated if the solution is incorrect. This approach is essentially the same as ABFT - only one row and column are added for ABFT, and the difference between factoring a $N \times N$ matrix and a $N + 1 \times N + 1$ matrix is small, especially when N is large.

As long as the expected time between failures is greater than the running time of the calculation, the number of times the calculation has to be run is a geometric random variable. If the expected time to failure is T_f and the running time of the calculation is T, the probability of an error occurring is $\frac{T}{T_f}$. The probability of an error-free run is $1 - \frac{T}{T_f} - \frac{T^2}{T_f^2} - \cdots \approx 1 - \frac{T}{T_f}$. So the expected time that it will take to complete an error-free run is

$$\frac{1}{1 - \frac{T}{T_f}} = \frac{T_f}{T_f - T}$$

When the expected time to a failure is less than the run time, as the error rate increases it becomes increasingly unlikely that the calculation will ever finish using ABFT.

5.5 Experimental Results

5.5.1 Platforms

We evaluate the proposed fault tolerance scheme on the following platforms:

Kraken at the University of Tennessee: 99,072 cores in 8,256 nodes. Each node has two Opteron 2435 "Istanbul" processors linked with dual HyperTransport connections. Each processor has six cores with a clock rate of 2600 MHz supporting 4 floating-point operations per clock period per core. Each node is a dual-socket, twelve-core node with 16 gigabytes of shared memory. Each processor has directly attached 8 gigabytes of DDR2-800 memory. Each node has a peak processing performance of 124.8 gigaflops. Each core has a peak processing performance of 10.4 gigaflops. The network is a 3D torus interconnection network. We used Cray MPI implementation MPT 3.1.02.

Ra at Colorado School of Mines: 2,144 cores in 268 nodes. Each node has two 512 Clovertown E5355 quad-core processor at a clock rate of 2670 MHz supporting 4 floating-point operations per clock period per core. Each node has 16 GB memory. Each node has a peak processing performance of 85.44 gigaflops. The network uses a Cisco SFS 7024 IB Server Switch. We used OpenMPI 1.4.

5.5.2 Overhead without failure

The overheads of this technique when no failure occurs are constructing the checksum at the beginning and verifying the results periodically during the computation. There are two types of verification, the panel and the trailing matrix. The panel verification must be done every iteration, so its overhead is the minimum possible. The trailing matrix verification takes longer, but can be done at a variable rate. The measured overhead is for the trailing matrix update done every iteration.

The overhead of verifying the checksums in each iteration is given in tables 5.1 and 5.2. The percentage overhead of the trailing matrix verification is close to $\frac{1}{N}$. The overhead of verifying the panel only is lower, but does not decrease as quickly with decreasing matrix size.

If only the elements involved in the verification are retrieved from memory, the ratio of the trailing matrix check to the panel check should be approximately the number of column blocks in a local matrix, or $\frac{N}{P\dot{n}b}$. For the experiments on Ra this value is 15.6,

Table 5.1: Ra: $N \times N$ matrix on $P \times P$ process grid, T_p is the panel verification time, and T_t is the trailing matrix verification time, block size 256, time in seconds, performance in Gflops

N	P	Total time	T_p (% overhead)	T_t (% overhead)	Performance
48000	12	128.10	3.40 (2.65)	31.66 (24.71)	575.6
64000	16	170.77	4.78 (2.79)	41.29 (24.17)	1023
80000	20	234.84	6.14 (2.61)	53.13 (22.62)	1454
96000	24	331.42	7.49 (2.25)	62.87 (18.96)	1780
112000	28	404.23	8.89 (2.19)	75.42 (18.65)	2317
128000	32	448.77	10.09 (2.24)	85.45 (19.04)	3115

Table 5.2: Kraken: $N \times N$ matrix on $P \times P$ process grid, T_p is the panel verification time, T_t is the trailing matrix verification time, block size 256, time in seconds, performance in Gflops

N	P	Total time	T_p (% overhead)	T_t (% overhead)	Performance
144000	72	242.38	5.63 (2.32)	16.85 (6.95)	8213
168000	84	368.16	6.57 (1.78)	19.68 (5.34)	8586
192000	96	307.70	7.51 (2.44)	22.50 (7.31)	15340
216000	108	405.24	8.46 (2.08)	25.32 (6.24)	16580
240000	120	459.27	9.39 (2.04)	28.13 (6.12)	20070
264000	132	553.96	10.32 (1.86)	30.96 (5.58)	22140
288000	144	606.51	11.28 (1.85)	33.75 (5.56)	26260
312000	156	725.67	12.23 (1.68)	36.57 (5.03)	27900

and for the experiments on Kraken it is 7.8. However, the ratios do not match these values, as shown in table 5.3. The panel verification times from both sets of experiments appear to be about a factor of two larger than the predicted values. The reason for this is most likely that the part of the matrix put in the cache during the operation includes elements that are not part of the panel. The trailing matrix verification needs the entire local matrix, so it ends up making a better use of the cache than the panel verification. If just the elements in the panel could be retrieved from memory, then the panel verification should take about half as long.

5.5.3 Overhead with Failure: Our approach compared to ABFT

In figure 5.10, the expected runtime using ABFT is shown assuming an average time between soft errors of both 40 and 80 minutes for the entire system. These values are chosen to illustrate a situation where our technique would be useful - it is not necessary to take these measures to survive failures unless the failure rate is close to the runtime of the program. When the runtime of the program is orders of magnitude less than the expected time between failures, the probability that a failure will actually occur during a run is low enough to make ABFT the best approach. On a very large system, the error rate would increase and the running time of some applications would also increase, bringing it into the range where our technique is useful.

With a time between failures that is between one and two orders of magnitude longer than the runtime, our technique shows its advantage over ABFT. As the time between failures gets shorter relative to the runtime, our technique gains even more advantage over ABFT.

All times shown include the runtime from an experiment without any fault tolerance and the expected overhead of the specified technique. For our technique, this is the time to verify the checksums and correct errors. For ABFT, this is the expected runtime given that a certain number of repetitions will be required due to propagated errors that make recovery from the checksums impossible.

As the number of processors increases, the expected time for ABFT increases at a faster rate than the time with our checksums. The difference that the error rate makes in the runtime for our method is much less than for ABFT. The expected time for ABFT increases as the error rate increases, and as the number of processors increases.

Table 5.3: With a $N \times N$ matrix, the ratio of trailing matrix verification to panel verification on Ra and Kraken

Ra N	Ra ratio	Kraken N	Kraken ratio
48000	9.32	144000	2.99
64000	8.66	168000	3.00
80000	8.67	192000	2.99
96000	8.43	216000	3.00
112000	8.52	240000	3.00
128000	8.50	264000	3.00
		288000	3.01
		312000	2.99

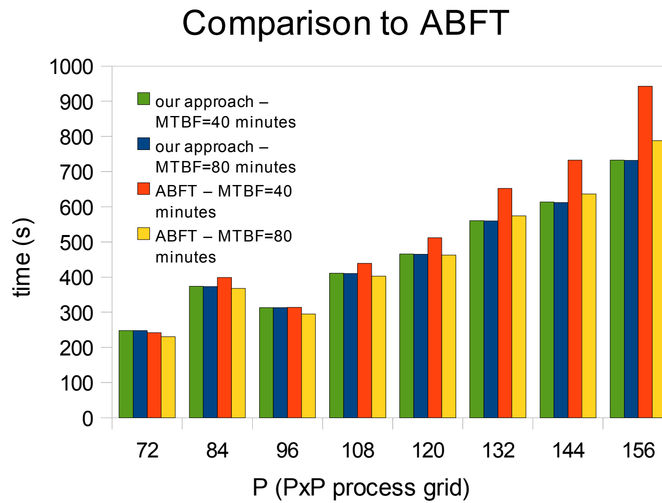


Figure 5.10: Assuming the times shown as the time between failures on average, the expected times using ABFT and our method are shown.

The increase in expected time with an increase in error rate for ABFT is large, while the increase in expected time for our approach is much less. For the numbers of processes shown, ABFT starts with an advantage for the less frequent error rate. However, its expected time increases much more quickly with an increase in the number of processes. To run very large calculations, ABFT becomes impractical. Even for the largest sizes shown here, the overhead of using ABFT is significant.

Chapter 6

Multiple error correction

The first technique allows one fail-stop failure to be corrected. The second technique allows at least one soft error to be both detected and corrected. Both of these techniques use just a sum of elements, that is

$$S = x_1 + x_2 + \cdots + x_n$$

The more errors the technique must be able to handle, the more sums it will need. Each simultaneous error might need another sum in order for it to be corrected.

The second technique can both detect and correct errors because it has two sets of sums that are calculated with different ways of dividing up the matrix elements. However, there is a limit to how the elements can be added up in different ways. A necessity for a technique that anticipates multiple simultaneous errors is coefficients to calculate different sums from the same elements, that is

$$S_1 = a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n S_2 = a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \vdots S_k = a_{k,1}x_1 + a_{k,2}x_2 + \cdots + a_{k,n}x_n$$

The problem of finding coefficients is difficult because they must be chosen so that when the sums are used to recover lost elements, the error is not too large. The problem is to find a matrix of coefficients for which any possible combination of errors will result in an acceptable error in the recovered elements.

6.1 Existing approaches

In [8], the optimal solution for a $2 \times n$ matrix is given:

$$\begin{pmatrix} \cos\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \dots & \cos\frac{(2n-1)\pi}{2n} \\ \sin\frac{\pi}{2n} & \sin\frac{3\pi}{2n} & \dots & \sin\frac{(2n-1)\pi}{2n} \end{pmatrix}$$

The supercondition number of this matrix is approximately $\frac{2n}{\pi}$, so the larger the matrix is, the larger its minimum supercondition number.

For very large matrices, a randomly generated matrix has better condition numbers on average than any other known method of generating it [48]. This means that there is a higher probability that the matrix needed to recover is well-conditioned. Still, if it is possible to find a matrix where every possible combination is well-conditioned, that is better because it guarantees that recovery is possible.

In [43], an evolutionary algorithm is used to find solutions for 2×100 and 3×10. The move used here tries to improve the condition number of the worst-conditioned submatrix by making a small change to the smallest singular value. This improves the condition number of this matrix, although it could make a different submatrix condition number worse.

6.2 Condition number and supercondition number

The condition number of a matrix A is defined as $||A|| \cdot ||A^{-1}||$, using the 2-norm here because it is used in [43], to make the results comparable. A lower condition number means that the solution of the system of equations will have more accuracy. The condition number is the target of the optimization for this problem. When the coefficients are used to recover from multiple failures, which ones are actually used depends on which processors fail. The relevant coefficients (in a $3 \times n$ matrix, for instance) could be from any 3 columns - whichever correspond to the failed processors. Therefore, the resulting system of equations should have good numerical stability no matter which columns are chosen. The name given to the maximum condition number out of any possible combination of columns is the supercondition number.

For the purpose of the evolutionary algorithm, the supercondition number indicates how good of a solution a particular matrix is. However, the supercondition number gives

very little information about the matrix, and does not uniquely identify a particular matrix. Because only a very small number out of the total number of columns contribute to it, enough diversity exists within one matrix so that the algorithm can be run with just one matrix as the evolving solution to the problem.

Often an evolutionary algorithm creates a population of possible solutions, and moves forward by creating new individuals through combinations and mutations, then keeping the best ones. In this algorithm, however, the columns of the matrix act more like the individuals in the population. Each evolutionary step, more candidates are introduced to the population and the least fit individuals are removed. The situation is somewhat complicated by the fact that the fitness of individual columns is only determined by their relationship to all of the other columns in the matrix. Nevertheless, this approach is successful at providing enough diversity that the evolution can continue moving forward, while taking into account the fact that each supercondition number evaluation takes a long time.

6.3 Simulated annealing approach

This problem has been approached with evolutionary algorithms before [43]. However, for large matrices this approach is very slow. It seems that taking a randomized method that optimizes just one matrix should be faster than one that optimizes a population of matrices. Because an exact solution is known for $2 \times n$ matrices, using simulated annealing on $3 \times n$ matrices seems like a good approach. The larger the matrix, the longer calculations take. Since the time to calculate the supercondition number is $O(n^m)$ for a $m \times n$ matrix, where m is determined by the number of sums that are required, finding coefficient matrices for realistically sized problems can become very costly.

The simulated annealing approach is intended here to add more diversity to the evolutionary search. Near the beginning of the calculation, there is a somewhat higher probability that a new column will be kept in spite of creating a higher supercondition number. This approach allows for the possibility that these columns might combine more favorably with other columns in the matrix later in the calculation. However, as the evolution proceeds, the probability of allowing unfavorable columns to be added is lowered. As the matrix converges to a solution, it becomes less likely that a column will eventually come to fit in. Later in the evolution, all of the columns in the matrix are more closely related

to each other, since every possible combination of columns has had a low enough condition number to be kept through all the previous steps of the evolution.

6.4 Origin of moves

The move that removes one column from the matrix while decreasing the supercondition number consists of finding which columns make up the submatrix that has the largest condition number, then deleting one of them. For example, when there are three rows the submatrix with the highest condition number could look like figure 6.1. When one of these columns is deleted, that submatrix no longer exists and the supercondition number of the matrix becomes the next highest condition number, from some other submatrix. In order to perform this move, it is necessary to decide which of the columns of the submatrix to delete. Deleting any one of them will have the effect of lowering the supercondition number, but each column interacts in different ways with the other columns of the matrix. Randomly picking one column to delete is a reasonable way to resolve this problem, given that the evolutionary algorithm already depends on some amount of randomness to generate its solution.

Figure 6.1 illustrates the fact that a large portion of the matrix is not directly affecting its evolutionary fitness. Only the columns of the submatrix with the highest condition number determine the value for the matrix. The remaining columns could be anything, as long as among all their combinations there is none with a higher condition number.

An approach that seemed promising at first was to randomly generate a much larger matrix, then delete columns that contribute to high condition numbers until the target size is reached. This seems like a promising approach because the supercondition number is guaranteed to improve with each step. The supercondition number is determined

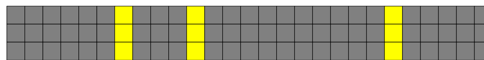


Figure 6.1: A $3 \times n$ matrix with one particular submatrix highlighted

by just a few columns, so if one of those columns is deleted, that condition number no longer exists. The one that replaces it as the highest must be lower, since no new combinations of columns have been added. Some factors that work against this approach are the time it takes to find the supercondition number of a very large matrix, and the fact that larger matrices are guaranteed to have higher supercondition numbers. Experimental results from this approach did not seem promising.

An alternative that is based on this first approach is to repeatedly add a small number of random columns, then delete down to the target size. This is essentially the same as the first approach, since the columns in that matrix were randomly generated to start with. It has the benefit of each column deletion taking less time on average than it would for the first method, and that it can be continued for an arbitrary number of moves instead of being constrained to the number that gets the matrix down to the desired size. A disadvantage of this approach is that it is possible for the supercondition number to increase when a new random column is added. However, it is possible to remove this effect, at least when only one column is added and deleted at a time. If the new column is part of the matrix that creates the supercondition number, then delete it. If it is not, that means that it does not create a larger condition number in combination with any of the columns already in the matrix, so it is favorable with respect to the existing columns.

In a test run on a 2×100 matrix with 100 deletions, which means that one approach starts with a 2×200 matrix and deletes columns until there are 100, and the other starts with 100 columns and adds and deletes one at a time 100 times, the first approach reached a supercondition number of 399.47 while the other had reached 352.41. Even if this is not always the pattern, for practical reasons the add-and-delete approach is better. Each operation on average takes less time since the time to find the supercondition number is $O(n^m)$ for a $m \times n$ matrix (m is 2 or 3 in these experiments, but could be larger; this assumes that n is much larger than m). The approach of starting with a much larger matrix is also fixed to a certain number of moves, while the add-and-delete approach can be continued for any number of moves.

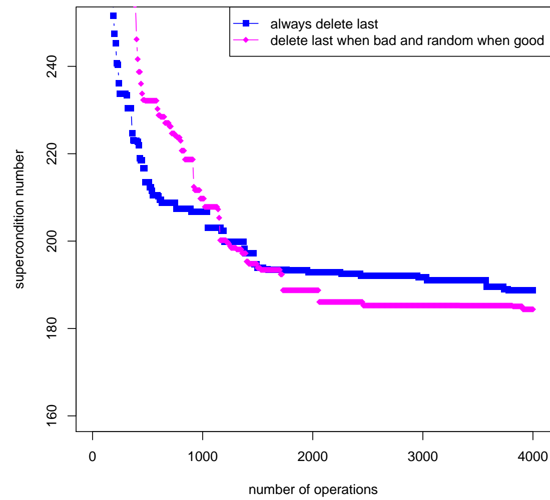


Figure 6.2: Comparison of two add-and-delete approaches when 10 columns are added then deleted at a time

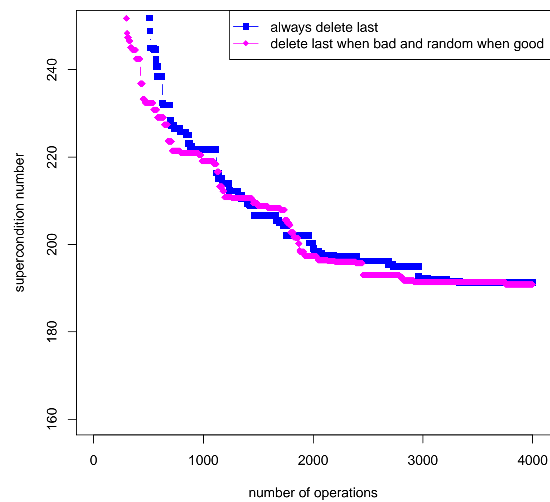


Figure 6.3: Comparison of two add-and-delete approaches when 1 column is added then deleted at a time

6.5 Variations on add-and-delete

When adding only one column then picking one to delete, there are two choices that seem reasonable. One option is to always delete the last of the columns that contributes to the supercondition number, and the other is to pick one at random when the added column is not part of the supercondition number submatrix, and delete the new column when it makes a worse supercondition number. It seems like the second option should be the best because the first option tends to keep more of the original matrix intact. It is unlikely to ever remove columns that are early in the matrix, which may actually be unfavorable.

It is easier to extend the first option to adding multiple columns than the second, but for the second option it is also possible. If the matrix dimension is n and k columns are added, then deleted each move, then by always deleting a column with an index greater than n if it is part of the worst-conditioned submatrix will almost be the same as deleting new columns if they turn out to be unfavorable. A new column could escape being deleted even though it is part of the current worst matrix if columns within the original matrix are deleted first, making its index less than n even if it is part of the k added columns. However, this occurrence is not very likely, and there is still a chance of the unfavorable column being deleted in this step or a later one. Also, the value of the columns depends only on the rest of the matrix, so a column that creates a high condition number may become more favorable when just a few other columns are removed.

An alternative to the two moves above that may seem reasonable is to always randomly choose a column to delete, not favoring the newly added columns for deletion. However, it turns out that the newly added columns are more likely to be bad with respect to the rest of the matrix, especially when the matrix is larger. This kind of move could be useful as a bad move for a simulated annealing approach, since it has the potential to introduce new columns that will keep the matrix from being stuck in a local minimum.

In figure 6.2, just the end of the run is shown. The values start much higher at the beginning, but if the entire range is shown the difference at the end is not visible. Here ten columns are added before being deleted. An operation is one column being removed. In this run, the approach of always deleting the last column of the worst-conditioned matrix does better earlier, but levels off at a higher value then the approach of only deleting the last column when it is one of the newly added columns. Even though the difference is small, it seems reasonable that the second approach would do better at the end of the run because

it can create more possible new matrices.

Figure 6.3 shows the mixed approach doing only slightly better. Nevertheless, it is used for the simulated annealing runs because it guarantees to always improve or not change the supercondition number while offering more diversity in moves.

6.6 Results

Our experiments show that the best approach is to delete the last column when it is a new one and a randomly chosen column otherwise. The approach that always chooses a random column to delete performs very badly alone, but also adds the most variety since it adds a lot more new columns: randomly generated columns are usually bad with respect to the existing matrix, so are usually deleted right away with the first approach. Therefore we chose to do a variation of simulated annealing where which of those two approaches to use is determined by the temperature, where the first approach is the "good" move and the second approach is the "bad" move. This seems to make sense because, using only one type of deletion approach, the unfavorable move typically is one that changes a column without changing the supercondition number. It makes more sense to use a move that will probably add a new column in order to add more variety.

Figure 6.4 shows a run on a 3×10 matrix. The best result had a supercondition number of 35.7, which is somewhat close to the value of about 21 obtained in [43] with less than half as many supercondition number evaluations. It seems promising that this method may give better results than the alternative evolutionary algorithm for larger matrices.

Figure 6.5 shows a run with a 2×100 matrix. It shows the effect of bad moves early in the run, then it converges to about 170. Unfortunately this is higher than the known miniumum value of about 63, indicating that this combination of parameters did not work well for this matrix size.

Figure 6.6 shows a run on a 3×40 matrix that ends with a supercondition number of 1126. While we cannot know how close this is to optimal, it seems likely that the optimal supercondition number of a $3 \times n$ matrix would grow at a faster rate than that of a $2 \times n$ matrix. The $2 \times n$ supercondition number is $O(n)$.

Figure 6.7 shows a run with a 3×100 matrix. In this run, the value of the supercondition number went as high as 9×10^{16} because of bad moves, which is not shown

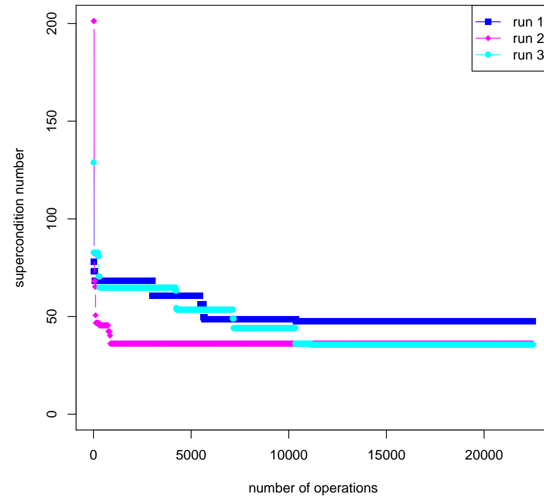


Figure 6.4: Three runs of 3×10 matrix with simulated annealing

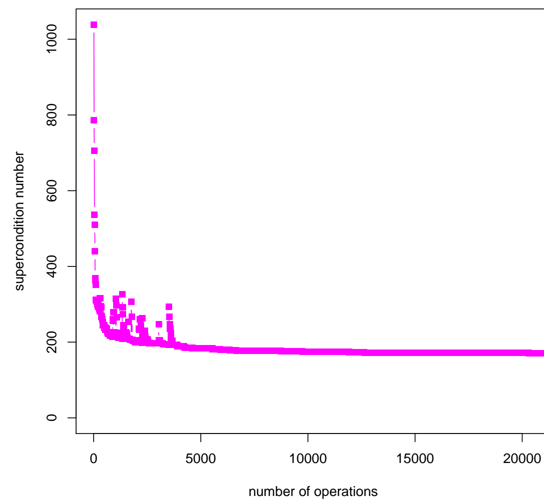


Figure 6.5: A run of 2×100 matrix with simulated annealing

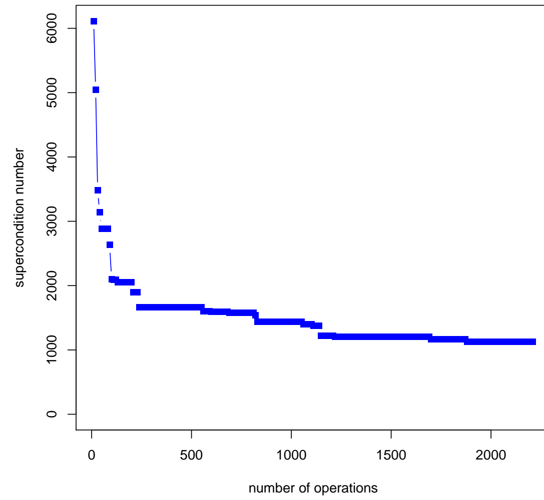


Figure 6.6: A run of 3×40 matrix with simulated annealing

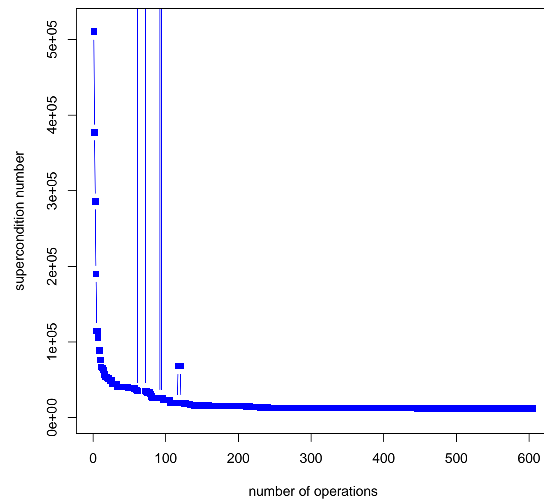


Figure 6.7: A run of 3×100 matrix with simulated annealing

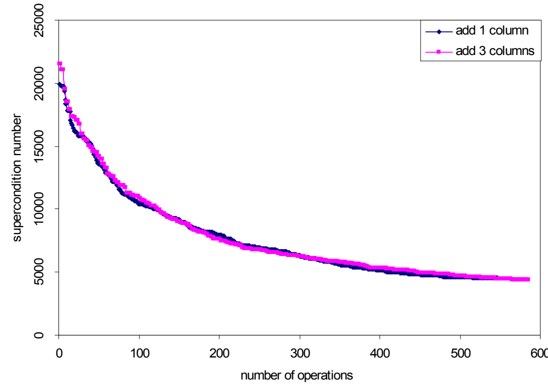


Figure 6.8: For a 2×1000 matrix, comparison in progress between adding 1 or 3 random columns per move

on the graph. The solution after 600 operations had a supercondition number of 11825, and it appears to be close to converging. Since at this point in a run the amount of improvement for additional moves is small, this solution is a reasonable one that can be obtained fairly easily.

Figure 6.8 shows the results of adding either one or three columns at a time when generating a 2×1000 matrix. In [43], the algorithm to find a 2×1000 matrix, run in parallel, took about a month of computing time and 30,000 supercondition number evaluations to come up with a matrix with a supercondition number that was 1.5 times the optimum. Our simulated annealing approach was able to find a matrix with a supercondition number one order of magnitude greater than the optimum (meaning one more digit of accuracy lost in recovery) in only a few hours, using about 400 supercondition number evaluations. As long as no better method for generating large matrices exists, this method seems to be a good way of generating usable, if not optimal, solutions.

6.7 Optimization

The biggest difficulty in using an evolutionary approach to find a matrix of coefficients that can be used in a large system is the time it takes to find the supercondition number. Since the time is $O(n^m)$ for a $n \times m$ matrix, where n is the number of elements in the sum and m is the number of sums, it can quickly become very time-consuming to attempt to find coefficients for a realistically sized problem. It is useful to reduce the num-

ber of supercondition number evaluations it takes to reach an acceptable solution. One aspect of this problem is the amount of error that is acceptable in the calculation that the fault tolerance is being used for. If a certain amount of error is acceptable, the search for the coefficient matrix can be stopped once the condition number is low enough, even if it is not optimal. Another consideration with the coefficient matrix is that the supercondition number is only the maximum possible condition number. When an error occurs, the coefficients for the lost elements are likely to be a different combination that results in a lower condition number. These considerations might mean that the time spent on slightly lowering the supercondition number is not worthwhile.

An important improvement on the time for the overall evolutionary algorithm is adding and removing multiple columns in one step. Adding a few extra columns when the matrix is large slightly increases the time to find the supercondition number, but just one supercondition number evaluation allows multiple new columns to be added to the matrix. As our results show, adding more than one new column at a time does not affect how quickly the calculation converges, but if, for instance, three columns are added per supercondition number evaluation, the same result as when one column is added at a time can be found with the amount of time spent on calculating supercondition numbers reduced by nearly a factor of three.

Here we only added a small additional number of columns, but for very large coefficient matrix it might be worthwhile to look for further ways to improve the time it takes to find a solution. One possibility is adding a much larger number of columns for each step. It would be necessary to find out to what extent this approach would affect how well the evolutionary algorithm can converge. Another possibility might be to start the process by finding multiple smaller matrices, then combining them in order to create the starting point for the evolution of the matrix of the required size. This possibility is less likely to be as effective, but could produce some useful information on techniques for finding much larger coefficient matrices.

Chapter 7

Conclusion

By adding a row checksum to the matrix, we are able to make the right-looking LU factorization with partial pivoting fault tolerant. We can recover lost data from the original matrix and from U resulting from one process failure. The overhead of the method consists mostly of the time to calculate the checksum at the beginning, using a reduce. The time to perform the checksum is approximately proportional to the size of the local matrix stored on one process, so that the overhead time can be kept almost constant when the matrix size is increased, decreasing the overhead as a fraction of the total time. This method can perform with much lower overhead than diskless checkpointing, which is a very good option in general. Although this method is specific to matrix operations, it can offer much better performance than diskless checkpointing for those operations.

We have also created a fault tolerance method that handles soft errors in the LU factorization. Our technique uses global and local checksums to both detect and correct soft errors when they occur, preventing repeated calculation. The frequency of checking can be adjusted to the error rate.

In this technique we have used only one checksum to handle one failure. However, with weighted checksums it is possible to recover from multiple failures, or to use the additional checksums to detect as well as recover from errors.

Bibliography

[1] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12), 2008.

[2] Y. Kim. *Fault Tolerant Matrix Operations for Parallel and Distributed Systems*. PhD thesis, University of Tennessee, Knoxville, June 1996.

[3] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, Philadelphia, PA, USA, June 2006.

[4] G. A. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. *CTWatchQuarterly*, 3(4), November 2007.

[5] Z. Chen and J. Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers*, July 2009.

[6] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[7] C. Wang, F. Mueller, C. Engelmann, and S. Scot. Job pause service under lam/mpi+blcr for transparent fault tolerance. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, March 2007.

[8] Z. Chen. Optimal real number codes for fault tolerant matrix operations. In *Proceedings of the ACM/IEEE SC2009 Conference on High Performance Networking, Computing, Storage, and Analysis*, Portland, OR, USA, November 2009.

[9] D. Hakkarinen and Z. Chen. Algorithmic cholesky factorization fault recovery. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, GA, USA, April 2010.

[10] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5(2):172–184, 1988.

[11] Charng da Lu. *Scalable diskless checkpointing for large parallel systems*. PhD thesis, Univ. of Illinois Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2005.

[12] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In *24th EUROMICRO Conference*, 1998.

[13] T.C. Chiueh and P. Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, 1996.

[14] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3), August 2009.

[15] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *IEEE/ACM Supercomputing Conference*, November 2010.

[16] Leonardo Arturo Bautista Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed diskless checkpoint for large scale systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010.

[17] Zizhong Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, Miami, FL, USA, April 2008.

[18] Zizhong Chen and Jack Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[19] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl, September 2008.

[20] Konrad Malkowski, Padma Raghavan, and Mahmut Kandemir. Analyzing the soft-error resilience of linear solvers on multicore multiprocessors. In *24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[21] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *International Conference on Supercomputing*, 2008.

[22] Greg Bronevetsky, Bronis R. de Supinski, and Martin Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientic applications. In *IEEE Workshop on Silicon Errors in Logic - System Effects*, 2009.

[23] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 385–396, New York, NY, USA, 2010. ACM.

[24] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 86 –94, 1999.

[25] Imran S. Haque and Vijay S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. *CoRR*, abs/0910.0505, 2009.

[26] Chong Ding, Christer Karlsson, Hui Liu, Teresa Davies, and Zizhong Chen. Matrix multiplication on gpus with on-line fault tolerance. In *Proceedings of the 9th IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE Computer Society Press, 2011.

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

[28] P. Banerjee, J. T. Rahmeh, C. B. Stunkel, V. S. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, C-39:1132–1145, 1990.

[29] J.G. Silva, P. Prata, M. Rela, and H. Madeira. Practical issues in the use of abft and a new failure model. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 26 –35, June 1998.

[30] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP' 12)*, 2012.

[31] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. High performance dense linear system solver with soft error resilience. In *IEEE Cluster*, 2011.

[32] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the 25th ACM International Conference on Supercomputing*. ACM Press, 2011.

[33] C. J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computers*, 37(12), December 1988.

[34] John A. Gunnels, Robert A. van de Geijn, Daniel S. Katz, and Enrique S. Quintana-Orti. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *The International Conference on Dependable Systems and Networks*, 2001.

[35] P. Banerjee and J.A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *IEEE Transactions on Computers*, 2006.

[36] J.Y. Jou and J.A. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. In *Proceedings of the IEEE*, volume 74, May 1986.

[37] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*, 2012.

[38] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the ACM SIG-PLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, USA, June 2005.

[39] J. S. Plank, Y. Kim, and J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *IEEE Journal of Parallel and Distributed Computing*, 43:125–138, 1997.

[40] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[41] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[42] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[43] Aaron Garrett, Zizhong Chen, and Daniel Eric Smith. Constructing numerically stable real number codes using evolutionary computation. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1163–1170, New York, NY, USA, 2010. ACM.

[44] Zizhong Chen and Jack J. Dongarra. Condition numbers of gaussian random matrices. *SIAM J. Matrix Anal. Appl.*, 27:603–620, July 2005.

[45] V. S. S. Nair. *Analysis and design of algorithms-based fault-tolerant systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1990.

[46] J. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[47] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17:530–531, September 1974.

[48] Z. Chen and J. Dongarra. Numerically stable real number codes based on random matrices. In *Proceeding of the 5th International Conference on Computational Science*, 2005.