High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing

Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen
Colorado School of Mines
Golden, CO, USA
{tdavies, ckarlsso, huliu, cding, zchen}@mines.edu

ABSTRACT

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. 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. In this paper, 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 without checkpointing. 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 fail-stop process failures in High Performance Linpack can be tolerated without checkpointing. 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.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Performance, Reliability

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.
Copyright 2011 ACM 978-1-4503-0102-2/11/05 ...$10.00.

Keywords

High Performance Linpack benchmark, LU factorization, fault tolerance, algorithm-based recovery

1. INTRODUCTION

Fault tolerance is becoming more important as the number of processors used for a single calculation increases [26]. When more processors are used, the probability that one will fail increases [14]. 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. 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.

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 [9, 20, 25, 28] or by maintaining some sort of checksum of the data [6, 8, 17, 21] from which it can be recovered. A method that can be used for any application is Plank's diskless checkpointing [4, 11, 12, 15, 22, 25, 27], 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 [20], 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 [5, 7]. 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 [17]. In this paper, we extend the checksum technique to the LU decomposition used by High Performance Linpack (HPL) [23].

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 show in this paper 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.

The rest of this paper will explain related work that leads to this result in section 2; the features of HPL that are important to our application of the checksum technique in section 3; the type of failure that this work is able to handle in section 4; the details of adding a checksum to the matrix in sections 5; proof that the checksum is maintained in section 6; and analysis of the performance with experimental results in sections 7 and 8.

2. RELATED WORK

2.1 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 [28]. 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 [25] 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, which would 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 processor in one group, all of the other processors can revert to their stored checkpoint, and the redundant data along with the data of all the other processors in the group is used to recover the data of the failed processor.

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 [10]. However, matrix operations are not susceptible to this optimization [20], 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 [24].

2.2 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance [1–3, 16, 18, 19, 21] is a technique that has 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 fact. A checksum of a matrix can be used to locate and correct entries in the solution matrix that are incorrect, although we are most interested in recovery. Failure location is determined by the message passing library in the case of the failure of a processor.

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 [8], 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 should 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.

2.3 Applications of Checksum-Based Recovery

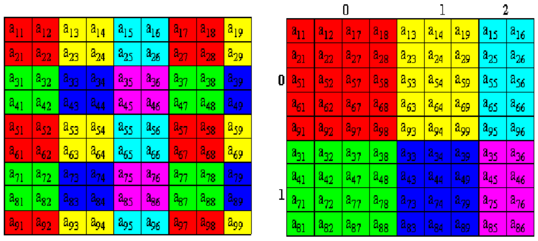
It has been shown in [8] how a checksum is maintained in the outer product matrix-matrix multiply. It is also made clear that not all algorithms will maintain the checksum in the middle of the calculation, even if the checksum can be shown to be correct at the end. So it is important to ensure that the 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 [17]. Here a checksum is used to recover from a failure during the ScaLAPACK Cholesky decomposition.

3. HIGH PERFORMANCE LINPACK

HPL performs a dense LU decomposition 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



a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

	0				1			2	
	a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
	a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
0	a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
	a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
	a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
	a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
	a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
1	a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
	a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

Figure 1: Global and local matrices under the 2D block cyclic distribution [8].

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.

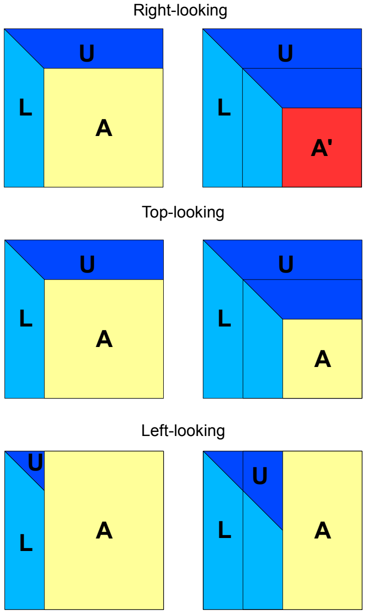
Matrices are often stored in 2D block cyclic fashion [8,17, 20]. Block cyclic distribution is used in HPL. This storage pattern creates a good load balance for many operations, since it is typical to go through the matrix row by row or column by column (or in blocks of rows or columns). With a block 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 1. As this figure indicates, the global matrix will not necessarily divide evenly among the processors. However, we currently are 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 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 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 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.

Figure 2: Three different algorithms for LU decomposition [20]. 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.



4. FAILURE MODEL

The type of failure that we focus on in this work 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. 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.

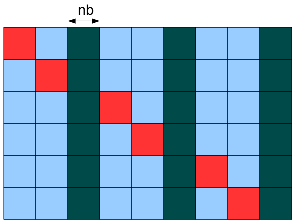


Figure 3: Global view of a matrix with checksum blocks and diagonal blocks marked. Because of the checksum blocks, the diagonal of the original matrix is displaced in the global matrix.

Soft failures are a task for later research, 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.

5. CHECKSUM

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 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 3 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.

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. 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.

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.

It is not guaranteed that the checksum can be maintained for any algorithm for calculating the LU decomposition. 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.

6. ALGORITHMS FOR LU DECOMPOSITION

Like all matrix operations, there are different algorithms for the LU decomposition, 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.

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.

Left-looking and top-looking LU decomposition do not maintain a checksum because various parts of the matrix are not updated in a given step, shown in figure 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.

6.1 Proof

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.

7. PERFORMANCE

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 [13]. 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.

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 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.

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.

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.

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.

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.

In addition to the time overhead, both techniques have the overhead of additional processors that are required to hold either the checksum or checkpoint. However, with the trend of using more and more processors, processors can be considered cheap. Additionally, 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.

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.

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

Table 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 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

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.

8. EXPERIMENTS

8.1 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 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

8.2 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 1, 2, and 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 shows the results on a different large system. Here also the overhead is typically less than 1%. Table 5 shows the results for small matrices. Even with few processes the overhead is low, and it decreases as the size increases.

Figure 4 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.

Table 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 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

Figure 4: 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.

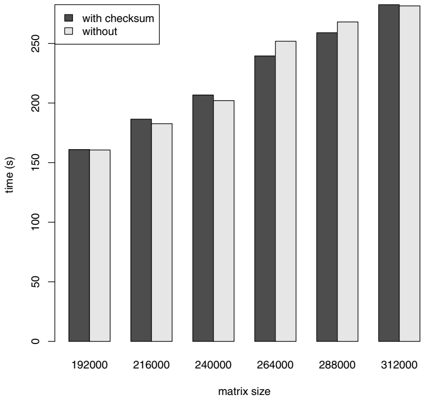


Table 6: Jaguar: local matrix size 2000×2000, block size 64

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

Table 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

8.3 Overhead with recovery

Tables 6 and 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 of the amount of work that is repeated, the first entry in table 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.

8.4 Algorithm-based recovery versus diskless checkpointing

According to [29], 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

Figure 5: Fault tolerance overhead without recovery: Algorithm-based recovery versus diskless checkpointing

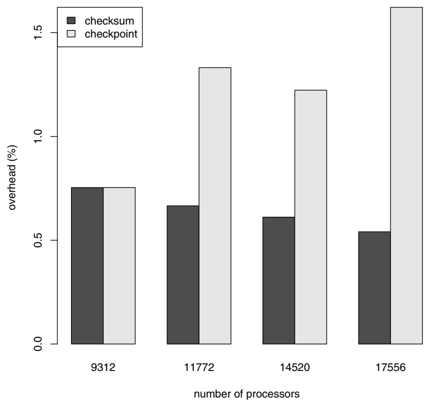
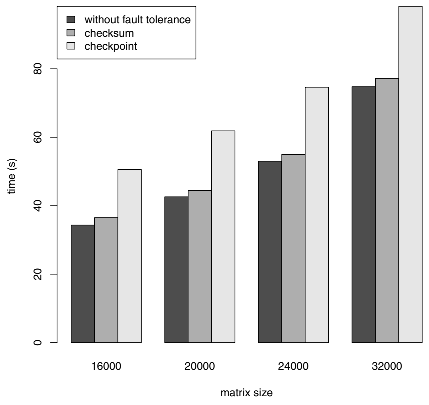


Figure 6: On smaller runs on Ra, the difference between checksum and checkpoint can be easily seen.

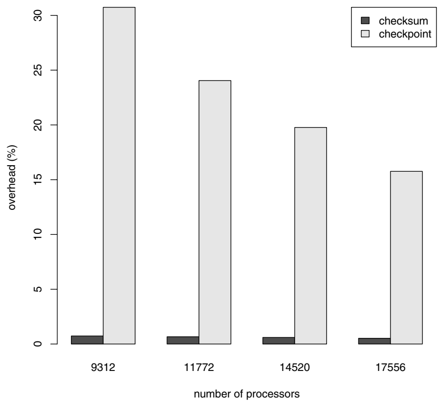


factor is the number of processes, which makes failures more likely and decreases the length of the checkpoint interval.

As an example of possible checkpoint overhead, figure 5 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 6 shows the results of smaller runs, comparing checksum and checkpoint overhead without a failure. Figure 7 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.

Figure 7: Fault tolerance overhead with recovery: Algorithm-based recovery versus diskless checkpointing



9. CONCLUSION

By adding a row checksum to the matrix, we are able to make the right-looking LU decomposition 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.

In this work 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. These possibilities will be explored in future work.

Acknowledgments

This research is partly supported by National Science Foundation under grant #OCI-0905019 and Department of Energy under grant DE FE#0000988.

We would like thank the following institutions for the use of their computing resources:

- The National Center for Computational Sciences: Jaguar
- The National Institute for Computational Sciences: Kraken
- The Golden Energy Computing Organization: Ra

10. REFERENCES

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

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

[3] 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.

[4] 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.

[5] Z. 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.

[6] 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.

[7] Z. Chen and J. 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.

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

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

[10] 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.

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

[12] C. 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.

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

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

[15] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010.

[16] J. A. Gunnels, R. A. van de Geijn, D. S. Katz, and E. S. Quintana-Orti. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *The International Conference on Dependable Systems and Networks*, 2001.

[17] 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.

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

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

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

[21] 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.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *IEEE/ACM Supercomputing Conference*, November 2010.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

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