UT-CS Technical Report #UT-CS-04-xxx

Recovery Patterns for Iterative Methods in a Parallel Unstable Environment

G. Bosilca*† Z. Chen*† J. Dongarra*‡† J. Langou*†
bosilca,zchen,dongarra,langou@cs.utk.edu

December 2004

Abstract

A simple checkpoint-free fault-tolerant scheme for parallel iterative methods is given. Assuming that when one processor fails, all its data is lost and the system is recovered with a new processor, this scheme computes a new approximate solution from the data of the non-failed system. The iterative method is then restarted from this new vector. The main advantage of this technique over standard checkpoint is that there is no extra computation added in the iterative solver. In particular, if no failure occurs, the fault-tolerant application is the same as the original application. The main drawback is that the convergence after failure of the method is no longer the same as the original method. In this paper, we present this recovery technique as well as some implementations of checkpoints in iterative methods. Finally, experiments are presented to compare the two techniques. The fault tolerant MPI library is the FT-MPI library. Iterative linear solvers and iterative eigensolvers are considered.

*This research was supported in part by the Los Alamos National Laboratory under Contract No. 03891-001-99 49 and the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

†Department of Computer Science, The University of Tennessee, Knoxville, TN, USA

‡Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA

1 Introduction

Among the most remarkable features of the ongoing computational revolution in science is the ease with which the aspirations of domain researchers have overtaken and outstripped the explosive growth in computing power described by Moore's law. The unquenchable desire of scientists to run ever larger simulations and analyze ever larger data sets is fueling an escalation in the size of supercomputing clusters from hundreds, to thousands, and even tens of thousands of processors. Unfortunately, the struggle to design systems that can scale up in this way also exposes the current limits of our understanding of how to efficiently translate such increases in aggregate computing resources into corresponding increases in scientific productivity.

One increasingly urgent aspect of this knowledge gap lies in the critical area of reliability and fault tolerance. Even making some generous assumptions (e.g. that the reliability of a single-processor system is several years), it is clear that as the processor count in high end clusters grows into the thousands, the mean time to failure (MTTF) will drop from a few days to a few hours, or less. The type of 100,000-processor machines projected in the next few years can expect experience a processor failure almost hourly. Although today's architectures are robust enough to incur process failures without suffering complete system failure, at this scale and failure rate, the only technique available to application developers for providing fault tolerance within the current parallel programming model "checkpoint/restart" has performance and conceptual limitations that makes it inadequate to the future needs of large scale simulation and modeling community who will use these systems.

To fulfill these needs a new message passing library has been created called FT-MPI [6, 8]. FT-MPI enables an implementer to create fault tolerant algorithms while providing the maximum of freedom to the user. Based on this library, it becomes possible to create more and more fault-tolerant algorithms and software without the need for specialized hardware; thus providing the numerical analyst the ability to explore a new area for implementation and development.

In order that applications survive faults, we design the following model. The recovery process for the application is made of three phases:

- Phase I : recover a correct computational environment,
- Phase II : recover the statistic data lost,
- Phase III: recover the dynamic data lost.

Phase I is the need to recover a correct MPI environment. In this paper, the recovered environment has the same number of processors as the failed one. This is the task of the FT-MPI library. Phase II consists in recovering the static data lost. By static data, we mean for example the matrix, the right-hand side, the preconditioner. This represents data that is computed once in the initialization phase of the application and is unchanged after. Phase III consists in recovering the dynamic data, this is the data that changes during the algorithm.

In this paper we mainly discuss the Phase III. Previous solutions to recover the dynamic data were based on checkpointing. Checkpointing is a way to provide fault tolerant applications that requires additional time and memory (or disk, or processors). In Section 3, we explain how to implement checkpoints efficiently in some iterative solvers. The particular checkpoint technique we consider in the experiments is named diskless checkpointing [11], and is particularly suited when there is a low MTTF. This technique is explained in Section 3.1.

Checkpoints involve global operations with large size data and their overhead is directly proportional to the number of nodes involved in the application. For iterative methods it is all the more bothering that their bottleneck regarding scalability is mostly in the computation of the scalar products essentially because this is a global operation [15, §12.2]. Adding checkpoints to an iterative method is just getting worse this problem. Even though the checkpoints of the dynamical variable would have scaled, it is still an additional cost to the application itself. For most computing systems today, applications are not that likely to encounter a failure and thus many users prefer to take their chance. The mode is to run the application without checkpoint and if a failure occurs restart the application from scratch. To solve both of these issues it is necessary to find a way to operate Phase III without any significant overhead in the original application.

Our primary concern is the iterative methods to solve the linear system $Ax = b$. Parts of the vectors are stored on each of the processors. A failure of one of the processors results in the lost of all the data stored in its memory (local data). Therefore when a failure occurs, a part of our approximate solution is lost. Assuming that no checkpoint of the dynamic variables has been performed and a failure occurs, what can be done? At this point, the local data of the approximate solution before failure $x^{(old)}$ is lost on a processor. Being positive, we prefer say that the approximate solution before failure $x^{(old)}$ is still known on all the processors but one. Thus our idea is to restore a new approximate solution from this data. This is done by solving the local equation associated with the failed processor. In the sequel, $A_{i,j}$ represents the sub-matrix which rows are stored on processor i and with column indexes corresponding to the rows stored on the processor j, x_j is the local part of the vector x stored on the processor j. If processor f fails then we propose to construct a new approximate solution $x^{(new)}$ via

$$\begin{aligned} x_j^{(new)} &= x_j^{(old)} \quad \text{for } j \neq f \\ x_f^{(new)} &= A_{f,f}^{-1}(b_f - \sum_{j \neq f} A_{f,j} x_j^{(old)}) \end{aligned} \tag{1}$$

providing $A_{f,f}$ is of full rank. (We assume that we use exact arithmetic in this discussion.) If $x^{(old)}$ was the exact solution of the system, Equation (1) constructs then $x_f^{(new)} = x_f^{(old)}$; the recovery of x is exact. In general the failure happens when $x^{(old)}$ is an approximate solution, in which case $x_f^{(new)}$ is not exactly $x_f^{(old)}$. After this recovery step, the iterative methods is restarted from $x^{(new)}$. The goal of Section 4 is to explain this technique and give some theoretical results about it. This method is sometimes referred as *lossy algorithm* (as opposed to loss-less for the checkpoint method), the reason is that the dynamic data of the failed processor (e.g $x_f^{(old)}$) is lost and is not recovered, we recover $x_f^{(new)}$ an approximation of it. In Section 4.3, we also explain how Equation (1) can be generalized for eigensolvers. In Section 5, we present some experimental results that compares the lossy method with checkpoint method.

The study is dedicated to single failure at a time. Theoretically it is not an issue to generalize the results to multiple failures at a time. Some hints addressing this problem are given in Section 4.2.

2 Description of the FT-MPI Library From the User Level

Although MPI [14] is currently the de-facto standard system used to build high performance applications for both clusters and dedicated MPP systems, it is not without its problems. Initially MPI was designed to allow for very high efficiency and thus performance on a number of early 1990s MPPs, that at the time had limited OS runtime support. This led to the current MPI design of a static process model. While this model was possible to implement for MPP vendors, easy to program for, and more importantly something that could be agreed upon by a standard committee. The second version of MPI standard known as MPI-2 [7] did include some support for dynamic process control, although this was limited to the creation of new MPI process groups with separate communicators. These new processes could not be merged with previously existing communicators to form intracommunicators needed for a seamless single application model and were limited to a special set of extended collectives (group) communications.

The MPI static process model suffices for small numbers of distributed nodes within the currently emerging masses of clusters and several hundred nodes of dedicated MPPs. Beyond these sizes the mean time between failures (MTBF) starts becoming a factor. As attempts to build the next generation Peta-flop systems advance, this situation will only become more adverse as individual node reliability becomes outweighed by orders of magnitude.

The aim of FT-MPI is to build a fault tolerant MPI implementation that can survive failures, while offering the application developer a range of recovery options other than just returning to some previous checkpointed state.

Current semantics of MPI indicate that a failure of a MPI process or communication causes all communicators associated with them to become invalid. As the standard provides no method to reinstate them (and it is unclear if we can even free them), we are left with the problem that this causes the communicator MPI_COMM_WORLD itself to become invalid and thus the entire MPI application will grid to a halt.

For more about how to get an application fault tolerant with the FT-MPI library, we refer to [6].

3 Description of the Iterative Methods With Checkpoint

For a complete description of iterative methods we refer the reader to [1]. In the paper we use the same notations Our discussion will focus on the GMRES method [12] but there is no difficulty in generalizing to the other iterative methods described in [1].

3.1 Diskless Checkpoint-Restart Technique

In order to recover the data from any of the processors while maintaining a low overhead in the storage, we are using a checksum approach to checkpointing. The information of the computing processors is saved in case of a failure. The information is encoded in the following way: if there is n processors for each of whom we want to save the vector x_k (for simplicity, we assume that the size of x_k is the same on all the processors), then the checkpoint unit stores the checksum x_{n+1} such that $x_{n+1} = \sum_{i=1,\ldots,n} x_i$. If processor f fails, we can restore x_f via $x_f = x_{n+1} - \sum_{i=1,\ldots,n; i \neq f} x_i$. The arithmetic used for the operations $+$ and $-$ can either be binary or floating-point arithmetic. (However note that, if the floating-point arithmetic is used, then one has to be aware that the recovered data is not the same as the initial one due to round-off errors; in particular, one shall expect important error if coefficients in the vector of x differs by large order of magnitude.) Our checkpoints are diskless, in the sense that the checkpoint is stored in memory of a processor and not on a disk. To achieve this, an additional processor is added to the environment. It will be referred as the checkpoint processor and its role is to store the checksum. The checkpoint processor can be viewed as a disk but with low latency and high bandwidth since it is located in the network of processors. For more information, we refer the reader to [11], where a special interest is given on how to deal simultaneous failures.

We will describe how we perform a recovery and only concentrate on the vector quantities. The scalar quantities (e.g. the number of iterations,...) are trivial to restore in case of a failure.

3.2 Classification of Checkpointing Strategies

To perform the checkpoint, we will classify the algorithms in different categories. But first of all, we need to classify the variables of the algorithm. The goal of this classification is to define which variables in memory need to be stored once (static, e.g. the system matrix A, the right-hand side b, or the preconditioner P), which variables are changing along the iterations (dynamic, e.g. the approximate solution x), which variables can be computed (using mathematical equivalence) after a failure rather than checkpointed (e.g. obtaining the residual via $r = b - Ax$ might be faster than via a checkpoint), which variables can be recomputed in case of a rollback but it is worth to save in order to save time (e.g. a scalar product is expensive to compute, easy to store and its value is the same on all the processors thus it makes sense to store those values in an array on all the processors, at recovery time, we provide those values to the failed processor, this avoids to recompute those values during the roll back).

Based on this classification of the variables, we give two different strategies of checkpointing. The first one, chkpt_R, is suited for GMRES with restart and Conjugate Gradient (e.g.), the other, chkpt_F, is suited for Full GMRES and Arnoldi (e.g.). In the experimental part (Section 5), both categories (i.e. GMRES with restart and Arnoldi method) are represented and compared with the lossy approach.

We note that checkpoint strategy can be derived to recover the static data (matrix, right-hand side, preconditioner) in Phase II (instead of a disk I/O e.g).

3.3 Full GMRES - Arnoldi Method (Eigenvalue Computation), An Example of Checkpointing at Each Iteration

In Full GMRES and Arnoldi methods, in order to perform iteration k, we need the knowledge of k vectors. This means basically that at each iteration a vector is constructed that is mandatory to continue the method, thus this vector has to be checkpointed. Full GMRES and Arnoldi method therefore have a very simple checkpoint strategy: each time a vector is computed, it is checkpointed. This strategy is called chkpt_F.

3.4 Conjugate Gradient Method and GMRES with Restart, an Example of Checkpointing with Rollback

The common point of CG and GMRES with restart is that, in both methods, the iteration k can be computed from the knowledge of only a constant number of vectors (independent of k).

For example in CG, to perform the iteration k, we need to know three vectors: x_{k-1}, p_{k-1} and r_{k-1}. This means that the vectors created at iteration $k-2$ are useless for the iteration k. (In practice, the CG implementation simply overrides those vectors by the new ones.) In this case, it makes sense to checkpoint all the vectors of a given iteration only from times to times. If a failure happens then we restart the computation from the last checkpointed version of those vectors. This is called a roll back and in case of a failure some of the computations have to be redone. It is clear that the rate of checkpoint has to be chosen carefully. In the one hand, distant checkpoints require roll back to a state long time ago and thus a large amount of computation have to be redone. In the other hand, close checkpoints imply a consequent overhead due to the large number of global communication introduced.

To know the optimum rate of checkpoints, we propose the following model.

Let us call:

$T_{\texttt{chkpt}}$	the time taken to make a checkpoint,
$T_{\texttt{MTTF}}$	the mean time to failure (the failure rate is assumed an exponential distribution),
$T_{\texttt{iter}}$	the time for an iteration (or any unit time step of the code),
$T_{\texttt{iter}} \cdot k$	the time between two checkpoints,
$T_{\texttt{iter}} \cdot N$	the time for a code without fault-tolerance and correct execution.

Given these definitions, we can write

$$T_{\texttt{total}} = N * T_{\texttt{iter}} + T_{\texttt{chkpt}} * \lfloor N/k \rfloor + \sum_{\text{for all the failures}} T_{\texttt{recover}}$$

which means that the total time is the sum of the time to perform the N iterations, the time to perform the checkpoints every k iterations and the time to perform the recovery of the encountered failures.

The time to perform the recovery of all the failures is given by the number of failure ($T_{\texttt{total}}/T_{\texttt{MTTF}}$) times the mean time to perform a recovery. The mean time to perform a recovery is the time of a checkpoint ($T_{\texttt{chkpt}}$) plus the time of the rollback, $T_{\texttt{rollback}}$. Thus we can write

$$T_{\texttt{total}} = N * T_{\texttt{iter}} + T_{\texttt{chkpt}} * N/k + T_{\texttt{total}}/T_{\texttt{MTTF}}(T_{\texttt{chkpt}} + T_{\texttt{rollback}})).$$

Taking into account that the distribution of failure is exponential of parameter $T_{\texttt{MTTF}}$, the distribution for a failure that happened between 0 and $kT_{\texttt{iter}}$ is:

$$\frac{1}{T_{\texttt{MTTF}}(1 - e^{-kT_{\texttt{iter}}/T_{\texttt{MTTF}}})} e^{-t/T_{\texttt{MTTF}}},$$

for t between 0 and $kT_{\texttt{iter}}$; and 0 elsewhere. The mean of this law, $T_{\texttt{rollback}}$, is given by

$$T_{\texttt{rollback}} = T_{\texttt{MTTF}}(1 - \frac{kxe^{-kx}}{1-e^{-kx}}) \quad \text{where } x = \frac{T_{\texttt{iter}}}{T_{\texttt{MTTF}}}.$$

Note that if kx is close to 0 then a good approximation for $T_{\texttt{rollback}}$ is $kT_{\texttt{iter}}/2$ which means that the failures happens statistically in the middle of the checkpoint interval. The interpretation is the following: if kx is close to 0 (i.e. $kT_{\texttt{iter}}$ is negligible against $T_{\texttt{MTTF}}$), then the exponential distribution of parameter $T_{\texttt{MTTF}}$ on 0 and $kT_{\texttt{iter}}$ is close to a uniform distribution, distribution for which the mean is $kT_{\texttt{iter}}/2$. When kx increases, the mean of the distribution deviates from $kT_{\texttt{iter}}/2$ and gets smaller.

Returning back to the expression of the total time we can write

$$T_{\texttt{total}} = \frac{N * T_{\texttt{iter}} + T_{\texttt{chkpt}} * N/k}{\frac{kxe^{-kx}}{1-e^{-kx}} - \frac{T_{\texttt{chkpt}}}{T_{\texttt{MTTF}}}}. \tag{2}$$

Our goal is to minimize the total time $T_{\texttt{total}}$ with respect to the parameter k. For the sake of simplicity we linearize the exponentials around $kx = 0$, we obtain

$$T_{\texttt{total}} = \frac{N * T_{\texttt{iter}} + T_{\texttt{chkpt}} * N/k}{1 - \frac{T_{\texttt{chkpt}}}{T_{\texttt{MTTF}}} - \frac{kT_{\texttt{iter}}}{2T_{\texttt{MTTF}}}}$$

the minimum is obtained for

$$k = \frac{\sqrt{T_{\texttt{chkpt}}(2T_{\texttt{MTTF}} - T_{\texttt{chkpt}})}}{T_{\texttt{iter}}} - \frac{T_{\texttt{chkpt}}}{T_{\texttt{iter}}}.$$

Giving the optimal time between two checkpoints

$$k \cdot T_{\texttt{iter}} = \sqrt{T_{\texttt{chkpt}}(2T_{\texttt{MTTF}} - T_{\texttt{chkpt}})} - T_{\texttt{chkpt}}. \tag{3}$$

If $T_{\texttt{MTTF}} >> T^2_{\texttt{chkpt}}$, then

$$k \cdot T_{\texttt{iter}} \sim \sqrt{2T_{\texttt{MTTF}} \cdot T_{\texttt{chkpt}}}. \tag{4}$$

The same formula as in [9].

In order to assess the validity of Formula (3) and Formula (4), we present an experimental simulation in Figure 1. Here are the characteristics of the simulation: $T_{\texttt{iter}}$ and $T_{\texttt{chkpt}}$ are assumed constant along the iteration (this is the case for CG, but not GMRES). In our model, N is set to 1500, $Titer$ is set to $1\texttt{ut}$ (unit of time), $T_{\texttt{chkpt}}$=$T_{\texttt{iter}}$ and $T_{\texttt{MTTF}} = 50\texttt{ut}$. The time to recover the MPI environment is set to 0 and the time to recover the data is the time to perform one checkpoint; we are mostly interested in the effect of the roll back.

For each time interval checkpoint `k`, we perform `nb_experiments`=60 experiments and we report the mean time to solution for those sample with the blue curve. First of all, note that for those value of $T_{\texttt{iter}}$ and $T_{\texttt{chkpt}}$, there is clearly an optimal value of checkpoint. We then plot the two models corresponding to Formula (3) and Formula (4). We see that the black curves corresponding to Formula (3) are closer to the blue curve, the experimental curve, than the red, corresponding to Formula (4) is. This was expected since the black curve takes into account second order term in $T_{\texttt{chkpt}}$ while the red curve does not. If the ratio is kept reasonably large enough $T_{\texttt{MTTF}}/T_{chkpt}$, we see that the Formula 4 is accurate enough to have a good estimation of the minimum, the Formula 2, for itself, is fully representative of the experiments.

For GMRES with restart m, to compute the vector v_{k+1} at iteration k, we need $k[m] + 1$ vectors. The checkpointing strategy we choose is to checkpoint the data when $k = 0$ (at the restart). In this case, we just have one vector to checkpoint (the approximate solution x) per m iterations. This strategy is called chkpt_R. Note that if the size of the restart is long relative to the mean time between failure of the machine (then the method becomes closer to Full GMRES), it might be more advantageous to checkpoint GMRES with restart as Full GMRES (at every iteration) in order to avoid long roll back (see Section 3.3).

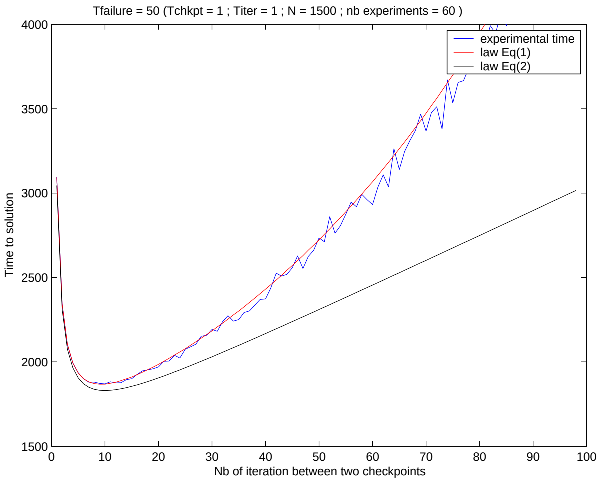

Figure 1: Comparison of the models for the checkpoint/rollback optimization

4 The Lossy Approach

The lossy approach with Block Jacobi step is defined by Equation (1). The lossy approach is strongly connected to the Block Jacobi algorithm. Indeed, a failure step with the lossy approach is a step of the Block Jacobi algorithm on the failed processor. Related work is by Engelmann and Geist [5], they propose to use the Block Jacobi itself as a scalable algorithm to failure. In fact the Block Jacobi step can be performed for the recovery but this can be embedded in any solver. Thus the user is free to use the iterative solver he wants. In particular one can choose one of the Krylov methods that are known to be more robust than the Block Jacobi method. On a related note, this is the work of Jacobi himself, during the time when computation were doing by hand. Gauss [10, p.321] states that the method was extremely tolerant to errors.

In the next section, we discuss about the convergence we shall expect after a recovery. In the Section 4.2 we give some issues about the lossy approach and how to tackle them away.

4.1 Quality of the New Approximate Solution Given by the Lossy Approach

The lossy approach implies that the iterations are different after a failure and recovery than from a run without failure. (This is directly opposed to the checkpoint strategy where the recovery provides the lost data.) In this section, we give some hints to describe whether the convergence (in term of iterations) will occur at the same iteration between failed and non-failed versions or that we should expect a long delay before the convergence of the lossy approach. Surprisingly enough, failures even enhance the convergence in some cases.

To quantify the convergence property of the lossy approach, we focus on the size of the residual norm "jump" made during the recovery and the speed of convergence after the recovery.

Since the lossy approach is nothing more than a step of Block-Jacobi-like method, a part of the theory of stationary iterative methods applies and we can prove that

$$\|x^{(new)} - x^*\| \leq \left(1 + \|A_{ff}^{-1}\|^2 \sum_{j \neq f} \|A_{fj}\|^2\right)^{1/2} \|(x_j^{(old)} - x_j^*)\| \tag{5}$$

$$\|r^{(new)}\| \leq \left(1 + \|A_{ff}^{-1}\|^2 \sum_{j \neq f} \|A_{jf}\|^2\right)^{1/2} \cdot \|r^{(old)}\| \tag{6}$$

As a result of these inequalities we can clearly quantify the jump of the residual norm after a recovery; the new residual norm is close to the previous one if $(\|A_{ff}^{-1}\|^2 \sum_{j \neq f} \|A_{jf}\|^2\|)^{1/2}$ is small compared to 1 (or at least of the same order). This assumes that the diagonal block is not ill-conditioned and the norm of the extra-diagonal blocks is small relative to the norm of diagonal blocks.

The residual norm of the approximate solution is not the only thing that matters in an iterative solver. The iterative solver computes other information in other vectors, losing those vectors and restarting from the new approximate solution could theoretically lead to some delay in the convergence. This problem is the same problem as the one induced by any restart in an iterative method. In a general manner, the lossy approach will perform well if the convergence behavior of the method is linear or sub-linear. Thus the lossy approach is justified in all the restarted method (in particular GMRES with restart) provided the "jump" of the residual norm (controlled by Equation (6)) is not too high.

GMRES with restart has the potential to stagnate during the iteration. We can view this as: along the restarts, GMRES finds a fixed point for itself. This fixed point can be described by its spectral properties. If the failure occurs during stagnation, the lossy approach computes another vector with the same quality in term of residual norm (Equation (5) and (6)) but this purges the vector from its bad spectral properties. In our experiments (see Table 3 and Table 4), we often observe that GMRES with restart with failure and the lossy approach performs better in term of number of iterations than the non-failed approach.

4.2 Remarks About the Lossy Approach

Block Jacobi preconditioner

The main cost of the recovery step in Phase III is to perform the LU factorization of the local matrix. However it is worth to note that if the preconditioner used is a Block Jacobi preconditioner those factors are available from step II and thus the recovery of x can be done to the price of a preconditioner step and a matrix-vector product.

What about a singular diagonal block $A_{i,i}$?

If the matrix A is nonsingular (which is assumed), we can extract rows from the column block $A_{:,i}$ such that these rows form a nonsingular square block. Thus in theory, this is not an issue; however in practice, we only focused to matrix with nonsingular (and even well-conditioned) diagonal block. Once more in the case of a Block Jacobi preconditioner, this property is assumed and thus the lossy approach fits well.

What about a matrix-free method?

The lossy approach needs to know the diagonal block corresponding to the failed processor. In some matrix-free method, those blocks are known; when they are not, the lossy approach will not work. An idea is to apply the global matrix-vector product to solve iteratively the local system (with restriction operators). Since the space in which we are working is smaller than the size of the initial matrix, the iterative solve should converge faster (than redoing the whole run).

What about multiple failures on a single instance ?

This is not a theoretical issue. If processors i and j fail, a lots of options are available. Basically we have to solve the following system

$$\begin{pmatrix} A_{i,i} & A_{i,j} \\ A_{j,i} & A_{j,j} \end{pmatrix}.$$

This can be done via any iterative method (GMRES), Block Jacobi steps, direct solve, whatever.

Restriction on the iterative method used
An important requirement of this recovery technique is to know the approximate solution x at each step of the iterative methods. This is true for most of the iterative methods (the stationary iterative methods, Conjugate Gradient, Orthomin, GCR, BiCGStab,...) but not all. For example, in the Full GMRES method, the approximate solution is computed only at the end of the algorithm. In this latter example, we use the following trick (see [1] for the notation). The solution $x^{(k)}$ at step k is implicitly known via the formula:

$$x^{(k)} = x^{(0)} + (v^{(1)}, \ldots, v^{(k)})y^{(k)}, \tag{7}$$

where the quantities $y^{(k)}$ is a small vector that can be computed from the data of any non-failed processors and the vectors $x^{(0)}$, $v^{(1)}$,...,$v^{(k)}$ are classically distributed among the processors. From Equation (7), if a failure occurs on the processor f, the local part of $x^{(k)}$ can be computed at this time on all the non-failed processor. Thus the lossy approach can also be used without modifying the generic algorithm (x computed at convergence only).

4.3 Generalization to Eigenvalue Computation

We believe that the concepts presented in the lossy algorithm for solving systems of linear equations could be applied to other methods as well. In this section, we move from the linear solves to the eigencomputation. We use the Arnoldi algorithm as described in [13, §7.5] and will take the same notation. For the sake of the simplicity, we suppose in this description that the blocksize of the method is 1 and that we are looking for the largest eigenvalue of the matrix A. (In the experimental Section 5, we take more complex case.) The lossy approach for the Arnoldi method is defined as follows. If the processor f fails at iteration n_f, Phase III is:

1. for each non failed processor, compute the largest eigenvalue $\lambda^{(Ritz)}$ and the associated eigenvector $w^{(Ritz)}$ of $H^{(n_f)}$,

2. for each non failed processor, k, compute its local part of the Ritz vector: $v_k^{(Ritz)} = (v_k^{(1)}, \ldots, v_k^{(n_f)})w^{(Ritz)}$,

3. the failed processor sets its local part of the Ritz vector to 0: $v_f^{(Ritz)} = 0$,

4. $x = Av^{(Ritz)} - v^{(Ritz)}\lambda^{(Ritz)}$,

5. the failed processor performs the local solve $x_f = (A_{f,f} - I\lambda^{(Ritz)})\ x_f$

6. the new vector x is an approximation of the eigenvector associated to the largest eigenvalue, then the Arnoldi method can be restarted with x as starting vector.

5 Experimental Results

Experiments are performed on the boba Linux cluster at The University of Tennessee composed of 64 dual processors Intel Xeon at 2.40 GHz with Myrinet interconnect. We use the double precision floating-point arithmetic. The MPI library used is FT-MPI. Test matrices are from The University of Florida sparse matrix collection [4]. We have chosen the matrices among the largest unsymmetric matrices available in the collection.

The experiments we report are simulations of what we aim in the sense that we aim machines with thousands of processors while we are working with tens of processors. However we want to make clear that the simulation stops there. The failure of one process is a real failure. It is simulated in the experiments by a forced `exit` in the process designed to fail. The software developed is ready to be used on a real large-scale system.

We recall that the recovery is performed in three phases: Phase I: recover of the MPI environment, Phase II: recover of the static data and Phase III: recover of the variable data. Even though the subject of the paper is neither Phase I nor Phase II, we give some clues here on what are our actual implementation choices.

Phase I is based on FT-MPI, we used the classical approach described in Section 2.

To recover the matrix A and the right-hand side b, we have chosen to perform a disk I/O. Since the matrix is stored in a file, this is a rather natural solution. We have changed the original storage format of the matrices. They are not stored on the Harwell Boeing compressed sparse column format but we rather preprocess them to a compressed sparse row format. Doing this, each processor need access a contiguous part of the file in disk. At the first reading of the matrix (initialization of the code), we store in an array the pointers where each processor starts to read the file (we use the C routine `ftell`), this part of the initialization is sequential. We spread these pointers on all the processors. If a failure occurs, at Phase II of the recovery, we first recover the pointer corresponding to the restarted processor and then we access the data in this huge file as if we had one small file for the failed processor (we use the C routine `fseek`). Another solution would have been to perform a diskless checkpoint of the matrix at the initialization. This solution is currently an option of the code we have, it performs similarly to a disk I/O for what we have observed. We will not discuss this subject in this paper; it will be an interesting subject only with a larger number of processors. If there is a preconditioner (static data) then our choice is to recompute the lost LU factors (no checkpoint neither).

For the lossy approach, the local solve is done via UMFPACK Version 4.3 [3]. The default parameters are used. We give two sets of results, first, in Section 5.1, we go in detail through a given matrix, in particular we justify some technical choices made in the code; then in Section 5.2, we present our main results.

5.1 Justifications of some choices

In this section, we study: (a) whether we shall perform the checksum either in floating-point arithmetic or in binary arithmetic; (b) whether it is advantageous to save the scalar product during the algorithm.

The matrix, A, we consider in this section is `cage14`. It is of order $n = 1,505,785$ with $nnz = 27,130,349$ nonzero elements. The run is performed on 32 computing processors (which means 33 for the checkpoint algorithm since there is an additional checkpoint processor in the configuration). The right-hand side is $b = Ax^*$ where x^* is the vector with all ones. The iteration stops when the iterative method has found an approximate solution x such that $\|b - Ax\|/\|b\|\cdot$ tol, where tol$= 10^{-6}$. The method we study is GMRES(30) without preconditioner and without failure it converges in 13 iterations and lasts 15.47s (Cf. Table 1).

The experiments consists in the same run but with the failure at iteration 10 of the processor #0, the recovery mode is chkpt_R. In this case (see Section 3.4), the chkpt_R algorithm performs checkpoints at each restart. Since the failure (iteration 10) is far before the first restart (iteration 30), the only checkpoint made is done at iteration 0. Thus, when the failure occurs at iteration 10, the algorithm has to rollback to the last checkpoint, that is to rollback to iteration 0. And then it performs the 13 iterations necessary to converge. Thus the 23 iterations for the chkpt_R algorithm reported in Table 1 comes with no surprise. In this particular case, the Full GMRES algorithm and the GMRES(30) algorithm are the identical. The choice of chkpt_R in this case is not appropriate and one shall certainly have performed checkpoint of the vectors at each iterations (chkpt_F) in order to have no rollback at the failure. (see Section 5 for a confirmation). This example is good to stress out that the optimization of the number of checkpoint versus the roll back (discussed in Section 3.4) is an important issue, in most of the cases this problem can be anticipated.

In Table 1, we present the results obtained where # iters represents the number of iterations for the algorithm to converge and $T_{\texttt{Wall}}$ the time to solution (in seconds). The detailed timing of the recovery will be discussed in the next section.

We compare three variants of the chkpt_R algorithm explained in Section 3.3: the first uses the double-precision arithmetic, the second uses the binary arithmetic and the third uses double-precision arithmetic in which we recompute the scalar products at the roll back.

1. At our scale of problems, reusing the scalar products seems not to have a large effect in the overall time. We expect that this effect shall be more important when the size of the problem gets larger or when the roll back is more important. In the sequel the code we present do not recompute the scalar products.

2. Using either binary arithmetic or double-precision arithmetic seems not to be a big issue. Both provide the same timing results and the errors due to the floating-point arithmetic are not damaging. In the sequel, the checkpoints are made using the binary arithmetic.

matrix	n	nnz	tol	# procs	n_f	nnz_f
`cage14`	$1,505,785$	$27,130,349$	10^{-6}	32	$47,056$	$414,240$

method	recovery		iter$_f$	# iters	$T_{\texttt{Wall}}$
GMRES(30)			no	13	15.47
	chkpt_R	double-precision	10	23	28.66
	chkpt_R	double-precision - dot recomputed	10	23	28.92
	chkpt_R	binary	10	23	28.80

Table 1: Comparison of three variants of the checkpoint fault-tolerant algorithms chkpt_R, times are given in seconds

5.2 Experiments with Unsymmetric Matrices

In Table 3, we give the main results of our experiments. In the following, we first go through the different parameters and explain their consistency among the different matrices tested then we are drawing the conclusions of the table.

Eight matrices are tested and the results for a given matrix are given in the Table 2 (GMRES(30) with Block Jacobi preconditioner), Table 3 (GMRES(30) without preconditioner) and Table 4 (Arnoldi Method).

For each of the matrices, we give the name, the order (n), the number of nonzero elements (nnz), the tolerance (tol) and the number of computational processors (# procs). We are considering that only one failure happen in the experiments, and for the sake of comparison between the methods, it happens always on the same processor and at the same iteration. For the local matrix of the failed processor we give its order(n_f) and its number of nonzero elements (nnz_f). These two numbers are representative of the amount of work that we would need to accomplish during a recovery step. The load balancing among the processors is done by setting $n_i = n/$ (# procs). For our matrices, this proves to equilibrate well nnz_i. The iterative solver is stopped at iteration i if: $\|b - Ax_i\|_2 \leq \|r_i\|_2 \cdot$tol. Regarding the Arnoldi method, defining $x_i^{(k)}$, the approximate solution of unit norm of the k^{th} eigenvector, and $\lambda_i^{(k)}$, the approximate value for the k^{th} eigenvalue, $\lambda_i^{(k)} = (x_i^{(k)})^T A x_i^{(k)}$, the eigensolver is stopped at iteration i if: $\|Ax_i - x_i\lambda_i\|_2 \leq |\lambda_i| \cdot$tol. Note that if the algorithm uses checkpoints the number of processors used is # procs +1 whereas for the lossy variant it is # procs.

For the different matrices and the different iterative methods, we test the three recovery modes explained in Section 3.1 and 4 (chkpt_R, chkpt_F or lossy). For the sake of comparison, we also provides the information for the scheme without failure (with or without checkpoint).

The iteration where the failure occurs is iter_f, we set it at (roughly) the half of the number of iterations for the scheme without failure, or we set two values one at the first third, the other at the second third. The LU factors of the block diagonals of the initial matrices used in the Block Jacobi preconditioner are computed via UMFPACK Version 4.3 [3].

Then we give the results: the number of iterations to converge (# iters), the time to solution ($T_{\texttt{Wall}}$), the time of the checkpoint ($T_{\texttt{chkpt}}$), the time lost in the rollback ($T_{\texttt{Rollback}}$: for the chkpt_F method, there is no roll back; for the chkpt_R method, there is a roll back; for the lossy approach if this time is positive, the method with failure performs more iterations than without and this quantify the time spent in those iterations, this time is negative when the Block Jacobi step of the lossy approach improves the convergence), the time for the recovery ($T_{\texttt{Recov}}$: it is the maximum time among all the processors of the difference between the time when the code enter the recovery routine and the time when the code exit it), the time for Phase I of the recovery (T_{I}: this is the time that it takes to the system and the FT-MPI library to provide a new MPI environment, we typically measure it on one of the non faulty processors), the time for Phase II of the recovery ($T_{\mathrm{II},A,b}$: the time to do the I/O to recover A and b; and if needed $T_{\mathrm{II},P}$: the time to compute the preconditioner, we measure them on the restarted processor) and finally the time for Phase III of the recovery (T_{III}: the time to recover a value on the restarted processor for x, it is measured on the restarted processor). All the time are given in seconds.

For all the experiments, we shall have the following identities (in theory):

$$T_{\texttt{Wall}} = T_{\texttt{Wall}}(\text{lossy}) + T_{\texttt{chkpt}} + T_{\texttt{Rollback}} + T_{\texttt{Recov}},$$
$$T_{\texttt{Recov}} = T_{\mathrm{I}} + T_{\mathrm{II},A,b} + T_{\mathrm{II},P} + T_{\mathrm{III}}.$$

Whereas the checkpoints for the strategy chkpt_R are not significant, they are more significant for chkpt_F. For example for Arnoldi method and `cage12`, the checkpoint represents up to 8.7% of the method.

For a given number of processors, we observe that the time for Phase I (recovery of a correct MPI environment) is constant. It is in fact almost proportional to the number of processors used: $0.60s$ for 8 processors, 1.10s for 16 and 2.00s for 32. The use of `ftell` and `fseek` in the I/O have tackles away the I/O problem. At this point, recover the static data (Phase II) is of the same order of magnitude as Phase I and Phase III. Phase III consists in the recover of x, it is either the time to do the *inverse* checkpoint or to solve the local problem.

In case of a failure, T_I and T_{II} shall be the same whether we use the checkpoint or the lossy variant. The time to recover will only differ from T_{III}. Note that if the preconditioner used is Block Jacobi (Table 2) then for the lossy algorithm, Phase III is almost free. The burden of the computation of the factorization of the local matrix is migrated in Phase II ($T_{\mathrm{II},P}$).

Even though, those problems are still toys problems of small size, it is important to note that both fault-tolerant techniques (checkpoint and lossy) have reached their initial goal. The number of iterations (resp. time for solution) for these variants with a fault is significantly smaller than the double of the number of iterations (resp. time for solution) for this variant without a fault. Therefore there are much better than the trivial fault-tolerant algorithm that consists in retrying the job from the beginning in the event of failure.

Since we lose part of the convergence theory of the initial method, the main fear with the lossy algorithm is to obtain a results that do not converge at all. As claimed in Section 4, we note that, for GMRES(30), the best number of iterations is given for the failed lossy algorithm not the algorithm without failure (four cases on five). So indeed, the lossy recovery improves the convergence. For Arnoldi, the lossy recovery performs more iterations than the original algorithm, but this remains reasonable.

We see from the table that the lossy and the checkpoint algorithm compare fairly in term of time and even though when one is better than the other the results are pretty closed.

One trend is however clear. Regarding Block Jacobi preconditioned GMRES with restart, the checkpoint versions (chkpt_R and chkpt_F) perform more iterations (due to the roll back) as the original algorithm (for chkpt_R at least), have some checkpoint overhead and have the same recover cost as the lossy algorithm. Thus, as long as, the lossy algorithm performs a less or equal number of iterations as the original algorithm (which is always the case in Table 3), it is faster.

Conclusions

The lossy technique (at least on the form presented in this paper) is intended to work on matrices where a Block Jacobi preconditioner is appropriate. In this paper, only matrices that satisfy this property are presented and, from our experience, and with no improvements of the technique, it does not generalize to other matrices. The lossy algorithm has its risks. Albeit Equation (5) and (6), the success of the lossy algorithm is hard to predict (in particular the speed of convergence after the recovery). The robust solution is at this point checkpointing. Regarding performance point of view, we have seen that the checkpointing algorithm performs well and that, for the size of problem we consider (less than 32 processors), by carefully adapting the checkpoint algorithm to the iterative methods, the overhead is acceptable.

A major advantage of the lossy algorithm resides in the fact that it enables fault tolerance with no overhead when there is no failure. We think that at this stage of early beginning of the implementation of the fault tolerance in end user codes it might be a convincing argument for the user. A consequence is that this method can be plugged as an external library to any existing software without modifications of the code. Another advantage of the lossy algorithm is that, for a sparse matrix, Phase III of the recovery only involves a few number of processors.

GMRES(30) with Block Jacobi preconditioner

matrix	n	nnz	tol	# procs	n_f	nnz_f
`fidap035`	$19,716$	$218,308$	10^{-6}	8	$2,465$	$26,848$

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Rollback}}$	$T_{\texttt{Recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
lossy	no	353	7.38							
chkpt_R	no	353	7.40	0.02						
lossy	150	348	7.95			0.72	0.60	0.04	0.04	0.01
chkpt_R	150	353	7.96	0.02	none	0.71	0.60	0.04	0.04	0.00

matrix	n	nnz	tol	# procs	n_f	nnz_f
`af23560`	$23,560$	$484,256$	10^{-6}	8	$2,945$	$59,841$

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Rollback}}$	$T_{\texttt{Recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
lossy	no	52	3.23							
chkpt_R	no	52	3.23	0.00						
lossy	30	51	4.30			1.08	0.62	0.09	0.32	0.02
chkpt_R	30	52	4.29	0.00	none	1.06	0.63	0.09	0.32	0.00

matrix	n	nnz	tol	# procs	n_f	nnz_f
`stomach`	$213,360$	$3,021,648$	10^{-10}	16	$13,335$	$185,541$

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Rollback}}$	$T_{\texttt{Recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
lossy	no	18	7.98							
chkpt_F	no	18	8.43	0.52						
chkpt_R	no	18	8.15	0.00						
lossy	10	18	14.11			5.50	1.05	0.33	3.61	0.35
chkpt_F	10	18	13.65	0.52		5.19	1.10	0.33	3.61	0.13
chkpt_R	10	28	16.00	0.00	2.29	5.15	1.11	0.33	3.61	0.01

Table 2: Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm, times are given in seconds

GMRES(30) (no preconditioner)

matrix	n	nnz	tol	# procs	n_f	nnz_f
stomach	$213,360$	$3,021,648$	10^{-10}	16	$13,335$	$185,541$

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Rollback}}$	$T_{\texttt{Recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
lossy	no	385	38.89						
chkpt_R	no	385	41.04	1.92					
lossy	100	372	42.38		−1.56	5.38	1.03	0.33	3.91
chkpt_R	100	395	45.49	1.92	2.40	1.68	1.02	0.32	0.20
lossy	200	374	42.44		−1.32	5.46	1.02	0.33	3.83
chkpt_R	200	395	47.34	1.92	3.60	1.83	1.02	0.33	0.20

matrix	n	nnz	tol	# procs	n_f	nnz_f
cage14	$1,505,785$	$27,130,349$	10^{-6}	32	$47,056$	$414,240$

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Rollback}}$	$T_{\texttt{Recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
lossy	no	13	15.47						
chkpt_F	no	13	16.88	1.50					
chkpt_R	no	13	15.49	0.02					
lossy	10	14	21.36			6.35	2.20	1.56	1.64
chkpt_F	10	13	22.92	1.50		5.51	2.20	1.73	1.39
chkpt_R	10	23	28.80	0.02	7.12	4.80	2.20	1.50	0.24

Table 3: Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm, times are given in seconds

Arnoldi Method

matrix	n	nnz	tol	n_e	bs	# procs	n_f	nnz_f
`fidap035`	19,716	218,308	10^{-6}	3	3	8	2,465	26,848

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Recov}}$	$T_{\texttt{I}}$	$T_{\texttt{II},A,b}$	$T_{\texttt{III}}$
lossy	no	69	2.32					
chkpt_F	no	69	2.51	0.19				
lossy	35	83	3.02		0.88	0.60	0.03	0.20
chkpt_F	35	69	3.69	0.19	0.88	0.61	0.04	0.23

matrix	n	nnz	tol	n_e	bs	# procs	n_f	nnz_f
`torso1`	116,158	8,516,500	10^{-6}	4	4	16	7,260	425,766

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Recov}}$	$T_{\texttt{I}}$	$T_{\texttt{II},A,b}$	$T_{\texttt{III}}$
lossy	no	60	16.35					
chkpt_F	no	60	17.28	0.92				
lossy	35	77	23.92		3.46	1.08	0.58	1.30
chkpt_F	35	60	21.28	0.92	2.90	1.08	0.56	0.33

matrix	n	nnz	tol	n_e	bs	# procs	n_f	nnz_f
`cage12`	130,228	2,032,536	10^{-2}	5	5	8	16,279	162,766

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Recov}}$	$T_{\texttt{I}}$	$T_{\texttt{II},A,b}$	$T_{\texttt{III}}$
lossy	no	120	24.28					
chkpt_F	no	120	24.33	0.10				
lossy	65	146	36.02		11.55	0.60	0.22	10.43
chkpt_F	65	120	26.31	0.10	1.44	0.90	0.22	0.31

matrix	n	nnz	tol	n_e	bs	# procs	n_f	nnz_f
`cage13`	445,315	7,479,343	10^{-6}	1	1	32	13,917	112,831

recovery	iter$_f$	# iters	$T_{\texttt{Wall}}$	$T_{\texttt{Chkpt}}$	$T_{\texttt{Recov}}$	$T_{\texttt{I}}$	$T_{\texttt{II}}$	$T_{\texttt{III}}$
lossy	no	63	44.11					
chkpt_F	no	63	47.97	3.86				
lossy	30	73	54.89		2.72	2.25	0.19	0.48
chkpt_F	30	63	50.15	3.86	2.84	2.06	0.18	0.38

Table 4: Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm, times are given in seconds

In this paper, we only have focused on the single failure at a time problem (either for the checkpoint or for the lossy approach). However our codes are able to deal with any number failures provided they occur separately (after a failure, Phase III has to be finished before another failure is allowed to happen). Generalization to deal with multiple failures at the same time is theoretically not an issue. Also note that current work of second and third authors is actively dealing with this issue for the checkpoint algorithms [2].

For the lossy approach, in the case where we are not using a Block Jacobi preconditioner (or for multiple failures at a time), the local solve is performed via UMFPACK Version 4.3 [3]. This is a sparse direct solver. Another idea is certainly to perform the local solve via an iterative method. When multiple failures occur simultaneously, this alternative is interesting. Also note that using an iterative method enables to play with the stopping criterion, if the failure occurs while the error is at a given level, it makes sense to solve the local system only at this level. Using a direct solver, the local solve are always done to full accuracy.

We have observed that performing a Block Jacobi step between two GMRES cycles often ameliorates the speed of convergence (in our case always). This confirms recent works that propose hybrid scheme (like Block Jacobi/GMRES) to cure the stagnation of GMRES with restart.

References

[1] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk A. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994. Also available as postscript file on http://www.netlib.org/templates/Templates.html.

[2] Zizhong Chen and Jack Dongarra. Numerically stable real-number codes based on random matrices. Technical Report UT-CS-04-526, University of Tennessee Computer Science Department, October 2004.

[3] Tim A. Davis. UMFPACK version 4.3 user guide. http://www.cise.ufl.edu/research/sparse/umfpack/.

[4] Tim A. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[5] Christian Engelmann and G. Al Geist. Development of naturally fault tolerant algorithms for computing on 100,000 processors. http://www.csm.ornl.gov/˜geist.

[6] Graham E. Fagg, Edgar Gabriel, George Bosilca, Thara Angskun, Zizhong Chen, Jelena Pjesivac-Grbovic, Kevin London, and Jack J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference*, June 2004.

[7] The Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm, 1995.

[8] Edgar Gabriel, Graham E. Fagg, Antonin Bukovsky, Thara Angskun, and Jack J. Dongarra. A fault-tolerant communication library for grid environments. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03), International Workshop on Grid Computing*, 2003.

[9] William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *International Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

[10] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[11] James S. Plank, Youngbae Kim, and Jack Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43:125–138, 1997.

[12] Youcef Saad and Martin H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[13] Yousef Saad. Arnoldi Method. In Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, pages 161–166. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[14] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI- The Complete Reference. Volume 1, The MPI Core, second edition*. MIT Press, Cambridge, MA, 1998.

[15] Henk A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, 2003.