SIAM J. SCI. COMPUT.
Vol. 30, No. 1, pp. 102–116

RECOVERY PATTERNS FOR ITERATIVE METHODS IN A PARALLEL UNSTABLE ENVIRONMENT*

J. LANGOU†, Z. CHEN‡, G. BOSILCA§, AND J. DONGARRA§

Abstract. Several recovery techniques for parallel iterative methods are presented. First, the implementation of checkpoints in parallel iterative methods is described and analyzed. Then a simple checkpoint-free fault-tolerant scheme for parallel iterative methods, the lossy approach, is presented. When one processor fails and all its data is lost, the system is recovered by computing a new approximate solution using the data of the nonfailed processors. The iterative method is then restarted with this new vector. The main advantage of the lossy approach over standard checkpoint algorithms is that it does not increase the computational cost of the iterative solver when no failure occurs. Experiments are presented that compare the different techniques. The fault-tolerant FT-MPI library is used. Both iterative linear solvers and eigensolvers are considered.

Key words. fault-tolerant algorithms, iterative methods, parallel distributed

AMS subject classifications. 65F10, 65F50, 68W10, 68W15

DOI. 10.1137/040620394

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 to 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 make it inadequate to the future needs

*Received by the editors December 7, 2004; accepted for publication (in revised form) May 3, 2007; published electronically November 16, 2007. This research was supported in part by the Los Alamos National Laboratory under contract 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.
http://www.siam.org/journals/sisc/30-1/62039.html

†Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, Denver, CO 80217 (julien.langou@cudenver.edu).

‡MCIS Department, Jacksonville State University, Jacksonville, AL 36265 (zchen@jsu.edu).

§Computer Science Department, University of Tennessee, Knoxville, TN 37996 (bosilca@cs.utk.edu, dongarra@cs.utk.edu).

of the large-scale simulation and modeling community who will use these systems. An excellent summary of current research in fault-tolerant algorithms is provided in [11].

To fulfill these needs, a new message passing library has been created called FT-MPI [7, 8]. FT-MPI enables an implementer to create fault-tolerant algorithms while maximizing 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. For more about how to make an application fault tolerant with the FT-MPI library, we refer the reader to [7].

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

- Phase I : recover a correct computational environment,
- Phase II : recover the static 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 of recovering the static data lost. By static data, we mean, for example, the matrix, the right-hand side, or the preconditioner. This represents data that is computed once in the initialization phase of the application and is unchanged after. Phase III consists of recovering the dynamic data, which is the data that changes during the algorithm.

In this paper we mainly discuss Phase III. Previous solutions to recover the dynamic data were based on checkpointing. Checkpointing is a way to provide fault-tolerant applications that require additional time and memory (or disks, or processors). In section 2, we explain how to implement checkpoints efficiently in some iterative solvers. The checkpoint technique used in the experiments is called diskless checkpointing (see section 2.1) and is particularly suited for parallel distributed computing.

Diskless checkpoints involve global operations with large size data, and their overhead is in direct relation with the number of nodes involved in the application.

In the context of iterative methods, where scalability bottlenecks reside mostly in global operations, such as scalar product computations [16, section 12.2], adding diskless checkpoints just makes the scalability worse. For most computing systems today, applications are unlikely to encounter a failure, and thus many users prefer to take their chances. The mode is to run the application with no checkpointing and, if a failure occurs, restart the application from scratch. The aim of this paper is to find a way to operate Phase III without any significant overhead in the original application.

Our primary concern are 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 loss 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 to 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 what follows, x_j is the local part of the vector x stored on processor j, and $A_{i,j}$ represents the submatrix whose rows are stored on processor i and with column indexes corresponding to the

rows stored on processor j. If processor f fails, then we propose constructing a new approximate solution $x^{(new)}$ via

$$\begin{aligned} x_j^{(new)} &= x_j^{(old)} \quad \text{for } j \neq f, \\ x_f^{(new)} &= A_{f,f}^{-1} \left(b_f - \sum_{j \neq f} A_{f,j} x_j^{(old)} \right), \end{aligned} \tag{1.1}$$

provided that $A_{f,f}$ is of full rank (see section 3.2 if not).

If $x^{(old)}$ is the exact solution of the system, (1.1) will construct $x_f^{(new)} = x_f^{(old)}$; the recovery of x will be exact. In general, the failure happens when $x^{(old)}$ is an approximate solution, in which case $x_f^{(new)}$ is not exactly $x_f^{(old)}$ but should be close. This assertion is justified theoretically in section 3.1. After the recovery step, the iterative method is restarted from $x^{(new)}$. The goal of section 3 is to explain this technique and give some theoretical results about it. This method is sometimes referred to as *lossy algorithm* (as opposed to lossless for the checkpoint method). This is because the dynamic data of the failed processor (e.g, $x_f^{(old)}$) is lost and not recovered, but we recover $x_f^{(new)}$ an approximation of it. In section 3.3, we also explain how (1.1) can be generalized for eigensolvers. In section 4, we present some experimental results that compare the lossy method with some checkpointing approaches.

This study is dedicated to one failure at a time. Theoretically, it is not an issue to generalize the results to multiple failures at a time. Some hints for addressing this problem are given in section 3.2.

2. Checkpoint techniques for parallel iterative methods. Our discussion will focus on the GMRES method [13], but there is no problem generalizing to the other iterative methods.

In this discussion, we describe how we perform a recovery of vector quantities. The scalar quantities (e.g, the number of iterations) are trivial to restore in case of a failure.

2.1. Diskless checkpoint-restart technique. The information of the computing processors is saved in a checkpoint in case of a failure. In order to save the data from any of the processors while maintaining a low overhead in the storage, we are using a checksum approach to checkpointing. If there are n processors for each of which we want to save the vector x_k (for simplicity, we assume that the local sizes of x_k are the same on all the processors), then we store 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 relative errors if the coefficients of x differ by large orders of magnitude.) Our checkpoints are diskless, in the sense that the checkpoint is stored in the memory of a processor and not on a disk. To achieve this, an additional processor is added to the environment. It will be referred to 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 [12], where special attention is given to simultaneous failures.

2.2. 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 algorithms. The goal of this classification is to define which variables

- need to be stored once, when they are referred to as "static" (e.g, the system matrix A, the right-hand side b, or the preconditioner P);
- are changing along the iterations, when they are referred to as "dynamic" (e.g, the approximate solution x);
- should be recomputed after a failure rather than checkpointed (e.g, obtaining the residual via $r = b - Ax$ might be faster than via a checkpoint);
- can be recomputed in case of a rollback but are worth saving their values in order to gain time (e.g, a scalar product is expensive to compute and easy to store and its value is the same on all processors, and thus it makes sense to store those values in an array on all processors; at recovery time, we provide those values to the failed processor, and this avoids recomputing those values during the rollback).

Once this classification of the variables is done, we give two different strategies for checkpointing. The first strategy (chkpt_F) checkpoints the data at each iteration (see section 2.3). It is suited for full GMRES and Arnoldi. The second strategy (chkpt_R) checkpoints the data every k iterations and implies rollback (see section 2.4). It is suited for GMRES with restart and conjugate gradient (CG). In the experimental part (section 4), both categories are represented and compared with the lossy approach. We note that a checkpoint approach can be used to recover the static data (matrix, right-hand side, preconditioner) in Phase II (instead of a disk I/O, for example).

2.3. Checkpointing at each iteration (chkpt_F). In full GMRES and Arnoldi methods, in order to perform iteration k, we need the knowledge of k vectors; thus all the vectors need to be checkpointed. The full GMRES and the Arnoldi method therefore have a very simple checkpoint strategy: each time a vector is computed, it is checkpointed. This strategy is called chkpt_F.

2.4. Checkpointing with rollback (chkpt_R). The common point of CG and GMRES with restart is that, in both methods, the iteration k can be performed using only the knowledge of a constant number of vectors (independent of k).

For example in CG, in order to perform the kth iteration, we need the knowledge of three vectors: $x^{(k-1)}$, $p^{(k-1)}$, and $r^{(k-1)}$. The vectors constructed at iteration $(k-2)$ are no longer needed. (Actually, in practice, the CG implementation simply overrides those vectors by the new ones.) In this case, it makes sense to checkpoint all the vectors involved in a given iteration only occasionally. If a failure happens, then we restart the computation from the last checkpointed version of those vectors. This is called a rollback. Rollback implies that some computations need to be performed again.

It is clear that the checkpointing rate has to be chosen carefully. On the one hand, distant checkpoints require long rollback. On the other hand, close checkpoints imply a large overhead due to a large number of global communications. Gropp and Lusk [9] explain how to choose the checkpointing frequency in an optimal manner. The analysis below is based on their initial work.

To know the optimal rate of checkpoints, we use the following notation:

$T_{\texttt{iter}}$	the time for an iteration (or any unit time step of the code),
$T_{\texttt{chkpt}}$	the time to perform a checkpoint,

$T_{\texttt{recov}}$	the mean time to repair and bring the application back to the last checkpoint,
k	the checkpoint frequency: a checkpoint is performed every k iterations (or units of time),
N	the number of iterations (or units of time) to converge (without failure),
$T_{\texttt{rollback}}(k)$	the mean time to perform the rollback.

Given these definitions, we can write

$$T_{\texttt{total}} = NT_{\texttt{iter}} + T_{\texttt{chkpt}}\frac{N}{k} + \frac{T_{\texttt{total}}}{T_{\texttt{MTTF}}}(T_{\texttt{recov}} + T_{\texttt{rollback}}(k)),$$

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 the encountered failures is the number of failures ($T_{\texttt{total}}/T_{\texttt{MTTF}}$) times the mean time for a recovery ($T_{\texttt{recov}} + T_{\texttt{rollback}}(k)$).

As in [9], we assume that the probability of failure is constant over time, which implies that the distribution of failures is exponential. Taking into account that the distribution of failures is exponential of parameter $T_{\texttt{MTTF}}$, the distribution of failures that have happened between $t = 0$ and $t = kT_{\texttt{iter}}$ 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 it is 0 elsewhere. The mean time of this law, $T_{\texttt{rollback}}(k)$, is given by

$$T_{\texttt{rollback}}(k) = T_{\texttt{MTTF}} - kT_{\texttt{iter}}\frac{e^{-kT_{\texttt{iter}}/T_{\texttt{MTTF}}}}{1 - e^{-kT_{\texttt{iter}}/T_{\texttt{MTTF}}}}.$$

When $kT_{\texttt{iter}} \ll T_{\texttt{MTTF}}$, then a good approximation of $T_{\texttt{rollback}}$ is $kT_{\texttt{iter}}/2$, which means that the failures happen, on average, in the middle of the checkpoint interval. This makes sense since, when $kT_{\texttt{iter}} \ll T_{\texttt{MTTF}}$, the exponential distribution of parameter $T_{\texttt{MTTF}}$ on 0 and $kT_{\texttt{iter}}$ is close to a uniform distribution.

Returning to the expression of the total time we can write

$$T_{\texttt{total}} = \left(NT_{\texttt{iter}} + T_{\texttt{chkpt}}\frac{N}{k}\right)\left(\frac{kT_{\texttt{iter}}}{T_{\texttt{MTTF}}}\frac{e^{-kT_{\texttt{iter}}/T_{\texttt{MTTF}}}}{1 - e^{-kT_{\texttt{iter}}/T_{\texttt{MTTF}}}} - \frac{T_{\texttt{recov}}}{T_{\texttt{MTTF}}}\right)^{-1}. \tag{2.1}$$

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, assuming $kT_{\texttt{iter}} \ll T_{\texttt{MTTF}}$, and get

$$T_{\texttt{total}} = \left(NT_{\texttt{iter}} + T_{\texttt{chkpt}}\frac{N}{k}\right)\left(1 - \frac{kT_{\texttt{iter}}}{2T_{\texttt{MTTF}}} - \frac{T_{\texttt{recov}}}{T_{\texttt{MTTF}}}\right)^{-1}. \tag{2.2}$$

The minimum is obtained for

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

This gives us the optimal time between two checkpoints:

$$kT_{\texttt{iter}} = \sqrt{T_{\texttt{chkpt}}(2T_{\texttt{MTTF}} + T_{\texttt{chkpt}} - 2T_{\texttt{recov}})} - T_{\texttt{chkpt}}. \tag{2.3}$$

With the assumptions $T_{\texttt{MTTF}} \gg T_{\texttt{chkpt}}$ and $T_{\texttt{chkpt}} = T_{\texttt{recov}}$, we recover the formula of Gropp and Lusk [9]:

$$k \cdot T_{\texttt{iter}} \sim \sqrt{2T_{\texttt{MTTF}} \cdot T_{\texttt{chkpt}}}. \tag{2.4}$$

The assumption $T_{\texttt{chkpt}} = T_{\texttt{recov}}$ holds well when the fault-tolerant library is unaware of the application; therefore checkpoints of the whole memory are made at regular intervals, which is the context of [9]. In our experiments (see section 4), there is a significant difference between $T_{\texttt{recov}}$ and $T_{\texttt{chkpt}}$. This is due to the fact that our checkpointing algorithm checkpoints only the dynamic data.

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 $k[m] = 0$ (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, it is more advantageous to checkpoint GMRES with restart as full GMRES (at every iteration) in order to avoid long rollback.

3. The lossy approach. The lossy approach with the block Jacobi step is defined by (1.1). 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 [6], where the authors propose using the block Jacobi method itself as a scalable algorithm to failure. In fact, the block Jacobi method needs only to be performed at the recovery step, but it can be embedded into any iterative solver. This way, one can choose the iterative solver desired, for example, a Krylov method. On a related note, we remind the reader of the work of Jacobi and Gauss at the time when computations were done by hand. Gauss (see [10, p. 321]) states that the method was extremely tolerant to errors.

3.1. Quality of the new approximate solution given by the lossy approach. The lossy approach implies that the method is no longer the same as the method without failure. In this section, we give some hints on the convergence of the lossy method. Surprisingly enough, failures sometimes enhance the convergence.

To quantify the convergence property of the lossy approach, we focus on the size of the residual difference norm between before and after the failure. We also discuss the speed of convergence after the recovery.

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

$$\|x^{(new)} - x^*\| \leq \left(1 + \|A_{f,f}^{-1}\|^2 \sum_{j \neq f} \|A_{f,j}\|^2\right)^{1/2} \|(x^{(old)} - x^*)\|, \tag{3.1}$$

$$\|r^{(new)}\| \leq \left(1 + \|A_{f,f}^{-1}\|^2 \sum_{j \neq f} \|A_{j,f}\|^2\right)^{1/2} \|r^{(old)}\|. \tag{3.2}$$

(A formal proof is omitted in this paper. The results follow easily from the discussion of Saad [15, section 4.2].)

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 $(\|A_{f,f}^{-1}\|^2 \sum_{j \neq f} \|A_{j,f}\|^2)^{1/2}$ is small compared to 1 (or at least of the same order). This assumes that the diagonal blocks are not ill-conditioned and the extradiagonal blocks have small norms relative to the diagonal block norms.

The residual norm of the approximate solution is not the only thing that matters. The iterative solver computes other information that is stored 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 sublinear. Thus, the lossy approach is justified in all the restarted methods (in particular, GMRES with restart) as long as the residual norm difference, (3.2), is not too high.

GMRES with restart has the drawback of stagnating fairly easily on practical examples. If a failure occurs during stagnation, the lossy approach computes a new approximate solution with the same quality in terms of error norm and residual norm, (3.2) and (3.1), but with different spectral properties. In our experiments (see Tables 3 and 4), we often observe that the GMRES with restart algorithm with a failure and a lossy recovery step performs better in terms of the number of iterations than the nonfailed GMRES with restart algorithm. This observation suggests that including block Jacobi steps inside GMRES cycles might cure the stagnation of restarted GMRES.

3.2. 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 noting that, if the preconditioner used is a block Jacobi preconditioner, those factors are available from the recovery step in Phase II, and thus the recovery of x can be done for the price of a preconditioner step and the local contribution of a matrix-vector product.

What about a singular diagonal block $A_{i,i}$? If the matrix A is nonsingular (which is given), we can extract rows from the column block $A_{:,i}$ such that these rows form a nonsingular square block. Thus in theory, a singular diagonal block $A_{i,i}$ is not an issue. In practice, we focused only on matrices with nonsingular (and even well-conditioned) diagonal blocks. 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 methods, 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 solver should converge faster than restarting the method from scratch.

What about multiple failures on a single instance? This is not a theoretical issue. If processors i and j fail at the same time, we have to solve a system of linear equations involving the coefficient matrix

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

Implicit knowledge of x. The lossy recovery requires the approximate solution

x at each step of the iterative method. This assumption is true for most of the iterative methods (stationary iterative methods, CG, Orthomin, GCR, BiCGStab, etc.) 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. The solution $x^{(k)}$ at step k is implicitly known via the formula

$$x^{(k)} = x^{(0)} + (v^{(1)}, \ldots, v^{(k)})y^{(k)}, \tag{3.3}$$

where $v^{(i)}$ represents the Krylov basis generated by GMRES. The quantities $y^{(k)}$ are contained in a small vector that can be computed from the data of any nonfailed processors, and the vectors $x^{(0)}$, $v^{(1)}, \ldots, v^{(k)}$ are classically distributed among the processors. From (3.3), if a failure occurs on the processor f, the local part of $x^{(k)}$ (i.e, $x_i^{(k)}$, $i \neq f$) can be computed on all the nonfailed processors. Thus the lossy approach can also be used without modifying the generic algorithm.

Superlinear convergence. The lossy algorithm performs a restart of the iterative method when a failure occurs. Adding an extra restart is fully justifiable in the context of restarted methods (e.g., GMRES with restart). In the context of nonrestarted methods (e.g., full GMRES or CG), we expect to lose the superlinear convergence after restart. As a rule of thumb, the lossy algorithm will perform well in term of iterations when the convergence of the iterative methods is linear (or sublinear).

3.3. 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 linear solvers to eigenvalue solvers. We use the Arnoldi algorithm (see, e.g, [14, section 7.5]). For the sake of simplicity, we assume in this description that the blocksize of the method is one and that we are looking for the largest eigenvalue of the matrix A. (In the experimental section, section 4, we take more complex cases.) If the processor f fails at iteration n_f, the lossy approach for the Arnoldi method is defined as follows:

1. For each nonfailed processor, compute the largest eigenvalue $\lambda^{(Ritz)}$ and the associated eigenvector $w^{(Ritz)}$ of the n_f-by-n_f Hessenberg matrix.
2. For each nonfailed processor k, compute the 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. Compute the residual: $x = Av^{(Ritz)} - v^{(Ritz)}\lambda^{(Ritz)}$.
5. Solve the residual equation on the failed processor f, $x_f = (A_{f,f} - I\lambda^{(Ritz)})\, x_f$.
6. The new vector x is an approximation of the eigenvector associated with the largest eigenvalue; then the Arnoldi method can be restarted with x as the starting vector.

4. Numerical experiments. Experiments were performed on the `boba` Linux cluster at the University of Tennessee composed of 64 dual Intel Xeon processors at 2.40 GHz with Myrinet interconnect. We used the double-precision floating-point arithmetic. The MPI library used was FT-MPI. Test matrices were from the University of Florida sparse matrix collection [1]. We chose the matrices among the largest unsymmetric matrices available in the collection at the time of publication. It turns out that, for those matrices, GMRES with restart and the block Jacobi preconditioner converge nicely.

The presented results are simulations of our final goals, but we are still far from the targeted thousands of processor experiments. However, we want to make clear that the simulation stops there. The process failures are real. They are simulated

in the experiments by a forced `exit` in the process designed to fail. The software developed could be used on a larger-scale system with real failures.

4.1. Experiment setup. We recall that the recovery is performed in three phases: Phase I: recover the MPI environment, Phase II: recover the static data, and Phase III: recover the variable data. The subject of the paper is neither Phase I nor Phase II; however, we give some clues about our actual implementation choices.

Phase I is based on FT-MPI, and we used the classical approach described in [7].

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 changed the original storage format of the matrices. They are not stored on the Harwell Boeing compressed sparse column format, but rather we preprocessed them to a compressed sparse row format. Doing this, each processor needs access to a contiguous part of the file in the disk. At the first reading of the matrix (initialization of the code), we store 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, and the performance is similar to disk I/O on our small examples. This subject is not discussed any further, but it would become an interesting subject when the number of processors gets larger. If there is a preconditioner (static data), then our choice is to recompute the lost LU factors (no checkpoint).

For the lossy approach, the local solve is done via UMFPACK Version 4.3 [2, 3, 4, 5]. The default parameters are used. Before going through our main results in section 4.2, we finish this section by explaining in detail the scenario for a given matrix. In particular, we will justify technical choices of our implementation.

In the remainder of this section, we will study two diskless checkpointing options. Namely, the first question is whether we shall perform the checksum in floating-point arithmetic or in binary arithmetic; and the second question is whether or not it is advantageous to save scalar products during the algorithm in case of a rollback.

The studied matrix is `cage14`, and 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 that there is a 33rd processor used to store the checkpoint data). 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 $\|b - Ax\|/\|b\| \leq \text{tol}$, where tol$= 10^{-6}$. The method is GMRES(30) without preconditioner. Without failure, this method converges in 13 iterations, and the run takes 15.47s (see Table 1).

The first experiment consists of the same run but with a failure at iteration 10 of processor 0, and the recovery mode is chkpt_R. In this case (see section 2.4), the chkpt_R algorithm performs checkpoints at each restart. Since the failure (iteration 10) is earlier than the first restart (iteration 30), the only checkpoint made is the one from iteration 0. Thus, when the failure occurs at iteration 10, the algorithm has to roll back to the last checkpoint, that is, to roll back to iteration 0. And then it performs the 13 iterations necessary to converge. This explains why it takes 23 iterations for the chkpt_R algorithm to converge (see Table 1). The choice of chkpt_R is not appropriate, and one should certainly have performed checkpoints of the vectors

TABLE 1

Comparison of three variants of the checkpoint fault-tolerant algorithms chkpt_R with GMRES(30): chkpt_R(1) performs checkpoints in double-precision arithmetic and stores scalar products; chkpt_R(2) performs checkpoints in double-precision arithmetic and recomputes scalar products during a rollback; chkpt_R(3) performs checkpoints in binary arithmetic and stores scalar products. Times are given in seconds, and the parameters for the problem are $n = 1,505,785$; $nnz = 27,130,349$; tol=10^{-6}; # $procs = 32$; $n_f = 47,056$; $nnz_f = 414,240$.

cage14			
Recovery	iter_f	# iters	$T_{\texttt{Wall}}$
	no	13	15.47
Chkpt_R(1)	10	23	28.66
Chkpt_R(2)	10	23	28.92
Chkpt_R(3)	10	23	28.80

at each iteration (chkpt_F) in order to have no rollback at the failure (see section 4 to see that chkpt_F is more efficient than chkpt_R in this example). This example is good to stress that the optimization of the number of checkpoints versus the rollback (discussed in section 2.4) is an important issue. In most of the cases, this problem can be anticipated.

In Table 1, we compare three variants of the chkpt_R algorithm. The first variant uses double-precision arithmetic, the second variant uses binary arithmetic, the third uses double-precision arithmetic, and the scalar products are recomputed during rollback. # 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. The first row of Tables 2–4 give the name of the matrix, the order (n), the number of nonzero elements (nnz), the tolerance (tol), and the number of computational processors (# procs). We are considering only one failure, and for the sake of comparison between the methods, it always happens 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 will 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 nnz_i well.

From Table 1, we can conclude the following points:

1. At our problem scale, reusing the scalar products does not seem to have a large effect on the overall time. We expect that this effect will be more important when the size of the problem or the rollback gets larger. In what follows, we do not recompute the scalar products at rollback.
2. Using either binary arithmetic or double-precision arithmetic does not seem to be a big issue. Both arithmetics lead to similar timings. The errors due to the floating-point arithmetic are not affecting the overall algorithm. In what follows, the checkpoints are made using the binary arithmetic.

4.2. Experimental results. Seven matrices are tested, and the results for a given matrix are given in Table 2 (GMRES(30) with block Jacobi preconditioner), Table 3 (GMRES(30) without preconditioner), and Table 4 (Arnoldi method).

The meaning of the tables is the same as described in section 4.1. Regarding the Arnoldi method, defining $x_i^{(k)}$, the approximate solution of unit norm of the kth eigenvector, and $\lambda_i^{(k)}$, the approximate value for the kth eigenvalue, $\lambda_i^{(k)} = (x_i^{(k)})^T A x_i^{(k)}$, and 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.

TABLE 2

Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm. Times are given in seconds.

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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{rollback}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
Lossy	no	353	7.38	none						
Chkpt_R	no	353	7.40	0.02						
Lossy	150	348	7.95	none	−1.04	0.72	0.60	0.04	0.04	0.01
Chkpt_R	150	353	7.96	0.02	0.00	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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{rollback}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
Lossy	no	52	3.23	none						
Chkpt_R	no	52	3.23	0.00						
Lossy	30	51	4.30	none	−0.06	1.08	0.62	0.09	0.32	0.02
Chkpt_R	30	52	4.29	0.00	0.00	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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{rollback}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	$T_{\text{II},P}$	T_{III}
Lossy	no	18	7.98	none						
Chkpt_F	no	18	8.43	0.52						
Chkpt_R	no	18	8.15	0.00						
Lossy	10	18	14.11	none	0.00	5.50	1.05	0.33	3.61	0.35
Chkpt_F	10	18	13.65	0.52	none	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 3

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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{rollback}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
Lossy	no	385	38.89	none					
Chkpt_R	no	385	41.04	1.92					
Lossy	100	372	42.38	none	−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	none	−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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{rollback}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
Lossy	no	13	15.47	none					
Chkpt_F	no	13	16.88	1.50					
Chkpt_R	no	13	15.49	0.02					
Lossy	10	14	21.36	none	1.19	6.35	2.20	1.56	1.64
Chkpt_F	10	13	22.92	1.50	none	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

For the different matrices and the different iterative methods, we test the three recovery modes explained in sections 2.1 and 3: chkpt_R, chkpt_F, or lossy (whenever they apply). For the sake of comparison, we also provide failure-free data (with or without checkpoints).

TABLE 4

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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
Lossy	no	69	2.32	none				
Chkpt_F	no	69	2.51	0.19				
Lossy	35	83	3.02	none	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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
Lossy	no	60	16.35	none				
Chkpt_F	no	60	17.28	0.92				
Lossy	35	77	23.92	none	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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{recov}}$	T_{I}	$T_{\text{II},A,b}$	T_{III}
Lossy	no	120	24.28	none				
Chkpt_F	no	120	24.33	0.10				
Lossy	65	146	36.02	none	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	$T_{\texttt{Wall}}$	$T_{\texttt{totchkpt}}$	$T_{\texttt{recov}}$	T_{I}	T_{II}	T_{III}
Lossy	no	63	44.11	none				
Chkpt_F	no	63	47.97	3.86				
Lossy	30	73	54.89	none	2.72	2.25	0.19	0.48
Chkpt_F	30	63	50.15	3.86	2.84	2.06	0.18	0.38

The iteration where the failure occurs is iter_f. By default, iter_f is set to (roughly) half of the number of iterations for the scheme without failure. To assess the robustness with respect to the location of iter_f, we also present experiments with two failure locations: one at the first third, the second at the second third. The checkpointing choices and timing of faults have been chosen so that there is no rollback for chkpt_R; this setup is made to favor the checkpointing methods in order to test the lossy algorithm against the harshest competition. The LU factors of the block diagonals of the initial matrices used in the block Jacobi preconditioner are computed via UMFPACK Version 4.3 [2, 3, 4, 5].

Then we give the results:

- the number of iterations to converge (# iters),
- the time to solution ($T_{\texttt{Wall}}$),
- the total time for all the checkpoints ($T_{\texttt{totchkpt}}$),
- the time lost in the rollback ($T_{\texttt{rollback}}$: for the chkpt_F method, there is no rollback, and so there is none; for the chkpt_R method, there is a rollback; for the lossy approach, if this time is positive, the method with failure performs more iterations than without, and this quantifies the time spent in those iterations; if this time is negative, 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 enters the recovery routine and the time when the code exits it),
- the time for Phase I of the recovery ($T_{\mathtt{I}}$: this is the time that it takes for the system and the FT-MPI library to provide a new MPI environment; we typically measure it on one of the nonfaulty processors),
- the time for Phase II of the recovery ($T_{\mathtt{II},A,b}$: the time to do the I/O to recover A and b; and, if needed, $T_{\mathtt{II},P}$: the time to compute the preconditioner; we measure them on the restarted processor),
- and, finally, the time for Phase III of the recovery ($T_{\mathtt{III}}$: the time to recover a value on the restarted processor for x; it is measured on the restarted processor).

All those times are given in seconds.

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

$$T_{\mathtt{Wall}} = T_{\mathtt{Wall}}(\text{lossy}) + T_{\mathtt{totchkpt}} + T_{\mathtt{rollback}} + T_{\mathtt{recov}},$$
$$T_{\mathtt{recov}} = T_{\mathtt{I}} + T_{\mathtt{II},A,b} + T_{\mathtt{II},P} + T_{\mathtt{III}}.$$

Although the checkpointing time for the strategy chkpt_R is not significant, it becomes more significant for chkpt_F. For example, for the Arnoldi method and `cage12` (see Table 4), the checkpointing time 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, proportional to the number of processors used: 0.60s for eight processors, 1.10s for 16, and 2.00s for 32. The use of `ftell` and `fseek` in the I/O has eliminated the I/O problem. At this point, recovery of the static data (Phase II) is of the same order of magnitude as Phase I and Phase III. Phase III consists of recovering the dynamic data. It is either the time to do a single checkpoint (chkpt_R) or several checkpoints (chkpt_F) or to solve the local problem (lossy).

In case of a failure, T_I and T_{II} should be the same whether we use a checkpoint mechanism or the lossy variant. The time to recover from a failure differs only from T_{III}. Our results report times for T_I and T_{II} that reflect slight but acceptable differences among the experiments.

Note that if the preconditioner used is block Jacobi (Table 2), then, for the lossy algorithm, the burden of the computation of the factorization of the local matrix is migrated into Phase II ($T_{\mathtt{II},P}$).

Even though those problems are small, it is important to note that both fault-tolerant techniques (checkpoint and lossy) have reached their initial goal. The number of extra iterations (resp., extra time for solution) for these variants with a fault is significantly smaller than the total number of iterations (resp., total time for solution). Therefore our fault-tolerant techniques are much better than restarting from scratch.

Since we lose part of the convergence theory of the initial method, the main concern with the lossy algorithm is losing the convergence. As claimed in section 3, 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 out of 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 tables that the lossy and the checkpoint methods compare fairly in terms of time, and, even when one is better than the other, the results are pretty close.

The main cost of the lossy algorithm during the recovery is to perform the LU factorization of the local submatrix. However, in the case of block Jacobi preconditioned

GMRES, this LU factorization is needed anyhow. Moreover, in the case of the block Jacobi preconditioner, the local diagonal blocks are well-conditioned. These two points make the lossy algorithm very attractive in the block Jacobi preconditioner case.

5. Conclusions. The lossy technique (at least in 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. Despite the theoretical background given in (3.1) and (3.2), 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. From a performance point of view, the checkpointing algorithm performs well, and, for the size of the problem we consider (fewer than 32 processors), by carefully adapting the checkpoint algorithm to the iterative method, the overhead remains 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 early stage of the implementation of the fault tolerance in end-user codes, it is a convincing argument. As a consequence, this method can be plugged as an external library for any existing software without modifications to the code. Another advantage of the lossy algorithm is that, for sparse matrices, Phase III of the recovery involves only a small number of processors.

In this paper, we have focused only on the one failure at a time problem (either for the checkpoint or for the lossy approach). However, our codes are able to deal with any number of failures, provided they occur separately. Generalizing to deal with multiple failures at the same time is theoretically not an issue.

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 [2, 3, 4, 5]. 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 us to adjust its stopping criterion since it makes sense to solve the local system only at the level where the failure has occurred.

We have observed that performing a block Jacobi step between two GMRES cycles often improves the speed of convergence (in our case always).

Acknowledgment. The authors would like to thank the referees for providing them with pertinent comments.

REFERENCES

[1] T. A. Davis, *University of Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/research/sparse/matrices; see also 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.

[2] T. A. Davis, *Algorithm* 832*: UMFPACK, an unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 196–199.

[3] T. A. Davis, *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 165–195.

[4] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 140–158.

[5] T. A. Davis and I. S. Duff, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Trans. Math. Software, 25 (1999), pp. 1–19.

[6] C. Engelmann and G. Al Geist, *Super-scalable algorithms for computing on* 100,000 *processors*, in Proceedings of International Conference on Computational Science (ICCS 2005), Part I, Lecture Notes in Comput. Sci. 3514, Springer, Berlin, 2005, pp. 313–320.

[7] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra, *Extending the MPI specification for process fault tolerance on high performance computing systems*, in Proceedings of the International Supercomputer Conference, 2004.

[8] E. Gabriel, G. E. Fagg, A. Bukovsky, T. Angskun, and J. J. Dongarra, *A fault-tolerant communication library for grid environments*, in Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03), 2003.

[9] W. D. Gropp and E. Lusk, *Fault tolerance in MPI programs*, Internat. J. High Performance Comput. Appl., 18 (2004), pp. 363–372.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.

[11] P. Hough and V. Howle, *Fault-tolerance in large-scale scientific computing*, in Parallel Processing for Scientific Computing, Software Environ. Tools 20, SIAM, Philadelphia, 2006, pp. 233–247.

[12] J. S. Plank, Y. Kim, and J. Dongarra, *Fault tolerant matrix operations for networks of workstations using diskless checkpointing*, J. Parallel Distrib. Comput., 43 (1997), pp. 125–138.

[13] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.

[14] Y. Saad, *Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., SIAM, Philadelphia, 2000, pp. 161–166.

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.

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