J. Parallel Distrib. Comput. 69 (2009) 410–416

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Algorithm-based fault tolerance applied to high performance computing

George Bosilca [a], Rémi Delmas [a], Jack Dongarra [a,*], Julien Langou [b]

[a] Department of Electrical Engineering and Computer Science, University of Tennessee, United States
[b] Department of Mathematical and Statistical Sciences, University of Colorado Denver, United States

ARTICLE INFO

Article history:
Received 20 June 2008
Accepted 11 December 2008
Available online 31 December 2008

Keywords:
Fault tolerance
Linear algebra
High performance computing

ABSTRACT

We present a new approach to fault tolerance for High Performance Computing system. Our approach is based on a careful adaptation of the Algorithm-Based Fault Tolerance technique [K. Huang, J. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Transactions on Computers (Spec. Issue Reliable & Fault-Tolerant Comp.) 33 (1984) 518–528] to the need of parallel distributed computation. We obtain a strongly scalable mechanism for fault tolerance. We can also detect and correct errors (bit-flip) on the fly of a computation. To assess the viability of our approach, we have developed a fault-tolerant matrix–matrix multiplication subroutine and we propose some models to predict its running time. Our parallel fault-tolerant matrix–matrix multiplication scores 1.4 TFLOPS on 484 processors (cluster `jacquard.nersc.gov`) and returns a correct result while one process failure has happened. This represents 65% of the machine peak efficiency and less than 12% overhead with respect to the fastest failure-free implementation. We predict (and have observed) that, as we increase the processor count, the overhead of the fault tolerance drops significantly.

1. Introduction

Much research has been conducted into the checkpointing of message passing applications [5]. The earlier project proposing checkpoint/restart facilities for a parallel application based on MPI was CoCheck [30] and some of the most current and widely used systems are LAM/MPI [6], MPICH-V/V2/V3 [5], CHARM++ [8] and Open MPI [4]. The foundation for all of these projects is that the system does not force the MPI application developers to handle the failures themselves, i.e., the underlying system, be it the Operating System or the MPI library itself, is responsible for failure detection and application recovery. The user of the system is not directly burdened with this at the MPI API layer.

System-level checkpointing in parallel and distributed computing settings has been studied extensively. The issue of coordinating checkpoints to define a consistent recovery line has received the attention of scores of papers, ably summarized in the survey paper of Elnozahy et al. [12]. Nearly all implementations of checkpointing (e.g. [1,7,11,13,19,20,23,25,27,29,30]) are based on globally coordinated checkpoints, stored to a shared stable storage. The reason is that system-level checkpointers are complex pieces of code, and therefore real implementations typically keep the synchronization among checkpoints and processors simple [17]. While there are various techniques to improve the performance of checkpointing (again, see [12] for a summary), it is widely agreed upon that the overhead of writing data to stable storage is the dominant cost.

Diskless Checkpointing has been studied in various guises by a few researchers. As early as 1994, Silva et al. explored the performance gains of storing complete checkpoints in the memories of remote processors [28]. Kim et al. [24] presented a similar idea of diskless checkpointing in 1994 as well. This technique was subsequently revised for SIMD machines by Chiueh [10]. More recently, diskless checkpointing of FFTs has been studied in [15]. Evaluations of diskless and diskless/disk-based hybrid systems have been performed by Vaidya [31]. Lu presents a comparison between diskless checkpointing and disk based checkpointing [21].

While these techniques can be effective in some specific cases, overall, automatic *application-oblivious* checkpointing of message passing applications do suffer from scaling issues and in some cases can incur considerable message passing performance penalties.

A relevant set-up of experimental conditions considers a constant failure rate per processor. Therefore, as the number of processors increases, the overall reliability of the system decreases accordingly. Elnozahy and Plank [14] proved that, in these conditions, checkpointing–restart is not a viable solution. Since the failure rate of the system is increasing with the number of processors, a scalable application requires its recovery time to decrease as the number of processors increases. The checkpoint–restart mechanism does not enjoy this property (at best the cost for recovery is constant).

* Corresponding address: Department of Electrical Engineering and Computer Science, University of Tennessee 1122 Volunteer Blvd., Suite 203, 37996-3450 Knoxville, TN, United States.
E-mail address: dongarra@cs.utk.edu (J. Dongarra).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.12.002

Our contribution with respect to existing HPC fault-tolerant research is to present a methodology to enrich existing linear algebra kernels with fault tolerance capacity at a low computational cost. We believe it is the first time that a technique is devised that enables a fault-tolerant application to be able to reduce the fault tolerance overhead while the number of processors increases and the problem size is kept constant. Moreover not only can our method recover from process failures, it can also detect, locate and correct "bit-flip" errors in the output data. The cause of these errors (communication error, software error, etc.) is not relevant to the overall detection/location/correction process.

Algorithm-based fault tolerance (ABFT), originally developed by Huang and Abraham [16], is a low-cost fault tolerance scheme to detect and correct permanent and transient errors in certain matrix operations on systolic arrays. The key idea of the ABFT technique is to encode the data at a higher level using checksum schemes and redesign algorithms to operate on the encoded data. These techniques and the flurry of papers that augmented them [2,3,22,26] opened the door for jettisoning system-level techniques and instead focused on high-performance fault tolerance of matrix operations.

The present manuscript focuses on exposing a new technique specific to linear algebra based on the ABFT approach from Huang and Abraham [16]. Our contribution with respect to the original work of Huang and Abraham is to extend ABFT to the parallel distributed context. Huang and Abraham were concerned with error detection, location and recovery in linear algebra operation. Once a matrix–matrix multiplication is performed (for example), then ABFT enables to recover from errors. This scenario is not ideal for HPC where we want to be able to recover an errorless environment immediately after a failure. The contribution is therefore to create algorithms for which ABFT can be used on the fly.

This manuscript focuses on obtaining an efficient fault-tolerant *matrix–matrix multiplication subroutine* (PDGEMM). This application does not respond well to memory exclusion techniques [18], therefore standard checkpointing techniques perform poorly. Matrix–matrix multiplication is a kernel of fundamental importance to obtain efficient linear algebra subroutines. Our claim is that we can encapsulate all the fault tolerance needed by the linear algebra subroutines in ScaLAPACK in a fault-tolerant *Basic Linear Algebra Subroutines* (BLAS).

The third contribution of this manuscript is the presentation of model to predict the running time of our routines.

The fourth contribution of this manuscript is a software based on FTMPI and some experimental results. Our parallel fault-tolerant matrix–matrix multiplication scores 1.4 TFLOPS on 484 processors (cluster `jacquard.nersc.gov`) and returns a correct result while one process failure has happened. This represents 65% of the machine peak efficiency and less than 12% overhead with respect to the fastest failure-free implementation.

2. A new approach for HPC fault tolerance

2.1. Additional processors to store redundant information

During a computation, our data is spread across different processes. If one of these processes fails, we need an efficient way to recover the lost part of the data. In that respect, we use additional processes to store redundant information in a checksum format. Checksums represent an efficient and memory-effective way of supporting multiple failures.

If a vector of data x is spread across p processes where x_i is held by process i, then an additional process is added for the storage of y such that $y = a_1x_1 + \cdots + a_px_p$. (For the sake of simplicity, we assume the size of x is constant on all the processes.) In case of a single process failure, using the information in the additional checksum process and the non-failed processes, the data on the failed process can trivially be restored. (Assuming the a_i are not 0.) This fault-tolerant mechanism is classically known as diskless checkpointing and was first introduced by Plank et al. [24]. The name diskless comes from the fact that checksums are stored on additional processes as opposed to being stored on disks.

In order to support f failures, f additional processes are added and f checksums are performed with the following linear relation

$$\begin{cases} y_1 = a_{11}x_1 + \cdots + a_{1p}x_p, \\ \vdots \\ y_f = a_{f1}x_1 + \cdots + a_{fp}x_p. \end{cases}$$

A sufficient condition to recover from any f-failure set is that any f-by-f submatrix of the f-by-p matrix A is non-singular.

Checksums are traditionally performed in Galois Field arithmetic. Another natural choice to encode floating-point numbers is to use the floating-point arithmetic. Galois Field always guarantees bit-by-bit accuracy. Floating-point arithmetic suffers from numerical errors during the encoding and the recovery. However, cancellation errors in the checksum are typically of the same order as the ones arising in the numerical methods, therefore of no concern for the quality of the final solution; additionally, the f-by-f recovery submatrix is statistically guaranteed to be well conditioned if we take the checkpoint matrix A with random values [9].

The ABFT technique presented in this manuscript is based on a floating-point checksum.

2.2. ABFT approaches to fault tolerance in FT-LA

In 1984, Huang and Abraham [16] proposed ABFT (Algorithm-Based Fault Tolerance) to handle errors in numerical computation. This section relies heavily on their idea. To illustrate the ABFT technique, we consider two vectors x and y, spread among p processes, where process i holds x_i and y_i. An additional checksum process has been added to store the checksum $x_c = x_1 + \cdots + x_p$ and the checksum $y_c = y_1 + \cdots + y_p$. The checksums are performed in floating-point arithmetic. Assume now that one wants to realize $z = x + y$. This involves the local computation on process i, $i = 1, \ldots, p$: $z_i = x_i + y_i$ and the update of the checksum z_c. Instead of computing z_c as $z_1 + \cdots + z_p$, as proposed in Section 2.1, ABFT simply performs $z_c = x_c + y_c$. While the traditional checkpoint method would require a global communication among the n processes and an additional computational step, ABFT performs a local operation (no communication involved) on an additional process during the time the active processes perform the same kind of operation. Therefore, to maintain the checksum of z_c consistent with the vector z, the penalty cost with respect to the non-fault-tolerant case is an additional process.

The same idea applies to all linear algebra operations. For example matrix–matrix multiplication, LU factorization, Cholesky factorization or QR factorization (see [2,3,16,22,26]).

As an example, we have studied ABFT in the context of the matrix–matrix multiplication. Assuming that A and B have been checkpointed such that

$$A_F = \begin{pmatrix} A & AC_R \\ C_C^T A & C_C^T AC_R \end{pmatrix} \quad \text{and} \quad B_F = \begin{pmatrix} B & BC_R \\ C_C^T B & C_C^T BC_R \end{pmatrix},$$

where C_C and C_R are the checksum matrices; then performing the matrix–matrix multiplication

$$\begin{pmatrix} A \\ C_C^T A \end{pmatrix} \begin{pmatrix} B & BC_R \end{pmatrix} = \begin{pmatrix} AB & ABC_R \\ C_C^T AB & C_C^T ABC_R \end{pmatrix} = (AB)_F, \tag{1}$$

we obtain the result $(AB)_F$ which is *consistent*, that is to say, it verifies the same checkpoint relation as A_F or B_F. This consistency

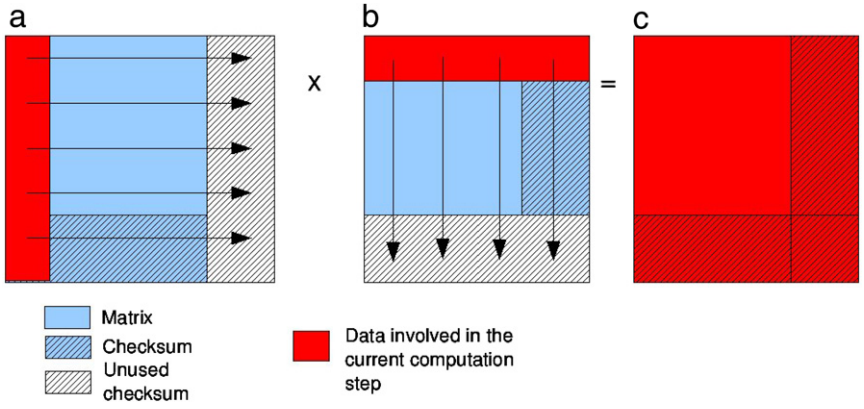

Fig. 1. Algorithm-Based Fault-Tolerant DGEMM.

enables us to detect, localize and correct errors. In the context of erasure, one has to be more careful since we want not only the input and output matrices to be consistent, but we also want any intermediate quantity to be consistent. Using the outer product version of the matrix–matrix product (`for k=1:n`, $C_k = C_k + A_{:,k}B_{k,:}$; `end`), as C_k is updated along the loop in k, the intermediate C_k matrices are maintained consistently. Therefore a failure at any time during the algorithm can be recovered (see Fig. 1).

If p^2 processes are performing a matrix–matrix multiplication, maintaining the checksum consistent requires $(2p-1)$ extra processes. The cost with respect to a non-failure-tolerant application occurs when the information of the matrix A is broadcasted along the process rows. In this case, one extra processor needs to receive the data. And vice versa for the matrix B: when the information of the matrix B is broadcasted along the process columns, one extra processor needs to receive the data. These are the only extra costs for being fault tolerant. From this analysis, we deduce that the main cost to enable ABFT in a matrix–matrix multiplication is to dedicate $(2p-1)$ processes over p^2 for fault tolerance sake. Therefore, the more the processors, the more advantageous the ABFT scheme!

3. Model

3.1. PBLAS PDGEMM

It is behind the scope of this manuscript to describe every aspect of parallel matrix–matrix multiplication, and we refer the reader to [32] for more information.

During a matrix–matrix multiplication, each block of A in the current column (of size $mloc * nb$) is broadcasted along the row to all other processes. The same happens to B: each block in the current row (of size $nb * nloc$) is broadcasted along the columns. After these two broadcasts, each process on the grid computes a local DGEMM. This step is repeated until the whole A and B matrices have been broadcasted to every process.

For the sake of simplicity, we consider that the matrices are square, of size n-by-n, and distributed over a square grid of $\sqrt{p}$-by-$\sqrt{p}$ processes.

In SUMMA, we implement the broadcast as passing a message around the logical ring that forms the row or column. In that case the time complexity becomes

$$(\sqrt{p}-1)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right)+(\sqrt{p}-1)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right) \tag{2}$$

$$+\,n\left(\frac{2n^2}{p}\gamma+\alpha+\frac{n}{\sqrt{p}}\beta+\alpha+\frac{n}{\sqrt{p}}\beta\right) \tag{3}$$

$$+(\sqrt{p}-2)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right)+(\sqrt{p}-2)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right) \tag{4}$$

$$+\frac{2n^3}{p}\gamma \tag{5}$$

$$=\frac{2n^2(n+1)}{p}\gamma+2(n+2\sqrt{p}-3)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right) \tag{6}$$

where β is the inverse of the bandwidth, α is the latency and γ is the inverse of the flop rate.

Eq. (2) is the time required to fill the pipe (that is the time for the messages originating from A and B to reach the last process in the row and in the column). The next term (3) is the time to perform the sequential matrix–matrix multiplication and passing the messages. Then, contribution (4) is the time for the final messages to reach the end of the pipe. The last term is the time for the final update at the node at the end of the pipe.

This complexity is then approximately

$$\frac{2n^3}{p}\gamma+2(n+2\sqrt{p}-3)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right) \tag{7}$$

and the estimated efficiency is

$$E(n,p)=\frac{1}{1+O\left(\frac{p}{n^2}\right)+O\left(\frac{\sqrt{p}}{n}\right)}. \tag{8}$$

We can see that the method is *weakly* scalable: if we increase the number of processors while maintaining constant the memory use per node (thus having $\frac{p}{n^2}$ constant), this algorithm maintains its efficiency constant.

In the remainder, we neglect the latency term (α) in the communication cost.

3.2. ABFT PDGEMM (0 failure)

In this section, we derive a model for ABFT PDGEMM. If we perform a traditional matrix–matrix multiplication, the result will be a checkpointed matrix (see Eq. (1)). In the outer-product variant of the matrix–matrix multiplication algorithm, rank-nb updates are applied to the global matrix C. Therefore, the checksum stays consistent throughout the execution of the algorithm provided that all the processes go at the same speed.

The last row of A (checksum) and the last column of B (checksum) are sent exactly in the same way as the rest of the data (see Fig. 1).

When no failure occurs, the overhead of the fault tolerance are:

- The initial checksum. However, if we call ABFT BLAS functions the ones after the other, we do not have to recompute the checksum between each call. As a consequence, we do not consider the cost of the initial checksum.

Table 1
Performance (GFLOPS/s/proc) of PBLAS PDGEMM, ABFT BLAS PDGEMM (0 failure), and ABFT BLAS PDGEMM (1 failure). The number without parenthesis is the experimental result, while the number in between parenthesis corresponds to the model value. This is a weak scalability experiment with $nloc = 3000$. (See also Fig. 5.)

	Performance per processor (GFLOPS/s/proc)					
	64	81	100	121	256	484
PBLAS PDGEMM	3.14 (3.09)	3.16 (3.09)	3.14 (3.10)	3.10 (3.10)	3.12 (3.12)	3.13 (3.13)
ABFTBLAS PDGEMM (0 failure)	2.43 (2.49)	2.51 (2.55)	2.56 (2.60)	2.62 (2.65)	2.74 (2.79)	2.86 (2.88)
ABFTBLAS PDGEMM (1 failure)	2.33 (2.40)	2.40 (2.46)	2.47 (2.52)	2.52 (2.53)	2.58 (2.63)	2.73 (2.74)
	Cumul performance (GFLOPS/sec)					
	64	81	100	121	256	484
PBLAS PDGEMM	201 (198)	256 (251)	314 (310)	375 (376)	799 (798)	1515 (1513)
ABFTBLAS PDGEMM (0 failure)	156 (159)	203 (207)	256 (260)	317 (320)	701 (714)	1384 (1395)
ABFTBLAS PDGEMM (1 failure)	149 (154)	194 (200)	247 (252)	305 (306)	660 (672)	1321 (1327)

Table 2
Overhead of the fault tolerance with respect to the non-failure-resilient application PBLAS PDGEMM. This is a weak scalability experiment with $nloc = 3000$. (See also Fig. 6.)

	64	81	100	121	256	484
PBLAS PDGEMM	100.0	100.0	100.0	100.0	100.0	100.0
ABFTBLAS PDGEMM (0 failure)	129.2	125.9	122.7	118.3	113.9	109.4
ABFTBLAS PDGEMM (1 failure)	134.8	131.7	127.1	123.0	120.9	114.7

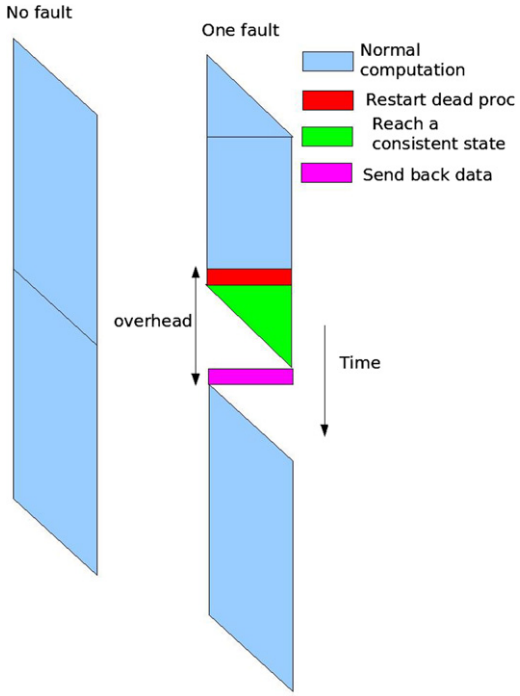

Fig. 2. Overhead between no fault and one fault in ABFT SUMMA.

- The computation of an $(n + nloc)$-by-n-by-$(n + nloc)$ matrix–matrix multiplication, instead of n-by-n-by-n for a non-fault-tolerant code.
- The broadcast which needs to be performed on $q + 1$ processes along the rows and $p + 1$ processes along the columns, instead of respectively p and q for a non-fault-tolerant matrix–matrix multiplication.

Using Eq. (6), we can then write the complexity for ABFT PDGEMM with no fault:

$$\frac{2(n+nloc)^2 n}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\frac{n+nloc}{\sqrt{p}}\beta\right). \quad (9)$$

The modification in the first term of the sum comes from the fact that the matrix–matrix multiplication now involves n-by-$(n+nloc)$ matrices on the same number of processors. The modification in the second term shows that the pipeline is now longer than in regular SUMMA.

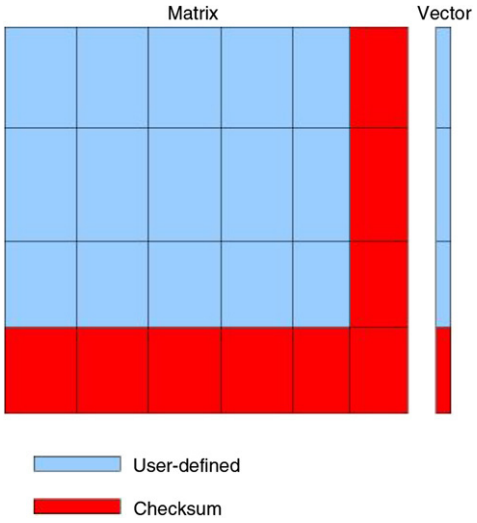

Fig. 3. Data distribution in the ABFT BLAS framework for vectors and matrices over a 4-by-6 process grid.

3.3. ABFT PDGEMM (1 failure)

When one fault happens, the timeline is illustrated in Fig. 2 and explained below.

- $T_{\text{detection}}$: All non-failed processes must be notified when a failure happens. Since processes are only notified when attempting an MPI communication, odds are that at least one process will be doing a full local DGEMM before being notified. So, basically, this overhead is more or less equal to the time of one local DGEMM.
- T_{restart}: FT-MPI must spawn a new process to replace the failed one. This is a blocking operation and the time for this step depends solely on the total number of processes involved in the computation.
- T_{pushdata}: The non-failed processes must reach the same consistent state. The time overhead consists in filling and emptying the pipe once.
- T_{checksum}: The last step in the recovery is to reconstruct the lost data of the failed process. This cost is the cost of an MPI_Reduce.

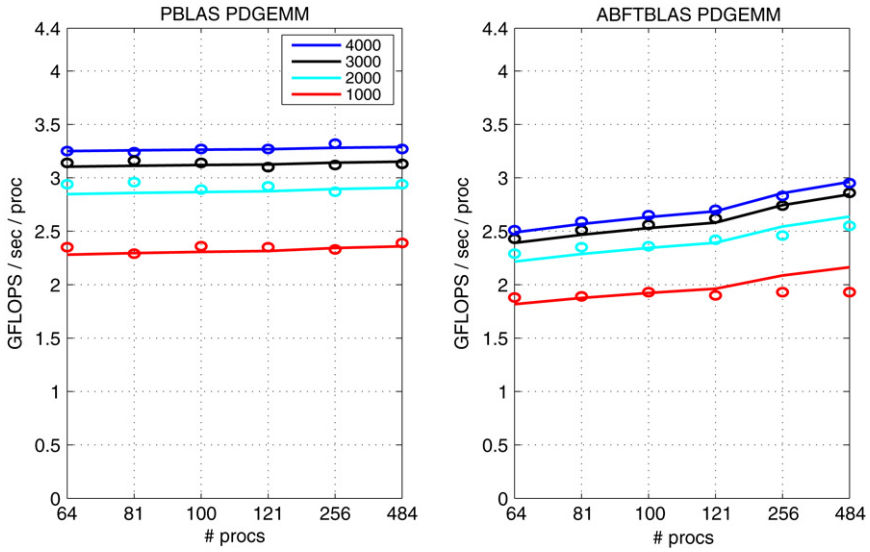

Fig. 4. Performance (GFLOPS/s/proc) of PBLAS PDGEMM (left) and ABFT BLAS PDGEMM with 0 failure (right). The solid lines represent model while the circles represent experimental points.

Fig. 5. Performance (GFLOPS/s/proc) of PBLAS PDGEMM, ABFT BLAS PDGEMM (0 failure), and ABFT BLAS PDGEMM (1 failure). The solid lines represent model while the circles represent experimental points. This is a weak scalability experiment with $nloc = 3000$. (See also Table 1.)

4. Experimental results

4.1. ABFT BLAS framework

A prototype framework ABFT BLAS has been designed and implemented. The ABFT BLAS application relies on the FT-MPI library for handling process failures at the middleware level. The ABFT BLAS responsibilities is to (efficiently) recover the data lost during a crash.

Vectors over processes are created and then destroyed. Several operations over the vectors are possible: scalar product, norm, vector addition, etc. Vectors are registered to the fault-tolerant context. When a failure occurs, all data registered in the fault-tolerant context is recovered and the application is continued. The default encoding mode is diskless checkpointing with floating-point arithmetic but an option is to perform Galois Field encoding (although this rules out ABFT).

The same functionality have been implemented for dense matrices. Dense matrices are spread on the processor in the 2D-block cyclic format, and checksums can be performed by row, by columns or both (see Fig. 3).

Fig. 6. Overhead of the fault tolerance with respect to the non-failure-resilient application PBLAS PDGEMM. The plain curves correspond to model while the circles correspond to experimental data. This is a weak scalability experiment with $nloc = 3000$. (See also Table 2.)

Regarding matrix and vector operations, available functionality are matrix–vector products and several implementations of matrix–matrix multiplications. One of the main features of the ABFT BLAS library is that the user is able to stack fault-tolerant parallel routines the ones on top of the others. The code looks like a sequential code but the resulting application is parallel and fault tolerant.

4.2. Experimental set-up

All runs where done on the `jacquard.nersc.gov` cluster, from the National Energy Research Scientific Computing Center (NERSC). This cluster is a 512-CPU Opteron cluster running a Linux operating system. Each processor runs at a clock speed of 2.2 GHz and has a theoretical peak performance of 4.4 GFLOPS/s. Measured peak (on a 3000-by-3000-by-3000 DGEMM run) is 4.03 GFLOPS/s. The nodes are interconnected with a high-speed InfiniBand network. The latency is 4.5 μs, while the bandwidth is 620 MB/s.

Below are more details on the experimental set-up:

- We have systematically used square processor grids.