# Exploiting Redundant Computation
# in Communication-Avoiding Algorithms
# for Algorithm-Based Fault Tolerance

Camille Coti

*LIPN, CNRS, UMR 7030*
*Université Paris 13, Sorbonne Paris Cité*
*F-93430, Villetaneuse, France*
*camille.coti@univ-paris13.fr*

*Abstract*—**Communication-avoiding algorithms allow redundant computations to minimize the number of inter-process communications. In this paper, we propose to exploit this redundancy for fault-tolerance purpose. We illustrate this idea with QR factorization of tall and skinny matrices. We give a theoretical evaluation of the number of failures our algorithm can tolerate under different semantics and an experimental evaluation of the performance overhead.**

## I. Introduction

Faut tolerance for high performance distributed applications can be achieved at system-level or application-level. System-level fault tolerance is transparent for the application and requires a specific middleware that can restart the failed processes and ensure coherent state of the application [5], [4].

Application-level fault tolerance requires the application itself to handle the failures and adapt to them. Of course, it implies that the middleware that supports the distributed execution must be robust enough to survive the failures and provide the application with primitives to handle them [11]. Moreover, it requires that the application uses fault-tolerant algorithms that can deal with process failures [3].

Recent efforts in the MPI-3 standardization process [13] defined an interface for a mechanism called *User-Level Failure Mitigation* (ULFM) [2] and *Run-Through Stabilization* [16].

This paper deals with the QR factorization of tall and skinny matrices. After a quick overview of techniques for fault tolerance (section II), we provide three fault-tolerant algorithms in the context of ULFM in section III. We give the robustness of each algorithm, the semantics of the fault tolerance and we detail the behavior during failure-free execution and upon failures. Then we present an experimental evaluation of the overhead of the fault tolerance mechanism in section IV.

## II. Algorithm-Based Fault Tolerance

FT-MPI [11], [12] defined four error-handling semantics that can be defined on a communicator. *SHRINK* consists in reducing the size of the communicator in order to leave no hole in it after a process of this communicator died. As a consequence, if one process $p$ which is part of a communicator of size $N$ dies, after the failure the communicator has $N - 1$ processes numbered in $[0, N - 2]$. On the opposite, *BLANK* leaves a hole in the communicator: the rank of the dead process is considered as invalid (communications return that the destination rank is invalid), and surviving processes keep their original ranks in $[0, N - 1]$. While these two semantics survive failures with a reduced number of processes, *REBUILD* spawns a new process to replace the dead one, giving it the place of the dead process in the communicators it was part of, including giving it the rank of the dead process. Last, the *ABORT* semantics corresponds to the usual behavior of non-fault-tolerant applications: the surviving processes are terminated and the application exits.

Using the first three semantics, programmers can integrate failure-recovery strategies directly as part of the algorithm that performs the computation. For instance, diskless checkpointing [18] uses the memory of other processes to save the state of each process. Arithmetic on the state of the processes can be used to store the checksum of a set of processes [7]. When a process fails, its state can be recovered from the checkpoint and the states of the surviving processes. This approach is particularly interesting for iterative processes. Some matrix operations exhibit some properties on this checkpoint, such as *checkpoint invariant* for LU factorization [10].

A proposal for *run-through stabilization* introduced new constructs to handle failures at communicator-level [16]. Other mechanisms, at process-level, have been integrated as a proposal in the MPI 3.1 standard draft [14, ch 15]. It is called *user-level failure mitigation* [2]. Failures are detected when an operation involving a failed process fails and returns an error. As a consequence, operations that do not involve any failed process can proceed unknowingly.

## III. Fault-Tolerant, Communication-Avoiding Algorithms

Communication-avoiding algorithms were introduced on [8]. The idea is to minimize the number of inter-process communications, should it involve additional computations. These algorithms perform well on current architectures, ranging from multicore architectures [9] to aggregations of clusters [1], because of the speed difference between computations and data movements.

As seen in the examples cited in section II, tolerating failures requires some form of *redundancy*, such as checkpoints or checksums stored in additional processes [3].

In this paper, we propose to exploit the redundant computations made by communication-avoiding algorithms for fault-tolerance purpose. In this section we illustrate this idea with a communication-avoiding algorithm for tall and skinny matrices (TSQR). This algorithm can be used to compute the QR factorization of matrices with a lot of rows and few columns, or as a panel factorization for QR factorization [15].

### A. Computing the R with TSQR

The TSQR relies on successive steps that consist of local QR factorizations, involving *no inter-process communications*, and one inter-process communication. Initially, the matrix is decomposed in submatrices, each process holding one submatrix. On the first step, each process performs a QR factorization on its local submatrix. Then odd-numbered processes send the resulting $\widetilde{R}$ to the previous even-numbered process: rank 1 sends to rank 0, rank 3 sends to rank 2.... The algorithm itself is given by Algorithm 1.

Even-numbered processes concatenate the two $\widetilde{R}$ matrices by creating a new matrix whose upper half is the $\widetilde{R}$ it has computed and whose bottom half is the $\widetilde{R}$ it has received. Then the odd-numbered process is done with its participation to the computation of the $R$. If the $Q$ matrix is computed, it will work again when the moment comes, after the computation of the $R$ is done.

Even-numbered processes perform a local QR factorization of the resulting matrix, and produce another $\widetilde{R}$. A similar communication and concatenation step is performed between processes of rank $r \pm 2^{step}$, if $r$ denotes the rank of each process. An illustration of this communication, recombination and local computation process on four processes is depicted by Figure 1.

At each step, half of the participating processes send their $\widetilde{R}$ and are done. The other half receive an $\widetilde{R}$, concatenate it with their own $\widetilde{R}$ and perform a local QR factorization. This communication-computation progression forms a binary reduction tree [17].

We can see on this example that once it has sent its $\widetilde{R}$, each process becomes idle. Eventually, process 0 is the only working process that remains. Half of the processes are idle after the first step, one quarter are idle after the second step, and so on until only one process is working at the end.

### B. Redundant TSQR

We have seen in section III-A that 1) only one process eventually gets the resulting $R$ and 2) at each step, half of the working processes get idle. The idea behind *Redundant TSQR* is to use these spare processes to produce copies of the intermediate $\widetilde{R}$ factors, in order to tolerate process failures during the intermediate steps.

*1) Semantics:* With *Redundant TSQR*, at the end of the computation, all the processes get the final $R$ matrix. If some processes crash during the computation but enough processes survive (see section III-B3), the surviving processes have the final $R$ factor.
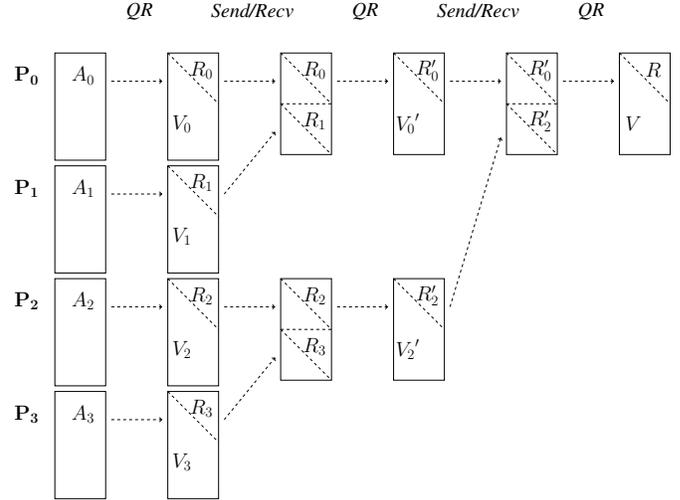


FIG. 1: *Computing the R of a matrix using a TSQR factorization on 4 processes.*

---

**Algorithm 1:** TSQR

**Data**: Submatrix A
1  Q, R = QR( A);
2  step = 0 ;
3  **while** *!* done() **do**
4    **if** isSender( *step*) **then**
       /* I am a sender                */
5      b = myBuddy( *step*) ;
6      send( *R, b*) ;
7      **return**;
8    **else**
       /* I am a receiver              */
9      b = myBuddy( *step*) ;
10     recv( *R', b*) ;
11     A = concatenate( *R, R'*);
12     Q, R = QR( A);
13    step++ ;
     /* The root of the tree reaches this
        point and owns the final $R$    */
14 **return** R;

---

*2) Algorithm:* The basic idea is that when two processes communicate with each other, instead of having one sender and one receiver that assembles the two $\widetilde{R}$ matrices, the processes *exchange* their matrices. Both of them assemble the two matrices and both of them proceed with the local QR factorization. This algorithm is given by Algorithm 2.

This algorithm is represented on four processes in Figure 2. Plain lines represent the regular TSQR pattern. During the first communication stage, the redundancies are represented by dashed lines: $P_1$ and $P_3$ exchange data with $P_0$ and $P_2$ respectively, and therefore obtain the same intermediate matrices. Then same data exchange is performed during the following step, resulting in a first level of redundancy (obtained from the
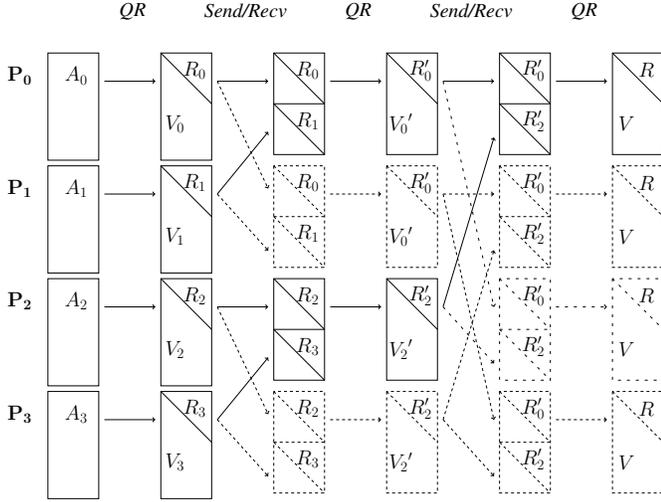
FIG. 2: *Computing the R of a matrix using a TSQR factorization on 4 processes with redundant $\widetilde{R}$ factors.*



FIG. 3: *Redundant TSQR on 4 processes with one process failure.*

$P_0 \leftrightarrow P_2$ exchange), represented by loosely dashed lines, and a secondary level of redundancy (obtained from the $P_1 \leftrightarrow P_3$ exchange), represented by dashed lines.

---

**Algorithm 2:** Redundant TSQR

**Data**: Submatrix A

1   Q, R = QR( $A$ );

2   step = 0 ;

3   **while** *!* done() **do**

4      b = myBuddy( *step*) ;

5      f = sendrecv( *R, R', b* ) ;

6      **if** $FAIL == f$ **then**

7         **return**;

8      A = concatenate( *R, R'*);

9      Q, R = QR( *A*);

10      step++ ;

    /* All the surviving processes reach this point and own the final $R$   */

11   **return** R;

---

*3) Robustness:* We can see that at each step, the data exchange creates one extra copy of each intermediate matrix. As a consequence, the redundancy rate doubles at each step of the algorithm. Therefore, if $s$ denotes the step number, the number of existing copies in the system is $2^s$. Hence, this algorithm can tolerate $2^s - 1$ process failures.

We can see that the number of failures that this algorithm can tolerate increases as the computation progresses. This fact is a direct consequence from the fact that the number of redundant copies of the data is multiplied by 2 at each step. For instance, the computation can proceed if no more than 1 process have failed by the end of step 1, no more than 3 processes have failed by the end of step 2, etc. In the meantime, the number of failures in the system increase with time: the longer a computation lasts, the more processes will
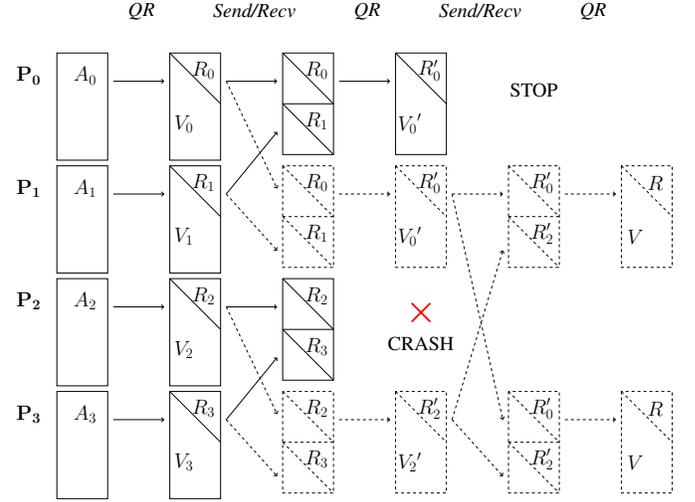
fail [19]. Therefore, the robustness of this algorithm increases with time, which is consistent with the need for robustness.

*4) Behavior upon failures:* When a process fails, the other processes proceed with the execution. Processes that require data from the failed process end their execution, and those that require data from ended processes end theirs as well (see line 7 of Algorithm 2).

For instance, Figure 3 represents the execution of Redundant TSQR on four processes. Process $P_2$ crashes at the end of the first step. The data it contained is also held by process $P_3$, therefore the execution can proceed. However, process $P_0$ needs data prom the failed process at the following step. Therefore, process $P_0$ ends its execution. As a consequence, $P_0$ ends its execution. At the end of the computation, the final result $R$ has been computed by processes $P_1$ and $P_3$ and therefore, the final result is available in spite of the failure.

### C. Replace TSQR

*1) Semantics:* The semantics of *Replace TSQR*, is similar to the one with *Redundant TSQR*: at the end of the computation, all the processes get the final $R$ matrix. If some processes crash during the computation but enough processes survive (see section III-C3), the surviving processes have the final $R$ factor.

*2) Algorithm:* The fault-free execution of the *Replace TSQR* algorithm is exactly the same as with *Redundant TSQR* (see section III-B2). The data held by processes along the reduction tree of the initial TSQR algorithm is replicated on spare processes that would normally stop their execution.

The difference comes when a failure occurs. In this case, the process that needs to communicate with another process gets an error when it tries to communicate with it. Then, it finds a *replica* of the process it is trying to communicate with (line 6 of Algorithm 3) and exchanges its matrix with it. If no replica can be found alive, it means that too many processes have failed and no extra copy of this submatrix exist. Then the process exits. The algorithm is described by Algorithm 3.

---

**Algorithm 3:** Replace TSQR

**Data**: Submatrix A

```
1  Q, R = QR( A );
2  step = 0; while ! done() do
3  │   b = myBuddy( step );
4  │   f = sendrecv( R, R', b );
5  │   while FAIL == f do
6  │   │   b = findReplica( b );
7  │   │   if None == b then
8  │   │   │   return;
9  │   │   f = sendrecv( R, R', b );
10 │   A = concatenate( R, R');
11 │   Q, R = QR( A );
12 │   step++ ;
   /* All the surviving processes reach
      this point and own the final R    */
13 return R;
```
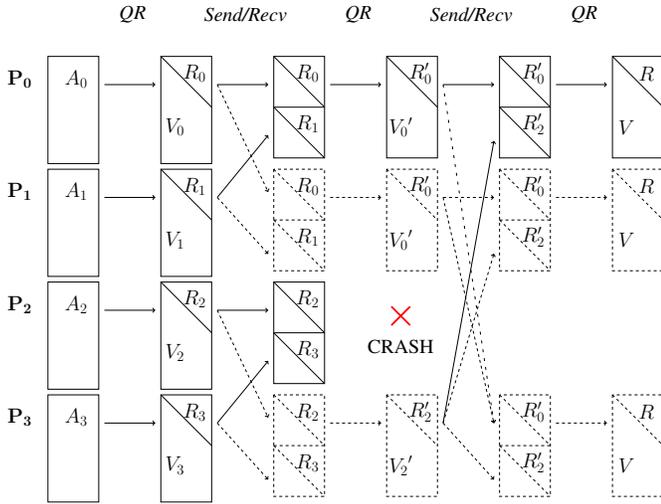
---



FIG. 4: *Replace TSQR on 4 processes with one process failure.*

*3) Robustness:* We have seen in section III-C2 that this algorithm can keen progressing as long as there exists at least one copy of each submatrix. We have seen in section III-B3 that at each step $s$, the number of existing copies in the system is $2^s$. Hence, this algorithm can tolerate $2^s - 1$ process failures, just like the *Redundant TSQR* algorithm (see section III-B3).

The difference between the *Redundant TSQR* and the *Replace TSQR* is that with the former, the processes that need to communicate with a failed process exit, whereas with the latter, they try to find a replica. Therefore, if the root of the tree does not die, it holds the final result $R$ at the end of the computation.

*4) Behavior upon failures:* If a process fails, the processes that try to communicate with it fail to do so and try to find a replica to communicate with.

For instance, Figure 4 represents the execution of Redundant TSQR on four processes. Process $P_2$ crashes at the end of

the first step. The data it contained is also held by process $P_3$, therefore the execution can proceed. However, process $P_0$ needs data prom the failed process at the following step. Therefore, process $P_0$ ends its execution. As a consequence, $P_0$ fails to communicate with $P_0$ and finds out that $P_3$ holds the same data as $P_2$. Then $P_0$ exchanges data with $P_3$.

### D. Self-Healing TSQR

The previous algorithms described here, *Redundant TSQR* (section III-B) and *Replace TSQR* (section III-C) proceed with the execution without the dead processes. Here we are describing an algorithm that replaces the dead process with a new one.

*1) Semantics:* With *Self-Healing TSQR*, at the end of the computation, all the processes get the final $R$ matrix. If some processes crash during the computation but enough processes survive at each step (see section III-D3), the final number of processes is the same as the initial number and all the processes have the final $R$ factor.

*2) Algorithm:* This algorithm follows the same basic idea as *Redundant TSQR* (see section III-B) in a sense that at each step of the computation, all the processes send or receive their $\widetilde{R}$ matrices and compute the $R$ of the resulting matrix. As a consequence, the data required by the computation (the intermediate submatrices represented in Figure 1) are *replicated*. This part is described by Algorithm 6 with the initialization described by Algorithm 4.

In this algorithm, the failed processes are replaced by newly spawned ones. We have seen that the data contained by the failed process has been replicated by the redundant computations. As a consequence, a failed process can be recovered completely and a newly spawned process can replace it: see Algorithm 5.

The fault-free execution of this algorithm is similar with the execution represented by Figure 2.

---

**Algorithm 4:** Self-Healing TSQR: initialization

**Data**: Submatrix A

```
1  Q, R = QR( A );
2  step = 0 ;
3  R = QR( R, step );
4  return R;
```

---

*3) Robustness:* We have seen in III-D2 and III-B3 that at each step $s$, the data necessary for each process from the original algorithm is replicated $2^s$ times on other processes. As a consequence, this algorithm can tolerate $2^s - 1$ process failures *at each step $s$*.

Similarly with *Redundant TSQR*, the robustness of the algorithm increases as the need for robustness increases (see section III-B3). The big difference with Redundant TSQR in terms of robustness is that Self-Healing TSQR replaces the failed processes. Therefore, this redundancy rate gives the number of failed processes that can be accepted *at each step*. For instance, 1 process can fail at step 1 ; it will be respawned and 3 additional processes can fail at step 2. As a

**Algorithm 5:** Self-Healing TSQR: process restart

```
  /* Gets my data from a process that
     holds the same as me.            */
1 t = mytwin() ;
2 R, step = recv( t ) ;
  /* Proceed with the computation     */
3 R = QR( R, step) ;
  /* At the end of the computation, this
     process holds the final R.       */
4 return R;
```

**Algorithm 6:** Self-Healing TSQR: computation

```
1 Function shtsqr( A, step ):
2     Q, R = QR( A );
3     while ! done() do
4         b = myBuddy( step ) ;
5         f = sendrecv( R, R', b ) ;
6         if FAIL == f then
7             spawnNew( b ) ;
8         A = concatenate( R, R');
9         Q, R = QR( A );
10        step++ ;
          /* All the processes reach this
             point and own the final R */
11    return R;
```

consequence, the total number of failures this algorithm can tolerate is $\sum_{k=1}^{p} 2^k$. Besides, at each step $s$ it can tolerate $2^s - 1$ process failures.

*4) Behavior upon failures:* When a process fails, the process that was supposed to communicate with it detects the failure and spawns a new process. The new process obtains the redundant data from one of the processes that hold the same data as the failed process. Then the computation continues normally.

## IV. PERFORMANCE EVALUATION

We evaluated the overhead of the replication during fault-free execution. The overhead of the recovery process of the self-healing algorithm (see section III-D) depends on many factors such as failure detection and the time to spawn a new process; it is beyond the scope of this paper.

We used the Griffon cluster of Grid'5000 [6], which is made of 92 nodes, each of which features two quad-core Intel Xeon L5420 running at 2.5 GHz, 16 GiB of RAM and both GigaEthernet and 20G InfiniBand network interconnection networks. The nodes were running a 64-bit Linux 3.2 kernel. All the code was compiled using the GNU gforfran and gcc 4.7.2 compilers with -O3 optimization flag. The computation kernels we used were provided by LAPACK 3.5.0 and OpenBLAS downloaded from its GitHub repository. The MPI library we used was the fault tolerant implementation OpenMPI 1.7ft_b3.
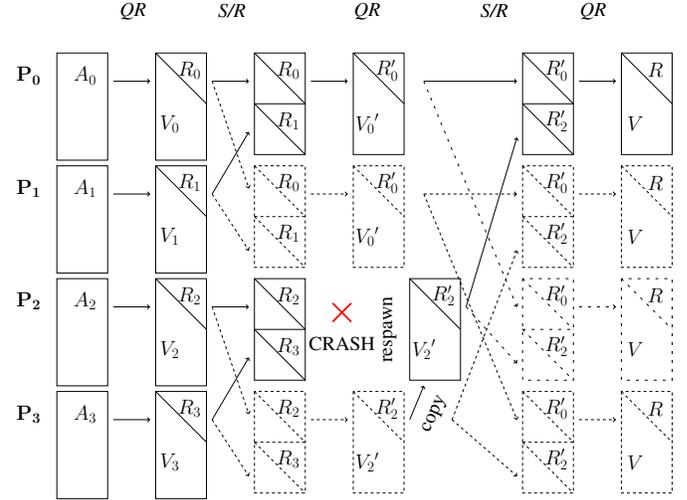


FIG. 5: *Self-Healing TSQR on 4 processes with one process failure.*

Every experiment was run 10 times and the plots present the average of these measurements. The plots present the number of megaflops per second, which is the number of floating point operations (computed using the $mn^2$ formula used for the QR factorization) divided by the computation time (obtained using `MPI_Wtime()`).

The performance obtained by the fault-tolerant and the non fault-tolerant for various matrix sizes and number of processes is given in Figure 6. The difference between the two curves give the overhead of the extra copy.

As expected, we can see that the fault-tolerance version has a relatively small overhead, possibly due to the extra memory movements (to create the intermediate matrices) and network communications: at each step, processes performs two communications (send+recv) instead of only one (send or recv).

Moreover, since, if $N$ denotes the number of columns of the matrix, each communication and each copy of an $\tilde{R}$ matrix has a size of $N^2$. Therefore, this overhead is more important if the matrix is wider.

## V. CONCLUSION AND PERSPECTIVE

In this paper we have presented three fault-tolerant algorithms for the QR factorization of tall-and-skinny matrices. These algorithms take advantage of properties from communication-avoiding algorithms. We have evaluated their resilience properties from a theoretical point of view, and we have evaluated their performance and seen that in practice, they have little overhead on the performance of the computation.

## REFERENCES

[1] Emmanuel Agullo, Camille Coti, Jack Dongarra, Thomas Herault, and Julien Langou. Qr factorization of tall and skinny matrices in a grid computing environment. In *24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*, Atlanta, Ga, April 2010.

[2] Wesley Bland, Aurelien Bouteiller, Thomas Hérault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.
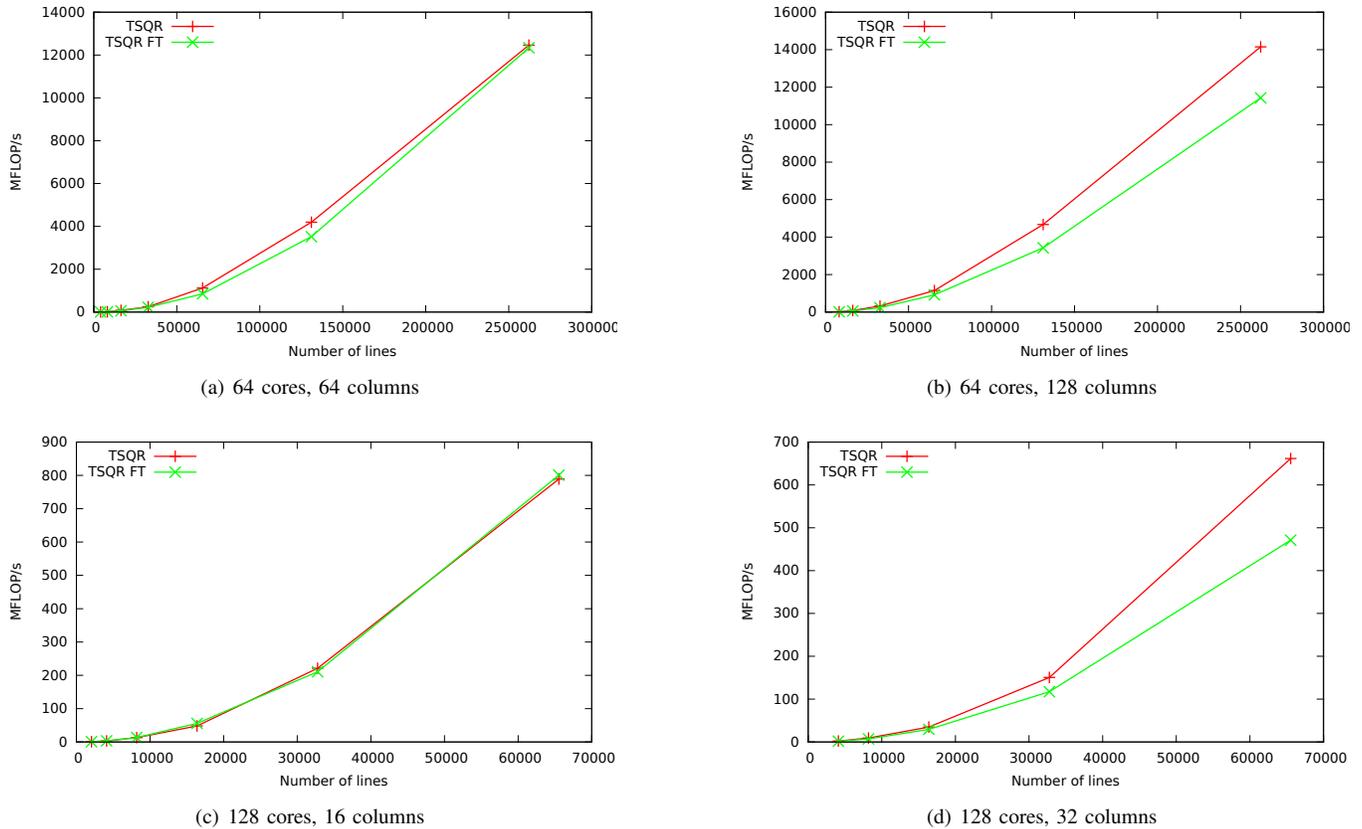
(a) 64 cores, 64 columns

(b) 64 cores, 128 columns

(c) 128 cores, 16 columns

(d) 128 cores, 32 columns

FIG. 6: *Comparison between the performance obtained by the fault tolerant and the non fault-tolerant implementation of TSQR during fault-free execution (overhead). The performance is given in MFLOPs/s (higher is better).*

[3] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[4] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *International Journal of High Performance Computing and Networking (IJHPCN)*, (3), 2004.

[5] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. *Future Generation Computer Systems*, 24 (1):73–84, 2008. Digital Object Identifier: http://dx.doi.org/10.1016/j.future.2007.02.002.

[6] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Yvon Jegou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Benjamin Quetier, and Olivier Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD (SC—05)*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.

[7] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223. ACM, 2005.

[8] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-avoiding parallel and sequential QR factorizations. *CoRR*, abs/0806.2159, 2008.

[9] Simplice Donfack, Laura Grigori, and Alok Kumar Gupta. Adapting communication-avoiding lu and qr factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[10] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[11] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353. Springer, 2000.

[12] 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 (ICS)*, 2004.

[13] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.0. Technical report, 2012.

[14] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, 09 2012.

[15] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tall and skinny qr matrix factorization using tile algorithms on multicore architectures. Technical report, Innovative Computing Laboratory, University of Tennessee, September 2009.

[16] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*, pages 329–332. Springer, 2011.

[17] Julien Langou. Computing the r of the qr factorization of tall and skinny matrices using mpi_reduce. *arXiv preprint arXiv:1002.4250*, 2010.

[18] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, October 1998.

[19] Daniel A. Reed, Charng da Lu, and Celso L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems*, 22(3):293 – 302, 2006.