Scalable, Robust, Fault-Tolerant Parallel QR Factorization

Camille Coti
LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, FRANCE
camille.coti@lipn.univ-paris13.fr

Abstract— **In this paper, we are presenting QR factorization algorithms that can tolerate process crashes and soft errors (bit flips). Our algorithms take advantage of structural properties of a QR factorization algorithm referred to as "communication-avoiding". We show that, exploiting these properties, our resilient, robust algorithms modify the communication pattern of the computation but do not add any significant computation in the critical path.**

I. Introduction

Large-scale parallel system are now more usual and accessible to scientists. The latest release of the Top 500[1], in June 2016, show that, in the world, 95 systems achieve a performance of 1 Petaflops and more. 53 systems ranked in the June 2016 list feature more than 100 000 cores and 466 feature more than 10 000 cores. The computations made at this scale need to be *scalable* in order to take advantage of the computational power of such systems.

Besides, at large-scale, failures are statistically frequent, and need to be taken into account by the computation. When failures are rare, the computations can start hoping that they will complete with no failure during the computation, and restarting the computation when a failure happens. However, at large-scale, failures are so common that it is expected that some nodes will fail during the computation [1]. As a consequence, the computation needs to expect these failures and adapt in order to proceed with the computation *in spite of these failures*. We call algorithms designed to work in such conditions *fault-tolerant* algorithms.

In this paper, we are considering two types of failures. The first one is *crash-type failures*: a process of the parallel computation works correctly until a failure happens, which causes it to stop working [2]. The second type is *bit errors*, which we are also called *soft errors*: the system seems to work correctly, but the result is wrong [3]. These errors can be caused by cosmic rays, especially on space systems This is different from what is usually called numerical error, caused by round-off errors and finite precision.

In this paper, we are focusing on the QR factorization, which has a broad range of applications, such as linear least-square problems, orthogonalization (using the Q matrix), diagonalization (using the R matrix), system solving, singular value decomposition... Our goal is to design algorithms that can be used in a *library approach*: we are providing a QR factorization that enjoys some properties, and the users that need such a QR factorization only needs to use it without taking care of what is necessary to obtain these properties.

In particular, we are presenting here algorithms for a fault-tolerant and robust QR factorization. The computation is fault-tolerant in a sense that it can tolerate process failures. It is robust in a sense that it can tolerate soft errors. These properties are enforced by the algorithm itself: the initial matrix is distributed like with any usual parallel QR factorization, and the result is the same as with any usual parallel QR factorization.

Our algorithms take advantage of a new model for fault-tolerance in parallel programs, called *User-Level Failure Mitigation* [4], that extends past MPI standards, and is being studied for inclusion in future MPI standards. Failures are handled by the application itself, and the behavior followed by the application is defined by the programmer. Failure detection and resolution are local, and depend on

[1] https://www.top500.org

communications: only the processes that try to communicate with a failed process are aware that it has failed. The rest of the system continue its execution unknowingly. This model is particularly interesting at large scale, since it involves limited synchronizations between processes.

The remainder of this paper is organized as follows. A short review of the literature on previous works on fault tolerance for distributed parallel programs is given in section II. The algorithms themselves are given in the next sections: section III recalls the communication-avoiding QR algorithm; section IV gives the fault-tolerant version of this algorithm; section V presents how it can be made numerically robust. Finally, section VI concludes the paper and gives some open perspectives on this topic.

II. Related works

Fault-tolerance, in the fail-stop model [2] in distributed applications can be approached from the *system level* and from the *application level.*

System-level fault tolerance is handled by the middleware and is transparent to the application. It uses a checkpointing mechanism to be able to save and restart the state of a process [5], and a distributed protocol to ensure the consistency of the parallel application after a process rollback. There protocols can be classified in two categories, *coordinated checkpoint* and *non-coordinated checkpoiting* [6]. Message-logging protocols save messages in order to be able to replay them after a rollback; the can use or not an event logger [7]. Coordinated checkpoints are, in most part, based on the Chandy-Lamport algorithm [8], and their implementation can be blocking or non-blocking [9].

However, the impact of such protocols on the performance of the application, especially when the number of processes increases [10], has the drawback of their benefit: they are too generic, whereas a more specific approach would offer better performance.

An application-level approach for MPI applications was initiated in FT-MPI [11]. The MPI standard was extended with functions to handle failures and define a behavior that must be followed by the application. It relies on a specific middleware, that needs to be able to survive failures and offer the corresponding features, such as tun-time spawning of new processes to replace the failed ones, or the absence of failure when a process tries to communicate with a failed one [12]. Such a middleware needs to be itself fault-tolerant: some work has been done to identify and characterize topologies with desirable properties to support it, such as k-ary sibling trees [13] or binomial graphs [14]. Following this application-level approach, some strategies have been developed to make parallel algorithms fault-tolerant [15][16][17]. For instance, some algorithms uses additional processes to store CRCs of parts of the matrices that are computed, and be able to recover the lost data upon failures [18].

More recently, the User-Level Failure Mitigation was introduced [4]. It is not part of the MPI standard (yet), but is under evaluation for further inclusion in forthcoming versions of the standard.

In the mean time, a new generation of parallel numerical algorithms has been developed: communication-avoiding algorithms [19]. In particular, we can cite communication-avoiding algorithms for matrix factorizations: LU, QR and Cholesky [20]. They turn out to be more efficient on current architectures, on which the computational power of the node is important compared to the latency of inter-node communications, on a wide range of architectures, such as multi-cores [21], GPUs [22] and federations of clusters [23]. These algorithms use a minimal number of inter-process communications, and a non-minimal number of computing operations. On current architectures, this approach is faster than former algorithms that were designed to maximize the parallelism between computations and involving a larger number of communications.

Additionally to their scalability, these algorithms feature some structural and algebraic properties that can be exploited for fault-tolerance. We have seen in [24] and [25] that partial redundancy and intermediate computations can be inserted with little modification in the critical path, and bringing a small overhead on the computation time (about 3%).

With *soft errors*, on the other hand, the program seems to work correctly, but the result it computes is wrong. The impact of bit flips on the numerical result was evaluated in [3]. They can be dealt with at several levels. Memory has error-correcting code

(ECC)

Compiling techniques can help, for instance by minimizing the probability that a soft error in registers can be propagated to other system components [26], or by minimizing the time spent by data in non-ECC protected register and preferring ECC-protected registers [27]. A specific kind of check-pointing can also be used [28].

III. Communication-Avoiding QR factorization

The QR factorization of a matrix is an operation that, for a matrix A, decomposes it in two matrices Q and R such that $A = QR$ with Q an orthogonal matrix and R an upper triangular matrix. If A is invertible and the diagonal terms of the R matrix are positive, this decomposition is unique otherwise, it is not.

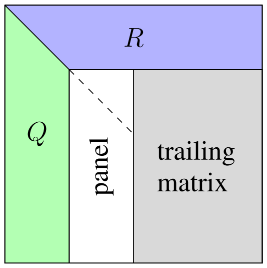

FIG. 1: *Panel/update QR factorization.*

It is usually computed following a *panel/update* scheme: a set of vectors (a panel) are factorized, then the trailing matrix (the submatrix on the right of the panel) is updated; the algorithm proceeds recursively on the trailing matrix until all the matrix has been factorized (see figure 1). To compute it in parallel, the processes are generally organized on a 2D (block-cyclic) grid, where one column of processes is computing the panel factorization and the other ones compute the update of the trailing matrix.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ 0 & A^1_{22} \end{pmatrix}$$

In this section, we are describing the *Communication-Avoiding QR* algorithm, given in [20]. We are describing the panel factorization in section III-A and the update of the trailing matrix in section III-B.

A. Panel factorization

The panel factorization is working on a small set of vectors. Therefore, the submatrix handled here has a specific shape: it is *tall-and-skinny*. A specific algorithm is used to factorize such matrices: *TSQR*.

The processes are organized as a column. Each process works on a submatrix of the panel. On the first step, each process performs a QR factorization of its local submatrix. Then the processes assemble these $\hat{R}$ matrices and compute the QR factorization of the resulting matrices, until the final R is obtained. This step is forming a tree, which can have any shape, since it is actually a reduction [29].

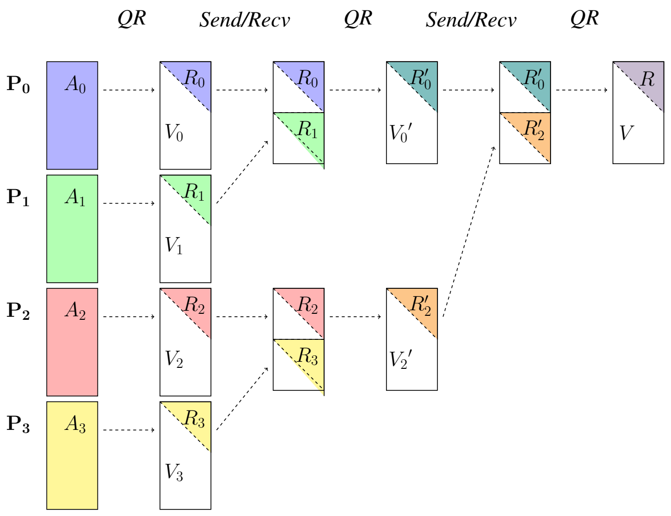

FIG. 2: *Panel factorization: TSQR*

The computation cost of the QR factorization of a M-by-N matrix on P processes is in $O(MN^2/P)$. Traditional parallel algorithms on P processes require $O(Nlog(P))$ communications. TSQR performs extra computation along this reduction tree: it takes $O(MN^2/P + N^3log(P))$ operations, and $O(log(P)$ communications. The same volume of data is exchanged: TSQR saves a factor N in the number of computation, to the cost of an additional factor $N^3log(P)$ in the computations. According to [19], the panel factorization is the bottleneck of the parallel QR factorization; making it significantly faster removes this bottleneck.

B. Trailing matrix update

Considering a matrix A in blocks, as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

The panel factorization is computed on the left-most blocks:

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$$

From the Householder reflectors computed by the panel factorization, we obtain the compact representation of the Q_1 matrix obtained by this panel factorization:

$$Q_1 = I - Y_1 T_1 Y_1^T$$

The Y_1 and T_1 matrices are used to update the trailing matrix:

$$(I - Y_1 T_1 Y_1^T) \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} - Y_1 (T_1^T (Y_1^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix})) = \begin{pmatrix} R_{12} \\ A_{22}^1 \end{pmatrix}$$

The top part R_{12} of the matrix obtained is part of the R of the matrix. The bottom part, A_{22}^1, is the new trailing matrix. The algorithm proceeds recursively on it until the whole matrix is computed.

If we denote the current matrix after the factorization of the first panel as follows:

$$\begin{pmatrix} R_0 & C_0' \\ R_1 & C_1' \end{pmatrix} = \begin{pmatrix} QR & C_0' \\ & C_1' \end{pmatrix}$$

The update consists of computing the $\hat{C}_i'$ factors on the right side of the panel:

$$A = Q \begin{pmatrix} R & \hat{C}_0' \\ & \hat{C}_1' \end{pmatrix}$$

The blocs of the left side of the matrix are decomposed into two parts: the top part contains as many lines as the number of columns of each block, the bottom part contains the rest of the lines. If the width of a block is denoted by N and $C[: N-1]$ denotes the first N lines of matrix C:

$$C_i = \begin{pmatrix} C_i' \\ C_i'' \end{pmatrix} = \begin{pmatrix} C_i[: N-1] \\ C_i[N :] \end{pmatrix}$$

The compact representation of the matrix is computed using the following pairwise computation along a tree:

$$\begin{pmatrix} \hat{C}_0' \\ \hat{C}_1 \end{pmatrix} = \left(I - \begin{pmatrix} I \\ Y_0 \end{pmatrix} T^T \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C_0' \\ C_1' \end{pmatrix}$$

Since the update of the trailing matrix depends on the computation of the Householder reflectors of the panel, we can see that the update of the trailing matrix is triggered by the panel factorization. In parallel, it forms a tree (the same tree as used by the panel factorization) depicted Figure 3. The parallel algorithm itself is given by Algorithm 1.

Algorithm 1: CAQR factorization: panel/update on a 2D grid.

```
Data: Initial matrix $A$, divided in panels $p$
1 for each panel $p$ do
2     On the column of processes that own p: TSQR( p );
3     Each process of this column broadcasts the
      Householder vectors and Householder multipliers $\tau_p$
      that it has just computed to the processes of the
      same row;
4     Using this $\tau_p$, each process forms its local $T$ and $Y$;
5     for step = 0 to log($P_{column}$) do
6         updateTrailingMatrix ( step );
7         if done ( step ) then
8             break;
```

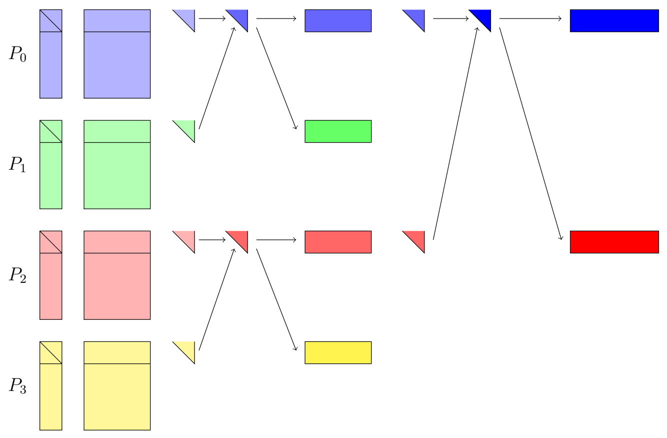

FIG. 3: *Tree formed by the parallel update of the trailing matrix.*

At each step of the tree, the update is computed between pairs of processes. The operations (communication and computation) performed are depicted Figure 4.

As proved in [19], the computational complexity of both the CAQR and traditional parallel algorithms of a $n \times n$ on a 2D-grid of P processes (*i.e.* a

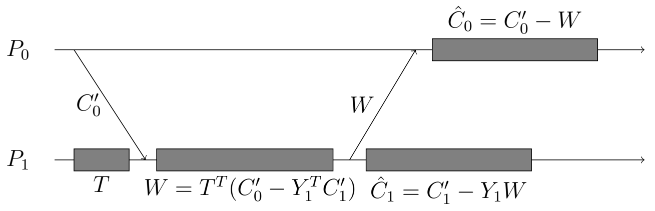

FIG. 4: *Pairwise computation of the update of the trailing matrix.*

grid of $\sqrt{P} \times \sqrt{P}$ processes) is in $O(\frac{4n^3}{3P})$. The communication volumes are also the same. The number of communications involved by $\frac{3}{8}\sqrt{P}log^3P$ communications, whereas traditional parallel algorithms involve $\frac{5n}{4}log^2P$.

IV. Fault-Tolerant Communication-Avoiding QR factorization

This section describes a fault-tolerant QR factorization algorithm. As presented in section III, the QR factorization algorithms is composed by two parts: a panel computation and the update of the trailing matrix. Therefore, we are presenting one algorithm for each of these parts: a fault-tolerant panel factorization (section IV-A) and a fault-tolerant trailing matrix update (section IV-B).

A. Panel factorization

We have seen in section III-A that, after the factorization of the local submatrix, the recombination of the intermediate $\hat{R}$ factors and the computation of the final R form a tree. If this tree is a binary one, at each step of the tree, half of the processes send their intermediate $\hat{R}$ and stop working.

Instead of that, the fault-tolerant TSQR factorization consists in *exchanging* the intermediate $\hat{R}$ factors: therefore, both processes have the same data and can proceed with the computation. This scheme is depicted figure 5.

The number of computing operations in the critical path of this algorithm is unchanged: the additional operations are performed on processes that would normally be idle. The only modification affecting the critical path is that, instead of having a send-receive communication between pairs of processes, they exchange this data. Therefore, fault-tolerance is achieved with *no additional process*,

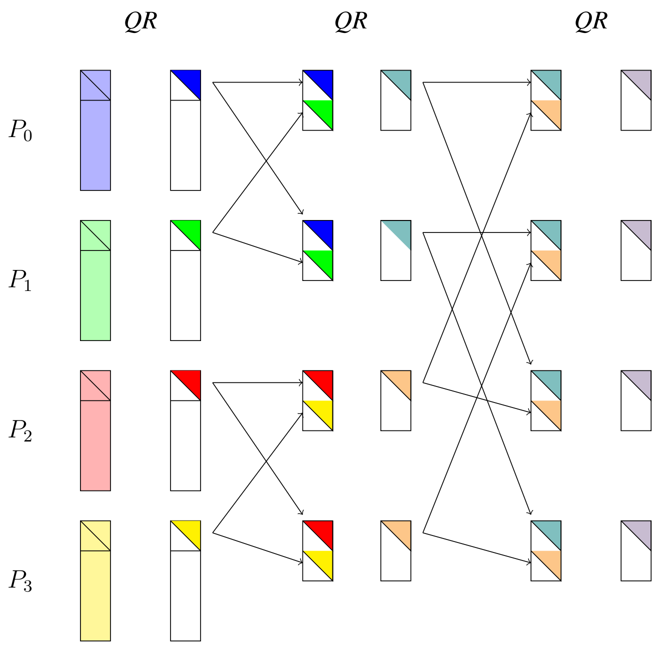

FIG. 5: *Fault-tolerant panel factorization.*

and *no significant addition* in the critical path of the computation.

At each step, the number of processes that holds the same data is doubled. Therefore, the resilience of the computation increases as the computation progresses and as the failure probability increases. If a process fails, a new process is spawned to replace it. Assuming that the initial A matrix is stored somewhere else and can be retrieved, it can obtain the lost data (the intermediate $\hat{R}$) from a peer process that holds the same data. *No recomputation is necessary*, since the friend process has performed the same computation.

B. Trailing matrix update

We have seen in section III-B that the update of the trailing matrix is triggered by the factorization of the panel, from which the Y and the T vectors can be computed. We have detailed this computation, made by pairs of processes, on figure 4. The upper process sends its C'_0 submatrix to the lower process. Then the lower process can compute $W = T^T(C'_0 - Y_1^T C'_1)$. This W is then sent to the upper process and both processes use it to compute their $\hat{C}$.

We have seen in section IV-A that, in the fault-tolerant version of the panel factorization, both processes compute the intermediate $\hat{R}$. Therefore, both of them can compute the T. If, in addition to the upper process sending its C_0' submatrix to the lower process, the lower process also sends its C_1' submatrix and Y_1, both process can compute the W and then their $\hat{C}$ (figure 6).

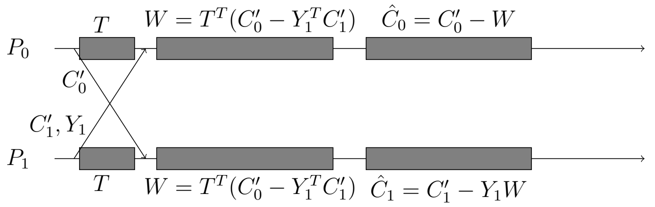
T
$W = T^T(C_0' - Y_1^T C_1')$
$\hat{C}_0 = C_0' - W$
P_0
C_0'
C_1', Y_1
P_1
T
$W = T^T(C_0' - Y_1^T C_1')$
$\hat{C}_1 = C_1' - Y_1 W$

FIG. 6: *Pairwise computation of the fault-tolerant update of the trailing matrix.*

Therefore, the subsequent communication scheme of the fault-tolerant update of the trailing matrix using the fault-tolerant panel factorization and this pairwise computation is depicted figure 7.

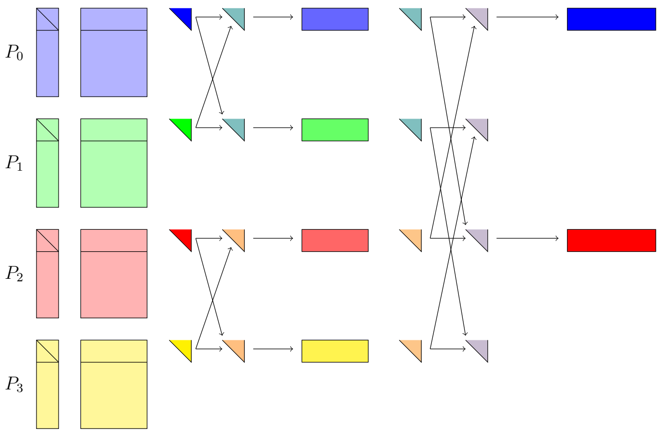

FIG. 7: *Tree formed by the fault-tolerant parallel update of the trailing matrix.*

This algorithm *changes the communication scheme* of the pairwise computation. It starts with an exchange (C_0' / C_1') and Y_1 is sent from the lower process to the upper process. However, it does not involve any second communication phase, whereas the original algorithm (figure 4) has a second communication phase, in which the lower process sends the W matrix to the upper process. *No computation is added in the critical path*: if both processes compute the T and the W, this is done in parallel on both processes and does not extend the critical path.

If a process fails, its data can be recovered using local data from the initial matrix and the W computed on the other process. Therefore, *only a few computations* are required to recover the state of the computation after a failure.

V. NUMERICAL ROBUSTNESS

Using the (partial) redundancy of intermediate results, the correctness of the computation can be verified. We are not dealing here with soft errors such as round-off errors, but about failures such as bit flips. For instance a bit can be flipped because of a cosmic ray or an electronic error. In this section, we are presenting an algorithm that verifies the correctness of each step of the computation. Due to the space limitations of this paper, we are giving this algorithm for the panel factorization only, because it is slightly more complex for performance reasons. However, the same idea can be applied to the trailing matrix update.

We have seen in section IV-A that redundancy can be introduced in the parallel computation with a minimal overhead. Several processes hold the same data and compute the same $\hat{R}$. As a consequence, *these results can be compared*. In order to reduce the cost of the communication and the comparison, a checksum can used. This algorithm is represented Figure 8.

The first step still needs to be handled a bit more carefully. If the number of processes that hold the same data doubles at each step of the reduction tree, as stated in section IV-A, the initial submatrices themselves are originally present on one process only. Therefore, a redundancy needs to be introduced. Before the beginning of the computation, each process exchanges its submatrix with another process. Then each process computes the QR factorization of the two submatrices it has in memory: its own one, and the other process's submatrix. Then they check the correctness of the two $\hat{R}$ they obtained. Since the processes already have two $\hat{R}_i$ and $\hat{R}_j$ intermediate factors, the first step of the tree does not require any additional communication between processes i and j.

If two processes notice that their checksum is not the same, they *rollback*, *i.e.* they compute the same $\hat{R}$ again. For this, a copy of the matrix used to compute this $\hat{R}$ (formed by two intermediate $\hat{R}$

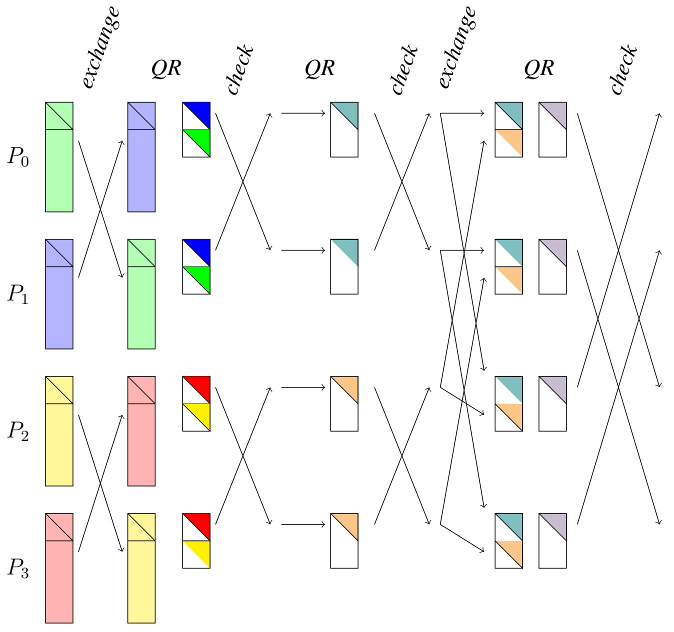

FIG. 8: *TSQR with correctness checks.*

or by the initial submatrix) is kept, and restored to perform the computation again.

Compared with running the whole computation several times and comparing the final result, this approach presents the advantage of verifying *intermediate* results, and, if necessary, running again *partial* computations.

If an error happens during the tree phase of the algorithm, the slow-down caused by the rollback is relatively small: the additional computation is in $O(N^3)$, which is supposed to be small compared to the cost of the overall computation.

However, if an error happens during the computation of the first QR factorization, the factorization that needs to be computed again is the one using the initial submatrix: the additional computation is in $O(MN^2)$, which is close to the cost of the overall computation. Therefore, in order to reduce this cost due to the rollback on soft errors, the original submatrix can be split into several parts and a (local) TSQR can be computed on it. A checksum is computed for each part of the local TSQR and all of them are compared with the peer process. Hence, only the subpart of the matrix that contains the error is computed again.

VI. CONCLUSION

In this paper, we have presented a set of algorithms for resilience in parallel, distributed computation of the QR factorization of matrices. We are considering two kinds of errors: fail-stop crashes and soft errors.

We have presented a quick recall of a scalable, communication-avoiding parallel algorithm for the QR factorization of a matrix. We have seen that, exploiting some structural and algebraic properties of this algorithm, fault tolerance and robustness properties can be obtained and exploited. In particular, the extra computations introduced by the family of algorithms we are interested in here can be exploited on idle processes in order to introduce some redundancy at minimal cost, *i.e. without adding any significant operation* in the critical path of the computation.

Not only do these algorithms have little overhead on the execution time, as presented for example in [24], but also the recovery after a failure implies little recomputation, and even no recomputation at all for the tall-and-skinny algorithm.

Using these partial redundancy of intermediate results, we have also seen that soft errors such as bit flips can be detected, making the computation *not only resilient, but also robust*. As a consequence, these algorithms are suitable for critical systems while achieving high performance.

The idea presented in this paper and applied to the QR factorization can be applied to other algorithms of the same family. For instance, we can cite the LU and Cholesky communication-avoiding algorithms.

Besides, the performance of these algorithms have been evaluated, for instance in [24], emphasizing on the overhead of the fault-tolerance algorithm with respect to the regular, non-fault-tolerant algorithm. It would be interesting to see how it behaves with specific tuning.

a) Acknowledgement: We would like to thank Bob Robey for the discussions on using partial redundancy for robust computing in the presence on bit flip errors.

REFERENCES

[1] D. A. Reed, C. da Lu, and C. L. Mendes, “Reliability challenges in large systems,” *Future Generation Computer Systems*, vol. 22, no. 3, pp. 293 – 302, 2006.

[2] R. D. Schlichting and F. B. Schneider, “Fail stop processors: An approach to designing fault-tolerant computing systems,” *ACM Transactions on Computer Systems*, vol. 1, pp. 222–238, 1983.

[3] J. J. Elliott, F. Mueller, M. K. Stoyanov, and C. G. Webster, “Quantifying the impact of single bit flips on floating point arithmetic,” Oak Ridge National Laboratory (ORNL), Tech. Rep., 2013.

[4] W. Bland, A. Bouteiller, T. Hérault, J. Hursey, G. Bosilca, and J. J. Dongarra, “An evaluation of user-level failure mitigation support in MPI,” *Computing*, vol. 95, no. 12, pp. 1171–1184, 2013.

[5] E. R. Jason Duell, Paul Hargrove, “The design and implementation of berkeley lab’s linux checkpoint/restart,” Berkeley Lab, Tech. Rep. publication LBNL-54941, 2003.

[6] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, “Coordinated checkpoint versus message log for fault tolerant MPI,” *International Journal of High Performance Computing and Networking (IJHPCN)*, no. 3, 2004.

[7] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello, “Impact of event logger on causal message logging protocols for fault tolerant MPI,” in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05)*. Washington, DC, USA: IEEE Computer Society, 2005, p. 97.

[8] K. M. Chandy and L. Lamport, “Distributed snapshots : Determining global states of distributed systems,” in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.

[9] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, “Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI,” in *Proceedings of the International Conference for High Performance Networking Computing, Networking, Storage and Analysis (SC—06)*, ACM/IEEE, Ed., Tampa, USA, November 2006, p. electronic.

[10] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, “Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI,” *Future Generation Computer Systems*, vol. 24 (1), pp. 73–84, 2008.

[11] G. E. Fagg and J. Dongarra, “FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users’ Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*, ser. Lecture Notes in Computer Science, J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds., vol. 1908. Springer, 2000, pp. 346–353.

[12] G. E. Fagg and J. J. Dongarra, “HARNESS fault tolerant MPI design, usage and performance issues,” *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1127–1142, Oct. 2002.

[13] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra, “Scalable fault tolerant protocol for parallel runtime environments,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users’ Group Meeting (EuroPVM/MPI’06)*. Springer, 2006, pp. 141–149.

[14] G. Bosilca, C. Coti, T. Herault, P. Lemarinier, and J. Dongarra, “Constructing resilient communication infrastructure for runtime environments,” in *International Conference in Parallel Computing (ParCo2009)*, Lyon, France, September 2009.

[15] 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 (ICS)*, 2004.

[16] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, “Fault tolerant high performance computing by a coding approach,” in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 213–223.

[17] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, “Algorithm-based fault tolerance for dense matrix factorizations,” *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 225–234, 2012.

[18] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, “Algorithm-based fault tolerance applied to high performance computing,” *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 410–416, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.12.002

[19] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, “Communication-avoiding parallel and sequential QR factorizations,” *CoRR*, vol. abs/0806.2159, 2008.

[20] ——, “Communication-optimal parallel and sequential QR and LU factorizations,” *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[21] S. Donfack, L. Grigori, and A. K. Gupta, “Adapting communication-avoiding lu and qr factorizations to multicore architectures,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[22] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, “Communication-avoiding qr decomposition for gpus,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 48–58.

[23] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. 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.

[24] C. Coti, “Exploiting redundant computation in communication-avoiding algorithms for algorithm-based fault tolerance,” in *2nd IEEE International Conference on High Performance and Smart Computing (IEEE HPSC 2016)*, April 2016.

[25] ——, “Fault tolerant QR factorization for general matrices,” *CoRR*, vol. abs/1604.02504, 2016.

[26] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956570

[27] J. Yan and W. Zhang, “Compiler-guided register reliability improvement against soft errors,” in *Proceedings of the 5th ACM international conference on Embedded software*. ACM, 2005, pp. 203–209.

[28] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, “Combining partial redundancy and checkpointing for hpc,” in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, June 2012, pp. 615–626.

[29] J. Langou, “Computing the R of the QR factorization of tall and skinny matrices using MPI_Reduce,” *arXiv preprint arXiv:1002.4250*, 2010.