Application-based fault tolerance for numerical linear algebra at large scale

Daniel Torres Gonzalez

UNIVERSITÉ PARIS XIII - SORBONNE PARIS NORD
École Doctorale Sciences, Technologies, Santé Galilée

Application-Based Fault Tolerance for Numerical Linear Algebra at Large Scale

Tolérance aux Pannes Basée sur les Applications pour l'Algèbre Linéaire Numérique à Grande Échelle

THÈSE DE DOCTORAT
présentée par

Daniel Alberto TORRES GONZÁLEZ
Laboratoire d'Informatique de Paris Nord, CNRS UMR 7030

pour l'obtention du grade de
DOCTEUR EN INFORMATIQUE

soutenue le 15/12/2021 devant le jury d'examen composé de :

WOLFLER CALVO Roberto, Université Sorbonne Paris Nord ..... Examinateur
VIALLE Stéphane, CentraleSupélec ..... Rapporteur
DAYDÉ Michel, INP-ENSEEIHT ..... Rapporteur
THIBAULT Samuel, Université de Bordeaux ..... Examinateur
BAUTISTA GOMEZ Leonardo, Barcelona Supercomputing Center .... Examinateur
PETRUCCI Laure, Université Sorbonne Paris Nord ..... Directrice de thèse
COTI Camille, University of Oregon ..... Co-encadrante de thèse

Abstract

The increasing size of supercomputers is allowing to compute solutions for various computational problems that, not so long ago, were consuming too much time and memory to be solved. However, at that scale, other challenges are arising. The optimal performance of dedicated algorithms running on these parallel and distributed architectures gets hampered by these new challenges. Large-scale systems involve their own challenges due to the total number of hardware and software components and the complexity of these components, including system reliability, availability, and scalability. In particular, hardware or software failures may occur at any moment during the execution of parallel applications that, therefore, cannot complete unless these failures are handled. Meanwhile, as high-performance systems have become larger, failures have become common and computation units are expected to fail during the execution of critical programs. As a consequence, fault tolerance has arisen as a major challenge in parallel and distributed computing, and parallel programs ought to rely on scalable algorithms that also ensure the correct completion of the program in spite of failures. In particular, parallel subroutines provided in libraries should be able to mitigate errors that occur in a call to these subroutines. For instance, linear algebra operations are building blocks used by domain scientists to solve problems or to simulate scientific phenomena. They are usually provided in highly optimized libraries and are typically software components that ought to be fault tolerant. One way to make sure fault tolerant algorithms are robust and can tolerate any failure scenario consists in designing a formal model of the system and prove properties on this model. In this thesis, I am presenting a set of fault tolerant algorithms for QR, LU and Cholesky matrix factorizations. These algorithms rely on a the corresponding communication-avoiding algorithms, for tall-and-skinny matrices and for general matrices; communication-avoiding algorithms have given good performance on multiple architectures and have properties that make them scale well. Using a model, I am proving robustness properties of these algorithms and, in particular, that as long as enough processes survive (and this number decreases as the execution progresses), they can tolerate any failure scenario. My algorithms can tolerate crash-type failures, respawn new processes to replace the failed ones and take advantage of algebraic and algorithmic properties to implement a forward-recovery approach, i.e., the processes that replace the failed ones to not roll back to a previous state but are inserted at a point of the execution that was never reached by the failed processes. I am presenting an experimental evaluation of the overhead introduced by the fault tolerance mechanisms on failure-free executions and the cost of a failure. Both scenarios have a small overhead, meaning that the fault tolerance mechanisms

have little impact on the critical path of the algorithms, and the post-failure restoration time is small.

Résumé

La taille croissante des supercalculateurs a permis de s'attaquer à des problèmes de calcul auparavant trop coûteux en temps et en mémoire pour être résolus. Cependant, d'autres problèmes apparaissent à cette échelle et la performance obtenue par les algorithmes parallèles et distribués conçus pour cette échelle est impactée. Les systèmes à grande échelle font face à des défis spécifiques du fait du grand nombre de composants matériels et logiciels dont ils sont composés, et parmi ces défis on compte la fiabilité du système, la disponibilité et le passage à l'échelle. En particulier, des défaillances matérielles et logicielles peuvent survenir à tout moment pendant l'exécution d'applications parallèles qui, par conséquent, ne peuvent pas s'exécuter intégralement et atteindre leur fin. Alors que la taille des systèmes parallèles augmente, les défaillances sont appelées à arriver fréquemment lors de l'exécution de programmes parallèles. Par conséquent, la tolérance aux défaillances est aujourd'hui un défi important pour le calcul parallèle distribué, et des programmes parallèles doivent se baser sur des algorithmes capables de passer à l'échelle mais aussi d'assurer la terminaison correcte du programme malgré les défaillances qui surviendraient. En particulier, des fonctions parallèles fournies dans des bibliothèques doivent être capables de tolérer des défaillances survenues dans les appels à ses fonctions. Les opérations d'algèbre linéaire sont des briques de base de calculs utilisés par les scientifiques pour résoudre des problèmes ou simuler des phénomènes scientifiques. Elles sont habituellement fournies dans des bibliothèques hautement optimisées et sont typiquement des composants logiciels qui devraient être capables de tolérer des défaillances. Une méthode pour s'assurer que des algorithmes tolérants aux défaillances sont robustes et peuvent tolérer n'importe quel scénario de défaillances consiste à concevoir un modèle formel du système et à prouver des propriétés sur ce modèle. Dans cette thèse, je présente un ensemble d'algorithmes tolérants aux pannes pour les factorisations LU, QR et de Cholesky. Ces algorithmes se basent sur les algorithmes à évitement de communications correspondants, pour matrices hautes et fines et pour matrices générales. Les algorithmes à évitement de communications ont montré de bonnes performances sur de multiples architectures contemporaines et ont des propriétés qui leur permettent de bien passer à l'échelle. En utilisant un modèle, je prouve des propriétés de robustesse de ces algorithmes et, en particulier, que tant que suffisamment de processus survivent (et leur nombre diminue au cours de l'exécution), ils peuvent tolérer n'importe quel scénario de défaillances. Ces algorithmes tolèrent des pannes de type *crash*, ou *fail-stop*, relancent des processus pour remplacer ceux atteints par des défaillances et exploitent des propriétés algorithmiques et algébriques pour mettre en place une approche de type *retour en avançant*, c'est-à-dire que

les processus qui remplacent ceux touchés par les défaillances n'effectuent pas un retour en arrière mais sont insérés dans l'exécution à un point qui n'avait jamais été atteint par les processus touchés par les défaillances. Je présente une évaluation expérimentale du surcoût introduit par les mécanismes de tolérants aux défaillances sur l'exécution en l'absence de pannes et le coût d'une défaillance. Ces deux scénarii ont un surcoût faible, impliquant que les mécanismes de tolérance aux défaillance ont un impact faible sur le chemin critique des algorithmes, et le temps de restauration de l'état après une défaillance est faible.

Acknowledgments

With a special dedication:

- To my family: for being aware of the progress of the PhD (baby boy, wife, father, mother, brother, sisters, brother-in-law, nephew and niece).

- To the partners in the laboratory: for sharing some delicious moments during the last three years, trying to teach me as much french as possible, each one of them with different accent.

- To Laure Petrucci: for accepting me as her PhD student, for giving me the necessary advices to fully develop and write the thesis work and for being aware of every single moment during my stay at the laboratory and in France.

- To Camille Coti: for accepting me as her PhD student too, despite of my english level, for the discussions about how to optimize some parts of the work and for her advices and help in the writing of the thesis.

- To the members of the laboratory: for being gentle at all times with each member.

- To the members of the jury: for accepting to be part of the thesis defense.

- To the Grid5000 team: for allowing me to perform my experiments on this large-scale infrastructure, regardless of how much time I needed to successfully complete all the experiments.

- To the National Council for Science and Technology (CONACyT): for the granted support.

Contents

Acronyms List

ABFT	Algorithm-Based Fault Tolerance
CA	Communication-Avoiding
CACH	Communication-Avoiding Cholesky
CALU	Communication-Avoiding LU
CAQR	Communication-Avoiding QR
CPN	Coloured Petri Nets
FTCACH	Fault-Tolerant Communication-Avoiding Cholesky
FTCALU	Fault-Tolerant Communication-Avoiding LU
FTCAQR	Fault-Tolerant Communication-Avoiding QR
FTTSCH	Fault-Tolerant Tall and Skinny Cholesky
FTTSLU	Fault-Tolerant Tall and Skinny LU
FTTSQR	Fault-Tolerant Tall and Skinny QR
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
OMPI	OpenMPI
TS	Tall and Skinny
TSCH	Tall and Skinny Cholesky
TSLU	Tall and Skinny LU
TSQR	Tall and Skinny QR
ULFM	User-Level Fault Mitigation

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Current High-Performance Computing (HPC) systems have been growing for three decades and continue to grow. Now that exascale is around the corner (with the announcement of machines such as El Capitan and the EuroHPC machines), a set of challenges have been identified to be addressed for exascale[1, 2, 3]. Fault tolerance is one of them [4].

The *Top500* ranks the 500 world's most powerful supercomputers (that submit a score). The June 2021 Top500 (`https://www.top500.org/lists/top500/2021/06/`) rank includes 5 machines that feature more than a million cores, and all the 500 machines listed have more than 10 000 cores. Hence, large-scale computing is no longer reserved for a handful of specialized machines, but accessible in many computation centers.

As the number of processors and nodes is increasing, HPC systems become more prone to failures [5]. Failures can arise anytime, stopping partially or totally the execution (crash-type failures) or providing incorrect results (bit errors). In this thesis, I focus on failures in the fail-stop model: processes work normally until they stop working completely. Failure detectors utilized in HPC use this model, such as [6].

The challenge for fault-tolerance in HPC presents two aspects: keep the overhead on failure-free execution low and recover the execution after a failure with as little overhead as possible. In other words, the cost of fault tolerance and the cost of failures must remain low.

1.1 Background

1.1.1 Current HPC Systems

Increasing the clock frequency in modern microprocessors faces physical limitations. Current HPC supercomputers are built by aggregating more and more powerful components, each with complex architectures (GPUs, multicore chips, etc.). But with this increase the difficulty for programs to reach extreme high-performance throughput increases too, due to failures.

The Mean Time Between Failures (MTBF) is a measure of system reliability which is defined as the probability that the system performs without deviations from agreed-upon behavior for a specific time. Equation 1.1 shows how MTBF can be estimated using the individual MTBF of each component and in figure 1.1 we can see how this measure decreases

as the number of components increases.

$$MTBF_{\mathcal{T}} = (\sum_{i=0}^{n-1} \frac{1}{MTBF_i})^{-1} \tag{1.1}$$

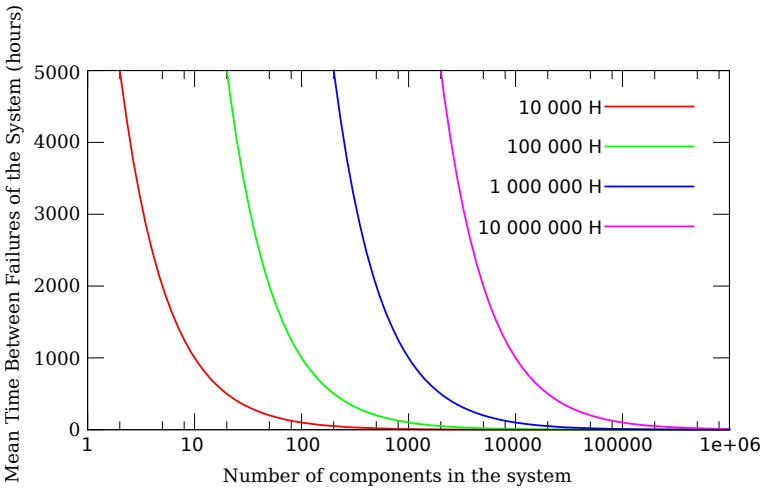


Figure 1.1: Mean Time Between Failures example

According to this estimation, at the scale the upcoming generation of HPC systems will be at, the system will experience a component failure every few hours or even minutes. For instance, the supercomputer Blue Waters located at the National Center for Supercomputing Applications (NCSA) at the University of Illinois has an MTBF of 4.18 hours, approximately. This means that an application that takes more than 4.18 hours to finish its computation in the whole machine has one out of two chances (50%) to finish its computation without experiencing a failure, and the other chance will be hit by an error. So, the expected behavior is the first case, that the launched program ends, but this will only happen 50 percent of the time the program is run. Therefore, computation intensive applications need fault tolerance to run in a sustainable way at large-scale. Furthermore, such applications should be designed to expect failures and take suitable actions, but the applications need to rely on a fault-tolerant environment. While the performance of the most powerful systems has almost doubled every 18 months, the overall system MTBF is reduced to just a few hours. Table 1.1 shows the five most powerful machines in *Top500* according to the June 2021 list(`https://www.top500.org/lists/top500/2021/06/`).

The road to exascale platforms requires simultaneous use and management of thousands or even millions of processing, networking, and storage components. In the not too distant future (*El Capitan* is expected in 2022[1]), we are expecting an exaflop machine with hundreds of CPU cores and thousands accelerator cores. But using this amount of components has a dramatic impact on the MTBF of the entire platform because the probability of a failure to

[1]https://www.llnl.gov/news/doennsa-lab-announce-partnership-cray-develop-nnsas-first-exascale-supercomputer

Table 1.1: Five Most Powerful Machines in *Top500* June 2021 list.

Rank	System	**Cores**	Rmax (TFlop/s)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science, Japan	**7,630,848**	442,010.0
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory, United States	**2,414,592**	148,600.0
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL, United States	**1,572,480**	94,640.0
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi, China	**10,649,600**	93,014.6
5	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC, United States	**706,304**	64,590.0

occur during the execution of a critical program on an exascale supercomputer gets closer to 1.

Lots of scientific programs designed specifically for these high-performance systems require long execution times. With this amount of components, element failure is more frequent, making it more difficult for programs to progress. To benefit from the computing power of these complex architectures and avoid restarting programs from the beginning when a failure occurs, the algorithms implemented in programs must be redesigned to tolerate failures efficiently.

1.1.2 Linear Algebra over Large Scale Infrastructures

Linear algebra operations are used by a lot of scientific applications and therefore, efficient linear algebra computation kernels that take advantage of the hardware described in the previous 1.1.1 section are in high demand, so the design of parallel linear algebra kernels

that take advantage of the hardware described in the previous section (1.1.1) should be considered important. However, extreme-scale is a challenging environment.

The major limiting factor to use HPC platforms to run dense linear algebra algorithms is the fact that such operations do not reach the highest performance and acceleration which is expected because of high data movement costs. Networks are getting faster, but at a slower rate than computation nodes. While computation nodes continue to increase their computing speed, the network that interconnects them is facing physical limits. Therefore, the cost imbalance is getting more and more in favor of the computation nodes as communications are getting more and more expensive with respect to computation. Unless the algorithms count with an implicit high level of parallelism due to repetitive operations, their performance will continue to be hampered by the time to move data between nodes (compared to the computation speed of the nodes). To take advantage of a large-scale distributed computing environment, the number of communications the programs use must be rethought to reduce and adjust them with the topology the cluster is built upon. Another possibility consists of hiding these communications by overlapping them with computations.

In this thesis, I am focusing on a class of algorithms that reduce the number of communications in matrix factorizations: the *Communication-Avoiding Algorithms*. Since communications are expensive over large-scale infrastructures, these algorithms aim to use a minimal number of inter-process communications, and a non-minimal number of computing operations. As an example of complexity reduction in a Communication-Avoiding algorithm we have the TSQR (see section 5.1) and CAQR algorithms. For example, CAQR that uses a right-looking QR factorization of a matrix on a two-dimensional grid of processes, distributing a 2D block cyclic layout over the grid. In this case, the number of arithmetic operations is almost the same as a parallel QR factorization implemented in ScaLAPACK, but the number of messages transferred in CAQR is reduced by a factor of the chosen block size. An example of this complexity reduction can be seen in table 1.2. For more details about complexity in Communication-Avoiding algorithms, see [7].

Table 1.2: Comparison of the communication and computation complexities for different parallel QR factorization algorithms.

	TSQR ($m \times n, P = P \times 1$)	CAQR	PDGEQRF
Flops	$\frac{4}{3}\frac{mn^2}{P} + \frac{2}{3}n^3 log(P)$	$\frac{4}{3}\frac{n^3}{P} + \frac{4}{3}\frac{n^2 b}{\sqrt{P}} log(P)$	$\frac{4}{3}\frac{n^3}{P}$
Bandwidth	$\frac{3}{4}mn \cdot log(P)$	$\frac{3}{4}\frac{n^2}{\sqrt{P}} log(P)$	Same
Latency	$log(P)$	$\frac{5}{2}\frac{n}{b} log(P)$	$\frac{3}{2}n log(P)$

1.1.3 Matrix Factorizations

A matrix factorization or a matrix decomposition is a procedure that factorizes a matrix into a product of matrices, to reduce the computational complexity of computing other matrix operations that can be performed on the decomposed matrix rather then on the original matrix. A matrix factorization is part of the basic linear algebra operations on computers,

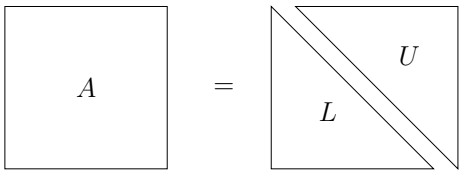
even for routine operations such as solving systems of linear equations, computing the inverse of a matrix or computing its determinant.

For example, if we need to solve a system of linear equations $Ax = b$, the LU, QR and *Cholesky* factorizations, casually known as *Los Tres Amigos*, can be used.

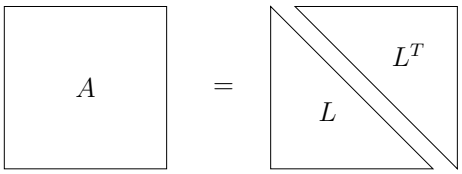
- LU factorization: decomposition of a matrix A into a product of matrices $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. It can be used, among other operations, to solve square systems of linear equations, to invert a matrix, or to compute the determinant of a matrix. The matrix computation can be represented as in figure 1.2a.

- QR factorization: decomposition of a matrix A into a product of matrices $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. It can be used to solve the linear least-squares problem. It is also the basis for finding the eigenvalues of a matrix using the QR factorizations. The matrix computation can be represented as in figure 1.2c.

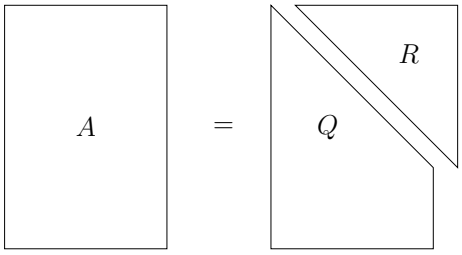
- *Cholesky* factorization: decomposition of a Hermitian, positive-definite matrix A into a product of matrices $A = LL^T$, where L is a lower triangular matrix and L^T is its conjugate transpose. This decomposition is useful for efficient numerical solutions and, when it is possible to apply the algorithm, it is roughly twice as fast as the LU factorization for solving systems of linear equations $Ax = b$. The matrix partitioning can be represented as in figure 1.2b.



(a) Representation of the LU factorization



(b) Representation of the Cholesky factorization



(c) Representation of the QR factorization

Figure 1.2: Matrix factorizations general representation.

1.1.4 Fault Tolerance in HPC

As mentioned in section 1.1.1, fault tolerance is necessary to sustain large-scale executions. Fault-tolerant methods aim at enhancing the use of the system at a high scale by handling failures occurring during the execution of applications. Indeed, the overhead on HPC applications due to fault tolerance is critical to the efficient use of the newest HPC systems. To use these systems efficiently and avoid restarting applications from the beginning when a failure occurs, the applications must be designed to tolerate failures, while keeping low overhead on failure-free executions.

Fault tolerance for high-performance parallel and distributed applications can be achieved at two different levels:

- System-level: transparent for the application (no modification is needed) and requires a specific middleware to restart the failed processes and maintain a consistent state of the application. It often uses a checkpointing mechanism and relies on rollback recovery to save and restart the state of a process in case of an error, and a distributed protocol to ensure the consistency of the parallel application after a process rollback. The protocols are usually classified into two categories: coordinated checkpoint and non-coordinated checkpoint.

- Application-level: requires the application itself to handle the failures and adapt to them. It implies that the specific middleware that supports the distributed execution must be robust enough to survive the failures and provide the application with primitives to handle them, such as run-time re-spawning of new processes to replace the failed ones or detect the absence of a failure process when an active process tries to communicate with the failed one. Such a middleware needs to be fault-tolerant.

The *de facto* message-passing standard capable to operate on a wide range of parallel and distributed computing architectures is the Message Passing Interface (MPI). The standard defines a friendly syntax and semantics to write efficient applications for large-scale architectures like clusters. Its set of libraries proves useful to a wide range of users that write message-passing programs in C, C++, and Fortran, turning it into the main paradigm to write parallel applications on distributed-memory, large-scale infrastructures.

Nowadays, MPI counts several efficient implementations (like *OpenMPI* or *MPICH*), many of which are open-source and/or derived from these two implementations. Because MPI facilitates the development of scalable large-scale parallel applications, the scientific community has encouraged the use of this interface on HPC systems. However, MPI is still evolving and benefits major research efforts to support the complexity of currently distributed architectures, interfacing with new programming paradigms and taking advantage of dynamic run-time systems.

Research topics related to fault-tolerant algorithms have led to the development of new distributed and parallel environments. Fault-tolerant programming environments should be used to ensure the safe execution of critical applications. Afault-tolerant specification that provides process-level fault tolerance for MPI applications was initiated in the mid 2000s: *FT-MPI*. It was extended with new functions to handle failures at run-time and define the

behavior that must be followed by the application after an error occurs, like respawning or restarting crashed processes. A more recent MPI Standard draft proposal called User-Level Fault Mitigation (ULFM) provides an interface specification to implement failure-recovery strategies in MPI: algorithms can be designed to be intrinsically fault-tolerant and programs can be implemented, stabilized, and executed using any implementation of this interface. This last specification is based on FT-MPI.

As previously said, a specific middleware is needed to be able to survive failures. It should offer the corresponding features to handle the replacement of failed processes, such as run-time spawning of new processes. Such middleware needs to be fault-tolerant. So FT-MPI and ULFM are attempting to unify a direction towards standardization of fault tolerance for MPI.

1.2 Objectives

The central idea of this work is to design, implement, evaluate and verify fault-tolerant dense linear-algebra algorithms, relying on communication-avoiding algorithms and exploiting properties of these algorithms. Those algorithms were chosen because, as presented in section 1.1.4, they achieve good performance on current large-scale machines. A specific approach is made on how they can be exploited to provide new fault-tolerant dense linear-algebra algorithms capable of giving the same results with low computational overhead. The produced libraries should be efficient and they should have a low impact recovery in case a run-time failure occurs. To completely achieve the main objective of the research, a series of previous objectives should be achieved first:

1. Analysis of existing non-fault-tolerant algorithms: how properties can be inserted in non-fault-tolerant algorithms to make them fault-tolerant.

2. Analysis of the user-level failure mitigation: what the model allows in terms of algorithm design and performance analysis, what the mechanisms implemented by the MPI libraries (OpenMPI, ULFM) allow.

3. Implementation of non-fault-tolerant algorithms: which steps of the algorithms must be emphasized, take advantage of the properties and get involved with the functionality.

4. Performance analysis of non-fault-tolerant algorithms: measure the performance of the non-fault-tolerant implementations, to serve as a baseline for the fault-tolerant algorithms; particularly, the overhead of the fault-tolerant algorithms must be measured.

5. Implementation of fault-tolerant algorithms: design a highly scalable model, such that all detected failures can be mitigated at the user level. It must take advantage of the communication-avoiding algorithms to improve the final results.

6. Theoretical analysis of the algorithms: overhead on the complexity, resilience (how many failures it can tolerate, etc.).

7. Design, analysis, and performance evaluation of other algorithms of the same family: it is not enough to take into consideration only a few non-fault-tolerant algorithms. Another set of algorithms must be considered to improve the fault-tolerant approximation.

8. Implementation of fault-tolerant techniques and practical performance evaluation: a scalable implementation requires its recovery time to decrease as the number of processors increases. Therefore, a set-up for experimental conditions must consider a constant failure for each processor. With this, the evaluation must consider the implementation overhead in execution without failures and the implementation overhead when a run-time failure occurs.

9. Apply last optimizations and evaluate performance of implementations: it must be proved that the final implementations can resist failures and be able to mitigate them at run-time, avoiding data losses and gaining efficiency at run-time.

1.3 Thesis Organization

This thesis is mainly organized as follows:

- Chapter 2 briefly presents an overview of the related literature and previous works that have addressed related issues with the current topic.

- Chapter 3 describes a special block-form representation of matrices, depending on the factorization that is going to be applied to a matrix.

- Chapters 4,5 and 6 present the non-fault-tolerant and fault-tolerant algorithms for LU factorization (TSLU/CALU), QR factorization (TSQR/CAQR) and Cholesky factorizations (TSCH/CACH), respectively. The TSLU algorithm includes the *tournament pivoting* algorithm. TSQR follows an approach similar to TSLU, forming an identical communication pattern. TSCH follows a linear communication pattern (only root process 0 distributes data).

- Chapter 7 introduces a formal verification model for the fault-tolerant algorithms, proving resilience and robustness. The presented formal model also verifies the correct termination of the communication-avoiding algorithms.

- Chapter 8 gives a summary about how all algorithms were implemented on a highly parallel and distributed environment.

- Chapter 9 presents how performance evaluations were made on a grid environment to experiment the execution/termination of non-fault-tolerant and fault-tolerant algorithms. It shows the most relevant results reached on large rectangular/square matrices. It provides an evaluation of the overhead injected to non-fault-tolerant algorithms by the fault-tolerant mechanisms during failure-free executions, to show that additional

overhead remains as low as possible. Here is shown an evaluation of the cost when a failure occurs on the overall execution time.

- Chapter 10 concludes the thesis work and provides some perspectives about possible future works on this topic.

CHAPTER 2

State of the Art

2.1 Current Fault Tolerance Approaches

Although reliability and high scalability have been studied for a long time [8], nowadays they have become critical for HPC infrastructures to exploit their power [5]. Fault tolerance has been identified as a major challenge to address towards exascale, due to the Mean Time Between Failures (MTBF), being only a few hours [9]. On the road to exascale [1, 2, 4, 10], the number of processes in the entire system will be millions. Over time, a large range of approaches have been studied to achieve these properties with satisfying performance [3], both at hardware-level and at software-level. Some examples of hardware-level approach include studies for detecting silent errors produced by hardware-level error correcting codes in DRAMS [11], a cache-flushing method to recover crash-consistence in non-volatile-memory for an iterative solver, a dense matrix multiplication and a Monte-Carlo simulation [12], and a strategy on monitoring events at system hardware-level across many computing nodes, replicating the most important events to a fault-tolerant runtime environment with a dynamic checkpointing scheme [13]. Another example can be seen in [14], where silent errors that occur in HPC programs are detected with a low-memory-overhead SDC detector, incurring low-performance overhead.

Considering software-level approaches for fault tolerance in parallel applications, they can be classified into two categories: system-level fault tolerance and application-level fault-tolerance. For system-level fault tolerance, most approaches rely on rollback recovery. A system-level approach is given in [15], proposing compiler instructions for allowing users to specify *checkpoint/restart* operations, supporting from basic to advanced mechanisms currently available on dedicated libraries and the using of fault-tolerance-dedicated threads. Another system-level strategy proposes extensions to the Distem emulator [16], enabling it to evaluate fault tolerance and load balancing mechanisms in real HPC Runtimes Charm++, MPICH, and OpenMPI. In system-level fault tolerance, checkpointing is widely used in large systems [17, 18]. About checkpointing algorithms, the two most important are coordinated checkpointing and non-coordinated checkpointing.

1. Coordinated checkpointing [19] requires a synchronization protocol between the processes that save their local state or snapshot and then, all together to form a global

snapshot or checkpoint. This global checkpoint is a consistent state of the system, allowing the system to recover in case of error, but it also forces all processes to roll back to the last stored state which is part of a complete global checkpoint [20]. Moreover, processes in the system perform a checkpoint operation, either stopping their execution (blocking implementation) or logging the messages that are being sent or received during the checkpoint wave (non-blocking implementation) [17]. This way to save coordinated checkpoints cause significant latency [21].

2. Non-coordinated requires a protocol to make sure the state of the distributed application will still be consistent after a process rolls back. After a roll back, messages sent to and by the failed process between the checkpoint and the failure need to be sent and received again. A first approach would be to force the remote processes to roll back, but it would lead to a *domino effect*: all the processes are likely to be forced to roll back up to the beginning of the execution. In order to avoid this, a solution is to log messages. *Message-logging protocols* have algorithms to reduce the information stored and the latency caused by storing the messages. However, message logging introduces a significant overhead on the communications [22, 23]. Communication-Induced Checkpointing [24] analyzes the causal dependencies between processes introduced by the communications and forces checkpoints when these dependencies would cause a domino effect. However, experimental results show that it forces a lot of checkpoints and therefore, it introduces a high overhead reducing the probability of reaching the desired performance in large-scale infrastructures [25, 26].

For this performance restriction, the application-level fault-tolerance approach proposes a more adaptable variety of solutions, promising better scalability at the cost of significant changes [27, 28]. With application-level fault tolerance, the run-time environment must be able to survive failures and support the rest of the application despite failures, while allowing to restart processes in the desired semantics. An extensively used software technique for fault tolerance is replication, which uses a large number of resources (memory or storage) to replicate relevant information at different levels, ensuring the reliable execution of applications. The most common replication techniques are used for verifying the correctness of computations, but for node failure errors, generating data redundancy to replace failed processes or even equipment seems to work much better [29, 30]. However, due to the loss of data in failed processes, this method often works together with checkpointing.

As MPI has been and continues to be the *de facto* standard to implement distributed applications, some efforts to adding fault tolerance to MPI have been attempted. FT-MPI [31], some recovery modes were available to the user: 1) Blank mode, which consisted in replacing failed processes with MPI_PROC_NULL; 2) Replace mode, which consisted in replacing failed processes with new ones. 3) Shrink mode, which consisted in replacing the corrupted communicator by reordering process ranks in a new communicator. However, despite all efforts, no standardization was attempted and it only served to people as the basis to design new constructs for MPI draft proposals [32, 33].

More recently, and using the knowledge gained from previous works such as FT-MPI, the MPI community proposed a new model for failure mitigation, called User-Level Fault

Mitigation (ULFM) [34, 35]. It provides fault-tolerant instructions to effectively extend MPI. With these fault-tolerant subroutines, the user is able to decide the behaviour his application requires: to "delete" dead processes from communicators after a failure (shrink) and only keep the alive ones, to re-start new processes (spawn) for replacing the dead ones, among other fault-tolerant operations.

On this same axis of application-level fault-tolerant environments we have Fenix [36, 37]. It is a fault-tolerant framework for helping in enabling recovery from process failures in an online and transparent manner, providing mechanisms for capturing failures, re-spawning new processes, fixing failed communicators, restoring the application state and continuing execution. This work relies on implicitly coordinated checkpointing and it can tolerate high failure rates with low overhead while sustaining performance. Diverse approaches for exploring failure recovery using stencil-based parallel applications to mask recovery overheads are presented in [38, 39, 40], showing how multiple failures can be masked to effectively reduce the impact on the total time.

Knowing these facts about application-level approaches, the Algorithm-Based Fault Tolerance (ABFT) approach [41] becomes promising. Essentially, it consists in avoiding "alive" processes to wait when a recovery operation needs to be perfomed (is in progress), but replace "dead" processes and feed them with redundant data to continue the program normal execution. Data redundancy should be ensured to allow that multiple failures can be tolerated, but the data to be replicated also needs to be selected carefully. Considering a full replication is expensive, because to replicate all information in different processes to perform the same operations could degrade the computation power [42]. So different techniques should be considered, such as saving checkpoints or snapshots and store them in a shared memory space [43]. According to [41], fault tolerance typically consists of fault detection/location, which can be done with the runtime environment, and fault recovery, which corresponds to the application for recovering lost data of the failed processes and reconstruct a correct state of the program. The work [44] proposes a scalable fault tolerance mechanism to detect and correct bit-flip errors on-the-fly on a matrix multiplication subroutine, giving less than 12 percent of injected overhead with respect to the failure-free implementation; and more studies have applied ABFT as an approach to solve diverse HPC problems [45, 46, 47].

2.2 Linear Algebra Applications at High Scale

Dense linear algebra applications running on HPC systems have been popular in the last decades and approaches have been adapted to the architecture they are targeting. Several libraries with a variety of algorithms dedicated to linear algebra have been implemented to allow users to take advantage of the computational resources the large-scale infrastructure counts. Table 2.1 lists some of these dedicated libraries.

Programs that require to work with large matrices can be distributed among *process grids* with specific, high-level subroutines that are in charge of managing the hardware resources available on the machine, freeing final users from this responsibility. As MPI has become the *de facto* interface for programming parallel applications on distributed architectures, programmers have taken advantage of MPI operations to develop and optimize linear al-

Table 2.1: Some linear algebra dedicated libraries capable to run on large-scale architectures.

Name	Description	Written in	First Release	Last Release
ScaLAPACK[1]	Contains high-performance linear algebra routines for parallel distributed memory; can use a block cyclic data distribution for dense matrices and a block data distribution for banded matrices; low-level modular components for parallelizing high level routines	Fortran, C	February 28, 1995	November 16, 2019
PLASMA[2]	Project to address linear algebra programs to reach the scalability on multi-core architectures; pretends to create frameworks that enable programmers to simplify the process of developing applications on new architectures	C	November 10, 2008	August 6, 2019
MAGMA[3]	Project that aims to develop a dense linear algebra library for heterogeneous/hybrid architectures; design linear algebra algorithms and frameworks for hybrid manycore and GPUs systems	Fortran, C++	September 14, 2009	July 13, 2021

gebra libraries [48]. However, the overhead generated due to the high cost of inter-process communications has grown more and more [49]. Complex hardware architectures prevent most of the existing algorithms to scale satisfactorly as the number of processes increases, and main memory size is a constrain when the amount of data (a matrix) is large enough to not fit directly on it [50]. Then, inter-process communication patterns need to be refactored directly in the design of algorithms implemented on dense linear algebra applications [51].

Recent advances on computation kernels for dense linear algebra for computing matrix factorizations use a (proven) minimal number of inter-process communications, overlapping (as much as possible) communication with computation [7, 52]. The core idea behind these algorithms is to minimize the number of messages sent by processes, reducing the total amount of data exchanged between them [53]. Minimizing communications is performed to the cost of some additional computations. However, in most situations, minimizing communications improves performance, because of the (growing) imbalance between communication and computation costs [54]. Several works propose a variety of solutions applied to dense linear algebra programs running on modern parallel computers [55, 56].

Focusing in matrix factorizations, it has been shown that the use of a tree-based algorithm can be exploited to obtain a fault-tolerant panel factorization. Then is possible to take advantage of algebraic properties on the trailing matrix update operation [57, 58], using matrix block form representations [59]. These results lead to more efficient implementations of matrix factorization. Similar approaches are promising in the sense that it is necessary to introduce as little modification as possible in the critical path, and practical experiments with these approaches show a small overhead [60, 61, 62].

Finally, different solutions have been proposed to check if a dense linear algebra algorithm can run without failures over an HPC system, over CPUs [63] or GPUs [64, 65]. It has been shown that numerical algorithms running in HPC have two costs: arithmetic and communication costs; communication costs often dominate arithmetic costs [66].

CHAPTER 3

Block Form Representations for Matrix Factorizations

In matrix factorizations or decompositions, we aim at factoring a given matrix A as the product of two matrices, generally finding a lower triangular matrix (all elements above the diagonal are zero) and an upper triangular matrix (all the elements below the diagonal are zero).

This way to decompose a matrix makes it simple to compute the solution of many problems associated with linear systems (Krylov subspaces, Least-Square Problem, etc.). It is also the first step to find the inverse of a matrix, or when computing the determinant of a matrix. In particular, solving a linear system of type $Ax = b$ gets reduced to solving smaller triangular systems.

A matrix factorization can be represented as the matrix form of Gaussian elimination. In a general view, a factorization can be seen as:

$$A = \begin{bmatrix} a_{0,0} & 0 & 0 \\ \vdots & \ddots & 0 \\ a_{m,0} & \dots & a_{m,n} \end{bmatrix} \begin{bmatrix} b_{0,0} & \dots & b_{0,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & b_{m,n} \end{bmatrix} = \begin{bmatrix} a_{0,0}b_{0,0} & \dots & a_{0,0}b_{0,n} \\ \vdots & \ddots & \vdots \\ a_{m,0}b_{0,0} & \dots & a_{m,0}b_{0,n} + \ldots + a_{m,n}b_{m,n} \end{bmatrix}$$

In modern machines with finite precision, it is very usual to solve square systems of linear equations using decompositions. It is important to consider that computed factors could or could not be numerically stable, meaning that minimal errors in the initial factors can cause a larger error in the final result. For matrices with specific properties, such as diagonal dominance by rows or columns, numerical stability is guaranteed, but in general, it is necessary to incorporate row interchanges, or row/column interchanges, to obtain a stable factorization.

Nowadays, current high-performance architectures are designed with hierarchical memories (cache, RAM, distributed memory, etc.). When we want to take advantage of modern machines and achieve a higher throughput on matrix decompositions, it is recommendable to implement them in a *block form*, expressing decompositions in terms of matrix multiplications and the solution of multiple right-hand side triangular systems. In this work, we use a right-looking and panel-update algorithm for partitioning a matrix and updating the trailing matrix, respectively.

3.1 LU Block Form

Supposing we have a matrix $A \in \mathbb{R}^{M \times N}$ and we want to apply the LU factorization on it, dividing the matrix in a block form with blocks of size $b \times b$, matrix A can be represented as follows:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & 0 \\ L_{1,0} & L_{1,1} \end{bmatrix} \begin{bmatrix} U_{0,0} & U_{0,1} \\ 0 & U_{1,1} \end{bmatrix}$$

Then, we can obtain equations:

$$A_{0,0} = L_{0,0}U_{0,0}, A_{0,0} \in \mathbb{R}^{b \times b} \tag{3.1}$$

$$A_{1,0} = L_{1,0}U_{0,0}, A_{1,0} \in \mathbb{R}^{M-b \times b} \tag{3.2}$$

$$A_{0,1} = L_{0,0}U_{0,1}, A_{0,1} \in \mathbb{R}^{b \times N-b} \tag{3.3}$$

$$A_{1,1} = L_{1,0}U_{0,1} + L_{1,1}U_{1,1}, A_{1,1} \in \mathbb{R}^{M-b \times N-b} \tag{3.4}$$

Taking $A_{0,0}$ and $A_{1,0}$ together, we can form a *panel* of A, that is the left-most part of the matrix. Then, with equations 3.1 and 3.2 we can perform an LU factorization on the panel. At the end of this computation, matrices $L_{0,0}$, $L_{1,0}$ and $U_{0,0}$ are known. The lower triangular system in 3.3 can be solved to give $U_{0,1}$. Rearranging 3.4 as:

$$A'_{1,1} = L_{1,1}U_{1,1} = A_{1,1} - L_{1,0}U_{0,1}, A'_{1,1} \in \mathbb{R}^{M-b \times N-b} \tag{3.5}$$

we can realize that the problem of finding $L_{1,1}$ and $U_{1,1}$ gets reduced to find the LU factorization of the matrix $A'_{1,1}$, that is known as the *trailing matrix* and the operation of calculating the LU factorization is known as the *trailing matrix update*. This update can be done by repeating the same procedure over $A'_{1,1}$ instead of the complete matrix A. At the end, $L_{0,0}/L_{1,1}$ are lower triangular with ones on the diagonal and $U_{0,0}/U_{1,1}$ are upper triangular. In figure 3.1 is shown a LU block-form of a matrix A and figure 3.2 represents a matrix being partitioned in sub-blocks $A_{i,j}$, with $i \in [0, rowM]$ and $j \in [0, colN]$.

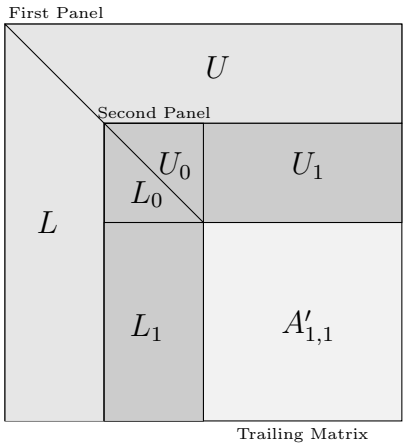


Figure 3.1: LU block form representation of a matrix.

	0	1	2	$\cdots$	$colN$
0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$\cdots$	$A_{0,colN}$
1	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,colN}$
2	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$\cdots$	$A_{2,colN}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$rowM$	$A_{rowM,0}$	$A_{rowM,1}$	$A_{rowM,2}$	$\cdots$	$A_{rowM,colN}$

Figure 3.2: Rectangular matrix partitioning by sub-blocks (sub-matrices), where $rowM = M - 1$ is the number of block-rows and $colN = N - 1$ is the number of block-columns.

3.2 QR Block Form

Supposing we have a matrix $A \in \mathbb{R}^{M \times N}$ and we want to apply the QR factorization on it, dividing the matrix in a block form with blocks of size $b \times b$, matrix A can be represented

as follows:

$$A = \begin{bmatrix} A_0 & A_1 \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = Q \begin{bmatrix} R_{0,0} & R_{0,1} \\ 0 & R_{1,1} \end{bmatrix}$$

where $A_{0,0} \in \mathbb{R}^{b \times b}$, $A_{1,0} \in \mathbb{R}^{M-b \times b}$, $A_{0,1} \in \mathbb{R}^{b \times N-b}$, $A_{1,1} \in \mathbb{R}^{M-b \times N-b}$, $R_{0,0} \in \mathbb{R}^{b \times b}$ is upper triangular, $R_{0,1} \in \mathbb{R}^{b \times N-b}$ and $R_{1,1} \in \mathbb{R}^{M-b \times N-b}$.

We can see that $A_0 = \begin{bmatrix} A_{0,0} \\ A_{1,0} \end{bmatrix}, A_0 \in \mathbb{R}^{M \times b}$ is a panel of A and it contains the first b left-columns of matrix A, and $A_1 = \begin{bmatrix} A_{0,1} \\ A_{1,1} \end{bmatrix}, A_1 \in \mathbb{R}^{M \times N-b}$ contains the remaining columns of A.

To compute Q, a series of *Householder transformations* can be applied to A_0, in the form:

$$H_i = I - \tau_i v_i v_i^T, i \in [0, b] \tag{3.6}$$

It can be shown that $Q = H_0 H_1 \cdots H_b = I - VTV^T$, where $T \in \mathbb{R}^{b \times b}$ is upper triangular and the column i of V equals v_i [67, 68]. Defining A'_1 as:

$$A'_1 = Q^T A_1 = (I - VT^T V^T) A_1 \tag{3.7}$$

reduces the problem to only find the QR factorization on A'_1, that is the trailing matrix, instead of the complete matrix A. This operation can be repeated again and again over the remaining trailing matrix until there are no more blocks to compute. This way to represent the QR factorization in a block form gives very good performances in matrix operations. In figure 3.3 is shown a QR block-form of a matrix A.

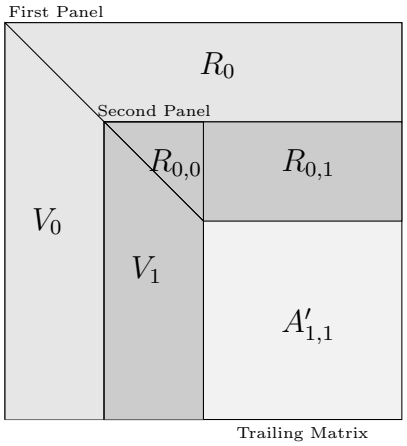


Figure 3.3: QR block form representation of a matrix.

3.3 Cholesky Block Form

Supposing we have a matrix $A \in \mathbb{R}^{M \times N}$ and we want to apply the Cholesky factorization on it, dividing the matrix in a block form with blocks of size $b \times b$, matrix A can be represented as follows:

$$A = \begin{bmatrix} A_{0,0} & A_{1,0}^T \\ A_{1,0} & A_{1,1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & 0 \\ L_{1,0} & L_{1,1} \end{bmatrix} \begin{bmatrix} L_{0,0}^T & L_{1,0}^T \\ 0 & L_{1,1}^T \end{bmatrix}$$

Then, we can obtain equations:

$$A_{0,0} = L_{0,0}L_{0,0}^T, A_{0,0} \in \mathbb{R}^{b \times b} \tag{3.8}$$

$$A_{1,0} = L_{1,0}L_{0,0}^T, A_{1,0} \in \mathbb{R}^{M-b \times b} \tag{3.9}$$

$$A_{1,0}^T = L_{0,0}L_{1,0}^T, A_{0,1} \in \mathbb{R}^{b \times N-b} \tag{3.10}$$

$$A_{1,1} = L_{1,0}L_{1,0}^T + L_{1,1}L_{1,1}^T, A_{1,1} \in \mathbb{R}^{M-b \times N-b} \tag{3.11}$$

where $A_{0,0}$ and $A_{1,1}$ are lower triangular matrices. If we compute the Cholesky factorization on $A_{0,0}$, then the lower triangular Cholesky factor $L_{0,0}$ will be known and, from equations 3.9 and 3.11, we can obtain:

$$L_{1,0} = A_{1,0}(L_{0,0}^T)^{-1} \tag{3.12}$$

$$A'_{1,1} = A_{1,1} - L_{1,0}L_{1,0}^T = L_{1,1}L_{1,1}^T \tag{3.13}$$

Now, the trailing matrix is $A'_{1,1}$ and the factorization can be completed by recursively applying the steps described above over $A'_{1,1}$. In figure 3.4 is shown a Cholesky block-form of a matrix A.

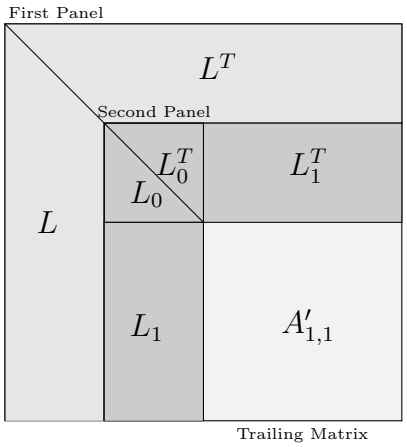


Figure 3.4: Cholesky block form representation of a matrix.

CHAPTER 4

LU Factorization

The LU factorization factors a given matrix A as the product of two matrices L and U as $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. This decomposition helps in reducing the complexity of solving a linear system $Ax = b$, simplifying a triangular system as $Ly = b$ and $Ux = y$.

Since not all matrices can be factorized in the LU form (non-singular matrices in the form $AB = BA = I$ cannot), we need a way to ensure that the LU factorization can be completed satisfactorily on a given matrix. For example, as the first step for finding the upper triangular matrix U is with a *Gaussian elimination* method and it is not a stable method, LU uses *pivoting* to improve stability by using the largest rows as pivots. Thus, if required, the resulting LU product can include a permutation matrix. This way, we can apply row and/or column interchanges to ensure that the pivots are nonzero (unless the pivot is already triangular). When we represent row permutations only, then the factorization is known as *LU factorization with partial pivoting* and is given by $PA = LU$ or $A = P^T LU$, where L and U are again lower and upper triangular matrices and P is a permutation matrix representing row re-orderings or permutations. If P is left-multiplied to A, the result will be the same initial matrix A, but with re-ordered rows. This technique is numerically stable, making LU decomposition with partial pivoting a very useful technique in practice. Another technique is the *LU factorization with full pivoting*. In this case, the decomposition involves both row/column permutations and the resulting product includes two extra matrices P and Q, as $PAQ = LU$, where L, U and P are defined as before, and Q is a permutation matrix representing column re-orderings or permutations.

As mentioned in section 3.1, when we want to obtain a higher throughput on LU decomposition, then it is recommendable to implement it in a *block form*, decomposing the matrix into blocks and distributing them between available processes, allowing each process to have one block. An example of a block decomposition is given in chapter 3, and it is the one that is going to be used for the algorithms presented in this dissertation. In the next sections, we will see how the LU block form can be used to factorize a matrix A.

4.1 TSLU and FT-TSLU

4.1.1 TSLU

A Tall and Skinny (TS) matrix is a matrix $A \in \mathbb{R}^{M \times N}$ with a specific shape; it has a large number of rows M and a small number of columns N; this is $M \gg N$. The Tall and Skinny LU (TSLU) factorization of a tall and skinny matrix A is the cornerstone of the LU factorization of a general square or rectangular matrix [7]. It can also be used alone by applications involving a few vectors in a high-dimension space. In this case, the parallel algorithms use a 1D data decomposition: columns are distributed between processes that hold the full lines.

The first phase finds *pivot rows* to improve the numerical stability of the overall computation, as mentioned in section 3.1. TSLU uses a set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ and a communicator $comm_p$ to compute the LU factorization of matrix A in parallel. At the first step $s = 0$, it decomposes the $M \times N$ matrix A into $b \times N$ block-rows, with $b = \frac{M}{t}$. Every resulting sub-matrix or sub-block $A_{i,s}$ is sent to its corresponding process P_i to compute the LU factorization of each sub-matrix independently. Then an exchange is performed between pairs of processes $\{P_i, P_j\}$ to group sub-matrices into successive pairs $U_{i,s}$ and $U_{j,s}$, to execute again the LU factorization algorithm over the grouped $U_{\{i,j\},s+1}$ pairs in parallel. In the case of process P_i, it receives the resulting sub-matrix process P_j sends. This procedure is repeated until there is only one final $U_{i,s}$ matrix left, which contains the best rows found on the whole initial matrix A. Process 0 broadcast the final matrix $U_{i,s}$. This operation, known as *tournament pivoting*, aims at finding at low communication cost the best b row-pivots that can be used to factor the entire matrix A and put them on the final matrix U. This technique gives good performance because it depends primarily on the size of the rows in A and uses a minimal number of inter-process communications. When the size of the tall and skinny matrix is considerable, *tournament pivoting* reduces the complexity of calculating the LU Factorization of the whole matrix A. Algorithm 1 describes in detail how the steps of the classical TSLU algorithm are executed in order to find final $U_{i,s}$ and $L_{i,s}$ matrices.

As can be seen in lines 5, 7, 9, 13 and 16 of the algorithm, there are five important functions that allow TSLU to work correctly:

- *myPartner(s)*: assigns a new partner (neighbor) j to the local process at step s. The value to the next partner can be assigned with equation 4.1.

$$j = \begin{cases} i + 2^s, \text{if } i \mod 2^{s+1} == 0 \\ i - 2^s, otherwise \end{cases} \tag{4.1}$$

  where i, j represent processes rank and s the step number.

- *send($U_{i,s}$, $comm_p$)*: sends the given matrix $U_{i,s}$ to the partner in communicator $comm_p$.

- *recv($U_{j,s}$, $comm_p$)*: receives the given matrix $U_{i,s}$ from the partner in communicator $comm_p$.

Algorithm 1: TSLU

Data: Sub-matrix $A_{i,0}$, Communicator $comm_p$, Integer i

```
1  s = 0 ;
2  myrank = i ;
3  U_{i,s} = LU(A_{i,0});
4  while !done() do
5      j = myPartner(s) ;
6      if myrank < j then
7          f = recv(U_{j,s}, comm_p) ;
8      else
9          f = send(U_{i,s}, comm_p) ;
10         break;
11     if FAIL == f then
12         return;
13     A_{i,s} = concatenate(U_{i,s}, U_{j,s});
14     s = s + 1 ;
15     U_{i,s} = LU(A_{i,s});
16 broadcast(U_{i,s}, comm_p, Process with rank 0) ;
17 L_i = A_{i,0} \ U_{i,s} ;
18 return L_i, U_{i,s};
```

- *concatenate($U_{i,s}$, $U_{j,s}$)*: concatenates both $U_{i,s}$ matrices, putting $U_{i,s}$ on top of $U_{j,s}$. Equation (4.2) represents the concatenation of both matrices.

$$U_{\{i,j\},s} = \begin{cases} U_{i,s}||U_{j,s}, \text{if } i < j \\ U_{j,s}||U_{i,s}, \text{if } i > j \end{cases} \tag{4.2}$$

- *broadcast($U_{i,s}$, $comm_p$, r)*: process with rank r distributes the given matrix $U_{i,s}$ to all processes in communicator $comm_p$. Processes with rank different from r only receive the matrix. This function is also used in future algorithms (3, 4, 7, 8, 11, 12) to distribute intermediate results in lines/columns, providing as source process the processes with rank equal to the current column number to be computed (see figures 3.2, 8.8).

Figure 4.1 shows a TSLU example with $t = 4$ processes. We can see that each process P_i starts with its own sub-matrix $A_{i,s}$ to compute a local LU decomposition, generate local sub-matrices $L_{i,s}$, $U_{i,s}$. Process P_i gets ready to receive the sub-matrix from P_j, while process P_j sends its results to its current partner P_i (in the image, process P_0 receives data from process P_1 and process P_2 receives from P_3). After process P_j has sent its results, it finishes its work (line 10) and remains waiting for the broadcast from process 0 (in the image, P_1 and

P_3 keep waiting). Then, process P_i reassings its new partner and receives new data from it. This operation is repeated until process 0 has collected all sub-matrices with the best-row pivots. At final step, process 0 broadcast the final matrix $U_{i,s}$ to waiting processes. At the end of the execution, every single process P_i has the same final $U_{i,s}$ matrix.

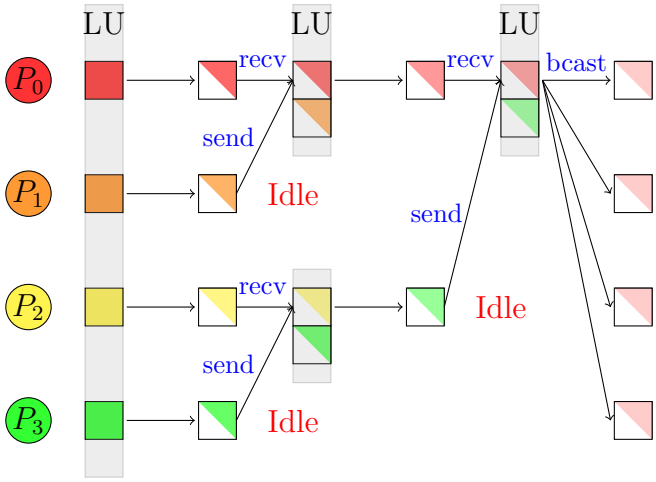


Figure 4.1: TSLU failure-free example with $t = 4$ processes. Gray bar represents the moment when a LU factorization is performed.

The processes assignment to communicate with each other in TSLU has been chosen as a binary tree communication pattern, in which the computation of a process depends on the data computed by all subtrees rooted in this process. The first A_i sub-blocks of the binary tree represent the leaves and the final $U_{i,s}$ matrix represents the root tree. Thus, the total number of steps that will take place at run-time is limited by equation 4.3:

$$T_{steps} = \log_2 t + 1 \tag{4.3}$$

where t is the total number of processes involved in the computation.

Following the previous example, with $t = 4$ and $T_{steps} = \log_2 t + 1 = 3$, the TSLU matrix algebra can be represented as follows:

- Step 0: $A = \begin{bmatrix} A_{0,0} \\ A_{1,0} \\ A_{2,0} \\ A_{3,0} \end{bmatrix} = \begin{bmatrix} L_{0,0}U_{0,0} \\ L_{1,0}U_{1,0} \\ L_{2,0}U_{2,0} \\ L_{3,0}U_{3,0} \end{bmatrix} = \begin{bmatrix} L_{0,0} & 0 & 0 & 0 \\ 0 & L_{1,0} & 0 & 0 \\ 0 & 0 & L_{2,0} & 0 \\ 0 & 0 & 0 & L_{3,0} \end{bmatrix} \cdot \begin{bmatrix} U_{0,0} \\ U_{1,0} \\ U_{2,0} \\ U_{3,0} \end{bmatrix}$

- Step 1: $\begin{bmatrix} U_{0,0} \\ U_{1,0} \\ \hline U_{2,0} \\ U_{3,0} \end{bmatrix} = \begin{bmatrix} U'_{\{0,1\},1} \\ U'_{\{2,3\},1} \end{bmatrix} = \begin{bmatrix} L'_{\{0,1\},1}U_{\{0,1\},1} \\ L'_{\{2,3\},1}U_{\{2,3\},1} \end{bmatrix} = \begin{bmatrix} L'_{\{0,1\},1} & 0 \\ 0 & L'_{\{2,3\},1} \end{bmatrix} \cdot \begin{bmatrix} U_{\{0,1\},1} \\ U_{\{2,3\},1} \end{bmatrix}$

- Step 2: $\begin{bmatrix} U_{\{0,1\},1} \\ U_{\{2,3\},1} \end{bmatrix} = U'_{\{0,1,2,3\},2} = L'_{\{0,1,2,3\},2}U_{\{0,1,2,3\},2}$

At each step s lower triangular matrices $\{L_{i,s}\ ,\ L'_{i,s}\}$ are generated too. Since solving a linear system of equations only requires the use of the final matrix $U_{i,s}$, the $\{L_{i,s},\ L'_{i,s}\}$ matrices can be set aside and, if required, they can be used to compute the final matrix $L_{i,s}$ by multiplying the intermediate $L_{i,s}$ matrices of all the steps. Of course, this extra computation requires more space and computing time. The algebra to generate the final matrix $L_{i,s}$ in the previous example is:

$$L_{\{0,1,2,3\},2} = \begin{bmatrix} L_{0,0} & 0 & 0 & 0 \\ 0 & L_{1,0} & 0 & 0 \\ 0 & 0 & L_{2,0} & 0 \\ 0 & 0 & 0 & L_{3,0} \end{bmatrix} \cdot \begin{bmatrix} L'_{\{0,1\},1} & 0 \\ 0 & L'_{\{2,3\},1} \end{bmatrix} \cdot L'_{\{0,1,2,3\},2}$$

The matrix multiplications involved in the final $L_{i,s}$ are quite expensive, but there are two ways to carry them out:

- On-the-fly at each step: each time an exchange has been completed, we have access to $\{L_{i,s}, L_{j,s}\}$ matrices; then we generate the $L'_{\{i,j\}}$ from $U'_{\{i,j\}}$. The matrix multiplication can be done when the three matrices are present at each process, generating the $L_{i,s}$ of the current step. This way reduces the amount of memory needed to store intermediate results but increases steps' execution times.

- At the final step: all intermediate results $\{L_{i,s}, L_{j,s}\}$ can be stored on the memory, increasing the amount of memory needed by the algorithm. In the end, when the final $U_{i,s}$ has been computed, we can solve a linear system $Ax = b$ using the initial matrix A_{init} and the final matrix $U_{i,s}$ as $A_{init}x = U_{i,s}$ to compute the final matrix $L_{i,s}$. So the operation $L_{i,s} = A_{init} \backslash U_{i,s}$ can be done in order to find efficiently the final lower triangular matrix.

Knowing these facts about TSLU, we will introduce in the next section its the fault-tolerant version.

4.1.2 FT-TSLU

The Fault-Tolerant Tall and Skinny LU (FT-TSLU) algorithm is the fault-tolerant version of TSLU. FT-TSLU relies on a fault-tolerant run-time environment to detect crash-type process failures at run-time and restore the communicators. When an error is detected, the failed processes are respawned, the communicator used by processes to exchange information is repaired and the current computation is recovered. To be able to recover the computation, the algorithm keeps track of the states of the matrices computed at each step (except the last one), storing them over the results generated at the previous step, thus enabling all the processes to share their previous known results with a restored process if necessary. Algorithm 2 shows the FT-TSLU algorithm, adding some operations to the TSLU original algorithm.

The basic pattern is similar to its non-fault-tolerant version. The main difference is that instead of having a sender and a receiver (the sender ending its participation after

Algorithm 2: FT-TSLU

```
   Data: Sub-matrix A_{i,0}, Communicator comm_p, Integer i
 1 s = 0 ;
 2 if I am a spawned process then
 3 |  U_{i,s} = update(s, comm_p) ;
 4 else
 5 |  U_{i,s} = LU(A_{i,0});
 6 while !done() do
 7 |  j = myPartner(s) ;
 8 |  f = sendRecv(U_{i,s}, U_{j,s}, comm_p) ;
 9 |  if FAIL == f then
10 |  |  restoreFailed(s, U_{i,s}, comm_p) ;
11 |  |  continue;
12 |  A_{i,s} = concatenate(U_{i,s}, U_{j,s});
13 |  s = s + 1 ;
14 |  U_{i,s} = LU(A_{i,s});
15 |  backup(U_{i,s}) ;
16 L_i = A_{i,0} \ U_{i,s} ;
17 return L_i, U_{i,s};
```

the communication), processes exchange local $U_{i,s}$ intermediate matrices and, owning the same concatenated matrix, compute the same intermediate $U_{i+1,s}$. As a result, intermediate computations are replicated taking advantage of processes that would have otherwise been idle. But the most significant changes reside in the new added functions to fully correct the failed processes, as can be seen in lines 3, 8, 10 and 15 of algorithm 2:

- *update(s, $comm_p$)*: when a process p has been re-spawned, it must receive from another process in communicator $comm_p$ the current state of the $U_{p,s}$ matrix, as well as the current step s.
- *sendrecv($U_{i,s}$,$U_{j,s}$, $comm_p$)*: performs an exchange between partners in communicator $comm_p$.
- *restoreFailed(s, $U_{p,s}$, $comm_p$)*: when a crash has been detected, all surviving processes try to restore the failed ones in communicator $comm_p$, sending the step s in which the error has occurred and the $U_{p,s}$ matrix. Here it must be noted that all processes do not hold the same information, so every process can share its previous results only with its previous partners. When the previous partners are part of the surviving processes, a branch has been lost and the new processes must be spawned from the beginning. The initial matrix should be sent to the restored processes.

- *backup($U_{p,s}$)*: performs a backup operation of the $U_{p,s}$ matrix, overwriting previous results. Backup can be performed in different ways:
  - As global storage: in case the node dies, the information can be recovered from anywhere.
  - As local storage: in case the node dies, the information cannot be recovered. However, a strategy can be carried out. Scheduling processes in a way that the same replicas are located on different machines, information cannot be lost.

As TSLU, FT-TSLU uses the same set of processes and the same communication pattern to exchange intermediate results. The initial distribution of matrix differs from the original one; instead of decomposing the $M \times N$ matrix into $b \times N$ block-rows and distributing them to its corresponding process, it shares the whole matrix with all the processes and every process P_i works on its corresponding sub-block. The mechanism was designed in this way to create more data redundancy and thus to enable every process to restore and share the initial information to all the processes from the beginning, regardless of its rank. Figure 4.2 shows a FT-TSLU example with $t = 4$ processes. We can see that each process P_i starts with its own sub-matrix $A_{i,s}$ to compute a local LU decomposition, generate local sub-matrices $L_{i,s}$, $U_{i,s}$ and share its results with its current partner P_j (in the image, process P_0 exchanges data with process P_1, process P_2 with P_3; then process P_0 with P_2 and process P_1 with P_3). After an exchange has been successfully completed, both processes P_i and P_j execute exactly the same operations, generating the same data. In other words, every pair of processes generates independently the same $U_{i,s}$ matrix, for later applying again the LU decomposition over the newly $U_{i,s}$ matrix and share its own results with its new assigned partner at next step $s+1$. At the end of the execution, every single process P_i has the same final $U_{i,s}$ matrix. Exchanging results between pairs of processes generates data redundancy; thus, FT-TSLU is provided with some level of fault tolerance, in the sense that if a process that has exchanged data already with its current partner fails, there will be another process that holds the same data, giving the algorithm a chance to continue with the computation.

Figures 4.3, 4.4, 4.5, 4.6 and 4.7 show a failure occurring on process P_2 at different steps in the algorithm.

- Figure 4.3: the error occurs at the final step. Only the failed process cannot complete the execution and cannot obtain the final result. The other processes continue their normal execution. Every surviving process is able to deliver the final $U_{i,s}$ matrix.

- Figure 4.4: the error occurs at a middle step. The pair of processes in which the error was detected are not able to complete the execution. First, process P_2 fails; then, process P_0 tries to exchange data with it, but it realizes that its partner has died, so it fails too since it cannot complete the whole computation. The other processes continue their normal execution. Every surviving process can deliver the final $U_{i,s}$ matrix.

- Figure 4.5: the error occurs at the first step. All processes fail and the final result cannot be reached. First, process P_2 fails; then, process P_3 tries to exchange data with the failed process P_2, but it realizes that its partner has died, so it will fail too since

it cannot complete the whole computation. The other processes continue their normal execution until a middle step is reached. Then, when they want to exchange data with their newly assigned partners, they realize their partners have died, so they fail too since they cannot complete the whole computation. There are no surviving processes and hence, the final $U_{i,s}$ matrix cannot be delivered.

- Figure 4.6: in this case, two errors occur at the middle step, in the same branch. Process P_2 and P_3 fail; then, process P_0 and P_1 try to exchange data with their corresponding partner, but they realize their partner has died, so they fail too since it is impossible to complete the computation. There are no surviving processes, so the final $U_{i,s}$ matrix cannot be delivered.

- Figure 4.7: again, two errors occur at the middle step, but now in different branches. Processes P_1 and P_2 fail; then, process P_0 and P_3 try to exchange data with their corresponding partner, but they realize their partner has died and they fail too. Once more, there are no surviving processes, and the final $U_{i,s}$ matrix cannot be delivered.

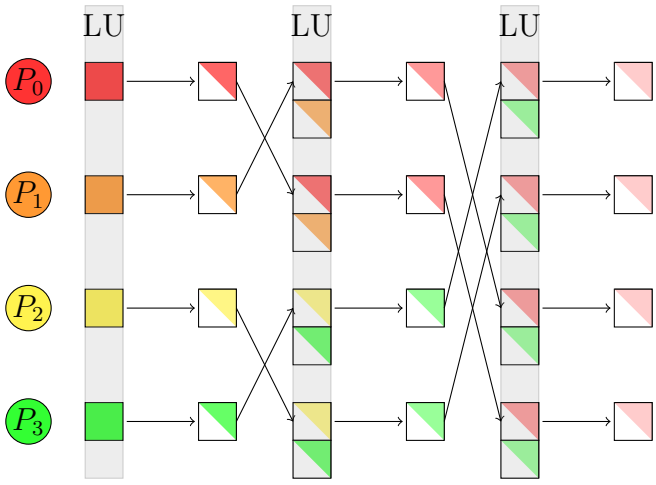


Figure 4.2: FT-TSLU failure-free example with $t = 4$ processes. Gray bar represents the moment when a LU factorization is performed.

As can be seen from the previous descriptions, depending on the moment a process fails, it changes the normal execution of the algorithm and it affects in different ways the execution of other processes too. The failed process stops its execution and the data it was processing is lost. As a consequence, it will not be able to communicate (send or receive) with its partner and therefore, the partner will not send or receive the data it needs to proceed with the computation. Thus, all the processes that (in a moment in time) could be designated as a partner of the failed process, and the future partners of its current designated partner will not be able to process the completed data and hence they will be forced to finish their execution too. This behavior does not always crash the execution if failures appear at the final step or a middle step, in different branches; but if failures start appearing from the

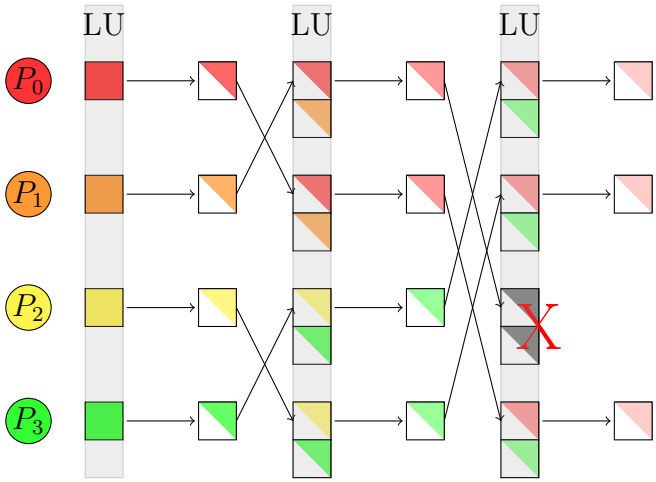


Figure 4.3: FT-TSLU example with $t = 4$ processes, showing one failure at the last step.

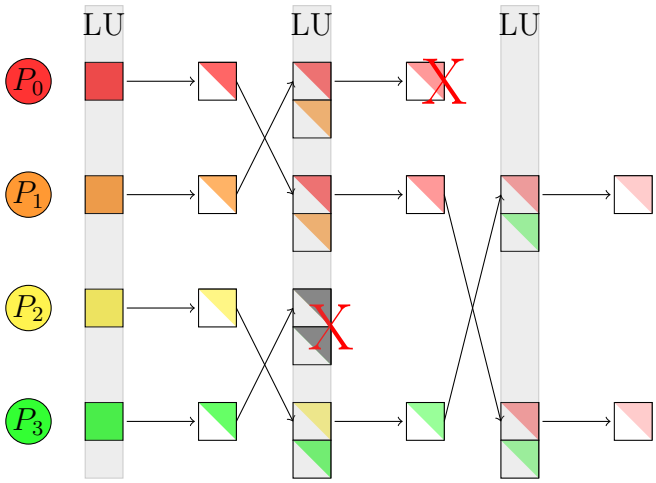


Figure 4.4: FT-TSLU example with $t = 4$ processes, showing one failure at the middle step.

beginning, all processes could be forced to stop their processing and computation could be terminated. So, at each successful data exchange, FT-TSLU doubles the number of failures it can tolerate, but also the longer it has been running, the more processes might fail. For example, at the first step $s = 0$ no failures can be tolerated; at the middle step $s = 1$ it is allowed to tolerate only one failure; at the final step $s = 2$ it is allowed to tolerate three failures. So it can be seen that after an exchange has been finalized, the level of redundancy doubles and the number of failures that can be supported are:

$$T_{failures} = 2^s - 1 \tag{4.4}$$

where s is the step number.

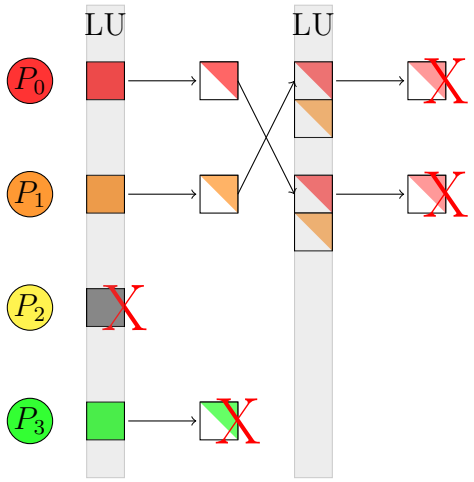


Figure 4.5: FT-TSLU example with $t = 4$ processes, showing one failure at the first step.

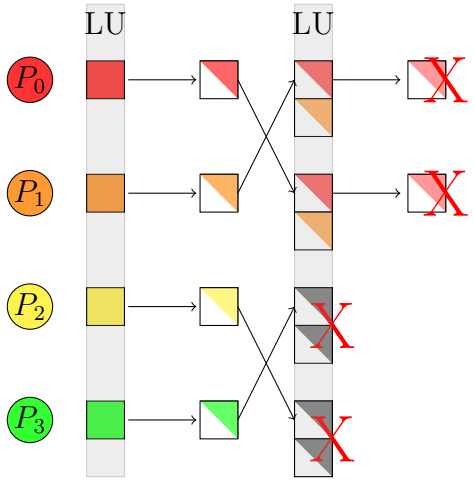


Figure 4.6: FT-TSLU example with $t = 4$ processes, showing two failures in the same branch at the middle step.

Another example of execution can be seen in figure 4.8: the failure occurs at the first step on process P_2 inducing the loss of the sub-block A_2; hence, it induces the whole procedure to fail. As no communication between processes has been carried out, if the initial matrix were not distributed to all processes, no process could restore the failed one (or even the failed ones). Another example can be seen in figure 4.9, but this time with an error in a middle step.

Thanks to these four new operations FT-TSLU can detect and handle the process failures at run-time, avoiding loss of data (branches in the binary tree), and ensuring the completion of the LU decomposition. Appendix A shows a numerical example of the execution by each process of the algorithms described.

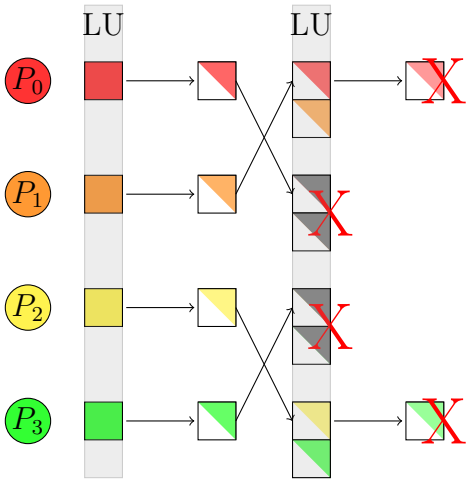


Figure 4.7: FT-TSLU example with $t = 4$ processes, showing two failures in different branches at the middle step.

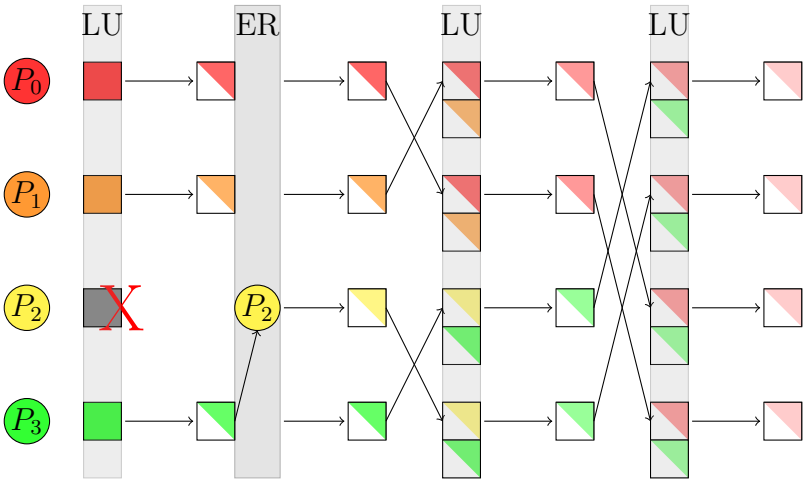


Figure 4.8: FT-TSLU recovering example with $t = 4$ processes, showing one failure at the first step.

4.2 CALU and FT-CALU

4.2.1 CALU

The Communication-Avoiding LU (CALU)[54] algorithm factors a wider (not necessarily square) matrix A into the product of two matrices L and U by iterating over block-column sub-matrices, as defined in section 3.1 and figure 3.1. Since a panel can be seen as a tall and skinny matrix, CALU uses TSLU to compute the LU factorization of each panel.

Initially, CALU divides the potentially square matrix $A \in M \times N$ into smaller sub-blocks

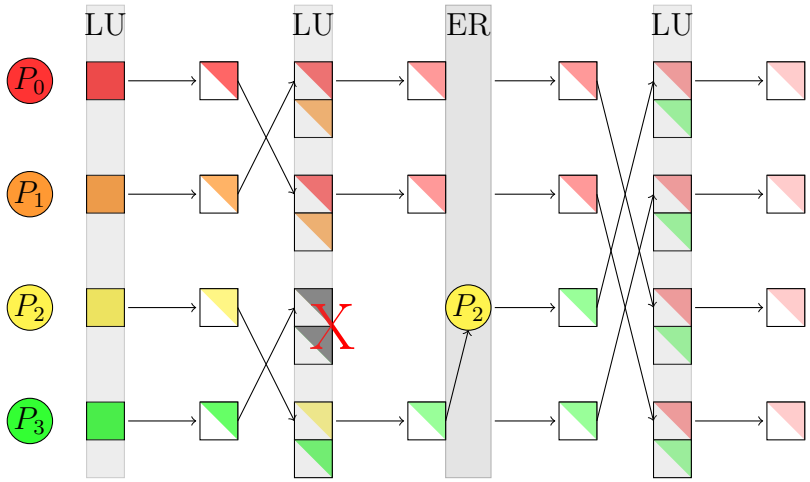


Figure 4.9: FT-TSLU recovering example with $t = 4$ processes, showing one failure at the middle step.

$A_{i,j}$ of size $b \times b$ each, forming a grid with $\frac{M}{b}$ lines and columns, as seen in figure 3.2. At each iteration, it forms a panel and computes the LU factorization on it, finding the best row pivots on the current panel and updating the trailing matrix, for later repeating the same procedure with the next non-processed panel in the matrix until the trailing matrix is small enough to execute a last single process LU factorization and complete the whole matrix. It must be noted that the current panel is also the current column *currCol* to be processed on the grid (see figure 8.8). Figure 4.10 displays an example of how the CALU algorithm partitions a square matrix into sub-blocks to accomplish the LU decomposition.

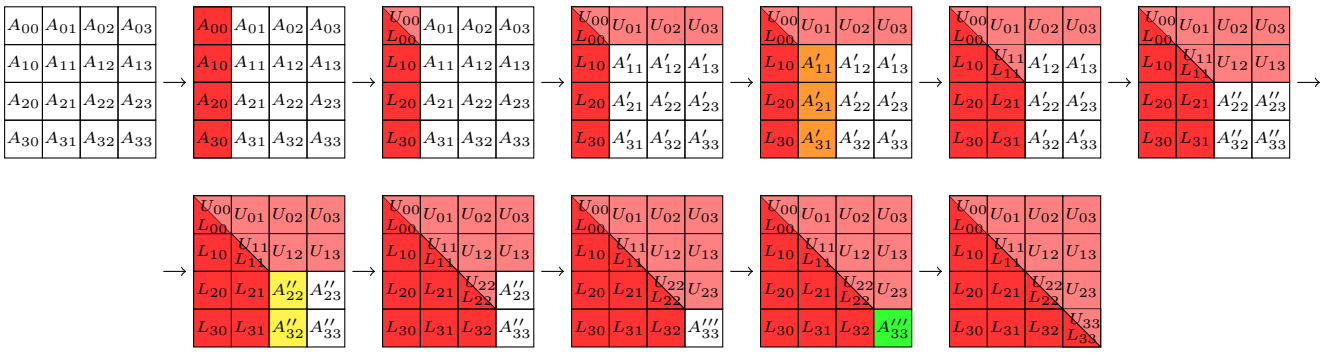


Figure 4.10: Square matrix partition by sub-blocks, panels selection and trailing matrix update example.

CALU relies on a 2D data distribution (or a 2D block-cyclic one). Processes belonging to the same column are in charge of decomposing the panel corresponding to their column number. During the pivoting step, the permutation matrices are broadcast on the process

line to swap the whole lines of the matrix. Then, after a panel decomposition, every process involved in the computation broadcasts its resulting $L_{i,j}$ sub-block on its row communicator to allow other processes to update its corresponding part of the trailing matrix at the right of the panel. Then, processes on the upper-row update the upper $U_{i,j}$ sub-blocks of the matrix by solving the linear system $U_{i,j} = L_{i,i} \backslash A_{i,j}$, and broadcast the result on their column communicator to enable others processes to update the remaining $A_{i,j}$ sub-blocks, doing the local operation $A'_{i,j}$=$A_{i,j} - L_{i,j} \times U_{i,j}$. At this stage, CALU can take the next leftmost panel $currCol + 1$ of the recently updated trailing matrix and repeat the same procedure iteratively on the trailing matrix. Algorithm 3 describes how the steps of the CALU algorithm are executed. A more detailed description can be found in [7].

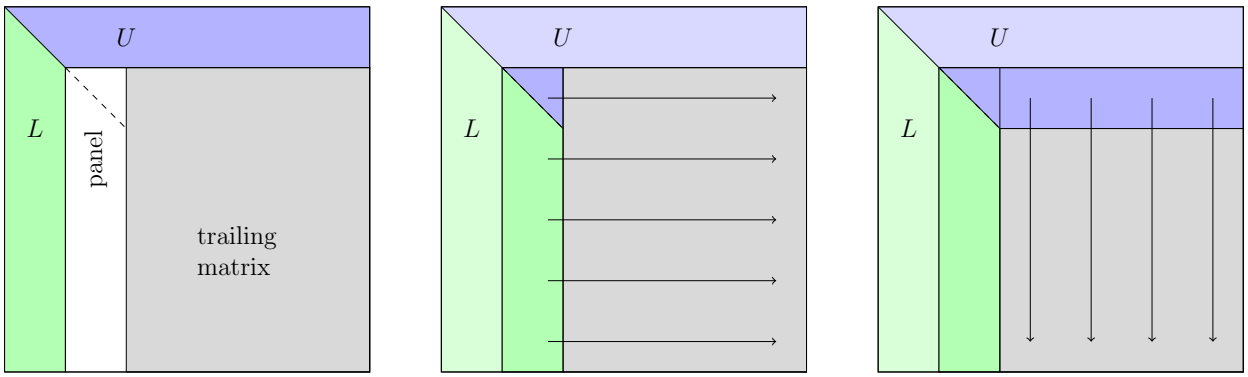
Algorithm 3: CALU

Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  while stillHasPanel() do
2      if computing a panel then
3          Panel = nextPanel() ;
4          L_{i,j}, U_{i,j} = TSLU(Panel) ;
5          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
6      else if comput. upper U_{i,s} then
7          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
8          U_{i,j} = L_{i,j} \A_{i,j} ;
9          broadcast(U_{i,j}, col_j, Process with rank currCol) ;
10     else if updating A_{i,j} then
11         broadcast(L_{i,j}, row_i, Process with rank currCol) ;
12         broadcast(U_{i,j}, col_j, Process with rank currCol) ;
13         A_{i,j} = A_{i,j} - L_{i,j} × U_{i,j} ;
14     updateStage() ;
15 return L_{i,j},U_{i,j};
```

In lines 1, 3, 4 and 14 of algorithm 3 new functions are used:

- *stillHasPanel()*: returns true if there are still panels to decompose; returns false otherwise.
- *nextPanel()*: returns the next panel.
- *TSLU(Panel)*: executes the TSLU algorithm over the panel, as described in algorithm 1.
- *updateStage()*: updates the current stage of the algorithm; i.e., the row and column currently being processed.



(a) Panel: select a panel, compute its LU factorization

(b) Broadcast $L_{i,j}$; pivot $A_{i,j}$; $U_{i,j} = L_{i,j} \backslash A_{i,j}$

(c) Update the trailing matrix: $A_{ij} - = L_{ij}U_{ij}$

Figure 4.11: CALU

As TSLU uses tournament pivoting to find the best rows in a matrix, each panel factorization can be performed at a low communication cost, and hence, we can take advantage of TSLU performance to gain throughput on CALU. Figure 4.11 represents how the panel selection, broadcasting, and trailing matrix update work.

4.2.2 FT-CALU

The Fault-Tolerant Communication-Avoiding LU factorization algorithm is the fault-tolerant version of the CALU algorithm. FT-CALU exhibits the same fault-tolerance characteristics as FT-TSLU: little modifications of the critical path of the algorithm, crash-type process failures detection at run-time, re-spawning all the failed processes at once, communicator and matrix state repairing to proceed with the rest of the execution. It relies on the fact that CALU is based on broadcast operations, that replicate the data over the processes of a communicator.

To have little impact on the memory footprint, it keeps track of the results obtained at the end of an operation backing up intermediate results on the local media storage device (HD, SSD, etc.). For example, when a process ends the decomposition of a panel with FT-TSLU or when a trailing matrix sub-block has been computed, each process stores its corresponding sub-block, replacing the results saved from a previous operation over the sub-block. This way, if a process restoration takes place, every process can have access to its last successful result and continue the execution with correct updated information. Algorithm 4 shows how fault tolerance can be achieved on CALU. We can appreciate the mention of new functions in lines 2, 7, 8 and 21:

- *readBackup()*: when a process has been re-spawned, it must perform a read operation of the last saved sub-block.
- *FTTSLU(Panel)*: executes the FT-TSLU algorithm over the panel, that is described in algorithm 2.

Algorithm 4: FT-CALU

Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  if I am a spawned process then
2      A_{i,j} = readBackup() ;
3      updateStage() ;
4  while stillHasPanel() do
5      if computing a panel then
6          Panel = nextPanel() ;
7          L_{i,j}, U_{i,j} = FTTSLU(Panel) ;
8          backup(L_{i,j}) ;
9          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
10     else if comput. upper U_{i,s} then
11         broadcast(L_{i,j}, row_i, Process with rank currCol) ;
12         U_{i,j} = L_{i,j} \A_{i,j} ;
13         backup(U_{i,j}) ;
14         broadcast(U_{i,j}, col_j, Process with rank currCol) ;
15     else if updating A_{i,j} then
16         broadcast(L_{i,j}, row_i, Process with rank currCol) ;
17         broadcast(U_{i,j}, col_j, Process with rank currCol) ;
18         A_{i,j} = A_{i,j} − L_{i,j} × U_{i,j} ;
19         backup(A_{i,j}) ;
20     if FAIL == f then
21         restoreFailed(current stage) ;
22         continue;
23     updateStage() ;
24 return L_{i,j},U_{i,j};
```

- *backup($L_{i,j}$)*: performs a backup operation of the given sub-block, overwriting the previous results on the local media storage device.
- *restoreFailed(current stage)*: performs a dead process restoration on the process grid.

The FT-CALU restoration algorithm proposed in this thesis has been designed to allow all the processes in the global communicator to detect errors at the same point in the algorithm, independently from the task a process is in charge of; thus, at the end of a broadcast and the backup operation, all processes can detect which processes have crashed and started the restoration procedure. This way, as with FT-TSLU, we can tolerate the possible errors that may appear in the CALU panel factorization and trailing matrix update, including the CALU upper panel update and the remaining trailing matrix update.

Complementing the backup operations (reading and writing), it should be noted that if a process fails before writing its results on the local media storage at a given stage S, the next re-spawned process will read the previous *state* $S-1$ the process use to had, considering the previous state was well written. In this case, it is considered as a *backward recovery*, because the state $S-1$ is re-taken. Otherwise, if a process fails after writing its results on the local media storage at stage S, then the next re-spawned process will read the current *state* S the process used to have. In this case, it is considered as a *forward recovery*, because the state S is taken from the crashing point. Another method to perform reading/writing operations is with *one-sided communications*, which allow a process to access an exposed memory block defined by another process (could be a dead one). However, the fault-tolerant subroutines needed to perform these operations are still "on the road" and it is still not possible to carry out this method with fault tolerance.

Both algorithms 3 and 4 seem very similar at communication and restoring level. They have the same communication pattern. Both algorithms use the same set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ and the same initial distribution of matrix A. They also use the same partner selection at every step, the same mechanism to detect and correct errors, and the same backup operation. The main difference between them resides in the backup and restore operations executed just after the information distribution ends.

The update operation over the $U^S_{i,s}$ sub-blocks from the upper panel is done by computing the operation $U^S_{i,s}$=$L^S_{i,i} \setminus A^S_{i,s}$. The update operation over the remaining $A^S_{i,s}$ sub-blocks from the trailing matrix located below the $U^S_{i,s}$ sub-blocks is performed by doing the operation $A^S_{i,s}$=$A^S_{i,s} - L^S_{i,j} \times U^S_{i,j}$.

Here it must be noted that every process (on the grid) computes, updates, distributes and backups only its corresponding $A^S_{i,s}$ sub-block on the row and column it belongs to. After that, depending on the position a process has on the grid, it is only responsible to act as a monitor on the computation, or to continue performing FT-TSLU, or updating the trailing matrix, and in case an error appears, it will be responsible for collaborating in the restoring procedure to recover the failed processes, and share the information needed to recover the state of the current matrix correctly. This characteristic is possible because all processes store the basic state of the matrix (stage, panel number, column/row number) in case they need to share it with a re-spawned process.

To complement a little bit more the operations a process does, if a process has already performed its computation, it *waits* until the remaining working processes finish all their computations. This way, some processes will execute FT-TSLU and others will not; some processes will update a sub-block in the trailing matrix and others will not. Although not all processes have the same tasks to run, they execute at least one dense operation (panel decomposition, linear solving, matrix multiplication) during the execution of the algorithm and thus the computation load is more evenly distributed. Thus, we can take advantage of the parallel resources our architecture has.

CHAPTER 5

QR Factorization

The QR factorization or decomposition factors a given matrix A as the product of two matrices Q and R as $A = QR$, where Q is an orthogonal matrix (its columns are orthogonal unit vectors, meaning $Q^T = Q^{-1}$ and $QQ^T = Q^TQ = I$) and R is an upper triangular matrix. For an invertible matrix A, the QR factorization is unique. The QR factorization can be used with various applications, such as solving linear least squares problems, or as a kernel in finding the eigenvalues of a matrix. Like LU, it can also simplify the procedure to solve a linear system $Ax = b$.

There are different methods to compute the QR decomposition (Gram–Schmidt process, Givens rotations), but in this thesis, we are going to focus only on the computation with Householder transformations, because it is a very numerically stable method. As mentioned in section 3.2, when we want to obtain a higher throughput on QR decomposition, then it is recommendable to implement it in a *block form*. In the next sections, we will see how we can use the QR block form to factorize a matrix A.

5.1 TSQR and FT-TSQR

5.1.1 TSQR

As with LU (see chapter 4), the Tall and Skinny QR (TSQR) factorization of a tall and skinny matrix A is a kernel of the QR factorization of a general square or rectangular matrix.

TSQR uses a set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ and a column block decomposition of the matrix (like with TSLU), to compute the QR factorization of matrix A, as previously defined for LU. The first step $s = 0$ consist in dividing the A matrix into $b \times N$ sub-blocks, with $b = \frac{M}{t}$, as mentioned in section 3.2. Every sub-matrix or sub-block $A_{i,s}$ is sent to its corresponding process P_i to compute locally the QR factorization of each sub-matrix. This operation will generate a set of Householder matrices $H_{i,s}$ instead of a set of $Q_{i,s}$ matrices. Notice that the $Q_{i,s}$ matrix can be computed from a collection of Householder vectors [68, 69]. After that, an exchange is performed between pairs of processes $\{P_i, P_j\}$ to concatenate upper $R_{i,s}$ sub-matrices into successive pairs $\{R_{i,s},R_{j,s}\}$, and also to collect the Householder vectors $\{H_{i,s},H_{j,s}\}$ needed for updating the trailing matrix in future steps. Then, the QR

factorization is executed again over the $R_{\{i,j\},s+1}$ pairs in parallel, generating Householder vectors $H_{\{i,j\},s+1}$ too. This procedure is repeated until there is only one final $R_{i,s}$ matrix left. In the case when it is only required the upper triangular factor $R_{i,s}$, this operation can be seen as a *reduction*. After each concatenation $\{R_{i,s},R_{j,s}\}$, a reduction operation is performed over the concatenation to obtain the final $R_{i,s}$ factor. A binary reduction tree is used and at each level of the tree, two upper triangular matrices are reduced to one [70].

This technique is very similar to LU, giving good performance because it uses a minimal number of inter-process communications. Algorithm 5 displays in detail how the steps of the TSQR algorithm are performed.

Algorithm 5: TSQR

Data: Sub-matrix $A_{i,0}$, Communicator $comm_p$, Integer i

```
1  s = 0 ;
2  myrank = i ;
3  H_{i,s}, R_{i,s} = QR(A_{i,0});
4  while !done() do
5      j = myPartner(s) ;
6      if myrank < j then
7          f = recv(H_{j,s},R_{j,s},comm_p) ;
8      else
9          f = send(H_{i,s},R_{i,s},comm_p) ;
10         break;
11     if FAIL == f then
12         return;
13     A_{i,s} = concatenate(R_{i,s}, R_{j,s});
14     s = s + 1 ;
15     H_{i,s}, R_{i,s} = QR(A_{i,s});
16 broadcast(H_{i,s}, R_{i,s}, comm_p, Process with rank 0) ;
17 Q_i = A_{i,0} \ R_{i,s} ;
18 return Q_i, H_{i,s}, R_{i,s};
```

We can appreciate that TSQR uses the same set of functions defined in 4.1, so we can also represent with figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7 TSQR examples with $t = 4$ processes, only changing the LU execution gray bar for QR.

Compared with the LU algorithm 1, the differences between them are that LU uses a pivoting operation to stabilize the factorization and QR does not, and the QR call generates both $H_{i,s}, R_{i,s}$ local matrices. Every collection of Householder vectors $H_{i,s}$ will be stored as a *list* for a later use in the QR communication-avoiding version. So each process P_i will start with its corresponding local sub-matrix $A_{i,s}$, will generate sub-matrices $H_{i,s}, R_{i,s}$ locally, will share them with a partner P_j and finally will backup $H_{i,s}$.

Following the examples shown in 4.1, with $t = 4$, $T_{steps} = \log_2 t + 1 = 3$, and considering we compute every $Q_{i,s}$ from $H_{i,s}$ at each step, the TSQR matrix algebra can be represented as follows:

- Step 0: $A = \begin{bmatrix} A_{0,0} \\ A_{1,0} \\ A_{2,0} \\ A_{3,0} \end{bmatrix} = \begin{bmatrix} Q_{0,0}R_{0,0} \\ Q_{1,0}R_{1,0} \\ Q_{2,0}R_{2,0} \\ Q_{3,0}R_{3,0} \end{bmatrix} = \begin{bmatrix} Q_{0,0} & 0 & 0 & 0 \\ 0 & Q_{1,0} & 0 & 0 \\ 0 & 0 & Q_{2,0} & 0 \\ 0 & 0 & 0 & Q_{3,0} \end{bmatrix} \cdot \begin{bmatrix} R_{0,0} \\ R_{1,0} \\ R_{2,0} \\ R_{3,0} \end{bmatrix}$
- Step 1: $\begin{bmatrix} R_{0,0} \\ R_{1,0} \\ \hline R_{2,0} \\ R_{3,0} \end{bmatrix} = \begin{bmatrix} R'_{\{0,1\},1} \\ R'_{\{2,3\},1} \end{bmatrix} = \begin{bmatrix} Q'_{\{0,1\},1}R_{\{0,1\},1} \\ Q'_{\{2,3\},1}R_{\{2,3\},1} \end{bmatrix} = \begin{bmatrix} Q'_{\{0,1\},1} & 0 \\ 0 & Q'_{\{2,3\},1} \end{bmatrix} \cdot \begin{bmatrix} R_{\{0,1\},1} \\ R_{\{2,3\},1} \end{bmatrix}$
- Step 2: $\begin{bmatrix} R_{\{0,1\},1} \\ R_{\{2,3\},1} \end{bmatrix} = R'_{\{0,1,2,3\},2} = Q'_{\{0,1,2,3\},2}R_{\{0,1,2,3\},2}$

We know that solving a linear system of equations only requires the use of the final matrix $R_{i,s}$, so if we are not planning to generate the $Q_{i,s}$ final matrix, we can dismiss it from the computation. In case we need it, the final matrix $Q_{i,s}$ can be computed by multiplying the intermediate $Q_{i,s}$ matrices. The algebra to generate the final matrix $Q_{i,s}$ in the previous example is:

$$Q_{\{0,1,2,3\},2} = \begin{bmatrix} Q_{0,0} & 0 & 0 & 0 \\ 0 & Q_{1,0} & 0 & 0 \\ 0 & 0 & Q_{2,0} & 0 \\ 0 & 0 & 0 & Q_{3,0} \end{bmatrix} \cdot \begin{bmatrix} Q'_{\{0,1\},1} & 0 \\ 0 & Q'_{\{2,3\},1} \end{bmatrix} \cdot Q'_{\{0,1,2,3\},2}$$

Again, the matrix multiplications involved in the final $Q_{i,s}$ require more computing time, but at the end, when the final $R_{i,s}$ has been computed, we can solve the linear system $A_{init}x = R_{i,s}$ as $Q_{i,s} = A_{init} \backslash R_{i,s}$ to compute the final matrix $Q_{i,s}$ efficiently.

Finally, it is important to mention that, compared to TSLU, TSQR uses much more space to save the Householder vectors $H_{i,s}$ each process generates at each step. In the next sections we will introduce the fault-tolerant version of this algorithm and we will see why every Householder vectors should be saved to correctly use them in the communication-avoiding version.

5.1.2 FT-TSQR

The Fault-Tolerant Tall and Skinny QR (FT-TSQR) algorithm is the fault-tolerant version of TSQR. As FT-TSLU, FT-TSQR requires a fault-tolerant middleware to be able to detect crash-type process failures at run-time and repair them. When an error is detected, it re-spawns all the failed processes at once, repairs both the communicator used by the processes to exchange information and the current state of the matrix. To achieve this last characteristic, it keeps track of the states of the matrices obtained at each step, storing them over the results generated at the previous step, thus enabling all processes to share their

previous known results with a restored process if necessary. Algorithm 6 shows how fault tolerance can be achieved on the FT-TSQR algorithm, making some modifications over the TSQR original algorithm.

Algorithm 6: FT-TSQR

Data: Sub-matrix $A_{i,0}$, Communicator $comm_p$, Integer i

```
1  s = 0 ;
2  if I am a spawned process then
3      H_{i,s}, R_{i,s} = update(s, comm_p) ;
4  else
5      H_{i,s}, R_{i,s} = QR(A_{i,0});
6  while !done() do
7      j = myPartner(s) ;
8      f = sendRecv(H_{i,s}, R_{i,s}, H_{j,s}, R_{j,s}, comm_p) ;
9      if FAIL == f then
10         restoreFailed(s, H_{i,s}, R_{i,s}, comm_p) ;
11         continue;
12     A_{i,s} = concatenate(R_{i,s}, R_{j,s});
13     s = s + 1 ;
14     H_{i,s}, R_{i,s} = QR(A_{i,s});
15     backup(H_{i,s}, R_{i,s}) ;
16 Q_i = A_{i,0} \ R_{i,s} ;
17 return Q_i, H_{i,s}, R_{i,s};
```

FT-TSQR works similar to its non-fault-tolerant version and, as FT-TSLU, generates data redundancy providing properties we can take advantage of for adding fault tolerance mechanisms into the algorithm. Compared to TSQR, the most significant changes reside in the added functions to fully correct the failed processes: *update(s, $comm_p$)*, *sendrecv($H_{i,s}, R_{i,s}$,$H_{j,s}, R_{j,s}$,$comm_p$)*, *restoreFailed(s, $H_{i,s}, R_{i,s}$, $comm_p$)* and *backup($H_{i,s}, R_{i,s}$)*. These functions are already defined in section 4.1.

FT-TSQR uses the same set of t processes $P = \{P_0, P_1, \cdots, P_{t-1}\}$, the same communication pattern to exchange intermediate $H_{i,s}, R_{i,s}$ results between them and the same distribution of matrix A as algorithm 2, described in section 4.1. Figures 4.8 and 4.9 illustrate an example of the complete computation of the QR factorization, including the re-spawning of a failed process.

5.2 CAQR and FT-CAQR

5.2.1 CAQR

The Communication-Avoiding QR (CAQR) algorithm factors a rectangular matrix A into the product of two matrices Q and R taking the left-most panel at each iteration, as defined in section 3.2 and figure 3.3. As a panel is a tall and skinny matrix, CAQR uses TSQR to factorize an entire panel.

CAQR divides the rectangular matrix $A \in M \times N$ into sub-blocks $A_{i,j}$ of size $b \times b$, forming a grid with $\frac{M}{b}$ lines and columns, as CALU. In every iteration, it forms a panel with all the sub-blocks $A_{i,j}$ belonging to the current processing column and computes the QR factorization on it, finding the upper $R_{i,j}$ and the initial set of Householder vectors $H_{i,j}$. Then, it updates the trailing matrix. In this case, the resulting upper $R_{i,j}$ panel-blocks are the final ones and they do not need to be distributed across columns. Finally, the same panel formation and construction is repeated until the trailing matrix is the last bottom-right sub-block and it can be fastly computed by the process located at that part on the grid.

CAQR processes are assigned to work on a specific sub-block on the grid. With this, processes belonging to the same column will be in charge of decomposing the panel corresponding to their column. At the end of the panel computation, the Householder vectors $H_{i,j}$ and the current state of the sub-block $A_{i,j}$ are broadcasted across row communicators on the grid to allow processes on the right of the panel to update its corresponding part of the trailing matrix. The operations needed to do the update of the trailing matrix are:

- Operation executed by all processes to apply the local Householder vectors to its current sub-block: $A_{i,j}\ C_i = A_{i,j} - H_i \times T^T \times H_i^T \times A_{i,j}$
- Operation executed by pairs of processes $\{P_i, P_j\}$ after an exchange has been completed. $W = T^T \times C_i + H_i^T \times C_j$
- Operation executed by the upper process (normally P_i) to complete the update procedure over its sub-block: $C_i = C_i + W$
- Operation executed by the lower process (normally P_j) to complete the update procedure over its sub-block: $C_i = C_i + H_i \times W$

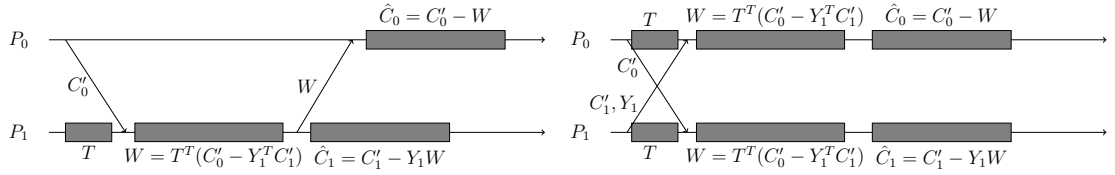
It must be noted that the trailing matrix update requires the computation of matrix T. This matrix can be generated locally by every process since they have exchanged the needed information [67]. At this stage, CAQR can take the next leftmost panel on the recently updated trailing matrix and repeat the same procedure iteratively on the trailing matrix. Algorithm 7 describes how the steps of the CAQR algorithm are executed.

CAQR uses the same set of functions defined in section 4.2: *stillHasPanel()*, *nextPanel()*, *broadcast(H_i,R_i,row_i,r)* and *updateStage()*. Compared with algorithm 3, the difference resides on the TSQR call. Figure 5.1 represents how the panel selection and trailing matrix update work, and figure 5.2 displays an execution example.

Algorithm 7: CAQR

Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  s = 0 ;
2  while stillHasPanel() do
3      if computing a panel then
4          Panel = nextPanel() ;
5          H_i, R_i = TSQR(Panel) ;
6          broadcast(H_i, R_i, row_i, Process with rank currCol) ;
7      else if updating trailing matrix then
8          broadcast(H_i, R_i, row_i, Process with rank currCol) ;
9          j = myPartner(s) ;
10         C_i = A_{i,j} − H_i × T^T × H_i^T × A_{i,j} ;
11         W = T^T × C_i + H_i^T × C_j ;
12         if i < j then
13             C_i = C_i + W
14         else
15             C_i = C_i + H_i × W
16     updateStage() ;
17 return Q_{i,j},R_{i,j};
```



(a) CAQR trailing matrix update per sub-block, without fault tolerance

(b) CAQR trailing matrix update per sub-block, with fault tolerance

Figure 5.1: CAQR trailing matrix update equations by pairs of process

5.2.2 FT-CAQR

The Fault-Tolerant Communication-Avoiding QR factorization algorithm is the fault-tolerant version of the CAQR algorithm. FT-CAQR has been given with the same fault-tolerance characteristics as FT-TSQR, FT-TSLU, and FT-CALU: minimal changes on the critical path and crash-type process failures restoration procedure at run-time (failed processes respawning, communication repairing, and matrix state repairing). FT-CAQR allows all processes to have access to the required data to execute the trailing matrix update operation,

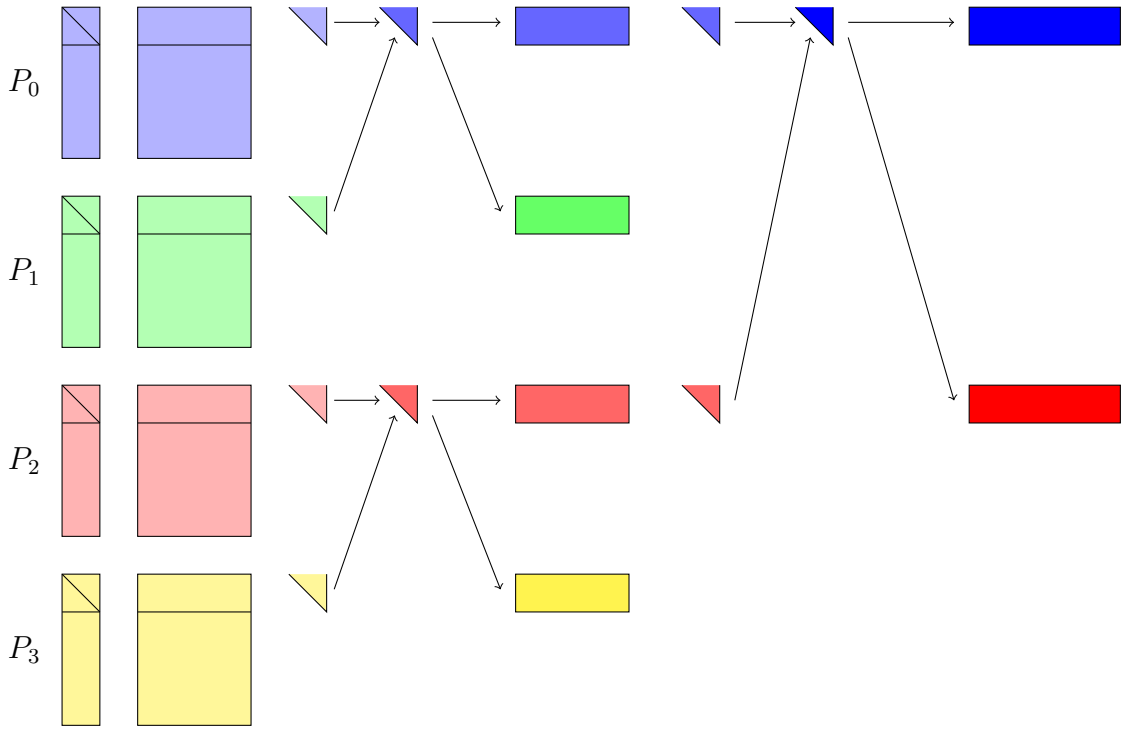


Figure 5.2: CAQR trailing matrix update execution example.

broadcasting only basic information over row communicators. When a process performs a broadcast operation over his row, he generates data redundancy across the grid rows, so there is always another process that holds the same data, and fault tolerance can be achieved.

To reduce the amount of memory used, FT-CAQR dumps the results obtained at the end of an important operation on the local media storage device (HD, SSD, etc.). For example, when a process ends the QR factorization of a panel or when a trailing matrix operation has been completed, each process stores its corresponding sub-block, overwriting previous results. As with previous fault-tolerant algorithms, this mechanism was designed like this to let the algorithm restore failed processes. If a process restoration takes place, every process can have access to its last valid result and continue with the execution. Algorithm 8 shows how fault tolerance can be achieved on CAQR, using the same function defined in section 4.1: *readBackup()*, *FTTSQR(Panel)*, *backup($H_{i,j}$,$R_{i,j}$)* and *restoreFailed(current stage)*.

The FT-CAQR restoration procedure synchronizes all processes in the global communicator to allow error detection, independently from the task a process is in charge of. So at the end of successful communication, all processes can realize which processes have crashed and start the restoration procedure. Both algorithms 7 and 8 have the same communication pattern (as with FT-TSLU and FT-CALU). Both algorithms use the same set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ and the same initial distribution of matrix A. The main difference between them resides again in the backup and restoration operation.

Algorithm 8: FT-CAQR

Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  s = 0 ;
2  if I am a spawned process then
3      A_{i,j} = readBackup() ;
4      updateStage() ;
5  while stillHasPanel() do
6      if computing a panel then
7          Panel = nextPanel() ;
8          H_i, R_i = FTTSQR(Panel) ;
9          broadcast(H_i, R_i, row_i, Process with rank currCol) ;
10         backup(H_i, R_i) ;
11     else if updating trailing matrix then
12         broadcast(H_i, R_i, row_i, Process with rank currCol) ;
13         j = myPartner(s) ;
14         C_i = A_{i,j} − H_i × T^T × H_i^T × A_{i,j} ;
15         W = T^T × C_i + H_i^T × C_j ;
16         if i < j then
17             C_i = C_i + W
18         else
19             C_i = C_i + H_i × W
20         backup(H_i, R_i, C_i) ;
21     if FAIL == f then
22         restoreFailed(current stage) ;
23         continue;
24     updateStage() ;
25 return Q_{i,j},R_{i,j};
```

CHAPTER 6

Cholesky Factorization

The Cholesky factorization or decomposition factors a given Hermitian, positive-definite matrix A as the product of matrices $A = LL^T$, where L is a lower triangular matrix and L^T is its conjugate transpose (takes the complex conjugate of each entry of the matrix A^T). This decomposition is useful for efficient numerical solutions and, when it is possible to apply the algorithm, the Cholesky factorization is roughly twice as fast as the LU factorization for solving systems of linear equations $Ax = b$.

As mentioned in section 3.3, when we want to obtain a higher throughput on Cholesky decomposition, then it is recommendable to implement it in a *block form*. In the next sections, we will see how we can use the Cholesky block form to factorize a matrix A.

6.1 TSCH and FT-TSCH

6.1.1 TSCH

As with LU and QR (see chapters 4,5), the Tall and Skinny Cholesky (TSCH) factorization of a tall and skinny matrix A is part of the usual, de facto standard set of kernels provided by the mainstream linear algebra libraries [65, 63]. TSCH uses the same set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ to compute the Cholesky factorization of matrix A, as previously defined for LU and QR. In this algorithm, we only have one step to execute. It consists in dividing the A matrix into $b \times N$ sub-blocks, with $b = \frac{M}{t}$, as mentioned in section 3.3. Every sub-matrix or sub-block A_i is distributed to its corresponding process P_i. Compared with LU and QR, TSCH only allows the upper process (normally process 0) to compute locally the Cholesky factorization of its corresponding sub-matrix. This operation will generate the lower triangular matrix L_i, which will be broadcast on the entire panel to let other processes compute its corresponding part of the matrix L_i by solving the linear system $A_i x = L_0$. This technique differs from LU and QR because in this case, we can only decompose diagonal sub-blocks with the Cholesky factorization, due to the form of matrix A. Thus, this factorization gives great performance because it uses only one broadcast operation and some other linear solving operations. Algorithm 9 shows how the TSCH algorithm works.

We can notice that TSCH does not have the same pattern as TSLU and TSQR (see

Algorithm 9: TSCH

Data: Sub-matrix $A_{i,j}$, Communicator $comm_p$, Integer i

```
1 myrank = i ;
2 if myrank = 0 then
3 |  L_{i,j} = CHOLESKY(A_{i,j});
4 f = broadcast(L_{i,j}, comm_p, Process with rank 0) ;
5 if FAIL == f then
6 |  return;
7 if myrank > 0 then
8 |  L_{i,j} = A_{i,j} \ L_{i,j} ;
9 return L_{i,j};
```

sections 4.1,5.1). Compared with these algorithms, the main differences are the Cholesky call and the communication pattern. While LU and QR have a butterfly communication pattern, Cholesky only generates a linear broadcast to distribute $L_{i,0}$. So each process P_i will start with its corresponding local sub-matrix $A_{i,s}$, process 0 will generate sub-matrix $L_{i,0}$ locally, will share it with all processes P_j on the panel, will send $L_{i,0}$ to them and finally processes with a rank higher than zero will compute its corresponding $L_{i,s}$.

The TSCH matrix algebra can be represented as follows:

- $A = \begin{bmatrix} A_{0,0} \\ A_{1,0} \\ A_{2,0} \\ A_{3,0} \end{bmatrix} = \begin{bmatrix} L_{0,0}L_{0,0}^T \\ A_{1,0} \\ A_{2,0} \\ A_{3,0} \end{bmatrix} = \begin{bmatrix} L_{0,0}L_{0,0}^T \\ A_{1,0} \setminus L_{0,0} \\ A_{2,0} \setminus L_{0,0} \\ A_{3,0} \setminus L_{0,0} \end{bmatrix} = \begin{bmatrix} L_{0,0}L_{0,0}^T \\ L_{1,0} \\ L_{2,0} \\ L_{3,0} \end{bmatrix}$

Finally, it is important to mention that, compared to TSLU and TSQR, TSCH uses much less space in memory or local media storage. In the next sections, we will introduce the fault-tolerant version of this algorithm.

6.1.2 FT-TSCH

The Fault-Tolerant Tall and Skinny Cholesky (FT-TSCH) algorithm is the fault-tolerant version of TSCH. As FT-TSLU and FT-TSQR, FT-TSCH requires a fault-tolerant middleware to be able to detect crash-type process failures at run-time and repair them. When an error is detected, it re-spawns all the failed processes at once, repairs the communication channels used by the processes to distribute information and the required data to let failed processes know which sub-block the dead process was in charge of. Processes backup their results obtained after calculating the Cholesky factorization (in the case of a process 0) or solving the corresponding linear system to find $L_{i,j}$ (in the case of processes which rank is greater than zero). The storing procedure is capable of backup sub-blocks on memory or in local media storage. Algorithm 10 describes how FT-TSCH functions.

Algorithm 10: FT-TSCH

Data: Sub-matrix $A_{i,j}$, Communicator $comm_p$, Integer i

```
1  myrank = i ;
2  while !done() do
3      if L_{i,j} has not being computed and myrank = 0 then
4          L_{i,j} = CHOLESKY(A_{i,j});
5      f = broadcast(L_{i,j}, comm_p, Process with rank 0) ;
6      if FAIL == f then
7          restoreFailed(L_{i,j}, comm_p) ;
8          continue;
9      if myrank > 0 then
10         L_{i,j} = A_{i,j} \ L_{i,j} ;
11     backup(L_{i,j}) ;
12 return L_{i,j};
```

This algorithm is less complex than LU and QR algorithms. It keeps the same fault tolerance intention and works similar to its non-fault-tolerant version. The only change is the *restoreFailed($L_{i,j}$)* and *backup($L_{i,j}$)* addition. These functions are already defined in section 4.1.

FT-TSQR uses the same set of t processes $P = \{P_0, P_1, \cdots, P_{t-1}\}$, keeps a linear communication pattern to distribute the first $L_{i,0}$ sub-block from process 0 to other processes and retains the same initial distribution of matrix A as algorithm 2.

6.2 CACH and FT-CACH

6.2.1 CACH

The Communication-Avoiding Cholesky (CACH) algorithm factors a necessarily square matrix A into the product of two matrices L and L^T taking the left-most panel at each iteration, as defined in section 3.3 and figure 3.4. CACH uses TSCH to factorize an entire panel at once and uses their computed $L_{i,j}$ sub-blocks to update the trailing matrix.

CACH divides the square matrix $A \in M \times N$ into sub-blocks $A_{i,j}$ of size $b \times b$, forming a grid with $\frac{M}{b}$ lines and columns, as CALU and CAQR. It forms the current processing panel with the sub-blocks $A_{i,j}$ belonging to the current processing column and computes the Cholesky factorization on it, as described in the previous section, finding all $L_{i,j}$ sub-blocks. Then, it updates the trailing matrix. In this case, the resulting $L_{i,j}$ sub-blocks are distributed across rows and the update procedure takes place. The same panel construction is repeated until the trailing matrix is the last sub-block and it can be fastly computed by one process.

CACH processes are assigned to work on a specific sub-block on the grid. Properties

provided by the form of the matrix let CACH dismiss all the upper parts of the matrix. Hence, processes working on the upper part of the main diagonal (belonging to the upper part of the grid) can be discarded to save some memory. Then, processes belonging to the same column will decompose the panel corresponding to their column. At the end of the panel computation, the sub-blocks $L_{i,j}$ are broadcasted across row communicators on the grid to allow processes on the right of the panel to update its corresponding part of the trailing matrix; with this data distribution, data redundancy is generated across rows and thus processes are holding the same data and fault-tolerant mechanisms can be added to the algorithm. Now, only one operation it is needed to perform the trailing matrix update: $A_{i,j} = A_{i,j} - L_{i,j} \times L_{i,j}^T$. We can see that the Cholesky trailing matrix update is much easier than CALU or CAQR. We only need to perform a matrix transposition, a multiplication, and an addition, which results in higher performance when implemented in parallel. In this part, CACH takes the next panel and repeats the same procedure iteratively on the trailing matrix. Algorithm 11 describes how the steps of the CACH algorithm are executed.

Algorithm 11: CACH

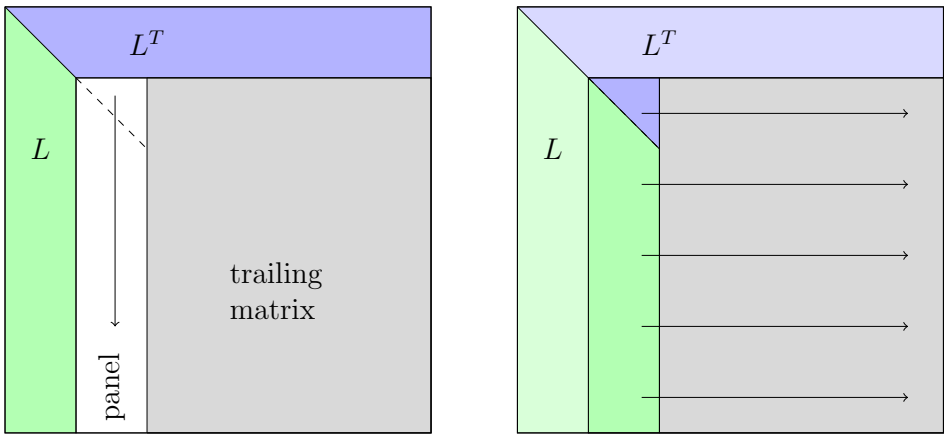
Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  while stillHasPanel() do
2      if computing a panel then
3          Panel = nextPanel() ;
4          L_{i,j} = TSCH(Panel) ;
5          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
6      else if updating trailing matrix then
7          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
8          A_{i,j} = A_{i,j} - L_{i,j} × L_{i,j}^T ;
9      updateStage() ;
10 return L_{i,j};
```

CACH uses the four functions defined in section 4.2: *stillHasPanel()*, *nextPanel()*, *broadcast($L_{i,j}$,row_i,r)* and *updateStage()*. Compared with algorithm 3 and 7, the difference resides on the TSCH call.

6.2.2 FT-CACH

The Fault-Tolerant Communication-Avoiding Cholesky factorization algorithm is the fault-tolerant version of the CACH algorithm. FT-CACH has been given the same fault-tolerance characteristics as all previous fault-tolerant algorithms, focusing on minimal changes on the critical path and crash-type process failures restoration procedure at run-time. As a process may need the $L_{i,j}$ sub-block corresponding to another row it is not part of, FT-CACH dumps the results obtained at the end of FT-TSCH on the local media storage device, allowing all



(a) Panel: select a panel, compute its Cholesky factorization, broadcast over L_i panel

(b) Broadcast $L_{i,j}$; update trailing matrix with $L_{i,j} = A_{i,j} - L_{i,j} \times L_{i,j}^T$

Figure 6.1: CACH panel selection and $L_{i,j}$ broadcast across panel and rows.

processes to have access to all $L_{i,j}$ sub-blocks to execute the trailing matrix update procedure, also broadcasting $L_{i,j}$ on row communicators.

When a process ends the Cholesky factorization of a panel and when a trailing matrix operation has been completed, each process stores its corresponding sub-block and overwrites its previous results. As with previous fault-tolerant algorithms, this mechanism allows the algorithm to restore failed processes. If a process restoration takes place, every process can have access to its last valid result and continue with the execution. Algorithm 12 displays how fault tolerance mechanism works, using the same functions defined in section 4.1: *readBackup()*, *FTTSCH(Panel)*, *backup($L_{i,j}$)* and *restoreFailed(current stage)*.

As with previous fault-tolerant versions, the restoration procedure synchronizes all processes in the global communicator. At the end of successful communication, all processes can detect which processes have died and re-start them. Both algorithms 11 and 12 have the same communication pattern on the grid. Both algorithms use the same set of t processes $P = \{P_0, P_1, \ldots, P_{t-1}\}$ and the same initial distribution of matrix A. The main difference between them resides again in the backup and restoration operations.

In chapter 7, a model is presented validating the approach described in all previous Communication-Avoiding algorithms, proving properties and demonstrating they can be reliable enough.

Algorithm 12: FT-CACH

Data: Square Matrix A, Communicator row_i, col_j, Integer i,j

```
1  if I am a spawned process then
2      A_{i,j} = readBackup() ;
3      updateStage() ;
4  while stillHasPanel() do
5      if computing panel then
6          Panel = nextPanel() ;
7          L_{i,j} = FTTSCH(Panel) ;
8          broadcast(L_{i,j}, row_i, Process with rank currCol) ;
9          backup(L_{i,j}) ;
10     else if updating trailing matrix then
11         broadcast(L_{i,j}, row_i, Process with rank currCol) ;
12         A_{i,j} = A_{i,j} − L_{i,j} × L_{i,j}^T ;
13         backup(A_{i,j}) ;
14     if FAIL == f then
15         restoreFailed(current stage) ;
16         continue;
17     updateStage() ;
18 return L_{i,j};
```

CHAPTER 7

Fault Tolerance Formal Verification

Interesting properties that can be used to provide fault tolerance to algorithms have arisen. These properties do not seem very intuitive at all when we consider them outside the algorithms. One way to represent fault-tolerant properties such that they appear more readable or understandable is with *formal models*. Coloured Petri Nets (CPN) [71] allow for easily representing the algorithm. Their ease in modeling and validating properties of parallel and distributed algorithms is very intuitive.

A CPN model contains *places* (represented with circles) that depict elements of the state of the system. It also contains *transitions* (represented with squares) that describe the actions carried out in the system, with their prerequisites and effects. CPN places contain tokens that hold a value that can be manipulated by transitions. Therefore, each place is associated with the type of tokens it can hold.

Formal verification allows for proving properties of the model. Then, we formally prove that Tall and Skinny algorithms are able to tolerate some failures that could arise during its execution and guaranteeing that the final results are always the expected ones. We also model the general fault tolerance mechanism used in our algorithms and prove it behaves as expected.

7.1 Tall and Skinny Formal Model

Figure 7.1 depicts a Coloured Petri Net modeling a Tall and Skinny algorithm (see algorithms 1, 2, 5, 6, 9 and 10). It focuses on the structure for the functioning of the algorithm, its different steps, and the communication between processes, and not the actual computation. The model mostly depends on a single parameter: the number of processes in the system. The other ones (number of steps, maximum number of failures) depend on the number of processes.

7.1.1 Tall and Skinny Model Description

- Place Processes contains triples (q, s, k) where q is a process number, s the current step, and k the index of the $\tilde{R}$ matrix it has already computed.

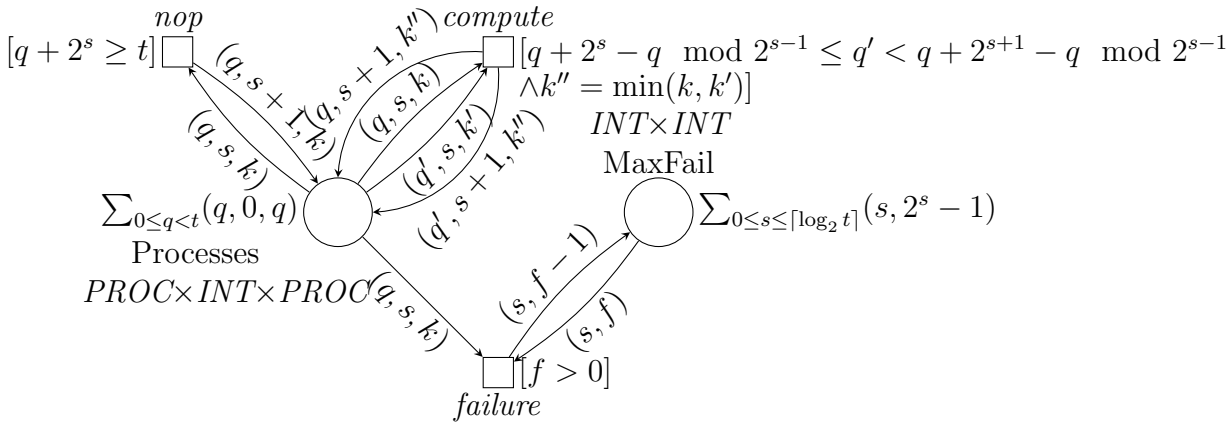


Figure 7.1: Model corresponding to algorithms 1, 2, 5, 6, 9 and 10.

- Place *MaxFail* contains pairs (s, f) which indicates how many failures are still allowed at step s. It thus limits the number of occurrences of transition *failure*.
- Transition *compute* assigns the fixed partner process q' and executes the current step s of the algorithm.
- Transition *failure* consumes a failed process at step s if it is still allowed.
- Transition *nop* captures the case when no partner can be found, and the process thus moves to the next step.

7.1.2 Tall and Skinny Structural Analysis

In this section, it is proved that:

1. The system modelled in section 7.1 can reach the end of the computation (prop. 1)
2. The final result is unique and, therefore is the expected one (prop. 2)

For proving, there is used projection functions Π_x to select the xth element of a token which has a tuple value $\Pi_{x,y}$ to select the xth and yth elements to form a pair. Π_x^s denotes the value of Π_x when the step number is s.

Property 1. *At every step $s > 0$, the system can tolerate at most $2^s - 1$ failures.*

Proof. Let $M()$ be the marking of places. During the first step, each process performs a local computation. Then at every step $s > 0$, transition *compute* takes the upper triangular $\tilde{U}$ and $\tilde{U}'$ from two processes q and q' and produces $\tilde{U}''$ on both q and q' or transition *failure* consumes a process. Formally, by a trivial recursion, it can be proved that at each step $s > 0$, it holds that: $|\Pi_3^s(M(Processes))| + \Pi_2^s(M(MaxFail)) = 2^s$.

However, the guard on transition *failure* ensures that $0 \leq \Pi_2^s(M(MaxFail)) \leq 2^s - 1$. Therefore, we have at each step $s > 0$: $1 \leq |\Pi_3^s(M(Processes))| \leq 2^s$: at least one process holds each intermediate $\tilde{U}$, which is sufficient for the computation to proceed with the next step. □

Property 2. *At the end of the computation, if the system did not suffer too many failures (as specified in property 1), at least one process holds the final R.*

Proof. This property is easily derived from the proof of property 1. At each step $s > 0$, we have $|\Pi_3^s(M(Processes))| \geq 1$. This is sufficient for the algorithm to reach the final step.

We also need to prove that this final $\tilde{R}$ is unique (and therefore, is the final R). We know that, for each $\tilde{R}$: $|\Pi_3^s(M(Processes))|+\Pi_2^s(M(MaxFail)) = 2^s$ Moreover, $0 \leq \Pi_2^s(M(MaxFail)) \leq 2^s - 1$. The final step is when $s = \log_2 t$. Hence, $|\Pi_3^s(M(Processes))| + \Pi_2^s(M(MaxFail)) = t$. As a consequence, all non-failed processes hold the same $\tilde{R}$, which is the final R. □

7.2 Communication-Avoiding Formal Model

Figure 7.2 shows a CPN model for the Fault-Tolerant Communication-Avoiding schemes (see algorithms 3, 4, 7, 8, 11 and 12). It is focused on the functioning of the algorithm: 1) how panel selection and stage increment is made, 2) how different processes are selected to run different tasks, 3) how processes communicate between them and 4) how dead processes are restored. As the Tall and Skinny model in previous section 7.1, the The Communication-Avoiding model mostly depends on the number of available processes in the system.

7.2.1 Communication-Avoiding Model Description

- Place *Ready* marked with an elementary token e indicates whether a new panel can be handled, otherwise it is empty.
- Place *Current Panel* contains the number of panel p to be processed.
- Place *Working Processes* contains a triple (l, S, p) composed of the (sub-)list of processes $[q_i, \ldots, q_j] \in Q$ in charge of processing the current panel p at stage S. It is empty when no panel is being processed and initialized when starting to process a new panel.
- Place *Process Stage* contains a pair (q, S) such that process q is executing stage S.
- Place *Alive* contains all processes q that are still alive.
- Place *Dead* contains all pairs (q, s) such that process q has failed at stage S.
- Transition *initialize* starts the initialization phase for handling the *Current Panel* p: it initializes place *Working Processes* with a triple containing the processes list $l = workers(p)$, stage $S = 0$, and the panel p, and place *Process Stage* with all pairs with

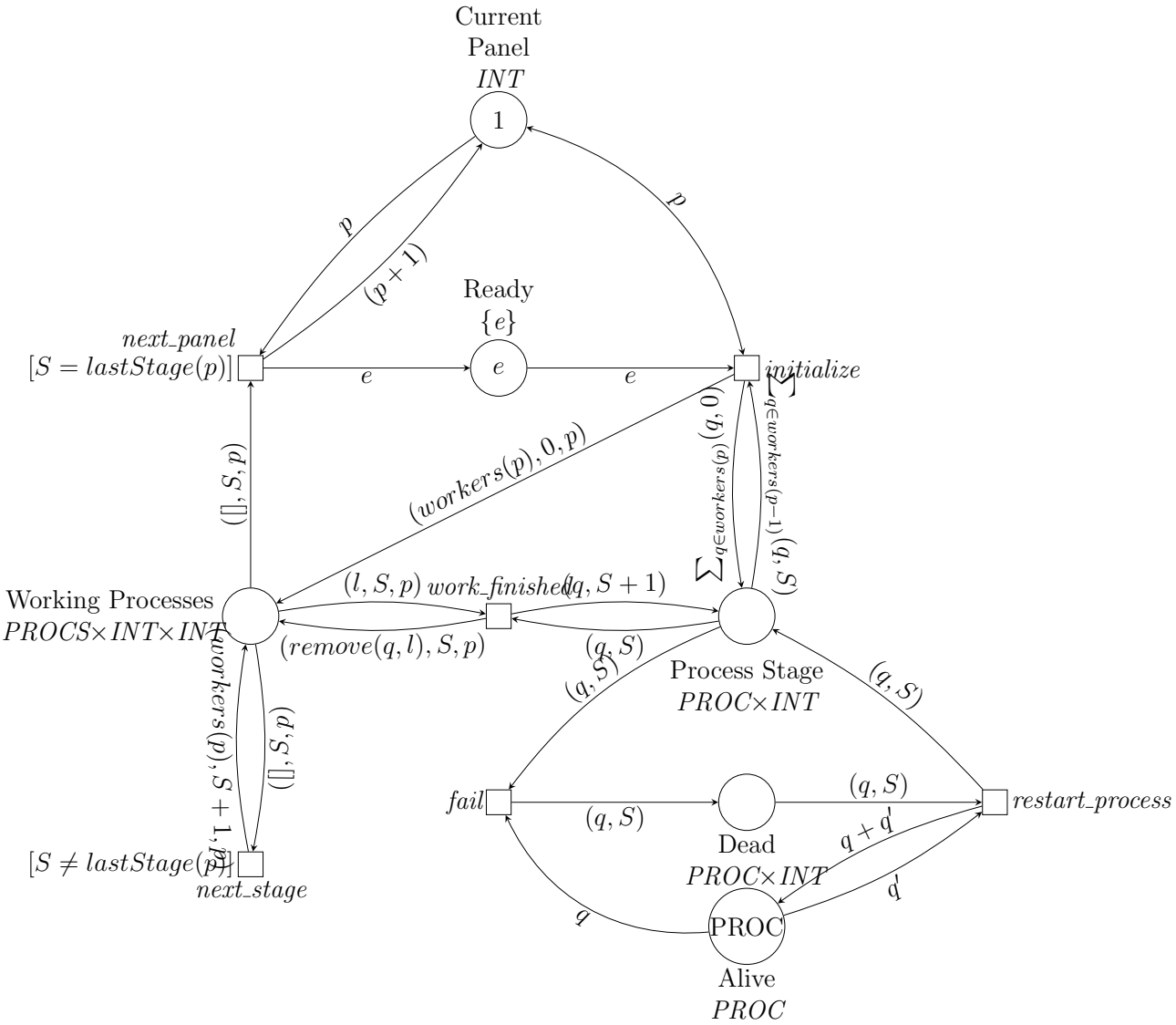


Figure 7.2: Model corresponding to algorithms 3, 4, 7, 8, 11 and 12.

a process in the list $workers(p)$, and their starting stage $S = 0$. At the same time, it removes from *Process Stage* all processes that operated on the previous panel and have now finished. This occurs only when place *Ready* is marked.

- Transition *next_stage* occurs when the list of processes at the current stage S is empty, *i.e.,* they can move on to the next stage (and in that case there still exists one). Thus it restarts with a new list of workers for panel p at stage $S + 1$.

- Transition *next_panel* updates, when stage S is the last one for the current panel p, the panel in place *Current Panel* with the next one $p + 1$. It also puts a token in place *Ready* to indicate that the next panel is ready to be processed. The empty list

of *Working Processes* at this last stage is also removed from its place.

- Transition *work_finished* indicates a process has finished the stage S and is ready to handle stage $S + 1$. It is also removed from the list of working processes in place *Working Processes*.

- Transition *fail* captures the failure of a process q at stage S. It is removed from place *Process Stage* and from place *Alive* and moved to place *Dead*.

- Transition *restart_process* restarts a dead process q at the stage S where it had failed, as indicated in place *Dead*, and puts it back in places *Alive* and *Process Stage*.

Note that the *workers(p)* function is not explicit here. It can choose the same worker processes for different panels. As referred to in section 8.3, the implementation uses a 2D process grid, hence in practice, they are different. However, this is not necessary because all processes are removed from place *process_stage* before the new ones are launched by transition *initialize*. The only constraint on this function should be that $workers(0) = \emptyset$ so that the first initialization does not require any token from the empty place *Process Stage*.

7.2.2 Communication-Avoiding Structural Analysis

This section exhibits the properties of the CPN model in figure 7.2, that ensures the correct functioning of the fault-tolerant scheme. The same projection functions as in section 7.1.2 are used to select elements of a token.

Property 3. *All processes are either alive or dead:* $M(Alive) + \Pi_1(M(Dead)) = PROC$.

Proof. The formula is a place invariant of the CPN. □

Property 4. *After a process q fails at stage s, it is restarted at the same stage.*

Proof. If a process q fails at stage S, by firing transition *fail*, the pair (q, S) "moves" from place *Process stage* to place *Dead*. The pair is kept and used by transition *restart_process*, therefore restarting the process at the same stage. Note that no other transition can use a token from place *Dead*. □

Property 5. *There is either one token in place Ready or Working processes:* $|M(Ready)| + |M(Working\ processes)| = 1$.

Proof. The formula is a place invariant of the CPN. □

Property 6. *The processes in the working list are either working or dead:*

$$\{q \in \Pi_1(M(Working\ processes)\} = \Pi_1^S(M(Process\ stage)) + M(Dead)$$

Proof. First, let us prove that there can only be one copy of a process q in place *Process Stage*. This place is initially empty. It can be filled only by transition *initialize* which removes the tokens corresponding to the processes in $workers(p-1)$ before filling it with tokens corresponding to processes in $workers(p)$, which are also in the list put in the list of place *Working Processes*. Note that this transition can only occur if there is no *Dead* process, since it requires all processes in $workers(p)$ to be in place *Process Stage*.

Then, transition *Work_finished* takes and puts a token in place *Process Stage*, both with the same process number. It increments the stage associated with the process, and removes the process from the *Working Processes* list.

Finally, transitions *fail* and *restart_process* are using the same process numbers as shown in Property 4.

The above arguments entail that processes in the list are members of the $workers(p)$ set. Those in *Process Stage* and *Dead* are exactly $workers(p)$. Hence the property. □

CHAPTER 8

Implementations

This chapter describes how the non-fault-tolerant and fault-tolerant algorithms presented in chapters 4, 5 and 6 were implemented, showing some examples and explaining the most important parts.

Before proceeding to the implementation of fault-tolerant mechanisms, it is first necessary to measure one single process execution to have a base reference for the next parallel implementations. Then, the implementation of the non-fault-tolerant algorithms was carried out. Knowing if the non-fault-tolerant implementations are scalable with our non-fault-tolerant implementations, they will provide base reference times that will be used to make the corresponding comparisons between fault-tolerant algorithms and the non-fault-tolerant ones. It also helps to get a better idea of where to inject the fault-tolerant mechanisms.

All the codes were written in the C programming language, using OpenMPI for non-fault-tolerant versions and ULFM for the fault-tolerant versions. The first utilized OpenMPI version is 4.1.0, but then a new release was published and it was decided to update the OpenMPI version to 5.1. The same happened with ULFM: the first utilized ULFM version is 4.1.0u1a1, but then the fault-tolerant extension was integrated into the OpenMPI main source code; so starting from this version, it is possible to get fault-tolerant subroutines with OpenMPI source, of course using the right configuration.

Initially, LU factorization algorithms were tested with OpenMPI version is 4.1.0 and ULFM 4.1.0u1a1, meaning that there was a comparison between two different software cores at runtime. When OpenMPI and ULFM versions were updated, there was only one software core at runtime, so comparisons for QR and Cholesky factorization algorithms were made only between a non-fault-tolerant core software and a fault-tolerant one.

All dense operations executed by a single process (local matrix factorization, matrix addition, linear equation solving, etc.) were not implemented. It was better to use OpenBLAS 0.3.8, which contains a collection of highly optimized subroutines dedicated to performing these dense operations locally and sequentially. In table 8.1, there are listed all OpenBLAS subroutines used in the implementations. All codes were compiled using `gcc` 8.3 with the `-O3` optimization flag.

Table 8.1: OpenBLAS subroutines used in the implementations.

Name	Description	Use
DAXPY	Perform the sum of two vectors	Matrix additions (or substractions) in LU/QR/Cholesky trailing matrix updates
DGEMM	Perform the product of two matrices	Matrix multiplications in LU/QR/Cholesky trailing matrix updates
DGEQRF	Computes the QR factorization of a matrix	Sub-matrix QR factorizations in TS/CA algorithms, generating Householder vectors used in trailing matrix update
DGETRF	Computes the LU factorization of a matrix	Sub-matrix LU factorizations in TS/CA algorithms, generating IPIV vectors used for row-swapping
DLARNV	Generates a random vector of real numbers, with uniform or normal distribution	Random input matrix generation (with uniform distribution) in LU/QR/Cholesky tests
DLASWP	Performs a series of row interchanges on a matrix	Row-swapping in TS/CA LU algorithms
DPOTRF	Computes the Cholesky factorization of a real symmetric positive definite matrix	Sub-matrix Cholesky factorizations in TS/CA algorithms
DSCAL	Scales a vector by a constant	Scalar vs matrix multiplication in QR trailing matrix updates
DTRSM	Solves a triangular system of linear equations	System solving to find lower triangular sub-blocks in LU/QR/Cholesky TS algorithms

8.1 Variables and Structures Definition

This section describes each fundamental variable for the development of the tall and skinny and communication-avoiding algorithms.

8.1.1 Variables and Structures for TS-Algorithms

As was mentioned previously, the thesis aims to implement fault-tolerant versions of matrix factorizations. For this, each matrix involved in the computations (input matrix, sub-matrix, Householder vectors, etc.) is seen as a large array of floating values, in which the elements of the matrix are store. As each matrix has particular fixed characteristics, it was defined a special structure to stored every matrix, which is presented in figure 8.1. As can be seen, the structure has the most important variables associated with a matrix, some of them are needed

to work with OpenBLAS subroutines and others are factorization-specific variables. Those algorithm-specific (array) variables are only used for the algorithm (LU/QR/Cholesky) the variable belongs to; if the algorithm does not need that variable, it will not initialized nor used.

```
typedef struct _matrix_data{
  int M;           // row size
  int N;           // column size
  double *A;       // stores the matrix A
  double *A_init;  // stores the initial matrix (copy of A)
  double *L;       // stores the matrix L (for Cholesky)
  double *U;       // stores the matrix U (for LU)
  double *R;       // stores the matrix R (for QR)
  double *Hv;      // stores the householder vectors (for QR)
  double *T;       // stores the T matrix (for QR)
  int *IPIV;       // stores the swapping index (for LU)
  double *TAU;     // stores the tau values (for QR)
  double *WORK;    // stores working values (for QR)
  int LDA;         // leading dimension of A
  int INFO;        // return value for an OpenBLAS subroutine call
  int Mb;          // row size (per block)
  int Nb;          // column size (per block)
}matrix_data;
```

Figure 8.1: Definition of C structure *matrix_data*

Then, we know that there will be processes exchanging information and they need a communication channel; so also implemented a structure to store all information related to the use of a classical MPI program. This special structure is shown in figure 8.2. The structure has relevant variables useful to maintain the process *state* in the MPI and Tall and Skinny environments. Particularly, the integer values *spawned* and *step_finished* have special use in a Tall and Skinny algorithm: variable *spawned* tells a process if he is a replacement of some other process that previously died; variable *step_finished* tells a process if he has already completed its work on the current step, in case a failure appeared and it is not necessary to (re-)execute again the operations the process already did.

The remaining variables used out of the structures for all Tall and Skinny algorithms are listed in figure 8.3. As with structure *matrix_data*, an algorithm will use only the variables it needs.

There are also some variables used only in TSQR to enable processes to know what is the current state of the algorithm, especially in data replication speaking. Those special variables are listed in figure 8.4. All of them are related to process knowledge of type *what processes are doing*. Processes use these variables to keep information about:

- it access to a Householder vector in order to distribute it.
- whether it should dump its current intermediate results.

```
typedef struct _MPI_data{
  int argc;                                    // argc value from main
  char **argv;                                 // argv array from main
  int processor_name_len;                      // size of the machine name
  char processor_name[MPI_MAX_PROCESSOR_NAME];// machine name
  int spawned;                                 // am I a spawned process?
  MPI_Comm world;                              // comm (row,column,global)
  int world_rank;                              // rank in comm
  int world_size;                              // elements in comm
  MPI_Status status;                           // receive calls status
  int step;                                    // step in a TS algorithm
  int step_finished;                           // did I finished the step?
  int *dest;                                   // partner ranks
  int *mirror;                                 // previous partner ranks
}MPI_data;
```

Figure 8.2: Definition of C structure *mpi_data*

```
int step_lim;                          // steps number to perform
int working_block[mpid->world_size];   // matrix index a process is working
int *dead_rank_list;                   // dead process ranks
int *surv_rank_list;                   // surviving process ranks

double time_exec_final;                // execution time
double time_tsch_final;                // tscholesky time
double time_tslu_final;                // tslu time
double time_tsqr_final;                // tsqr time
double time_mult_final;                // dgemm time
double time_solv_final;                // dtrsm time
double time_add_final;                 // daxpy time
double time_swap_final;                // dlaswp time
double time_rest_final;                // restoration time
double time_comm_final;                // communication time
double time_copy_final;                // copy time
double *time_steps;                    // step by step time
```

Figure 8.3: More useful variables in a Tall and Skinny algorithm.

- whether it has a partner to share information with (in case the number of processes is not a power of two).
- whether it should send a broadcast with its current intermediate results.
- whether it should receive a broadcast from a process with no partner in the previous step.
- whether it should forget the broadcast operation, in case some other process has already performed the broadcast with the same information.

The next sections give examples in which TSQR has a process number not a power of two and some data could be lost if no broadcast operation is performed.

```
int avail;                  // householder vector is available?
int already_back;           // did I dumped my results?
int broadcast;              // am I broadcasting?
int broadcast_dim[DIM];     // broadcast sizes
int broadcast_list[SIZE];   // ranks with no partner
int prev_alone_list[SIZE];  // previous ranks with no partner
int broadcast_count;        // how many processes received the broadcast?
int alone_count;            // how many ranks with no partner?
int first_alone;            // should I broadcast later?
int forget_broadcast;       // did someone else performed my broadcast?
```

Figure 8.4: Variables used in TSQR to allow householder vector broadcasts between processes.

Finalizing with the Tall and Skinny variables, we have the ones related to matrix concatenations in TSLU and TSQR algorithms. Figure 8.5 describes those special concatenation variables. They are related to let processes know the current size of the upper triangular matrix concatenation, since it increases after an exchange has been successful. They serve to store Householder vectors, too. It should be mentioned that again when the number of process is not a power of two, there will be processes that do not have an extra matrix to concatenate with (they had no partner at some point of the execution), and hence, the concatenation operation will not be performed. Finally, there are some variables dedicated to storing previous concatenations, with the aim of sending this concatenation to a respawned process in case an error occurs.

```
int concat_sizes[CONCAT];       // concatenation sizes
int l_calc;                     // should I compute L?
int q_calc;                     // should I compute Q?
int MU,NU;                      // U concatenation sizes
int MR,NR;                      // R concatenation sizes
int MU_prev,NU_prev;            // previous U concatenation sizes
int MR_prev,NR_prev;            // previous R concatenation sizes
double *concatU;                // stores the U concatenation
double *concatR;                // stores the R concatenation
double *concatU_prev;           // stores the previous U concatenation
double *concatR_prev;           // stores the previous R concatenation
int **Hv_sizes;                 // householder vector list
int Hv_list_size;               // householder vector list size
```

Figure 8.5: Variables used in TSLU/TSQR to enable intermediate results to be concatenated and remembered in case of an error.

8.1.2 Variables and Structures for CA-Algorithms

As mentioned before, Communication-Avoiding algorithms work over a process grid, using a minimal number of inter-process communications, minimizing the number of transferred messages across the grid. Processes communicate over a 2D process grid and use their coordinates on this grid to determine what they are doing at a given step (panel factorization, trailing matrix update). We know that there will be processes exchanging information in their row and/or column communicators and that they will perform different tasks at the same time, depending on their position on the grid and the current matrix state. The process grid has specific characteristics, so there were defined a set of particular variables to represent it. Variables related to the process grid are presented in figure 8.6.

```
1  MPI_Comm cart_2D;              // grid communicator
2  MPI_Comm row_comm;             // row communicator
3  MPI_Comm col_comm;             // column communicator
4  MPI_Comm panel_comm;           // panel communicator
5
6  int stage;                     // current matrix stage
7  int rowM;                      // number of rows in the grid
8  int colN;                      // number of columns in the grid
9  int rowNum;                    // process row number in the grid
10 int colNum;                    // process column number in the grid
11 int currRow;                   // current processing row
12 int currCol;                   // current processing column
13
14 double *panel;                 // stores the current panel
15 int panelRowNum;               // process row number in the panel
16 int panelIndex;                // panel index a process is working
17 int panelM;                    // number of rows in the panel
18 int panelN;                    // number of columns in the panel
19
20 double time_cach_final;        // cacholesky time
21 double time_calu_final;        // calu time
22 double time_caqr_final;        // caqr time
23 double time_exec_tm_final;     // trailing matrix update time
24 double time_read_final;        // reading time
25 double time_write_final;       // writing time
```

Figure 8.6: Variables used in Communication-Avoiding algorithms to form the process grid with its respective communicators and maintain the current state of a matrix.

These variables are mainly defined to maintain the state the grid has through time. Particularly, the integer value *stage* serves to know what operation is now being executed over the grid. For example, from the starting point of an algorithm, processes belonging to the first panel will generate it and they will perform the corresponding Tall and Skinny algorithm, depending on the factorization is being applied. Other processes not belonging to the currently processed column, will be waiting for the panel factorization to finish to advance to the next stage, preparing themselves to exchange information across row communicators.

Different specific stages for every matrix factorization were defined, as specified in figure 8.7. There are identical stages on different factorizations, like the column division stage; the main difference between them is the trailing matrix update preparation.

```
#define STAGE_COMM_DIVISION          0 //column comm division (Cholesky)
#define STAGE_PANEL_GENERATION       1 //panel generation (Cholesky)
#define STAGE_FTTSCH_INITIALIZATION  2 //FT environment init (Cholesky)
#define STAGE_FTTSCH_EXECUTION       3 //FT facto exec (Cholesky)
#define STAGE_SEND_RECEIVE_L         4 //L broadcast (Cholesky)
#define STAGE_SPLIT_COL_COMM         5 //column comm division (Cholesky)
#define STAGE_UPDATE_TM              6 //trailing matrix update (Cholesky)
#define STAGE_LAST_BLOCK             7 //final block? (Cholesky)

#define STAGE_COMM_DIVISION          0 //column comm division (LU)
#define STAGE_PANEL_GENERATION       1 //panel generation (LU)
#define STAGE_FTTSLU_INITIALIZATION  2 //FT environment init (LU)
#define STAGE_FTTSLU_EXECUTION       3 //FT facto exec (LU)
#define STAGE_SEND_RECEIVE_L         4 //L broadcast (LU)
#define STAGE_SEND_RECEIVE_U         5 //U broadcast (LU)
#define STAGE_UPDATE_TM              6 //trailing matrix update (LU)
#define STAGE_LAST_BLOCK             7 //final block? (LU)

#define STAGE_COMM_DIVISION            0 //column comm division (QR)
#define STAGE_PANEL_GENERATION         1 //panel generation (QR)
#define STAGE_FTTSQR_INITIALIZATION    2 //FT environment init (QR)
#define STAGE_FTTSQR_EXECUTION         3 //FT facto exec (QR)
#define STAGE_SEND_RECEIVE_HV_TAU_SIZE4 //householder size broadcast (QR)
#define STAGE_SEND_RECEIVE_HV_TAU      5 //householder vector broadcast(QR)
#define STAGE_SPLIT_COL_COMM           6 //column comm division (QR)
#define STAGE_UPDATE_TM_HOUSEHOLDER    7 //trailing matrix update (QR)
#define STAGE_LAST_BLOCK               8 //final block? (QR)
```

Figure 8.7: Possible *stages* utilized in Communication-Avoiding algorithms, depending on the factorization that is being applied.

Finally, again in figure 8.6, variables referring to the number of rows/columns the grid has and the current processing row/column are used to know what are the already processed columns (panels), what operation a process is going to perform, and which processes have finished their tasks. A more detailed description of the grid construction and representation with these variables is shown in figure 8.8. The remaining variables serve to know the panel size and time measurements for Communication-Avoiding specific times.

8.2 Tall and Skinny Matrix Factorizations

In this section it will be described the most relevant information about the Tall and Skinny implementations.

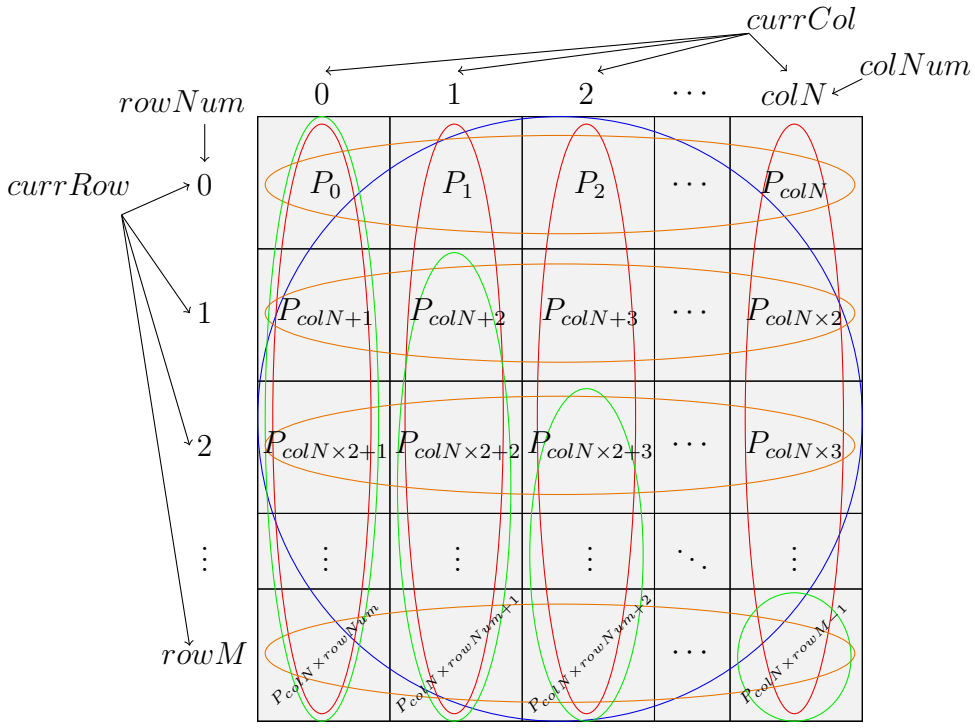


Figure 8.8: Communication-Avoiding grid partitioning description: red=column communicators, orange=row communicators, green=panel communicators, blue=global communicator; $rowM = M - 1$, $colN = N - 1$, $\{currRow, rowNum\} \in [0, rowM]$, $\{currCol, colNum\} \in [0, colN]$

8.2.1 TSLU and FT-TSLU

As described in previous section 4.1, the most important functions to correctly develop TSLU/FT-TSLU algoritms are *myPartner*, *sendrecv*, *concatenate*, *update*, *restoreFailed* and *backup*. With all these functions implemented (and many others that are not described here), TSLU was a little reduced to design the logic the algorithm has, like selecting the pairs of processes to exchange data, write some functions according to equations in 3.1 and 4.1, etc. Particularly, in FT-TSLU implementation, the hardest part was to design the fault-tolerant mechanisms to recover the current computation. Great care had to be taken in the information contained in each process over time because not all processes have the same data in the initial steps. This is one of the reasons why it was decided to use previous states of the matrix concatenations, for example.

Special care was also taken when having a number of processes that is not a power of two. In this case, the process without a partner will only compute its respective sub-matrix factorization and simulate it has finished the exchange procedure, to later continue to the next step and find a partner. In the next section, some examples will be presented in which this *not a power of two* number of processes becomes a hard issue to solve.

8.2.2 TSQR and FT-TSQR

As described in section 5.1, TSQR/FT-TSQR uses the same set of functions as TSLU/FT-TSLU to work correctly, but compared to LU previous algorithms, the main and most important difference resides in that TSQR/FT-TSQR counts with a more sophisticated mechanism to decide when to exchange information and when not to do it, since the number of processes is not a power of two. So, the hardest part in TSQR/FT-TSQR implementation was the design of the fault-tolerant mechanisms to avoid loss of data. For a better understanding about this issue, figure 8.9 shows a TSQR execution example with $t = 3$ processes.

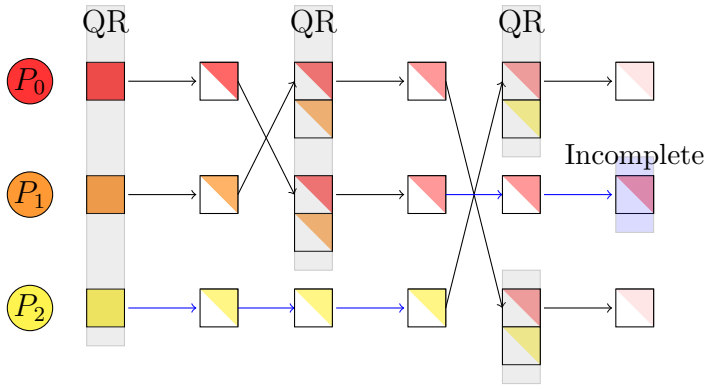


Figure 8.9: FT-TSQR example with $t = 3$ processes.

We can see that each process P_i starts with its own sub-matrix $A_{i,s}$ to compute a local QR decomposition, generate local sub-matrices $H_{i,s}$, $R_{i,s}$ and, in the case of P_0 and P_1, share their results between them. In the case of process P_2, it only advances to the next step keeping its current results. Then process P_0 and P_1 execute the second QR factorization over the $R_{i,s}$ matrix concatenation. Process P_2 does not re-execute again the QR factorization. When the next partner selection is carried out, P_0 exchanges with P_2 and P_1 only advances to the next step. At the end of the execution, P_0 and P_2 have the same results, but as no more exchanges will take place, process P_1 is not able to access the information P_2 computed, so the final result for P_1 is incomplete and it is not possible to share it with other processes in the Communication-Avoiding algorithm, because the incorrect result in TSQR will produce an incorrect result in CAQR. The Householder vectors $H_{i,s}$ held by every process at each step in this same example can be seen in table 8.2.

Table 8.2: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 3$ processes.

	0	1	2
P_0	H_0, H_1	$H_{0,1}, H_2$	$H_{0,1,2}$
P_1	H_0, H_1	$H_{0,1}$	$H_{0,1}$
P_2	H_2	$H_{0,1}, H_2$	$H_{0,1,2}$

More examples of this behaviour are presented in figures 8.10 and 8.11, with Householder vectors $H_{i,s}$ in tables 8.3 and 8.4, respectively. In both cases, the lower-most process stays without a partner until the last step, in which it exchanges its result with the upper-most process (normally P_0) and get the final result correctly, but it implies that the rest of the processes (all intermediate processes) cannot have a partner and hence, they cannot have the final result.

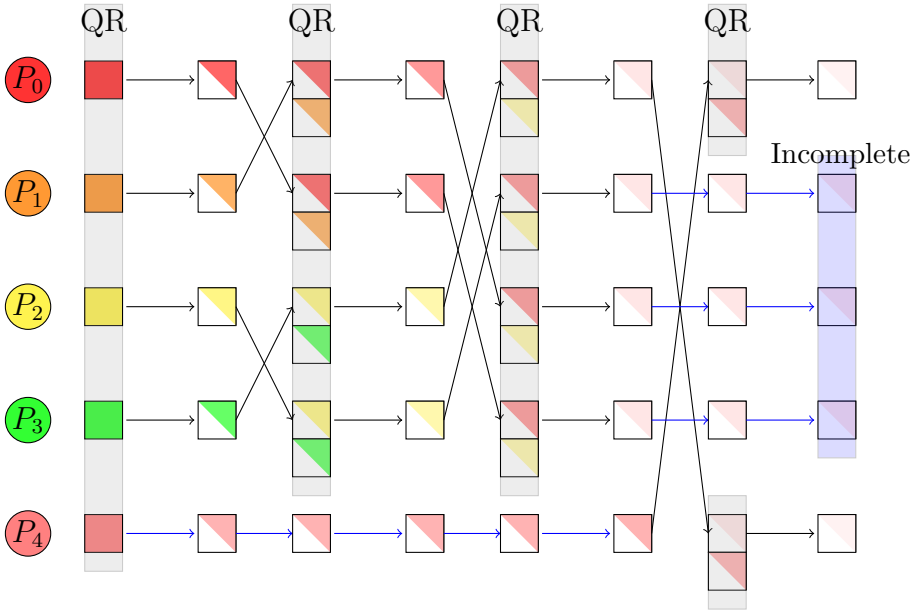


Figure 8.10: FT-TSQR example with $t = 5$ processes.

Table 8.3: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 5$ processes.

	0	1	2	3
P_0	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_4$	$H_{0,1,2,3,4}$
P_1	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_2	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_3	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_4	H_4	-	$H_{0,1,2,3}, H_4$	$H_{0,1,2,3,4}$

Slightly different examples are shown in figures 8.12 and 8.13, with Householder vectors $H_{i,s}$ in tables 8.5 and 8.6, respectively. In both cases, in the first step, all processes can have a partner to exchange results with, but starting from step $s = 1$, the two lower-most processes stay without a partner until the last step, in which finally they find a partner and exchange their results with the two upper-most processes (normally P_0 and P_1) and get the final result correctly; but (again) this implies that the rest of the intermediate processes cannot have a partner and they cannot have the final result.

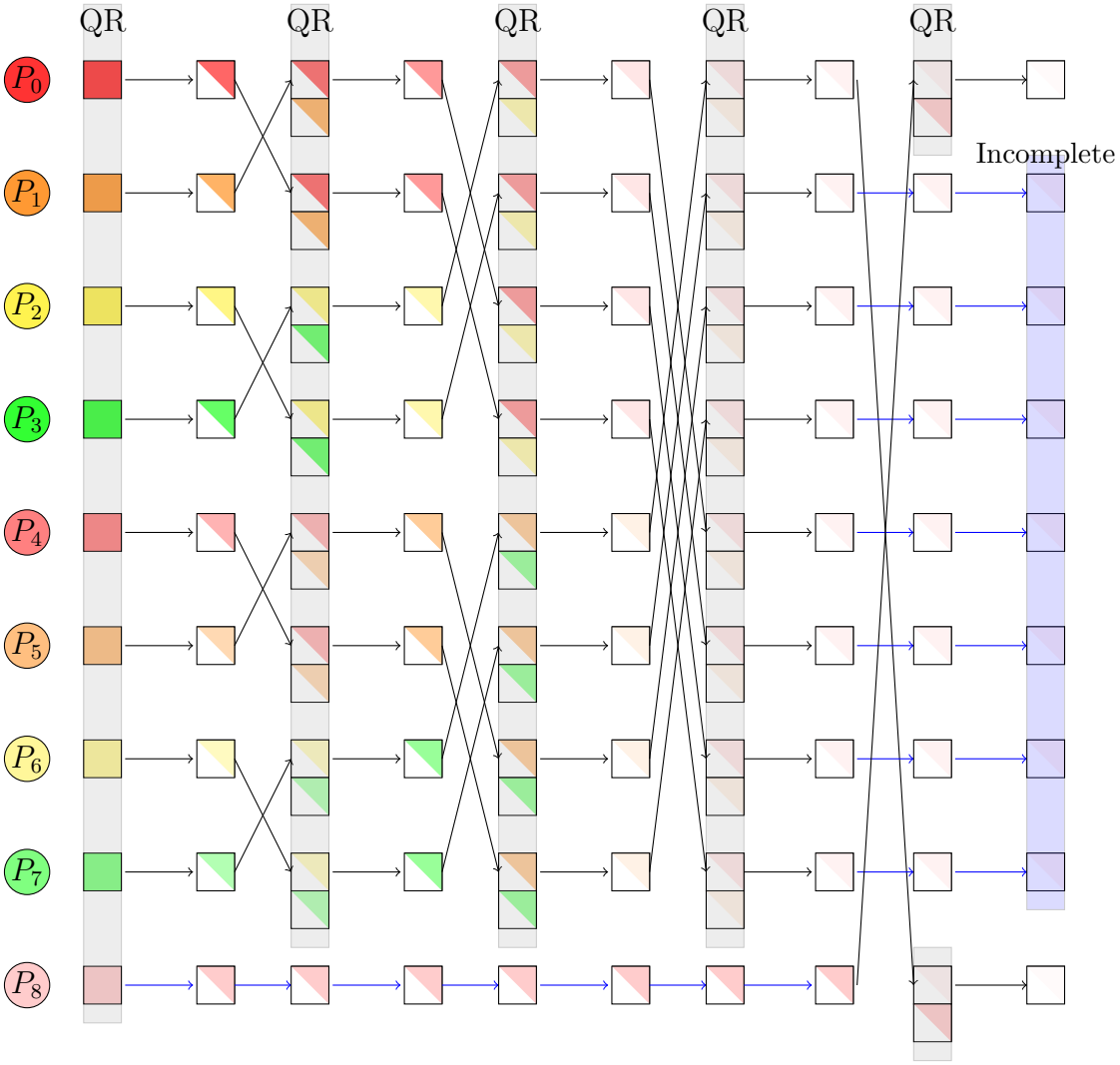


Figure 8.11: FT-TSQR example with $t = 9$ processes.

Another incomplete data case is illustrated in figure 8.14 with Householder vectors in table 8.7. In this case, process P_6 stays without partner only until step $s = 1$; then process P_5 is the one without partner, losing the bottom-part of the concatenation; finally, in the last step, process P_3 loses its partner and it loses essential information to get the final result too. So at the end, processes P_1, P_3 and P_5 have incomplete information: P_5 and P_3 because they lost their partner, and P_1 because P_5 sent him incomplete results.

With this set of examples, it is obvious that loss of data and redundancy can take place if we do not take care of TSQR implementation. During the TSQR implementation, to avoid loss of redundancy, a great care was taken regarding the information contained in each process with the advancement of the algorithm over time. Taking into consideration that not all processes can have access to the same data at different steps and that not all processes

Table 8.4: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 9$ processes.

	0	1	2	3	4
P_0	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}, H_8$	$H_{0,1,2,3,4,5,6,7,8}$
P_1	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_2	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_3	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_4	H_4, H_5	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_5	H_4, H_5	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_6	H_6, H_7	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_7	H_6, H_7	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_8	H_8	-	-	$H_{0,1,2,3,4,5,6,7}, H_8$	$H_{0,1,2,3,4,5,6,7,8}$

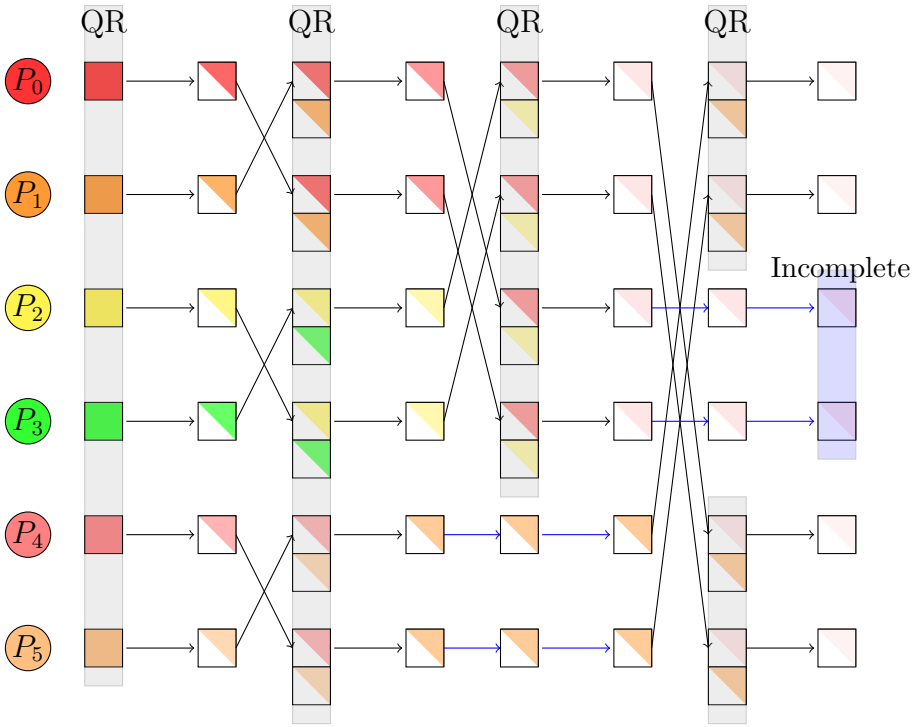


Figure 8.12: FT-TSQR example with $t = 6$ processes.

can get the final result, it was decided to implement a special distribution mechanism, using the variables described in figure 8.4. This mechanism allows "lonely" processes to get ready for sending broadcasts (in future steps) to future "lonely" processes, avoiding data loss for those future "lonely" processes. For example, seeing figure 8.14, when process P_6 realizes he has no partner, he "lights" a flag telling him it is necessary to send his results to the "lonely" processes in the next step (P_5). When step $s = 1$ is achieved, P_5 realizes he is now alone

Table 8.5: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 6$ processes.

	0	1	2	3
P_0	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$
P_1	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$
P_2	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_3	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_4	H_4, H_5	$H_{4,5}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$
P_5	H_4, H_5	$H_{4,5}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$

Table 8.6: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 10$ processes.

	0	1	2	3	4
P_0	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}, H_{8,9}$	$H_{0,1,2,3,4,5,6,7,8,9}$
P_1	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}, H_{8,9}$	$H_{0,1,2,3,4,5,6,7,8,9}$
P_2	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_3	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_4	H_4, H_5	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_5	H_4, H_5	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_6	H_6, H_7	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_7	H_6, H_7	$H_{4,5}, H_{6,7}$	$H_{0,1,2,3}, H_{4,5,6,7}$	$H_{0,1,2,3,4,5,6,7}$	-
P_8	H_8, H_9	$H_{8,9}$	-	$H_{0,1,2,3,4,5,6,7}, H_{8,9}$	$H_{0,1,2,3,4,5,6,7,8,9}$
P_9	H_8, H_9	$H_{8,9}$	-	$H_{0,1,2,3,4,5,6,7}, H_{8,9}$	$H_{0,1,2,3,4,5,6,7,8,9}$

Table 8.7: Storing of Householder vectors $H_{i,j}$ in TSQR after an exchange between partners is successful, with $t = 7$ processes.

	0	1	2	3
P_0	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6}$	$H_{0,1,2,3,4,5,6}$
P_1	H_0, H_1	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$
P_2	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}, H_{4,5,6}$	$H_{0,1,2,3,4,5,6}$
P_3	H_2, H_3	$H_{0,1}, H_{2,3}$	$H_{0,1,2,3}$	-
P_4	H_4, H_5	$H_{4,5}, H_6$	$H_{0,1,2,3}, H_{4,5,6}$	$H_{0,1,2,3,4,5,6}$
P_5	H_4, H_5	$H_{4,5}$	$H_{0,1,2,3}, H_{4,5}$	$H_{0,1,2,3,4,5}$
P_6	H_6	$H_{4,5}, H_6$	$H_{0,1,2,3}, H_{4,5,6}$	$H_{0,1,2,3,4,5,6}$

and some other process is sending some data, so P_5 prepares to receive data and P_6 sends the data, continuing then with his normal exchange with P_4. With this, P_5 can generate the complete intermediate results. Later, in $s = 2$, P_5 is in charge of sending his data to the lonely process P_3. When exchanges are completed, all processes can have the correct final result.

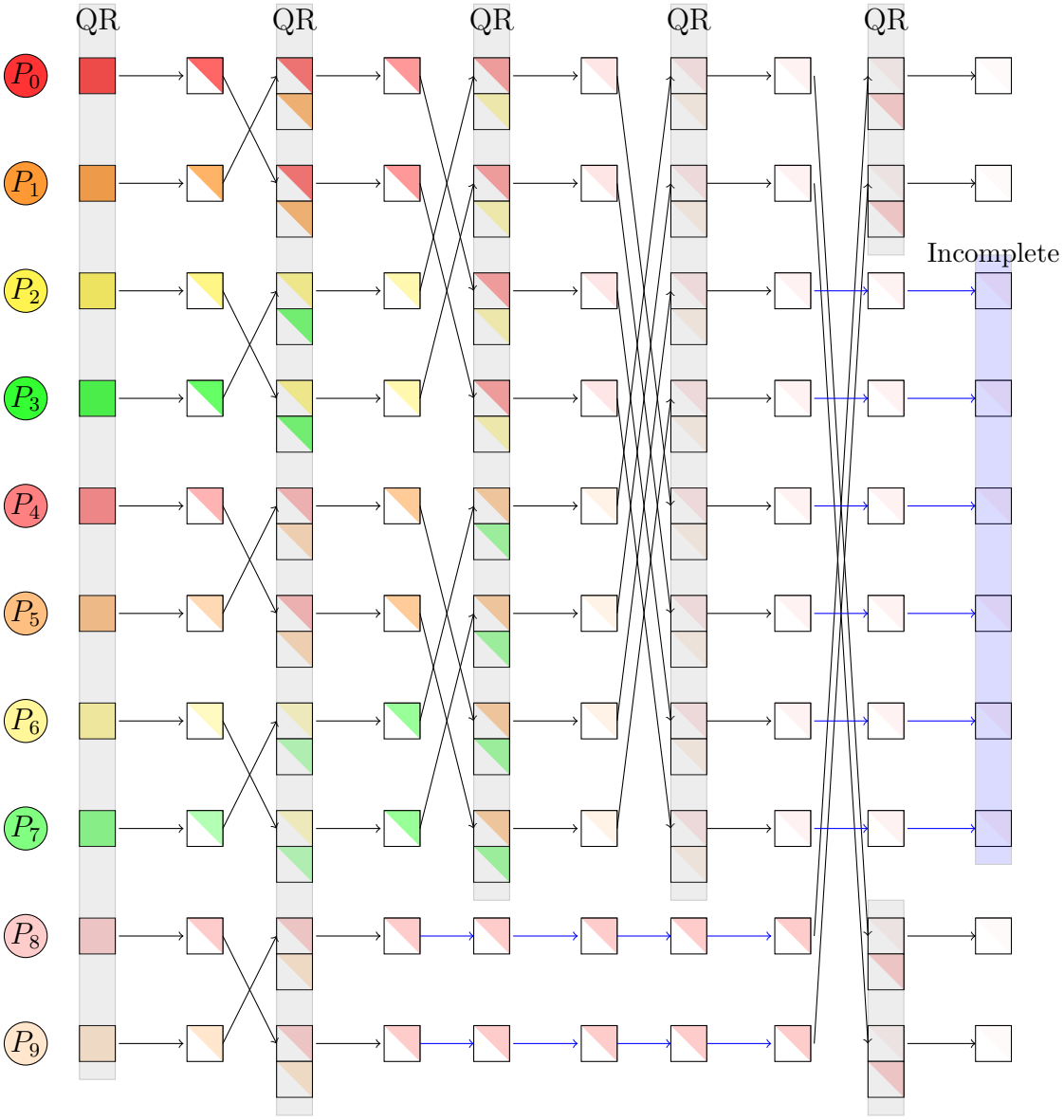


Figure 8.13: FT-TSQR example with $t = 10$ processes.

This mechanism was developed to handle the Householder vectors $H_{i,j}$ used in the CAQR/FT-CAQR algorithms, in particular in the trailing matrix update subroutine. The great advantage is that all processes can find the final upper triangular matrix $R_{i,s}$ correctly. It should be mentioned that every Householder vector $H_{i,j}$ generated during the execution is also written on the local media storage device, as part of the fault-tolerant mechanisms.

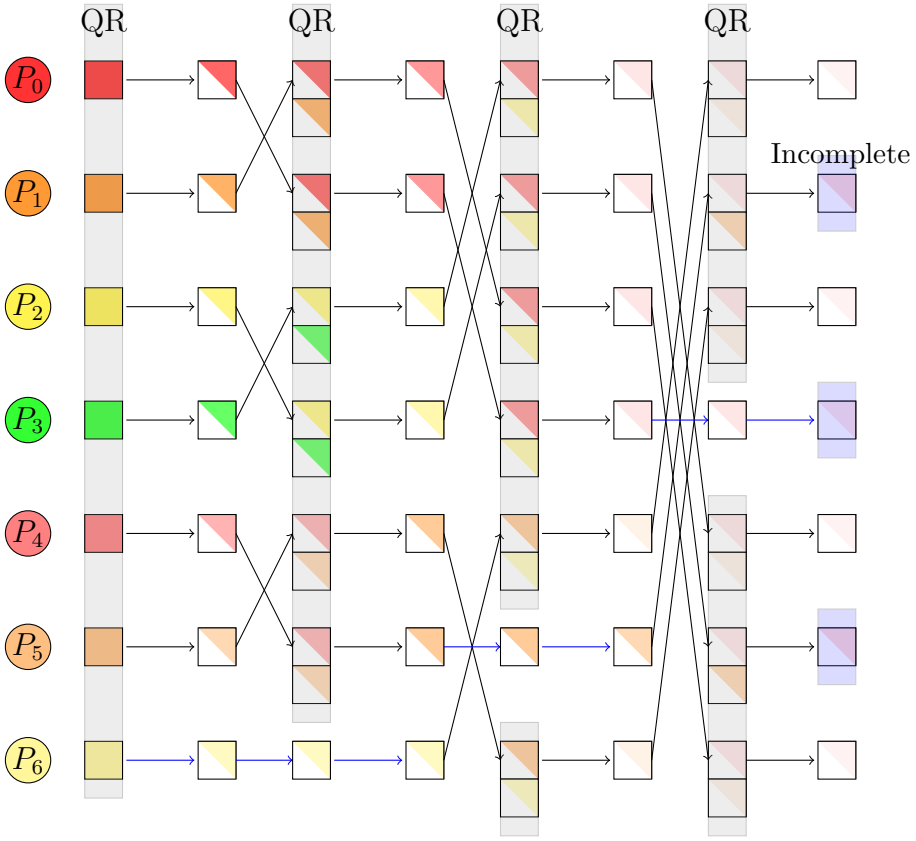


Figure 8.14: FT-TSQR example with $t = 7$ processes.

8.2.3 TSCH and FT-TSCH

TSCH/FT-TSCH algorithms were the last ones implemented. As described in the previous section 6.1, TSCH/FT-TSCH use the same set of functions as TSLU/FT-TSLU/TSQR/FT-TSQR, both for communication and for error recovery. But the biggest difference, compared to both LU and QR algorithms is that the TSCH algorithm only performs one Cholesky factorization over the sub-block located at the main diagonal of the matrix; that is, it performs the Cholesky factorization over the upper sub-block of the panel, using only one step $s = 0$. This property is derived from the shape the matrix has (see section 3.3). So there is only one process executing this factorization (normally P_0). In the end, the upper process distributes the found lower triangular $L_{i,0}$ to processes below him to let them compute their corresponding sub-block $L_{i,j}$ with a linear solving operation.

In this case, the number of processes involved in the algorithm it is not important, since only the first process is in charge of running the Cholesky factorization and distribute its results. The others only keep waiting for receiving the results. If an error occurs, all processes can restore the failed one without taking care of any intermediate result; they only take care of the basic process information (rank, which sub-block was working, etc.), the restored one, and if the operation has been completed or not.

8.3 Communication-Avoiding Matrix Factorizations

This section describes the most relevant information about the Communication-Avoiding implementations.

8.3.1 CALU and FT-CALU

CALU/FT-CALU algorithms were the first Communication-Avoiding implementations. The chosen perspective to represent an environment in the form of a process grid is the one illustrated in figure 8.8. As described in section 4.2, the most important functions to develop CALU/FT-CALU algorithms are *stillHasPanel*, *nextPanel*, *TSLU*, *broadcast*, *updateStage*, *readBackup*, *backup* and *restoredFailed*. With all these functions implemented, CALU/FT-CALU most involved parts to implement were dealing with the amount of space needed to store the sub-blocks in memory and local storage, ensuring that sub-block distribution by rows and columns is completed correctly, and enabling each process to know exactly what tasks he is executing in a particular stage. Particularly, in FT-CALU implementation, the most challenging part was to design, implement and test the functioning of the fault-tolerant mechanisms to recover the current state of the grid environment, especially knowing that, if a process fails, it will fail in all communicators (row, column, global, maybe panel). All the logic to respawn the failed process, let it know in which position it was on the grid before the failure, and reintegrate it in its respective communicators was the hardest challenge. Fortunately, the proposed solution to deal with these issues injects low overhead (depending on the number of errors) and performance results scale correctly as the number of processes increases (see experimental results in section 9.3).

8.3.2 CAQR and FT-CAQR

CAQR/FT-CAQR algorithms were the second (and hardest) Communication-Avoiding implementations. As described in the previous section 5.2, CAQR/FT-CAQR uses the same set of functions as CALU/FT-CALU to work correctly. Compared with LU algorithms, CAQR/FT-CAQR uses much more space (memory and local media) to store all Householder vectors generated in TSQR/FT-TSQR, which are necessary for the trailing matrix update. To perform correctly the CAQR trailing matrix update, each process belonging to a non-processed column (at the right of the current panel) must share its Householder vectors with the other processes belonging to the same column. This trailing matrix update follows the same communication pattern as a Tall and Skinny algorithm but is now performed in all non-processed columns, instead of the current panel. As previously seen in algorithm 7, we can see that local operations require more computational time when performing this procedure. Also, the same mechanism described in 8.2.2 was utilized to deal with trailing matrix updates with a number of a process not a power of two. The fault-tolerant mechanisms in FT-CAQR are the same described in FT-CALU (see 8.3.1). Experimental results are shown in section 9.4.

8.3.3 CACH and FT-CACH

CACH/FT-CACH algorithms were the last ones being implemented. As described in section 6.2, CACH/FT-CACH use the same set of functions as CALU/FT-CALU/CAQR/FT-CAQR, both for communication and for error recovery. The main difference, compared to both LU and QR algorithms, is that the CACH algorithm only uses processes below the main diagonal to perform the Cholesky factorization of the entire matrix. That is, processes located above the main diagonal are not used in the algorithm, or they could be used as backup processes in the fault-tolerant algorithm. If the second option is chosen, in case a lower process crashes, its transposed process can act as the replacement, since it has the same (transposed) sub-block to work with, derived from the shape the matrix has (see section 3.3). With this, we can see that only processes located in the main diagonal perform the Cholesky factorization and the rest perform matrix addition and multiplication to update the *almost half* of the trailing matrix and proceed with the algorithm. Finally, as FT-CAQR, after a sub-block of the entire matrix has been completed, FT-CACH dumps the result in the local media storage device. Experimental results are shown in section 9.5.

CHAPTER 9

Experiments

By having the implementations of both non-fault-tolerant and fault-tolerant algorithms, the next step is to run a performance evaluation to verify that the overhead injected by restoration mechanisms and the overhead on failure-free execution is minimal. This chapter specifies the selected architecture for testing the algorithms. Time and performances graphics obtained in the different algorithms versions are shown too.

9.1 Grid5000 Test Architecture

Grid5000 is a large-scale infrastructure available to address research experiments oriented to diverse areas of computer science. It is supported by a scientific interest group hosted by *Inria* and including *CNRS*, and several Universities, as well as other organizations (see `https://www.grid5000.fr`). It is connected by the *Renater French Education and Research Network*. Grid5000 is a computer science project dedicated to the study of grids. The main areas using it for testing are parallel and distributed computing (Cloud, HPC, Big Data or AI).

Algorithm implementations described in chapter 8 were evaluated on this architecture. The cluster *Gros* in *Nancy* site was chosen for testing, because it is more likely to have access to a large amount of nodes, which is mandatory to launch experiments with large amounts of processes. It counts with the hardware specified in table 9.1. Nodes on the selected cluster run a *Debian 9.2* Linux-based operating system.

Table 9.1: Grid5000 Gros cluster hardware characteristics

Cluster	Nodes	CPU	Cores	Memory	Storage	Network
gros	124	124 x Intel Xeon Gold 5220 Cascade Lake-SP	18	96 GiB	480 GB SSD + 960 GB SSD	2 x 25 Gbps (SR-IOV)

9.2 Input and Measured Times

Input matrix sizes used for testing are: $16k \times 16k$, $32k \times 32k$, $64k \times 64k$ and $100k \times 100k$; OpenBLAS *DLARNV* subroutine with uniform distribution and fixed seed was used to generate them, except for the input matrices for Cholesky algorithms tests. For generating matrices with the particular properties the Cholesky factorization needs, algorithm 13 was implemented.

Algorithm 13: Random Positive-Definite Symmetric Matrix

Data: Integer M

1 $I =$ `identityMatrix(`M`)` ;
2 $A =$ `randomMatrix(`M`)` ;
3 $A^T =$ `transposeMatrix(`A`)` ;
4 $A = (\frac{1}{2} * (A + A^T)) + (M * I)$;
5 **return** A;

The number of processes utilized for every test varies from 64 to 1024, corresponding to 4 to 57 Gros nodes. In Communication-Avoiding algorithm tests, executions are launched with a number of processes with an exact square root $(64, 144, \cdots, 784, 1024)$ to be able to form an exact square process grid. Every experiment was run 10 times and the plots presented here were computed taking the average values of all executions. The maximal throughput achieved in every algorithm were computed with the next equation:

$$\tau_{max} = \frac{2MN^2}{t_b * 10^9} \tag{9.1}$$

where M and N are the number of rows and lines in the original matrix, respectively, t_b represents the base time chosen for an algorithm execution (failure-free with OpenMPI, failure-free with ULFM or one failure with ULFM) and τ_{max} is the maximal throughput represented in Gigaflops.

Failures were injected by sending a *SIGKILL* signal to the processes. In this case, the operating system sends closing notifications on the TCP sockets used by the run-time environment and the failures are detected immediately. In real life, detecting failures is a challenge on its own and this cannot happen when a failure occurs. A more advanced failure detection mechanism could be used[6]. Although there exist more realistic techniques to inject failures, we chose not to use them in order to isolate the *algorithmic* cost from the *system* cost, and evaluate the performance of the algorithms separately from some system-specific costs.

In every test, for every algorithm, elapsed times to perform a particular task were measured:

- Execution time: total execution time, since MPI environment is created until all processes finish their execution. It includes all the initializations, including memory allo-

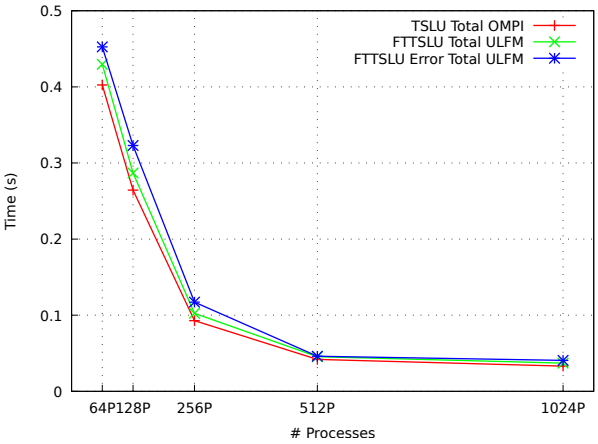
cations for storing matrices, DLARNV call, etc., so it is not used to evaluate scalability and injected overheads.

- DGEMM, DTRSM, DAXPY, DLASWP times: total time spent in calls to OpenBLAS subroutines.
- TSCH, TSLU, TSQR times: total time spent in panel factorizations, since the first step is started until all processes finish their computations. It includes all the communications, memory allocations, etc. It is used to evaluate scalability and injected overheads in panel factorization algorithms.
- Step by step time: total time spent to execute successfully a step in panel factorization. It includes all the communications, matrix concatenations, etc.
- CACH, CALU, CAQR times: total time spent for the rectangular factorizations, since the grid environment is created until all processes finish their executions. This time is used to verify scalabitily.
- Trailing Matrix Update time: total time spent updating the trailing matrix.
- Restoration time: total spent time to respawn dead processes, restore communicators and share required information with the new ones.
- Communication time: total time spent to communicate between processes (for exchanging information across communicators).
- Copy time: total time spent to copy (sub-)matrices with *memcpy* (backup, restoration, sub-block extraction, etc.).
- Writing/Reading time: total time spent to write/read a backup. Various strategies were explored and no significant difference in terms of performance was perceived.

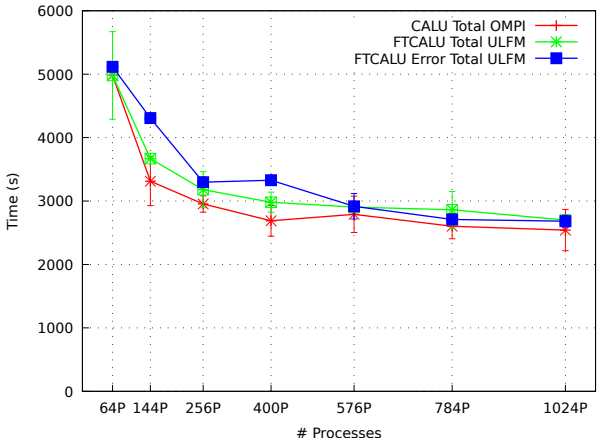
In the next sections, graphics with the most relevant times are presented.

9.3 LU Tall and Skinny/Communication-Avoiding Executions

Figure 9.1a displays execution times for TSLU/FT-TSLU and figure 9.1b shows execution times for CALU/FT-CALU. It can be noted that both algorithms scale satisfactorily as the number of processes increases. It can also be seen that OpenMPI and ULFM failure-free executions are similar, and when a failure occurs the ULFM execution gets slightly degraded. This means that added fault tolerance mechanisms generate a small overhead over non-fault-tolerant algorithms, but it is a minimal overhead compared against the total execution time; and we have the advantage of having gained fault tolerance. As both algorithms perform parallel LU decompositions over sub-blocks, complexity for factorizing panels is reduced successfully with small overhead.



(a) Execution times for TSLU/FT-TSLU



(b) Execution times for CALU/FT-CALU

Figure 9.1: Total times obtained in (FT-)TSLU/(FT-)CALU algorithms (input: $100k \times 100k$).

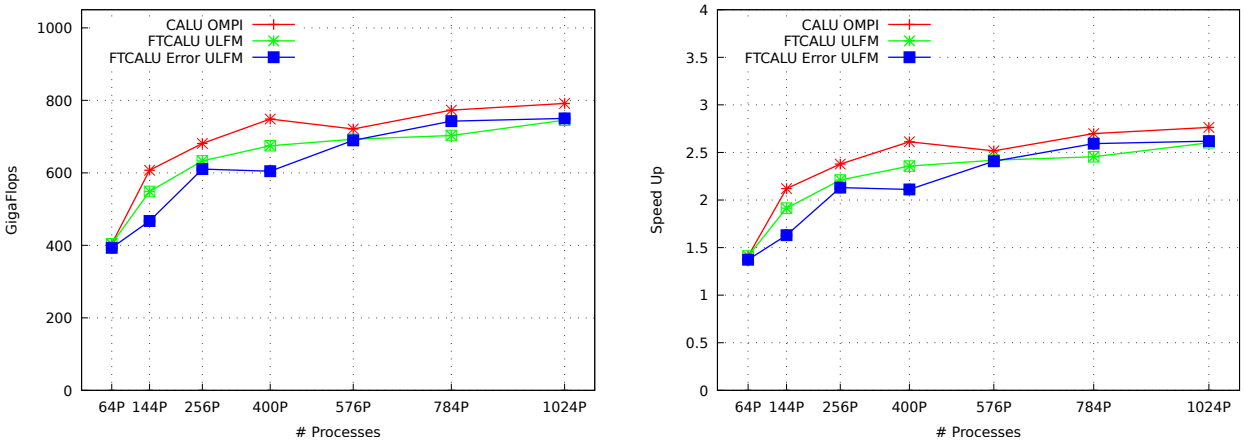
With a small number of processes it is noticeable that non-fault-tolerant version slightly overcomes fault-tolerant versions, but as the amount of processes grows, this overhead tends to shrink, even when a failure occurs. When an error is detected and corrected, execution times increase due to all the extra information that must be shared between processes to restore the previous state the failed process had. Also, process synchronization on communicators must be done to successfully restore the processes grid environment.

Figures 9.2a and 9.2b present the maximal flops and speed up (respectively) achieved for both algorithms. As expected, both measures increase with the amount of processes used; but just as it happened with execution times, when a failure occurs on FT-TSLU execution, it decreases its performance at the level of TSLU or even less.

Execution times for restoring processes were measured too. On every test including a failure, a script in charge of sending *SIGKILL* signals to FT-CALU processes was used, in order to simulate pseudo-random crashes. The selected number of errors detected and corrected in every test was 1. Tests with more variations are those with more processes. When an error is detected and a new process is spawned, the recovery step involves data exchanges to restore the state of the failed process before the crash. This recovery time is presented in figure 9.3. It can be seen time increases slightly as the number of processes increase, because of the synchronizing operations on the communicator. As the size of the grid increases, the recovering time of the process environment also grows, but without reaching even one second, even with the bigger process number.

Figure 9.4 presents a comparision of our FT-CALU executions with all tested matrix sizes. As expected, scalability swiftly increases as the amount of processes increase as well as the matrix size, specially failure-free executions. When a failure occurs, performance looks minimally hit compared against a failure-free FT-CALU execution.

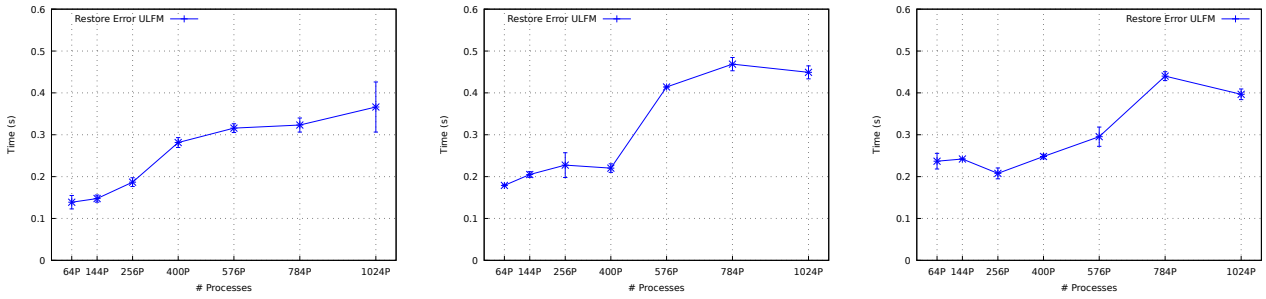
In general, the overall performance increases with the number of processes. As expected, the scalability and the performance per process increase as the size of the matrix increases



(a) Maximal throughput reached in CALU/FT-CALU executions

(b) Maximal speed-up reached in CALU/FT-CALU executions

Figure 9.2: Maximal throughput and speed-up reached in (FT-)CALU algorithms (input: $100k \times 100k$).



(a) Restoration times for TSLU/FT-TSLU (input: $32k \times 32k$)

(b) Restoration times for TSLU/FT-TSLU (input: $64k \times 64k$)

(c) Restoration times for TSLU/FT-TSLU (input: $100k \times 100k$)

Figure 9.3: Restoration times obtained in FT-CALU algorithm for all inputs, with one error.

and the algorithms' good parallelism properties get satisfactory performance on the parallel platform, for both fault-tolerant and non-fault-tolerant versions of TSLU and CALU.

9.4 QR Tall and Skinny/Communication-Avoiding Executions

Figure 9.5 compares the execution times of TSQR/FT-TSQR (figure 9.5a) and CAQR/FT-CAQR (figure 9.5b) implementations, respectively. Like TSLU/FT-TSLU, TSQR/FT-TSQR scale well and the relative overhead of the proposed fault-tolerance mechanisms are very small. It must be noticed that the impact of a failure on the total execution time is also very small. This result shows (again) that exploiting intrinsic redundancies that exist in

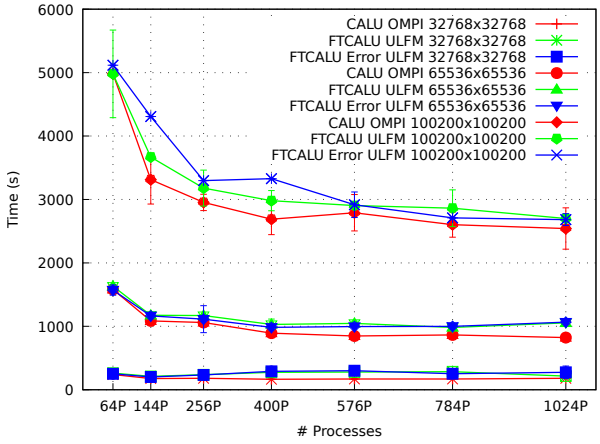
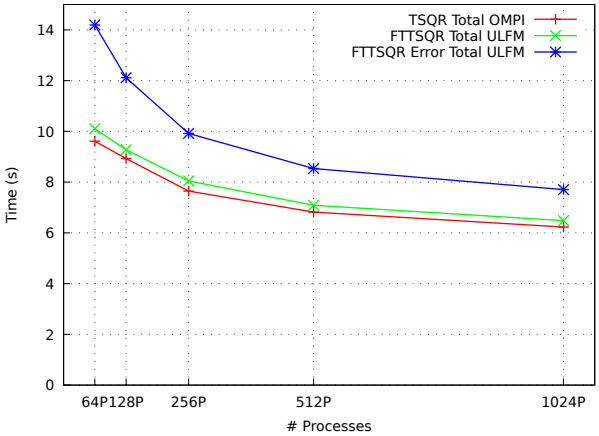
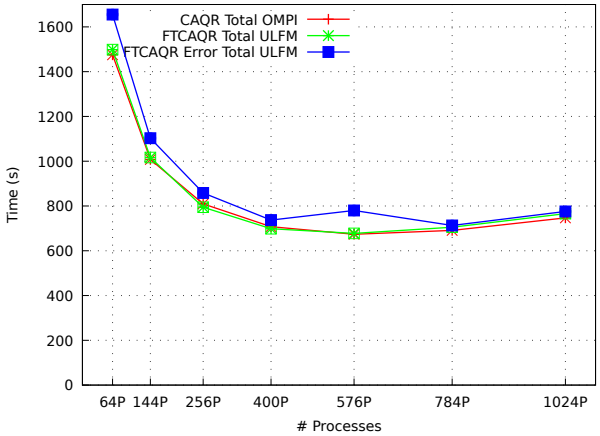


Figure 9.4: Comparision of the scalability of FT-CALU for different matrix sizes.

Tall and Skinny algorithms is a good approach to recover from failures and proceed with the rest of the execution. However, CAQR/FT-CAQR presents a different behavior; with a small amount of processes, total time starts decreasing, proving good scalability, but as the number of process grows the execution time also starts increasing. In this case, with an input size $16k \times 16k$, the best result is given with 576 processes; that is, a process grid of size 24×24. This increase begins to grow due to the operations of backup and distribution of Householder vectors (which implies disk reading), in addition to the dense operations the trailing matrix update requires (matrix multiplication, addition, transposing, etc.). So, in CAQR/FT-CAQR, when applying the algorithm to "small" matrices, it is recommended to find the best number of processes to achieve good scalability.



(a) Execution times for TSQR/FT-TSQR

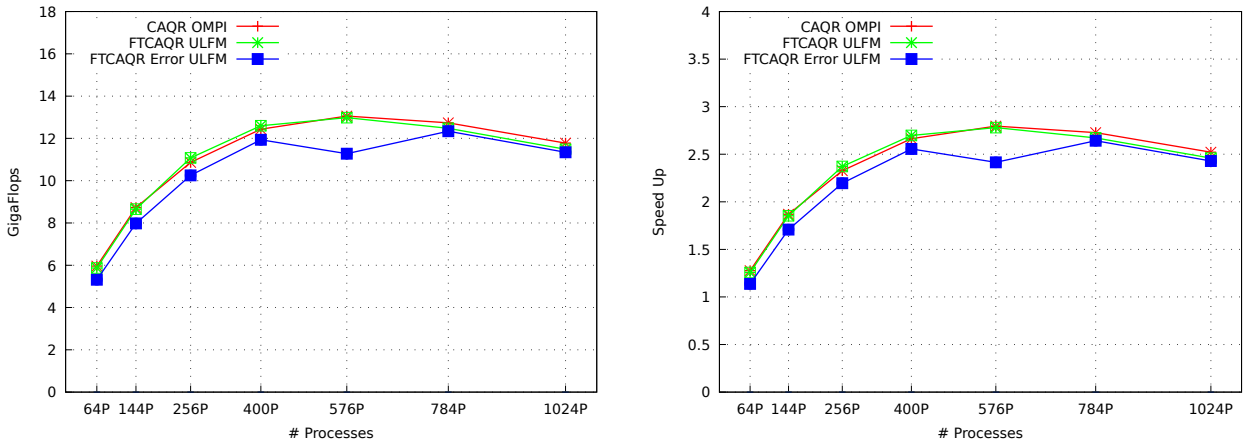


(b) Execution times for CAQR/FT-CAQR

Figure 9.5: Total times obtained in (FT-)TSQR/(FT-)CAQR algorithms (input: $16k \times 16k$).

This behavior can be seen in figure 9.6. Figure 9.6a present the maximal throughput achieved for CAQR/FT-CAQR, and figure 9.6b present the maximal speed up. As described,

both measures start increasing with a small amount of processes, but then a maximal point is reached and performance starts decreasing. But just as it happened with previous LU algoritms, when a failure occurs on FT-CAQR, performance is marginally decreased at the level of CAQR as maximum, which means that practically the same performance is kept, but with fault tolerance included.



(a) Maximal throughput reached in CAQR/FT-CAQR executions

(b) Maximal speed-up reached in CAQR/FT-CAQR executions

Figure 9.6: Maximal throughput and speed-up reached in (FT-)CAQR algorithms (input: $16k \times 16k$).

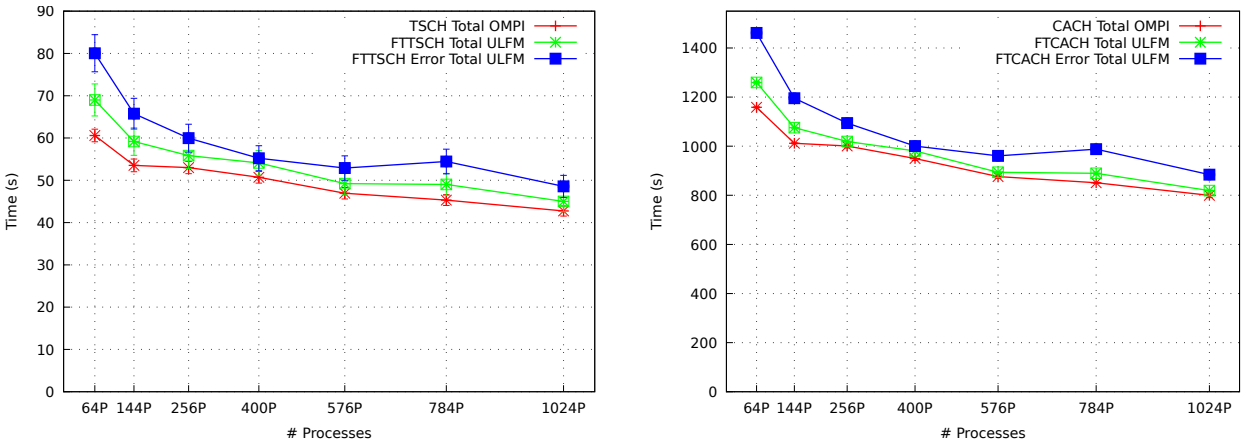
The same testing fault tolerance method was used in FT-CAQR (sending *SIGKILL* signals, with one error). When an error is detected and a new process is spawned, the recovery step requires only basic data exchanges, and mainly local media reading to recover Houlseholder vectors to use in the trailing matrix update, that in fact, is the most expensive operation in CAQR/FT-CAQR. So it can be seen time increases as the number of processes increase, because of this expensive operations.

More tests with FT-CAQR algorithm were not possible because of the amount of space required to store all Householder vectors and sub-blocks the algorithm needs, specially when a big number of processes is used.

9.5 Cholesky Tall and Skinny/Communication-Avoiding Executions

Figure 9.7a displays execution times for TSCH/FT-TSCH and figure 9.7b shows execution times for CACH/FT-CACH. Both algorithms scale satisfactorily as processes increase, but since it requires only one fast step (Cholesky factorization and result distribution), scalability does not form a curve as TSLU or TSQR. Here, non-fault-tolerant and fault-tolerant versions present the same behavior: with a process increase, total execution time gets reduced, but

benefit tends to be not as big as with TSLU/TSQR, since TSCH does not require exchanges between partners and it does not execute a dense local operation per process (only process 0). So, added fault tolerance mechanisms generate a small overhead, as TSLU or TSQR, since if a process fails, it will onlly be re-launched and it will verify what was its corresponding sub-block. The same behavior as FT-CAQR can be appreciated in FT-CACH, since they perform the same backup subroutines on local disk; this is, Cholesky stores lower triangular sub-blocks to let other processes to take and used them in trailing matrix update.



(a) Execution times for TSCH/FT-TSCH

(b) Execution times for CACH/FT-CACH

Figure 9.7: Total times obtained in (FT-)TSCH/(FT-)CACH algorithms (input: $100k \times 100k$).

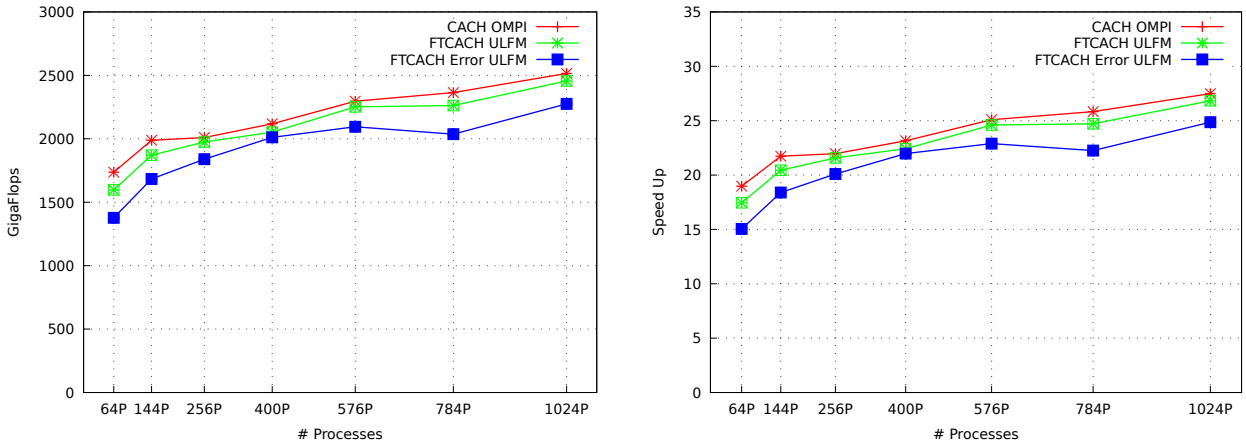
Figures 9.8a and 9.8b present the maximal flops and speed up (respectively) reached for both algorithms. As expected, both measures increase with the amount of processes used and they are bigger than FT-CALU/FT-CAQR. When a failure occurs, performances decrease as with CALU/CAQR. But thanks to their inherent parallelism, TSCH/FT-TSCH and FT-CACH/FT-CACH can generally achieve good scalability on cluster infrastructures.

When an error is detected and a new process is spawned, the recovery step involves minimal data exchanges to restore the state of the failed process.

Figure 9.9 presents a comparison of our FT-CALU executions with all tested matrix sizes. As expected, scalability swiftly increases as the amount of processes increases as well as the matrix size, especially failure-free executions. When a failure occurs, performance looks minimally hit compared against a failure-free FT-CACH execution.

In general, the overall performance increases with the number of processes. As expected, the scalability and the performance per process increase as the size of the matrix increases and the algorithms' good parallelism properties get satisfying performance on the parallel platform, for both fault-tolerant and non-fault-tolerant version of TSCH and CACH.

Appendix B shows more graphs obtained with different sizes of matrices, for each factorization.



(a) Maximal throughput reached in CACH/FT-CACH executions

(b) Maximal speed-up reached in CACH/FT-CACH executions

Figure 9.8: Maximal throughput and speed-up reached in (FT-)CACH algorithms (input: $100k \times 100k$).

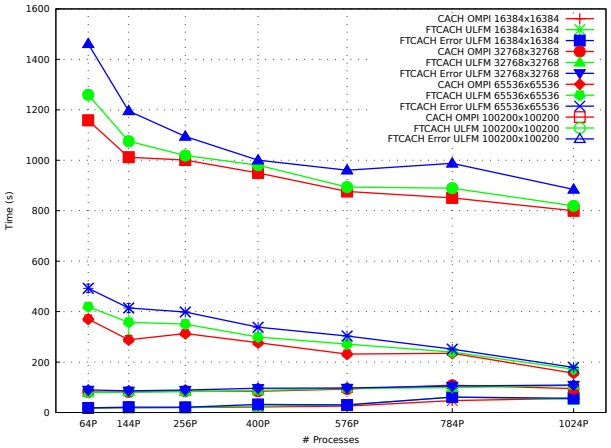


Figure 9.9: Comparision of the scalability in (FT-)CACH algorithm for all inputs.

CHAPTER 10

Conclusions and Future Perspectives

10.1 Summary

In this thesis, I have worked on an approach to tolerate failures at application-level by exploiting properties of the algorithms in three linear algebra operations: the LU, QR and Cholesky matrix factorizations. I have designed, implemented, and presented a collection of algorithms to perform these important matrix factorizations of a dense matrix in parallel in a context of crash-type failures. The algorithms presented here can be used in two contexts: as a computation kernel, guaranteeing the user that if a failure happens during the factorization, it will be dealt with automatically, or implementing these algorithms in fault-tolerant applications that require to run a matrix factorization.

I have exploited structural properties of communication-avoiding factorization algorithms [54] to introduce intrinsic redundancies upon failures and let the algorithms restore the state of the failed processes, minimizing the amount of computation required to resume computation. I have applied these ideas to the factorization of a tall-and-skinny matrix (TSLU, TSQR, TSCH, FT-TSLU, FT-TSQR, FT-TSCH), of a rectangular matrix (CALU, CAQR, FT-CALU, FT-CAQR) and a square matrix (CACH, FT-CACH). The robustness of the tall-and-skinny algorithms has been previously modeled and proved in [72].

Execution times on failure-free executions and executions with one failure were measured. The performance evaluation that was conducted to verify the scalability in the developed algorithms show that the introduced fault-tolerant mechanisms do not represent a large overhead on the final cost. The cost of a failure on the total execution time is very important in a context where failures can happen at any time and must not be significantly harmful for the computation. Therefore, I have seen that the approach based on exploiting intrinsic (partial) redundancies in this communication-avoiding algorithms is promising, both since it has little impact on the failure-free performance and the existence of intermediate results allows new processes to start from a partial point, providing a quick post-failure recovery.

This work has produced new libraries that can be used in the context previously described. These libraries are available for any person who needs them. However, more efforts can be done to generate new fault-tolerant mechanisms in the development of new mathematical libraries highly demanded in the scientific world.

Finally, as being part of the design of concurrent system modeling and verification, a formal model for fault-tolerant tall and skinny algorithms was derived. The core contributions are to prove how failures can be represented and modeled using the abstraction provided by the model. It also helps in the proof design of fault tolerance properties for general parallel algorithms.

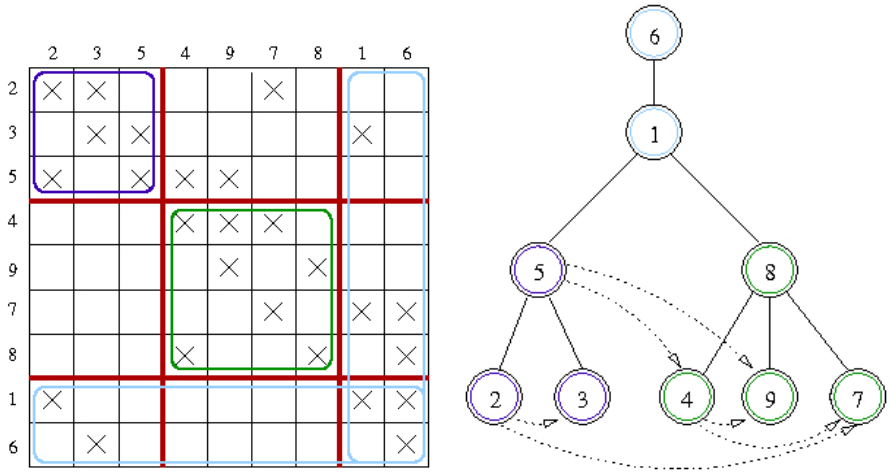
10.2 Future Perspectives

Regarding the issue of formal verification, the models derived and presented in this work can be useful serving as a basis, or being extended to a general model that represents how errors are modeled formally. A future perspective is to derive a model and verification approach for fault-tolerant algorithms, showing that they are capable of mitigating failures that are present at runtime, and also, that they are capable of repairing them on the fly regardless of when they occur.

As experiments over the collection of fault-tolerant algorithms have given hopeful results, this thesis work could be extended to add fault tolerance mechanisms into any algorithm that enjoys of interesting intrinsic properties that allow creating data redundancy and add fault-tolerant mechanisms on large-scale parallel algorithms, like sparse linear algebra.

In sparse linear algebra, for example, a temporary matrix is generated from the nonzeros entries in the first rows/columns of the original matrix. Then, a variant of Gaussian elimination factorizes a temporary matrix with an LU factorization, generating *frontal matrices* and dispersing the columns of the lower triangular matrix and the rows of the upper triangular in a structure called *elimination tree*. In this procedure, there are also generated the *contribution blocks*, which can be seen as a trailing matrix, and later they will be used during the updating process of some rows and columns in the original matrix. During the forward elimination, the elimination tree is processed; at each step, part of the solution LU is computed, and the right-hand side is modified using the partial computed solution.

With a first analysis over this algorithm, I have realized that there are possibilities to add similar fault-tolerant mechanisms as in Tall-and-Skinny and Communication-Avoiding algorithms specified in chapters 4, 5 and 6. For example, some data redundancy can be generated in the building of temporary matrices, distribution of frontal matrices and contribution blocks. If we share information between processes, the elimination tree could be processed in parallel. In figure 10.1 is represented a basic form of a sparse matrix (10.1a) and the elimination tree generated from the initial matrix.



(a) Example of a sparse matrix (b) Example of an elimination tree

Figure 10.1: Example of a sparse matrix with an elimination tree.

The conclusions from this work are encouraging and open perspectives for applying this approach to other computation kernels.

Appendices

APPENDIX A

TSLU/FT-TSLU execution examples

To provide a better understanding of the way the TSLU and FT-TSLU algorithms work, this section provides an example of their execution steps. The matrix used is a tiny one, an 8×2 matrix, to make the easier and quicker the description. All the matrix factorizations are computed with subroutine OpenBLAS *DGETRF*.

A.1 TSLU execution example

A.1.1 TSLU single-process execution

$$A = \begin{bmatrix} 7.0000 & 7.0000 \\ 7.0000 & 2.0000 \\ 9.0000 & 2.0000 \\ 5.0000 & 8.0000 \\ 1.0000 & 5.0000 \\ 3.0000 & 7.0000 \\ 7.0000 & 3.0000 \\ 8.0000 & 6.0000 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.5556 & 1.0000 \\ 0.7778 & 0.7903 \\ 0.7778 & 0.0645 \\ 0.1111 & 0.6935 \\ 0.3333 & 0.9194 \\ 0.7778 & 0.2097 \\ 0.8889 & 0.6129 \end{bmatrix} \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = LU, IPIV = \{3, 4\}$$

A.1.2 TSLU multi-process execution

The matrix A is divided into $t = 4$ sub-blocks and each block is sent to a process. Each process computes its block and shares its result with its current partner.

1. Step 0 execution:

$$P_0 \rightarrow A_0 = \begin{bmatrix} 7.0000 & 7.0000 \\ 7.0000 & 2.0000 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 1.0000 & 1.0000 \end{bmatrix} \begin{bmatrix} 7.0000 & 7.0000 \\ 0.0000 & -5.0000 \end{bmatrix} = L_0 U_0$$

$$IPIV_0 = \{1, 2\}$$

$$P_1 \rightarrow A_1 = \begin{bmatrix} 9.0000 & 2.0000 \\ 5.0000 & 8.0000 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.5556 & 1.0000 \end{bmatrix} \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = L_1 U_1$$

$$IPIV_1 = \{1, 2\}$$

$$P_2 \to A_2 = \begin{bmatrix} 1.0000 & 5.0000 \\ 3.0000 & 7.0000 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.3333 & 1.0000 \end{bmatrix} \begin{bmatrix} 3.0000 & 7.0000 \\ 0.0000 & 2.6667 \end{bmatrix} = L_2 U_2$$

$$IPIV_2 = \{2, 2\}$$

$$P_3 \to A_3 = \begin{bmatrix} 7.0000 & 3.0000 \\ 8.0000 & 6.0000 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.8750 & 1.0000 \end{bmatrix} \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & -2.2500 \end{bmatrix} = L_3 U_3$$

$$IPIV_3 = \{2, 2\}$$

Each process exchanges its results with its partner:

$$P_0 \longleftrightarrow P_1, P_2 \longleftrightarrow P_3$$

2. Step 1 execution:

   - U_i, U_j concatenation: when the exchange has been completed, the processes execute *tournament pivoting* to select the best rows from the U_i, U_j matrices they have and concatenate them.

$$P_i, P_j : U_i, U_j \to U'_{i,j} = \begin{bmatrix} U_i \\ U_j \end{bmatrix}$$

$$P_0, P_1 : U_0 = \begin{bmatrix} 7.0000 & 7.0000 \\ 0.0000 & -5.0000 \end{bmatrix}, U_1 = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} \to U'_{0,1} = \begin{bmatrix} 7.0000 & 7.0000 \\ 9.0000 & 2.0000 \\ 0.0000 & -5.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

$$P_2, P_3 : U_2 = \begin{bmatrix} 3.0000 & 7.0000 \\ 0.0000 & 2.6667 \end{bmatrix}, U_3 = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & -2.2500 \end{bmatrix} \to U'_{2,3} = \begin{bmatrix} 3.0000 & 7.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 2.6667 \\ 0.0000 & -2.2500 \end{bmatrix}$$

   - DGETRF is executed on the $U'_{i,j}$ new matrix:

$$P_i, P_j : U'_{i,j} = L_{i,j} U_{i,j}$$

$$P_0, P_1 : U'_{0,1} = \begin{bmatrix} 7.0000 & 7.0000 \\ 9.0000 & 2.0000 \\ 0.0000 & -5.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.0000 & 1.0000 \\ 0.0000 & -0.7258 \\ 0.7778 & 0.7903 \end{bmatrix} \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = L_{0,1} U_{0,1}$$

$$IPIV_{0,1} = \{2, 4\}$$

$$P_2, P_3 : U'_{2,3} = \begin{bmatrix} 3.0000 & 7.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 2.6667 \\ 0.0000 & -2.2500 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.3750 & 1.0000 \\ 0.0000 & 0.5614 \\ 0.0000 & -0.4737 \end{bmatrix} \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix} = L_{2,3} U_{2,3}$$

$$IPIV_{2,3} = \{2, 2\}$$

At this part, each process has the resulting $U_{i,j}$ of the step.

$$P_0, P_1 : U_{0,1} = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

$$P_2, P_3 : U_{2,3} = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix}$$

Each process exchanges its results with its new partner:

$$P_0 \longleftrightarrow P_2, P_1 \longleftrightarrow P_3$$

3. Step 2 execution:

   - $U_{i,j}, U_{k,l}$ concatenation: when the exchange has been completed, the processes execute *tournament pivoting* to select the best rows from the $U_{i,j}, U_{k,l}$ matrices they have and concatenate them.

$$P_{i,j}, P_{k,l} : U_{i,j}, U_{k,l} \rightarrow U'_{i,j,k,l} = \begin{bmatrix} U_{i,j} \\ U_{k,l} \end{bmatrix}$$

$$P_{0,1}, P_{2,3} : U_{0,1} = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix}, U_{2,3} = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix} \rightarrow U'_{0,1,2,3} = \begin{bmatrix} 9.0000 & 2.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 6.8889 \\ 0.0000 & 4.7500 \end{bmatrix}$$

   - DGETRF is executed on the $U'_{i,j,k,l}$ new matrix:

$$P_{i,j}, P_{k,l} : U'_{i,j,k,l} = L_{i,j,k,l} U_{i,j,k,l}$$

$$P_{0,1}, P_{2,3} : U'_{0,1,2,3} = \begin{bmatrix} 9.0000 & 2.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 6.8889 \\ 0.0000 & 4.7500 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.0000 & 1.0000 \\ 0.8889 & 0.6129 \\ 0.0000 & 0.6895 \end{bmatrix} \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = L_{0,1,2,3} U_{0,1,2,3}$$

$$IPIV_{0,1,2,3} = \{1, 3\}$$

To generate the final $L_{i,j,k,l}$, DTRSM is executed as follows:

$$Ax = U_{i,j,k,l} \rightarrow \begin{bmatrix} 7.0000 & 7.0000 \\ 7.0000 & 2.0000 \\ 9.0000 & 2.0000 \\ 5.0000 & 8.0000 \\ 1.0000 & 5.0000 \\ 3.0000 & 7.0000 \\ 7.0000 & 3.0000 \\ 8.0000 & 6.0000 \end{bmatrix} x = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \\ 0.0000 & 0.0000 \end{bmatrix} \rightarrow \begin{bmatrix} 0.7778 & 0.7903 \\ 0.7778 & 0.0645 \\ 1.0000 & 0.0000 \\ 0.5556 & 1.0000 \\ 0.1111 & 0.6935 \\ 0.3333 & 0.9194 \\ 0.7778 & 0.2097 \\ 0.8889 & 0.6129 \end{bmatrix} = L_{i,j,k,l}$$

At this part, each process has the final $L_{i,j,k,l}$ and $U_{i,j,k,l}$:

$$P_{0,1,2,3} : L_{0,1,2,3} = \begin{bmatrix} 0.7778 & 0.7903 \\ 0.7778 & 0.0645 \\ 1.0000 & 0.0000 \\ 0.5556 & 1.0000 \\ 0.1111 & 0.6935 \\ 0.3333 & 0.9194 \\ 0.7778 & 0.2097 \\ 0.8889 & 0.6129 \end{bmatrix}, U_{0,1,2,3} = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

Compared with the single process L,U matrices, the final $L_{0,1,2,3}$ only need to be sorted (with DLASWP):

$$L = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.5556 & 1.0000 \\ 0.7778 & 0.7903 \\ 0.7778 & 0.0645 \\ 0.1111 & 0.6935 \\ 0.3333 & 0.9194 \\ 0.7778 & 0.2097 \\ 0.8889 & 0.6129 \end{bmatrix}, U = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

A.2 FT-TSLU multi-process execution

For a better understanding, the example given here shows how a failed process is re-spawned in a middle step.

1. Step 0 execution: the same as TSLU, point 1.

2. Step 1 execution:

   - U_i, U_j concatenation: when the exchange has been completed, the processes execute *tournament pivoting* to select the best rows from the U_i, U_j matrices they have and concatenate them. In this case, process P_2 crashes after the exchange with P_3 has finished.

$$P_i, P_j : U_i, U_j \rightarrow U'_{i,j} = \begin{bmatrix} U_i \\ U_j \end{bmatrix}$$

$$P_0, P_1 : U_0 = \begin{bmatrix} 7.0000 & 7.0000 \\ 0.0000 & -5.0000 \end{bmatrix}, U_1 = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} \rightarrow U'_{0,1} = \begin{bmatrix} 7.0000 & 7.0000 \\ 9.0000 & 2.0000 \\ 0.0000 & -5.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

$$P_3 : U_2 = \begin{bmatrix} 3.0000 & 7.0000 \\ 0.0000 & 2.6667 \end{bmatrix}, U_3 = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & -2.2500 \end{bmatrix} \rightarrow U'_{2,3} = \begin{bmatrix} 3.0000 & 7.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 2.6667 \\ 0.0000 & -2.2500 \end{bmatrix}$$

- DGETRF is executed on the $U'_{i,j}$ new matrix:

$$P_i, P_j : U'_{i,j} = L_{i,j}U_{i,j}$$

$$P_0, P_1 : U'_{0,1} = \begin{bmatrix} 7.0000 & 7.0000 \\ 9.0000 & 2.0000 \\ 0.0000 & -5.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.0000 & 1.0000 \\ 0.0000 & -0.7258 \\ 0.7778 & 0.7903 \end{bmatrix} \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix} = L_{0,1}U_{0,1}$$

$$IPIV_{0,1} = \{2, 4\}$$

$$P_3 : U'_{2,3} = \begin{bmatrix} 3.0000 & 7.0000 \\ 8.0000 & 6.0000 \\ 0.0000 & 2.6667 \\ 0.0000 & -2.2500 \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 \\ 0.3750 & 1.0000 \\ 0.0000 & 0.5614 \\ 0.0000 & -0.4737 \end{bmatrix} \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix} = L_{2,3}U_{2,3}$$

$$IPIV_{2,3} = \{2, 2\}$$

At this part, each surviving process has the resulting $U_{i,j}$ of the step.

$$P_0, P_1 : U_{0,1} = \begin{bmatrix} 9.0000 & 2.0000 \\ 0.0000 & 6.8889 \end{bmatrix}$$

$$P_3 : U_{2,3} = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix}$$

Each process exchanges its results with its new partner:

$$P_0 \longleftrightarrow P_2, P_1 \longleftrightarrow P_3$$

P_1 and P_3 exchange correctly their results, but by the time P_0 tries to exchange its results with P_2, all processes realize it has died. The process restoration starts: P_2 is re-spawned and all the processes restore the communicator to include the new restored process. The process in charge to share its current information with the failed one is P_3, because it is the previous partner that had the same data as P_2.

$$P_3 \longrightarrow P_2$$

Here, P_2 has received the resulting $U_{2,3}$ of the step.

$$P_2 : U_{2,3} = \begin{bmatrix} 8.0000 & 6.0000 \\ 0.0000 & 4.7500 \end{bmatrix}$$

When the restoration exchange finishes, P_0 exchanges its results with its newly restored partner P_2 and the calculation continues.

3. Step 2 execution: the same as TSLU, point 3.

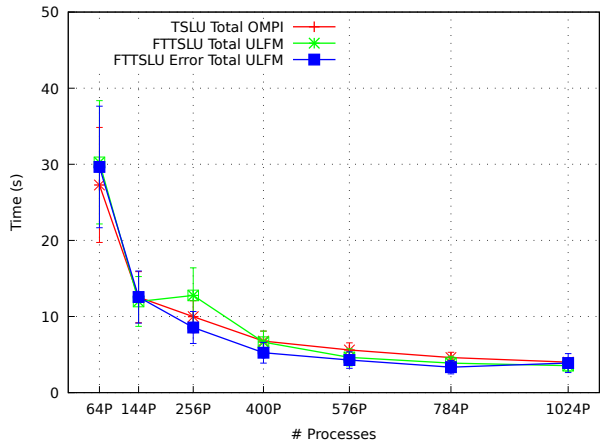
APPENDIX B

More Tall and Skinny/Communication-Avoiding Graphics

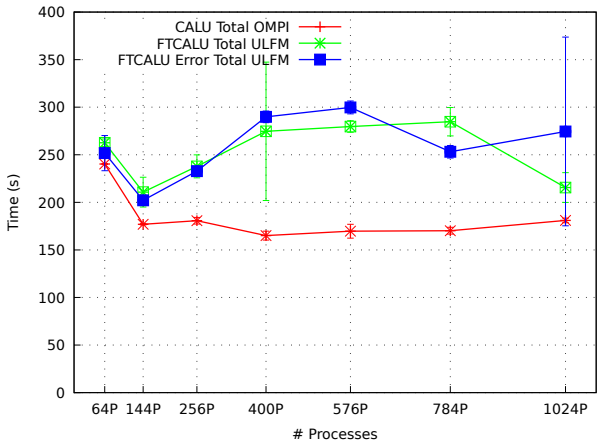
In this section, more graphics obtained from the executions of the LU and Cholesky factorizations algorithms are shown.

B.1 More LU Execution Graphics

In figure B.1 it is shown the total execution times that the Tall and Skinny and Communication-Avoiding algorithms generated, for LU factorization, with an input size of $32k \times 32k$. For the same input and algorithms, figure B.2 is illustrated the maximal reached throughput and speed-up.



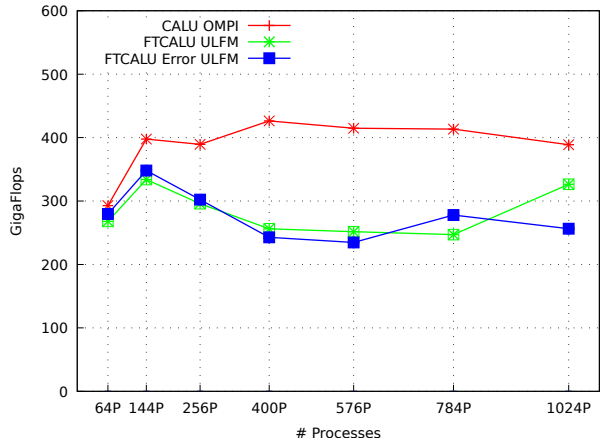
(a) Execution times for TSLU/FT-TSLU



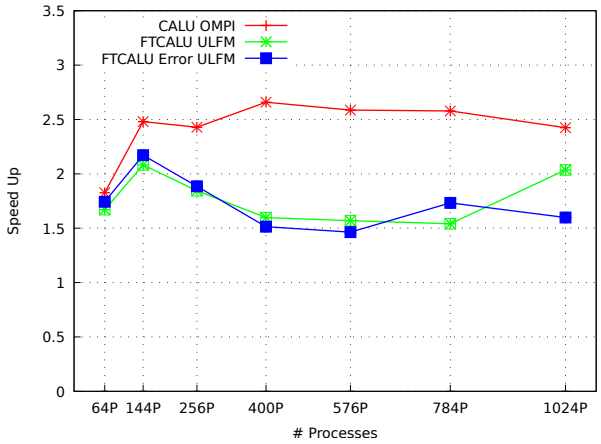
(b) Execution times for CALU/FT-CALU

Figure B.1: Total times obtained in (FT-)TSLU/(FT-)CALU algorithms (input: $32k \times 32k$).

Additionally, figures B.3 and B.4 display the total execution times and the the maximal achieved throughput and speed-up, for the same algorithms, with an input size of $64k \times 64k$.

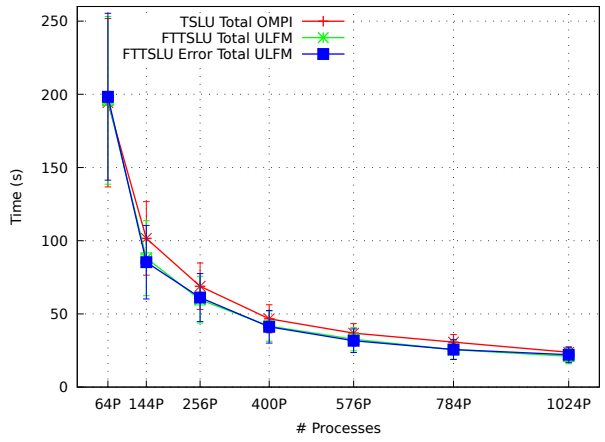


(a) Maximal throughput reached in CALU/FT-CALU executions

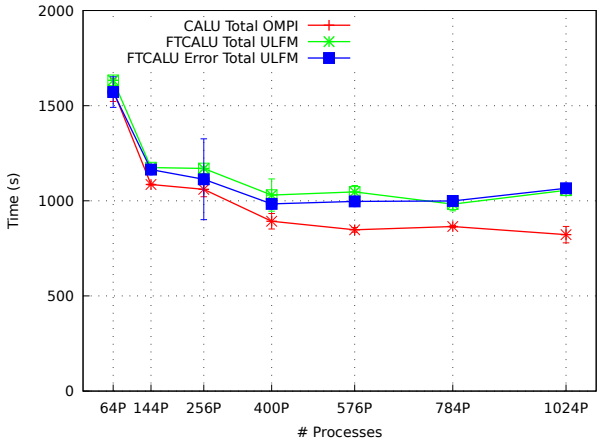


(b) Maximal speed-up reached in CALU/FT-CALU executions

Figure B.2: Maximal throughput and speed-up reached in (FT-)CALU algorithms (input: $32k \times 32k$).



(a) Execution times for TSLU/FT-TSLU



(b) Execution times for CALU/FT-CALU

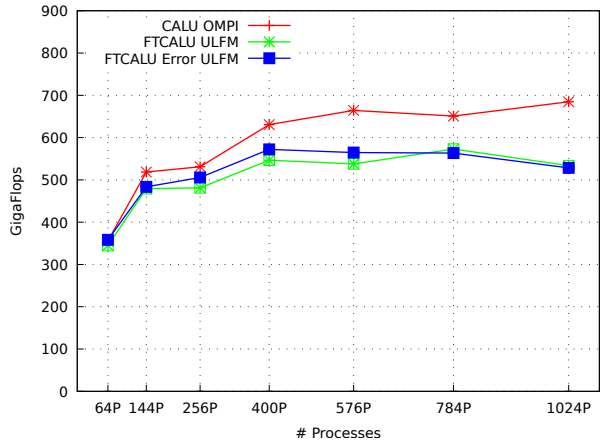
Figure B.3: Total times obtained in (FT-)TSLU/(FT-)CALU algorithms (input: $64k \times 64k$).

B.2 More Cholesky Execution Graphics

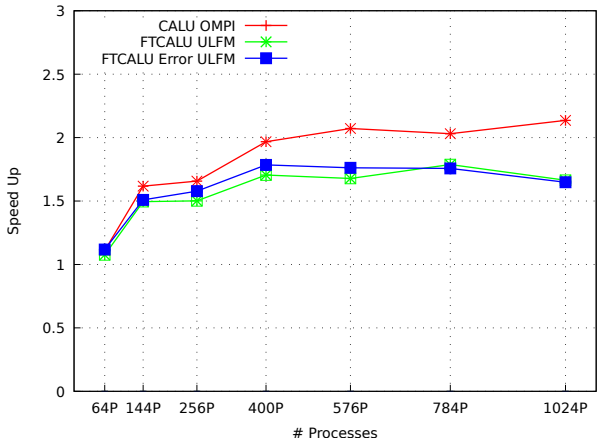
In figure B.5 it is shown the total execution times that the Tall and Skinny and Communication-Avoiding algorithms generated, for Cholesky factorization, with an input size of $16k \times 16k$. For the same input and algorithms, figure B.6 is illustrated the maximal reached throughput and speed-up.

Additionally, figures B.7 and B.8 display the total execution times and the the maximal achieved throughput and speed-up, for the same algorithms, with an input size of $32k \times 32k$.

Finally, figures B.9 and B.10 exhibit the total execution times and the maximal accom-

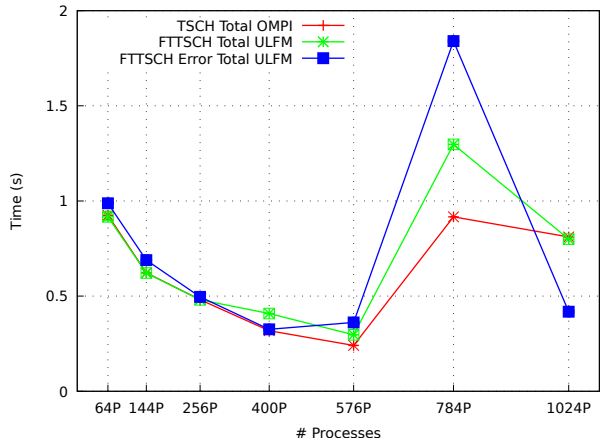


(a) Maximal throughput reached in CALU/FT-CALU executions

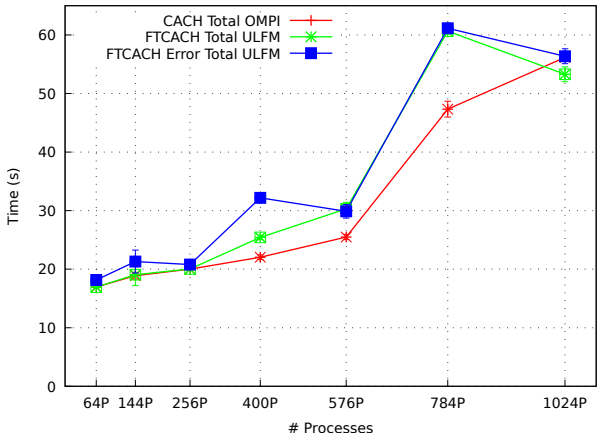


(b) Maximal speed-up reached in CALU/FT-CALU executions

Figure B.4: Maximal throughput and speed-up reached in (FT-)CALU algorithms (input: $64k \times 64k$).



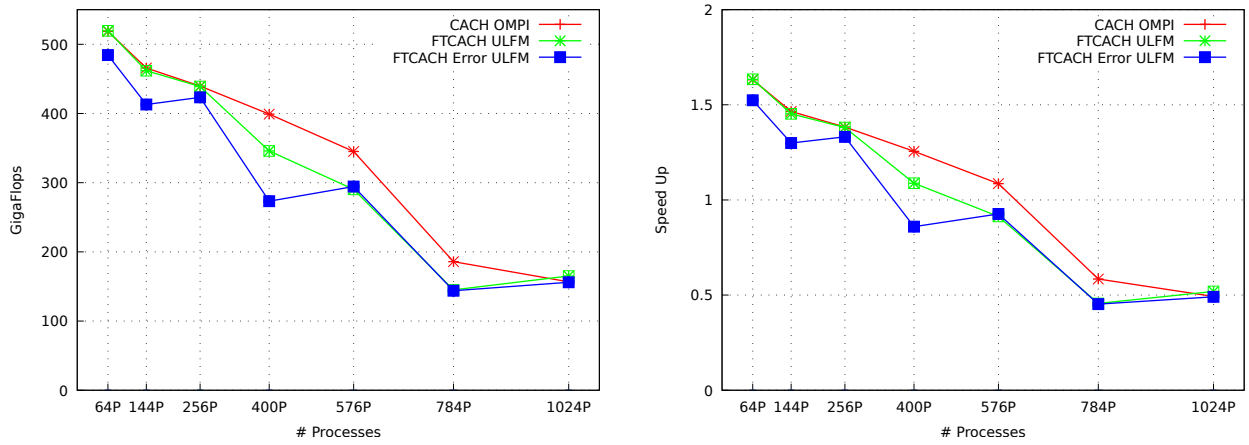
(a) Execution times for TSCH/FT-TSCH



(b) Execution times for CACH/FT-CACH

Figure B.5: Total times obtained in (FT-)TSCH/(FT-)CACH algorithms (input: $16k \times 16k$).

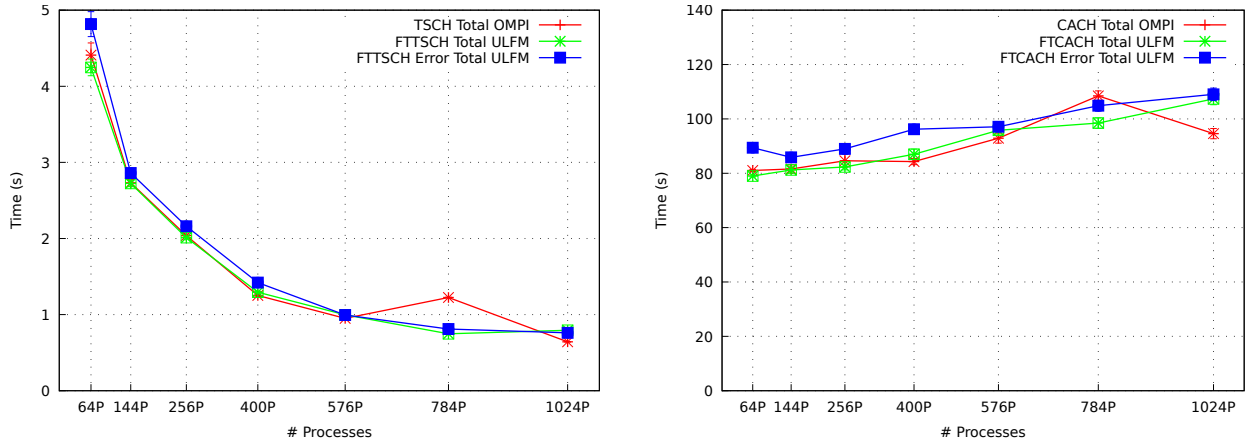
plished throughput and speed-up, for the same algorithms, with an input size of $64k \times 64k$.



(a) Maximal throughput reached in CACH/FT-CACH executions

(b) Maximal speed-up reached in CACH/FT-CACH executions

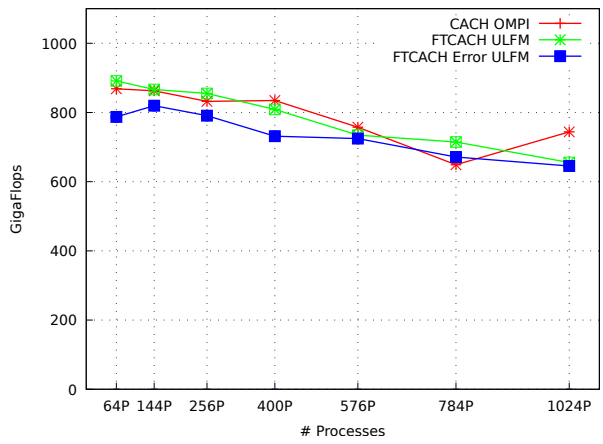
Figure B.6: Maximal throughput and speed-up reached in (FT-)CACH algorithms (input: $16k \times 16k$).



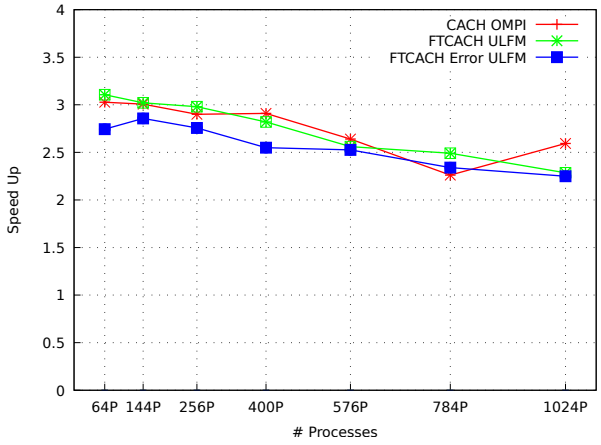
(a) Execution times for TSCH/FT-TSCH

(b) Execution times for CACH/FT-CACH

Figure B.7: Total times obtained in (FT-)TSCH/(FT-)CACH algorithms (input: $32k \times 32k$).

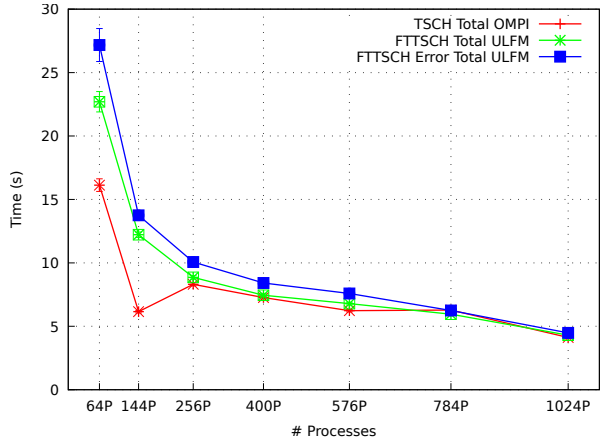


(a) Maximal throughput reached in CACH/FT-CACH executions

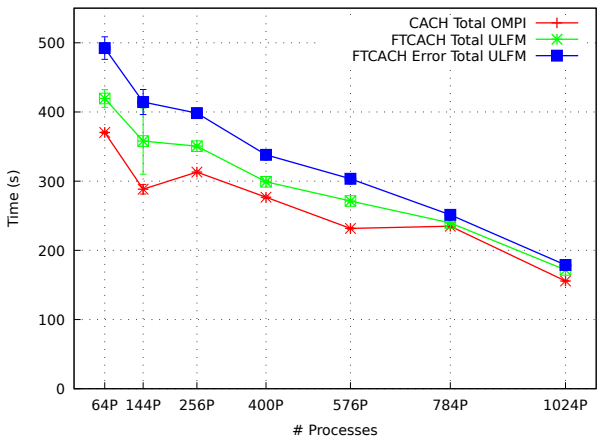


(b) Maximal speed-up reached in CACH/FT-CACH executions

Figure B.8: Maximal throughput and speed-up reached in (FT-)CACH algorithms (input: $32k \times 32k$).

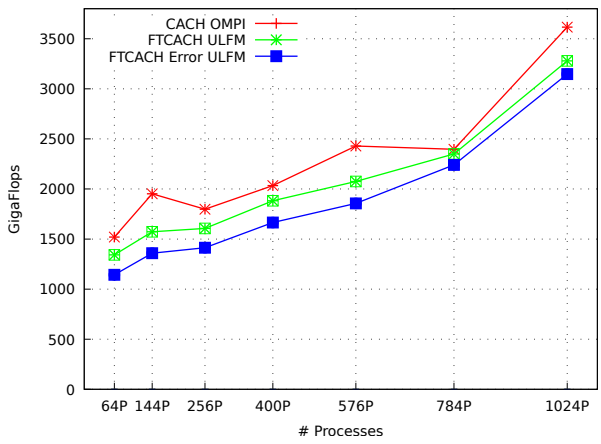


(a) Execution times for TSCH/FT-TSCH

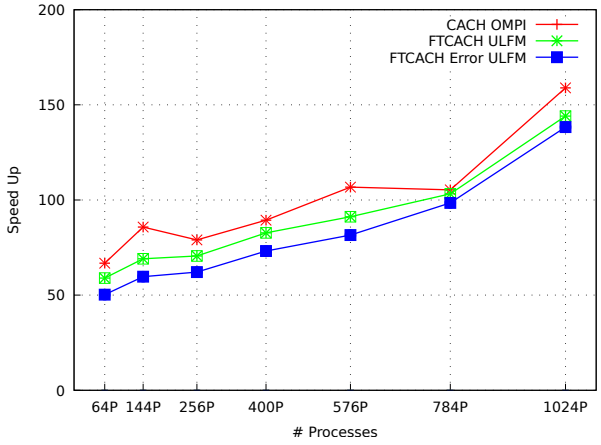


(b) Execution times for CACH/FT-CACH

Figure B.9: Total times obtained in (FT-)TSCH/(FT-)CACH algorithms (input: $64k \times 64k$).



(a) Maximal throughput reached in CACH/FT-CACH executions



(b) Maximal speed-up reached in CACH/FT-CACH executions

Figure B.10: Maximal throughput and speed-up reached in (FT-)CACH algorithms (input: $64k \times 64k$).

Bibliography

[1] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 2011.

[2] William Gropp and Marc Snir. Programming for exascale computers. *Computing in Science & Engineering*, 15:27, 2013.

[3] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[4] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[5] Daniel A. Reed, Charng Da Lu, and Celso L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems*, 22(3):293–302, 2 2006.

[6] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra. Failure detection and propagation in hpc systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 312–322, 2016.

[7] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM J. Sci. Comput.*, 34(1):206–239, February 2012.

[8] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[9] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 610–621, Washington, DC, USA, 2014. IEEE Computer Society.

[10] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and Jesper Larsson Träff. MPI at exascale. In *Procceedings of SciDAC 2010*, Jun. 2010.

[11] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith. Unprotected computing: A large-scale study of dram raw error rate on a supercomputer. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 645–655, 2016.

[12] Shuo Yang, Kai Wu, Yifan Qiao, Dong Li, and Jidong Zhai. Algorithm-directed crash consistence in non-volatile memory for hpc. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 475–486, 2017.

[13] S. Perarnau and L. Bautista-Gomez. Monitoring strategies for scalable dynamic checkpointing. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2016.

[14] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, and F. Cappello. Spatial support vector regression to detect silent errors in the exascale era. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 413–424, 2016.

[15] Marcos Maroñas, Sergi Mateo, Kai Keller, Leonardo Bautista-Gomez, Eduard Ayguadé, and Vicenç Beltran. Extending the openchk model with advanced checkpoint features. *Future Generation Computer Systems*, 112:738 – 750, 2020.

[16] C. Ruiz, J. Emeras, E. Jeanvoine, and L. Nussbaum. Distem: Evaluation of fault tolerance and load balancing strategies in real hpc runtimes through emulation. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 267–272, 2016.

[17] 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 *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 18–18, Nov 2006.

[18] Ifeanyi P. Egwutuoha, Shiping Chen, David Levy, and Bran Selic. A fault tolerance framework for high performance computing in cloud. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 709–710, 2012.

[19] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[20] Sangho Yi, Derrick Kondo, Bongjae Kim, Geunyoung Park, and Yookun Cho. Using replication and checkpointing for reliable task management in computational grids. In *2010 International Conference on High Performance Computing & Simulation*, pages 125–131. IEEE, 2010.

[21] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *[1992] Proceedings 11th Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[22] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22:2196–2211, 11 2010.

[23] Lemarinier Pierre, Bouteiller Aurelien, Herault Thomas, Krawezik Geraud, and Cappello Franck. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant {MPI}. In *19th International Parallel and Distributed Processing Symposium*, Denver, USA, United States, April 2005.

[24] J.-M Helary, Achour Mostéfaoui, and Michel Raynal. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. pages 68–77, 07 1997.

[25] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[26] Ifeanyi Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65, 09 2013.

[27] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 38–38, 2004.

[28] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, June 2003.

[29] Y. Zhu, Y. Liu, and G. Zhang. Ft-pblas: Pblas-based fault-tolerant linear algebra computation on high-performance computing systems. *IEEE Access*, 8:42674–42688, 2020.

[30] M. Artioli, D. Loreti, and A. Ciampolini. Fault tolerant high performance solver for linear equation systems. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 113–11309, 2019.

[31] Graham E Fagg and Jack J Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 346–353. Springer, 2000.

[32] William Hoarau, Pierre Lemarinier, Thomas Hérault, Eric Rodriguez, Sébastien Tixeuil, and Franck Cappello. Fail-mpi: How fault-tolerant is fault-tolerant mpi? *2006 IEEE International Conference on Cluster Computing*, pages 1–10, 2006.

[33] 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*, pages 329–332, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[34] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. 27(3):244–254, August 2013.

[35] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 193–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[36] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906, 2014.

[37] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar. Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 346–355, 2016.

[38] Marc Gamell, Keita Teranishi, Michael A. Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Exploring failure recovery for stencil-based applications at extreme scales. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 279–282, New York, NY, USA, 2015. Association for Computing Machinery.

[39] M. Gamell, K. Teranishi, J. Mayo, H. Kolla, M. A. Heroux, J. Chen, and M. Parashar. Modeling and simulating multiple failure masking enabled by local recovery for stencil-based applications at extreme scales. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2881–2895, 2017.

[40] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[41] Erlin Yao, Mingyu Chen, Rui Wang, Wenli Zhang, and Guangming Tan. A new and efficient algorithm-based fault tolerance scheme for a million way parallelism. *CoRR*, abs/1106.4213, 2011.

[42] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies — Ada-Europe '96*, pages 38–57, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[43] J.S. Plank, Kai Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[44] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[45] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Algorithm-based fault tolerance for convolutional neural networks, 2020.

[46] C. Coti. Exploiting redundant computation in communication-avoiding algorithms for algorithm-based fault tolerance. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, pages 214–219, April 2016.

[47] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(06):518–528, jun 1984.

[48] E. Agullo, C. Coti, J. Dongarra, T. Hérault, and J. Langem. Qr factorization of tall and skinny matrices in a grid computing environment. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–11, April 2010.

[49] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[50] Eran Rabani and Sivan Toledo. Out-of-core svd and qr decompositions. In *PPSC*, 2001.

[51] Laura Grigori. *Introduction to Communication Avoiding Algorithms for Direct Methods of Factorization in Linear Algebra*, pages 153–185. Springer International Publishing, Cham, 2017.

[52] Edward Hutter and Edgar Solomonik. Communication-avoiding cholesky-qr2 for rectangular matrices. 10 2017.

[53] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond

Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[54] Laura Grigori, James W. Demmel, and Hua Xiang. Calu: A communication optimal lu factorization algorithm. *SIAM J. Matrix Anal. Appl.*, 32(4):1317–1350, November 2011.

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

[56] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. On algorithmic variants of parallel gaussian elimination: Comparison of implementations in terms of performance and numerical properties. 01 2013.

[57] C. Coti. Scalable, robust, fault-tolerant parallel qr factorization. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pages 626–633, Aug 2016.

[58] Jack J. Dongarra, Sven Hammarling, and David W. Walker. Key concepts for parallel out-of-core lu factorization. *Parallel Comput.*, 23:49–70, 1997.

[59] James Demmel, Nicholas Higham, and Robert Schreiber. Block lu factorization. 03 2000.

[60] Erik Elmroth and Fred Gustavson. Applying recursion to serial and parallel qr factorization. *IBM Journal of Research and Development*, 44:605 – 624, 08 2000.

[61] Erik Elmroth. New serial and parallel recursive qr factorization algorithms for smp systems. 11 1998.

[62] Camille Coti. Fault tolerant QR factorization for general matrices. *CoRR*, abs/1604.02504, 2016.

[63] Joseph Dorris, Jakub Kurzak, Piotr Luszczek, Asim Yarkhan, and Jack Dongarra. Task-based cholesky decomposition on knights corner using openmp. volume 9945, pages 544–562, 06 2016.

[64] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, and Zizhong Chen. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 68:1–68:12, Piscataway, NJ, USA, 2018. IEEE Press.

[65] Tingxing Dong, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. A fast batched cholesky factorization on a gpu. In *2014 43rd International Conference on Parallel Processing*, pages 432–440, 2014.

[66] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *Computing Research Repository - CORR*, 32, 02 2009.

[67] Robert Schreiber and Charles VanLoan. A storage-efficient wy representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10, 02 1989.

[68] Christian H. Bischof and Charles Van Loan. The wy representation for products of householder matrices. In *PPSC*, 1985.

[69] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and Edgar Solomonik. Reconstructing householder vectors from tall-skinny qr. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1159–1170, 2014.

[70] Julien Langou. Computing the r of the qr factorization of tall and skinny matrices using mpi_reduce. 02 2010.

[71] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[72] Camille Coti, Laure Petrucci, and Daniel Alberto Torres Gonzalez. Fault-tolerant matrix factorisation: a formal model and proof. 6th International Workshop on Synthesis of Complex Parameters (SynCoP) 2019, 2019.