Adaptive and generic parallel exact linear algebra

Ziad Sultan

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques Appliquées**

Arrêté ministériel : 7 août 2006

Présentée par

Ziad SULTAN

Thèse dirigée par **Jean-Guillaume DUMAS**
et codirigée par **Clément PERNET**

préparée au sein du **Laboratoire Jean Kuntzmann**
et de l'**Ecole Doctorale ED-MSTII**

Algèbre linéaire exacte parallèle générique et adaptative

Thèse soutenue publiquement le **17 juin 2016**,
devant le jury composé de :

Mme Laura GRIGORI
Directeur de recherche, Inria-Paris Rocquencourt, Rapporteur
M. Arne STORJOHANN
Professeur, Université de Waterloo, Rapporteur
M. Denis TRYSTRAM
Professeur, Grenoble INP, Président
M. Pascal GIORGI
Maître de Conférence, Université de Montpellier, Examinateur
M. Jean-Guillaume DUMAS
Professeur, Université Grenoble Alpes, Directeur de thèse
M. Clément PERNET
Maître de conférence, Université Grenoble Alpes, Co-Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier et à témoigner toute ma reconnaissance à mes directeurs de thèse, Jean-Guillaume DUMAS et Clément PERNET pour leur soutien, convivialité et pour m'avoir accompagné jusqu'au bout de ce travail. Leur compétence et leur rigueur scientifique m'ont permis de mener à bien ce projet.

Je tiens à remercier aussi, tous les membres du Jury: M. Denis TRYSTRAM, M. Pascal GIORGI, et aussi Mme Laura GRIGORI et M. Arne STORJOHANN d'avoir passé du temps à lire et rapporter mon manuscrit.

Je remercie aussi tous les membres de l'équipe CASYS et de l'équipe MOAIS, en particulier M. Jean Louis ROCH et M. Thierry GAUTIER pour leur soutien, leur aide et les collaborations que nous avons eu ensemble pendant cette thèse. Merci également à toutes les personnes formidables travaillant au laboratoire LJK et Inria pour leur soutien moral et les bons moments passés avec certains.

J'ai eu la chance de partager mon bureau à l'Inria et au LJK avec plusieurs collègues que j'aimerais remercier, notamment Jean-Baptiste, Burak, Joseph, Millian, Rodrigo qui ont fait, par leur bonne humeur et leur sympathie, que cette thèse reste pour moi une excellente expérience.

Je ne saurai terminer sans adresser mes remerciements à tout le groupe d'amis proches pour leur motivation et leur encouragements, en particulier Amro et Fabio qui ont toujours été présents à mes côtés.

Ma reconnaissance va à ceux qui ont plus particulièrement assuré le soutien affectif sans faille de ce travail doctoral, ma famille:

- *Mes parents - Malgré mon éloignement depuis de nombreuses années, leur confiance, leur soutien et leur amour me portent et me guident tous les jours. Merci pour avoir fait de moi ce que je suis aujourd'hui.*

- *Mes frères qui sont mes meilleurs amis à la fois, Ahmad et Tarek, à qui j'adresse ma gratitude pour leurs encouragements, leur support et leur confiance et qui ont su de par leur expérience scientifique me donner les bons conseils et éthiques pour mener à bien cette thèse.*

- *Ma merveilleuse et splendide fiancée Nissrine pour son soutien qui a été sans faille. Toujours à mes côtés, elle a su donner une nouvelle saveur à cette réalisation.*

Enfin, je remercie l'agence nationale de la recherche pour avoir financé cette thèse dans le cadre du projet HPAC (ANR 11 BS02 013).

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The elementary components of computer algebra consist in numbers and polynomials, and the basic domains are the integers, rational numbers, finite fields and polynomial rings. Motivated by a large range of applications, ranging from symbolic manipulation, algebraic cryptanalysis, computational number theory, linear programming, and formal proof verification, computation in this field is required to be exact. In general, it boils down to computations over prime fields $\mathbb{Z}/p\mathbb{Z}$ through the use of RNS or p-adic lifting techniques. Therefore, sequential exact linear algebra over prime fields of machine word size has been developed [36, 22].

The classic computation of the product of two $n \times n$ dense matrices over a field is done with $O(n^3)$ operations. In 1969, Strassen [83] showed that it is possible to perform the same computation with only $O(n^{2.807})$. This sub-cubic complexity opened a new area of research with two main axes. The first one focuses on finding the best exponent ω such that two $n \times n$ matrices can be multiplied in $O(n^\omega)$. Currently the best known value for ω is approximately 2.3728639 [71]. The second axis aims at reducing all dense linear algebra problems to the matrix multiplication operation to reduce their complexity. This is done by considering block algorithms that gather arithmetic operations in matrix multiplication.

In this process, dense linear algebra has become an optimized building block for problems that are dense by nature but also for large sparse problems that are reduced to smaller dense problems that are still large. Indeed, sparse elimination switches to dense elimination on blocks after fill-in, in the case of sparse direct methods [30], or induces dense elimination on blocks of iterated vectors when using sparse iterative methods [66].

Computing efficiently in dense exact linear algebra does not only rely on optimized time and space complexities. It also depends on algorithms that can be made relevant in practice. It is thus important to design algorithms that are sensitive to nowadays machine architectures. The latter performance have shown an extremely rapid evolution in the past decades by doubling every 18 months. Nonetheless, the pace of advancement has slowed drastically a decade ago due to physical limitations such as energy consumption and heat dissipation. In order to carry on in terms of performance growth, the solution has been to increase the number of cores per processor. Hence, in

order to take advantage of the new architecture of parallel processing units, sequential algorithms need to be adapted and parallelism needs to be introduced. The key challenge is to deliver powerful implementations of algorithms in computer algebra within this high-performance context.

This manuscript summarizes our contributions to the design of high performance computer algebra and shows the thorough encroachment between two areas: exact linear algebra and numerical linear algebra. A large body of research efforts addressing these two areas has resulted in efficient sequential implementations and mature software in numerical [59, 50, 88, 3, 1] and exact linear algebra [22, 35, 14, 43, 12, 79]. Moreover, parallel implementations in numerical computation have been intensively studied and reported in the literature [27, 7, 77, 70, 87, 53]. However, in dense exact linear algebra, very few parallel implementations exist and the design aspects in this field are still to be investigated.

1.1 Issues in the design of parallel dense exact linear algebra

Triangular matrix decomposition is a fundamental building block in computational linear algebra. It is used to solve linear systems, compute the rank, the determinant, the null-space or the echelon form of a matrix.

The LU decomposition, defined for matrices whose leading principal minors are all nonzero, can be generalized to arbitrary ranks and rank profiles by introducing pivoting on sides, leading e.g. to the LQUP decomposition of [58] or the PLUQ decomposition [49, 63]. Many algorithmic variants exist allowing fraction free computations [63], in-place computations [36, 62] or sub-cubic rank-sensitive time complexity [82, 62]. More precisely, the pivoting strategy is the key difference between these PLUQ decompositions.

In numerical linear algebra [49], pivoting is used to ensure a good numerical stability, good data locality, and reduce the fill-in. In the context of exact linear algebra, the computation of the echelon form is crucial in many applications using exact Gaussian elimination, such as Gröbner basis computations [44] and computational number theory [81]. Indeed, only certain pivoting strategies for these decompositions will reveal the echelon form or the rank profile of the matrix [62, 39].

Computation over a finite field shapes the heart of exact linear algebra. Yet, it implies performing additional arithmetic operations through modular reductions. The latter operations can be costly depending on the algorithm used and should be taken into account in a delayed design.

In exact computation, one can benefit from asymptotically faster complexities by using Strassen [83] and Strassen-Winograd [29] variants for matrix multiplication, especially when applied on sufficiently large blocks. This implies using recursive algorithms that on one hand insures coarser grain cutting than iterative algorithms and on the other hand implies multiple level of parallelism. Moreover, over a finite field, the elimination algorithms can discover rank deficiencies during the computation and implies having heterogeneous tasks. Thus, these problems should be addressed by using work-stealing based schedulers that handle unbalanced workloads and parallel runtime

systems that handle efficiently the management of recursive tasks.

Nevertheless, the design of a parallel software in numerical linear algebra differs from that in exact linear algebra. These differences are detailed in 1.2.1.

1.2 Exact and Numerical linear algebra

Efficient sequential exact linear algebra routines benefit from the experience in numerical linear algebra. In particular, a key point there is to embed the finite field elements in integers stored as floating point numbers [36], and then rely on the efficiency of the floating point matrix multiplication `dgemm` of the BLAS. Hence a natural ingredient in the design of efficient dense linear algebra routines is the use of block algorithms that result in gathering arithmetic operations in matrix-matrix multiplications. Those can take full advantage of vector instructions and have a high computation per memory access rate, allowing to fully overlap the data accesses by computations and hence deliver close to peak performance efficiency.

1.2.1 Main differences between Exact and Numerical linear algebra

Computation in exact and in numerical linear algebra shares similarities in algorithmic aspects as well as in implementation aspects. The use of block algorithms is important to ensure better data locality. Thus iterative and recursive block algorithms are investigated. A key common point is to reduce as much as possible the data movements that leads to distant memory accesses in parallel especially when computation are performed on Non-Uniform Memory Access machines. The latter machine architecture is explained in section 1.3.3.

However, we illustrate here how numerical and exact LU factorization mainly differ in the following aspects:

- the pivoting strategies,
- the cost of the arithmetic (of scalars and matrices),
- the treatment of rank deficiencies.

Those have a direct impact on the shape and granularity of the block decomposition of the matrix used in the computation.

1.2.1.1 The cost of the arithmetic

In numerical linear algebra, the cost of arithmetic operations is more or less associative: with dimensions above a rather low threshold (typically a few hundreds), the BLAS sequential matrix multiplication attains about 80% of the peak efficiency of the processor. Hence the granularity has very little impact on the efficiency of a block algorithm run sequentially. On the contrary, over a finite field, a small granularity can imply a larger number of costly modular reductions, as we will show in Section 4.5. Moreover, numerical stability is not an issue over a finite field, and asymptotically fast matrix multiplication algorithms, like Winograd's variant of Strassen algorithm [45, §12] can be used on top of the BLAS. The cost of sequential matrix multiplication over finite field is therefore not associative: a larger granularity delivers a better sequential efficiency [17].

1.2.1.2 Pivoting strategies and rank deficiencies

In dense numerical linear algebra, a pivoting strategy is a compromise between the two competing constraints: ensuring good numerical stability and avoiding data movement. In the context of dense exact linear algebra, stability is no longer an issue. Instead, only certain pivoting strategies will reveal the echelon form or, equivalently, the rank profile of the matrix. These strategies are detailed in Chapter 5.

In the case of numerical LU factorization, most often all panel blocks have full rank. Therefore the splitting can be done statically according to a granularity parameter. Over exact domains, on the contrary, depending on the application, the large blocks can be rank deficient. Thus, the tiles or slabs have unpredictable dimensions and the block splitting necessarily dynamic, as will be illustrated in Chapter 5.

1.2.2 Consequences on the design of exact matrix multiplication

Consequently the design of an exact matrix factorization necessarily differs from the numerical algorithms as follows:

- granularity should be as large as possible, to reduce modular reductions and benefit from fast matrix multiplication;
- exact algorithms should preferably be recursive, to group arithmetic operations in matrix products as large as possible;
- block splitting and pivoting strategies must preserve and reveal the rank profile of the matrix.
- block dimensions depend on rank deficiencies which are unknown in advance. This leads to dynamic computation of block size when input matrices are rank deficient.
- modular reduction can be costly and should be delayed as much as possible.

1.3 Parallel linear algebra libraries

We list in this section the state of the art numerical and exact linear algebra parallel libraries. This is not an exhaustive list of all existing high performance libraries. We present the most commonly used libraries in numerical linear algebra to which we will compare.

1.3.1 State of the art numerical linear algebra libraries for shared memory architectures

We focus first on high performance numerical linear algebra libraries that are designed for shared memory architectures:

1.3.1.1 Intel MKL

Intel Math Kernel Library (Intel MKL) [60, 86] is a computing math library of highly optimized, threaded routines. The library provides Fortran and C programming language interfaces. It includes highly vectorized and threaded Linear Algebra, Fast

Fourier Transforms (FFT), Vector Math and Statistics functions. It is the reference library in numerical computation and it is parallelized using OpenMP standard and Intel Threading Building Blocks (TBB).

1.3.1.2 OpenBLAS

OpenBLAS is an open source implementation of the BLAS (Basic Linear Algebra Subprograms) API. It is a continuation of GotoBLAS2 [50] that adds optimized implementations of linear algebra kernels for several processor architectures achieving performance comparable to the Intel MKL on Intel Sandy Bridge [87] and Loongson [91]. OpenBLAS can be compiled sequentially, or using a multithreaded version. It provides parallelism using pthreads or using OpenMP standard.

1.3.1.3 PLASMA-Quark

The Parallel Linear Algebra Software for Multicore Architectures [77, 70, 69, 26] is a dense linear algebra package at the forefront of multicore computing. PLASMA currently offers a collection of routines for solving linear systems of equations, least square problems, eigenvalue problems, and singular value problems. It uses the QUARK (QUeuing And Runtime for Kernels) [1] library as a parallel runtime. QUARK provides a library that enables the dynamic execution of tasks with data dependencies in a multi-core and multi-socket shared-memory environment. QUARK infers data dependencies and precedence constraints between tasks from the way that the data is used, and then executes the tasks in an asynchronous, dynamic fashion in order to achieve a high utilization of the available resources. PLASMA-QUARK also supports different data mapping and block storage strategies that helps to deliver the highest possible performance.

1.3.1.4 High performance numerical linear algebra libraries for distributed memory architectures

All the following libraries are designed for distributed memory architectures and will not be used to compare with our shared-memory oriented implementations. We thus don't give an exhaustive description of these libraries, we list the most important of them:

ScaLAPACK

The ScaLAPACK (or Scalable LAPACK) library[2] is a high-performance linear algebra library written in Fortran and includes a subset of LAPACK [3, 2] routines. The latter is a standard software library for sequential numerical linear algebra. The ScaLAPACK library deals with dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems. However, it is only designed for distributed memory parallel computers.

[1] http://icl.cs.utk.edu/quark
[2] http://www.netlib.org/scalapack

Elemental

Elemental [78] is an open-source library for distributed-memory dense and sparse-direct linear algebra . It's build on top of BLAS, LAPACK, and MPI using modern C++ and additionally exposes interfaces to C and Python. It supports a wide collection of distributed-memory operations.

MTL

The Matrix Template Library [80] is a high performance library that includes a large number of data formats and algorithms, including most popular sparse and dense matrix formats and functionality. Benchmarks with MTL4 showed good performance on the parallel Supercomputing Edition [3] where it was tested on an HPC cluster.

1.3.2 State of the art exact linear algebra libraries

1.3.2.1 NTL

Victor Shoup's NTL [79] is a Library for Number Theory that provides implementations of state-of-the-art algorithms for:

- basic linear algebra over the integers, finite fields, and arbitrary precision floating point numbers.
- arbitrary length integer arithmetic and arbitrary precision floating point arithmetic; NTL can be used in conjunction with GMP (the GNU Multi-Precision library) for enhanced performance;

Recently, it its latest releases, NTL focused on parallelization and multithreading. As of version 9.5, NTL has thread boost feature. Yet, for now, this feature is only supported for polynomial factorization.

1.3.2.2 MAGMA

Magma [12] is a large, well-supported software designed for computations in algebra, number theory, algebraic geometry and algebraic combinatorics. It has developed its own mathematically rigorous environment for defining and working with structures such as groups, rings, fields, modules, algebras, schemes, curves, graphs, designs, codes and many others. Magma often sets the reference in efficiency. Moreover, LinBox library, detailed in 1.3.2.3 and where implementation contributions of this thesis are developed, has been constructed recently with close comparisons to Magma on sequential matrix multiplication [31] and on the computation of the characteristic polynomial [41, 65].

1.3.2.3 LinBox

Project LinBox [15] is a collaborative effort among researchers at a number of locations. The goals are to produce algorithms and software for symbolic linear algebra, particularly using black box matrix methods, i.e. iterative methods requiring only the

[3] http://www.simunova.com/en/node/186

linear transform property of the matrix (that one can compute Ax $\rightarrow$ y). The LinBox project relies on several libraries, such as Givaro for finite fields and also `FFLAS-FFPACK` for dense linear algebra over a finite field.

1.3.2.4 Givaro

Givaro [32] is a C++ library for arithmetic and algebraic computations. Its main features are implementations of the basic arithmetic of many mathematical entities: primes fields, extensions fields, finite fields, finite rings, polynomials, algebraic numbers, arbitrary precision integers and rationals (C++ wrappers over gmp) It also provides data-structures and templated classes for the manipulation of basic algebraic objects, such as vectors, matrices (dense, sparse, structured), univariate polynomials (and therefore recursive multivariate).

1.3.2.5 The `FFLAS-FFPACK` library

The `FFLAS-FFPACK` [14, 36] is a LGPL-2.1+ source code library for basic linear algebra operations over a finite field. It is inspired by BLAS interface (Basic Linear Algebra Subprograms) and the LAPACK library for numerical linear algebra, and shares part of their design. Yet it differs in many aspects due to the specifities of computing over a finite field. It is generic with respect to the finite field, so as to accommodate a large variety of field sizes and implementations.

1.3.3 Parallelization of the `FFLAS-FFPACK` library

We are interested in the parallelization of the `FFLAS-FFPACK` library in this thesis. However, parallelism also introduces additional concerns. A sequential module encapsulates the code that implements the functions provided by the module's interface and the data structures accessed by those functions. In parallel programming, we need to consider not only code and data but also the tasks created by a module, the way in which data structures are partitioned and mapped to processors, and internal communication structures. The latter depends greatly on the machine architecture.

Shared machine architectures

The computer architecture, microprocessor or supercomputers, is strongly influenced by the harness of a fundamental property of applications: parallelism. Today, most servers are distributed memory machines (clusters) or shared memory machines (multiprocessors). Multiprocessor shared memory architectures can be:

- Uniform Memory Access (UMA): all the processors share the physical memory uniformly. UMA architectures mainly use bus-based Symmetric MultiProcessing (SMP) architectures. SMP is the processing of programs by multiple processors that share a common operating system and memory. In symmetric (or "tightly coupled") multiprocessing, the processors share memory and the I/O bus.
- Cache-only memory architecture (COMA): the local memories for the processors at a node is used as cache. The hardware transparently replicates the data and migrates it to the memory module of the node that is currently accessing it. This

increases the chances of data being available locally. One can refer to [23] for more details.

- Non-Uniform Memory Access (NUMA): memory access time depends on the memory location relative to a processor. A NUMA system is a multiprocessor system in which memory areas are separated and placed in different locations (and different buses). Depending on each processor access times therefore differ according to the accessed memory area.

In the SMP architecture all the memory space is accessible through a single bus. This causes a performance bottleneck when different processors try to access concurrently the bus. The NUMA is a more suitable architecture for systems with many processors and is designed to overcome the limitations of the SMP architecture. In a NUMA architecture, processors may access local memory quickly and remote memory more slowly. This can dramatically improve memory throughput as long as the data are localized to specific processes (and thus processors). NUMA represents a middle position between the SMP and clustering (various machines).

A dedicated interface to parallel exact linear algebra

As will be explained in Chapter 2, an efficient parallelization of exact linear algebra libraries can attain good performance using a high level runtime system. Today, many high level runtime systems exist with various parallel programming models and paradigms. We want to offer the option of using different runtime systems and be able to compare and achieve high speed-up. To do so, the user needs to be able to plug a parallel runtime system into the library. This helps not only to benchmark parallel performance using different runtime systems but also to be able to deal with specific problems by using the most adapted runtime system to the type of computation performed. The user thus needs an interface that fills the following requirements:

- genericity and portability
- performance and scalability
- high enough level of abstraction
- take into account large range of machine architectures

But also:

- to support runtime system with good performance for recursive tasks
- to handle efficiently unbalanced workloads
- to use optimized and efficient range loop cutting for parallel for

In chapter 2 we describe how we tackled these issues by presenting the PALADIn language that is an interface with runtime system as a pluggin.

1.4 Methodology of experiments

All experiments have been conducted on a 32 cores (4 NUMA nodes with 8 cores each) Intel Xeon E5-4620 2.2Ghz (Sandy Bridge) with L3 cache(16384 KB). All implemented routines are in the `FFLAS-FFPACK` library[4] with git repository #6d0c995 relying on givaro #44a003a. The numerical BLAS used is OpenBLAS r0.2.9. MKL version is sp1.1.106, LAPACK version is v3.4.2 and PLASMA version is v2.5.0. We used the X-KAAPI runtime version 2.1 with last git commit: xkaapi_40ea2eb. The gcc compiler version is 5.3 (supporting OpenMP 4.0 and higher versions), the clang compiler version is 3.5.0 and the icpc compiler version is sp1.1.106 (using some gcc 4.7.4).

In our experiments, we use the effective Gfops (Giga field operations per second) metric defined as $Gfops = \frac{\#\text{ of field ops using classic matrix product}}{\text{time}}$. This is $\frac{2mnk}{\text{time}}$ for the product of an $m \times k$ by a $k \times n$ matrix, and $\frac{2n^3}{3\text{time}}$ for the Gaussian elimination of a full rank $n \times n$ matrix. We note that the effective Gfops are only true Gfops (consistent with the Gflops of numerical computations) when the classic matrix multiplication algorithm is used. Still this metric allows us to compare all algorithms on a uniform measure: the inverse of the time, normalized by an estimate of the problem size; the goal here is not to measure the bandwidth of our usage of the processor's arithmetic instructions.

1.5 Contributions

This work has been vastly supported by the collaboration framework of the HPAC[5] project, funding this Ph.D. program. Its research focuses on the design of efficient parallel dense exact linear algebra kernels. The contribution to most algorithmic and implementation aspects of the parallel Gaussian elimination and the computation of rank profiles will be presented here.

In this essay, we emphasize our contribution to algorithmic aspects and to implementation by advocating three thesis:

- *Sub-cubic exact linear algebra algorithms scale up in parallel.*
- *While recursive algorithms are undesired in numerical linear algebra they are good candidates to maintain scalability of sub-cubic algorithms with the help of optimized runtime systems in exact linear algebra, where they scale better than iterative algorithms.*
- *PALADIn proves that we are able to develop a generic parallel linear algebra library with runtime system as a plugin.*

The project of this thesis is to develop high performance shared memory parallel implementations of exact Gaussian elimination algorithm. We propose in this manuscript a recursive Gaussian elimination that can compute simultaneously the row and column rank profiles of a matrix as well as those of all of its leading sub-matrices, in the same

[4] http://linalg.org/projects/fflas-ffpack
[5] High Performance Algebraic Computing, ANR 11 BS02 013

time as state of the art Gaussian elimination algorithms. We also studied the conditions to make a Gaussian elimination algorithm reveal this information by defining a new matrix invariant, the rank profile matrix.

In Chapter 2 we deal with the parallelization of linear algebra routines. In order to abstract the computational code from the parallel programming environment, we develop a domain specific language, PALADIn: Parallel Algebraic Linear Algebra Dedicated Interface, that is based on C/C++ macros. This domain specific language allows the user to write C++ code and benefit from sequential and parallel executions on shared memory architectures using the standard OpenMP, TBB [61] and X-Kaapi [46] parallel runtime systems and thus providing data and task parallelism. Depending on the runtime system, task parallelism can use explicit synchronizations or data-dependency based synchronizations. Also, this language provides different matrix cutting strategies according to one or two dimensions. Moreover, block algorithms, such as block iterative and recursive matrix multiplication, can involve splitting according to three dimensions. The latter is also a feature that is provided to the user. The PALADIn interface can be used in any C++ library for linear algebra computation and gets the best performance from the three supported parallel runtime systems.

In Chapter 3 we present algorithms (matrix multiplication and triangular solving matrix algorithms) that are the building blocks for the design of parallel Gaussian elimination over a finite field on shared memory architectures. Specificities of exact computations over a finite field include the use of sub-cubic matrix arithmetic and of costly modular reductions. As a consequence coarse grain block algorithms perform more efficiently than fine grain algorithms and recursive algorithms are preferred. We incrementally build efficient kernels, for matrix multiplication first, then triangular system solving, on top of which a recursive PLUQ decomposition algorithm is built. We study the parallelization of these kernels using several algorithmic variants: either iterative or recursive and using different splitting strategies. Experiments show that recursive adaptive methods for matrix multiplication, hybrid recursive-iterative methods for triangular system solve and recursive versions of PLUQ decompositions, together with various data mapping policies, provide the best performance on 32 cores NUMA architecture.

Chapter 4 explores a panorama of block algorithms for the computation of Gaussian elimination. We study existing algorithms, present a new recursive algorithm for block Gaussian elimination and compare the implementations of the most important of them. We then study the cost of modular reduction for these algorithms in terms of arithmetic complexity.

In Chapter 5, we focus on the computation of echelon forms regardless of the parallel aspects. The row (resp. column) rank profile of a matrix describes the stair-case shape of its row (resp. column) echelon form. We propose a recursive Gaussian elimination that can compute simultaneously the row and column rank profiles of a matrix, as well as those of all of its leading sub-matrices, in the same time as state of the art Gaussian elimination algorithms. Here we first study the conditions making a Gaussian elimination algorithm reveal this information. We propose the definition of a new matrix invariant, the rank profile matrix, summarizing all information on the row and column rank profiles of all the leading sub-matrices. We also explore the conditions

for a Gaussian elimination algorithm to compute all or part of this invariant, through the corresponding PLUQ decomposition. As a consequence, we show that the classical iterative CUP decomposition algorithm can actually be adapted to compute the rank profile matrix. Used, in a Crout variant, as a base-case to our recursive Gaussian elimination implementation, it delivers a significant improvement in efficiency. Second, the row (resp. column) echelon form of a matrix are usually computed via different dedicated triangular decompositions. We show here that, from some PLUQ decompositions, it is possible to recover the row and column echelon forms of a matrix and of any of its leading sub-matrices thanks to an elementary post-processing algorithm.

In Chapter 6 we propose efficient parallel algorithms and implementations on shared memory architectures of LU factorization over a finite field. It is thus important to design parallel algorithms that preserve and compute the rank profile of the matrices. Moreover, as the rank profile is only discovered during the algorithm, block size has then to be dynamic. We propose and compare several block decompositions: tile iterative with left-looking, right-looking and Crout variants, slab and tile recursive. Overall, we show that the overhead of modular reductions is compensated by the fast linear algebra algorithms and that exact dense linear algebra matches the performance of full rank reference numerical software even in the presence of rank deficiencies.

Chapter 2

The PALADIn domain specific language

The aim of this chapter is to study how the `FFLAS-FFPACK` library can be updated to support parallelism. As mentioned in the introduction of this thesis, the `FFLAS-FFPACK` library relies on the BLAS interface and implements all basic linear algebra routines over finite fields. It provides:

- high performance implementations of basic linear algebra routines over word size prime fields,
- exact alternative to the numerical BLAS library,
- exact triangularization, system solving, determinant computation, rank computation, matrix inversion, polynomial characteristic.

To parallelize the `FFLAS-FFPACK` library, one needs to :

- explore several algorithms and variants adapted for parallel computation,
- investigate parallel runtime systems that provides abstraction to the user and that has scheduling as a plugin,
- and study the trade-off between using parallel loops of parallel tasks

The parallel computation constraints in numerical linear algebra and in exact linear algebra differ. First, the state of the art numerical libraries often deal with non singular matrices with fixed static cutting. This allows the user to manually map and schedule tasks or threads easier. Second, the use of iterative algorithms in numerical computation often implies one or two levels of parallelism. Whereas in exact computation, one can benefit from sub-cubic algorithms, such as Strassen algorithms for matrix multiplication, by using recursive algorithms. This implies multiple level of parallelism. Moreover, over finite fields, the elimination algorithms can discover rank deficiencies during the computation and implies having heterogeneous tasks. This issue can be treated using schedulers that handle unbalanced workloads. This motivated us to look for high level parallel programming environments to write a parallel `FFLAS-FFPACK` library.

Thus, we need a parallel programming environment that preserves portability, scalability and performance and also that offers features that can tackle the constraints that arise from the computation in exact linear algebra. However, no existing parallel

environment offers all these functionalities. We thus need to design a code that is independent from the parallel programming environment used and that uses runtime systems as a plugin.

We present here PALADIn [48] that stands for Parallel Algebraic Linear Algebra Dedicated Interface which is a specific language for parallel exact linear algebra. We first describe existing parallel environments and then focus on the parallelization issues in exact linear algebra. This allows to refine the search and select the most suited parallel environments for the computation over finite fields. Afterwards we give the description, the grammar and the implementation of the `PALADIn` language. The comparison of the parallel behavior of two runtime systems are then presented.

2.1 State of the art of parallel shared memory environments

2.1.1 Native threads parallel programming environments

POSIX Threads, usually referred to as pthreads, is a POSIX standard that defines an API for creating and manipulating threads. It is a set of C programming language types, functions and constants that allows individual threads of execution to be created, synchronized, and terminated manually. Most of high level parallel runtime systems (e.g. OpenMP [9], TBB [61] and xKaapi [46], ...) are implemented using pthreads at least in some of their implementation. This standard is considered as building block for high level parallel programming environments.

Porting a native threads-based solution onto another OS often requires code changes and increases the initial development/debugging effort and the maintenance burden, especially if one user wants portability. POSIX threads are typically used in Unix OS and Windows threads are typically used in Windows OS.

Pthreads allows the user to benefit from data parallelism and task parallelism, whereas using a high level library, one can also benefit from dataflow parallelization. The latter is a programming style based on scheduling tasks that relies on computing data dependencies. This allows to have finer synchronizations between tasks.

2.1.2 Parallel shared memory high level programming environments

The multitasking and multithreading parallelization has long existed in some manufacturers systems (CRAY, NEC, IBM, ...), but each had its own set of instructions. The resurgence of multiprocessor machines with shared memory pushed to define a standard. A significant majority of manufacturers and builders have adopted OpenMP [9] (Open Multi Processing) as a standard for shared memory parallelization.

Many parallel programming libraries exists alongside OpenMP, and can be pooled in two groups. We do not give here an exhaustive list of parallel shared memory libraries but we focus on the most used of them:

- Annotation based API:
  - OpenMP is an API that supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment

variables that influence run-time behavior. OpenMP is still considered as standard for shared-memory architecture thanks to several advantages:

* good performance and scalability,
* maturity,
* portability,
* simple syntax, requiring little programming effort,
* allows incremental parallel implementation.

– SMPSS - SMP SuperScalar is a task based programming environment for parallel applications based on function level parallelism. Tasks are defined with a pragma annotation right before their function definition. This annotation indicates that the following function is a task and specifies the directionality of each of the task parameters.

- Function-class based libraries
  – TBB [61] implements work stealing to balance a parallel workload across available processing cores. It is a library that implements task parallelism (fork-join) and data parallelism (parallel for)
  – CILK++ [8] is an extension to the C and C++ languages to support data and task parallelism using work-stealing policy.
  – xKaapi [46], as CILK++, implements both data and task parallelism using work-stealing policy but also with dataflow dependency between tasks.
  – StarPU is a task programming library for hybrid architectures

2.1.3 Parallel programming environments supported in the PALADIn interface

We focus here on three parallel programming environments that are supported in the PALADIn language: The standard OpenMP, TBB and xKaapi. We chose TBB and xKaapi along with the OpenMP standard for different reasons:

- The xKaapi runtime proved to be more efficient than OpenMP especially for recursive tasks [18].
- Nested parallelism is supported by OpenMP but it may be hard to avoid resource over-utilization. TBB has been designed to naturally support nested and recursive parallelism. A fixed number of threads are managed by the TBB task scheduler's task stealing technique.

2.1.3.1 OpenMP

The OpenMP standard specification started in 1997 and was mainly based on loop parallelization. The concept of tasks appeared in the OpenMP standard in the version 3.0 released in 2008. Thanks to these features both coarse-grain and fine-grain parallelism are possible. The latest release of OpenMP in 2013, version 4.0, adds some new features: mainly support for accelerator, thread affinity and tasking extensions by adding new OpenMP clauses. In the OpenMP standard various types of clause exist

to help the user set data environment management. In this work, we are interested in only few of them mostly in data sharing attribute clauses, synchronization clauses and some scheduling clauses.

OpenMP 3.0 allows two types of parallelization: Parallel loops and Fork-join using OpenMP tasks. In both types, data sharing attributes can be set using mainly the following clauses:

- `shared`: data within a parallel region are visible and accessible by all threads simultaneously.
- `private`: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable.
- `firstprivate`: like private clause except that data are initialized to original value.
- `lastprivate`: like private clause except that original value is updated after construct.
- `reduction`: a safe way of joining work from all threads after construct.

By default OpenMP passes all data as `firstprivate`. So, if needed, shared data can be specified by the user. One can refer to [9] for more details.

In the latest release of OpenMP 4.0 [10], dataflow parallelization model is supported via the `depend` clause, but is implemented in version 4.9 or newer of gcc, or version 3.7 or newer of Clang++ compiler.

OpenMP scheduler uses `libgomp` runtime library to handle thread and task creation and management.

2.1.3.2 TBB

Soon after the introduction of the first multicore CPU pentium D, Intel releases the first version of Threading Building Blocks (TBB) in 2006. TBB is a C++ template library that provides parallel algorithms and data structures avoiding to the user the need to deal with native threading. Unlike OpenMP, the TBB library does not use an extension of the language and can be used with any compiler.

The library implements task parallelism with work stealing strategy to circumvent unbalanced work load. Moreover, TBB algorithms (`parallel_for`, `parallel_reduce`, ...) are designed using fork-join tasks. Hence, every algorithm benefits from the work stealing strategy, unlike OpenMP `parallel for`. In TBB every loop based algorithm takes a functor that decides the cutting strategy of the loop range.

Since version 2.1 release in 2008, TBB integrates many C++ 11 features to simplify the interface. For example, one can easily create a task by using a C++ 11 lambda function, hence avoiding the need to define a specific functor.

Finally, the last feature provided by TBB is a memory allocator that takes into account many parameters to allow better scaling.

2.1.3.3 xKaapi

KAAPI stands for "Kernel for Adaptative, Asynchronous Parallel and Interactive programming". It is a C++ library that allows to execute fine/medium grain multithreaded computation with dynamic data flow synchronizations. It is a work-stealing based parallel library that originally [46] aimed to exploit with great efficiency the computation resources of a multiprocessor cluster. The latter is an efficient work-stealing algorithm for a macro data flow computation based on a minor extension of the POSIX thread interface.

Expressing parallelism using tasks allows the programmer to choose a finer grain parallelization. But the success of such an approach depends greatly on the runtime system used. Today, the KAAPI project focuses on shared memory and CPU/GPU computation. The xKaapi [47] library relies on the `libkomp` [18] runtime that provides an implementation of the OpenMP norm to be used as a replacement of the `libgomp` [64] library. Thus, it takes OpenMP directives and generates xKaapi tasks.

The `libkomp` runtime handles task creation and scheduling better than the `libgomp` runtime for recursive tasks [18]. Using the `libkomp` runtime, the xKaapi library thus supports also dataflow parallelism as the OpenMP standard. Using the version 4.9 of gcc or newer data dependencies are detected thanks to the "depend" clause of the OpenMP environment.

2.2 Parallelization issues in Exact linear algebra

In parallel exact linear algebra we focus on the use of high level parallel environments. This is motivated by several aspects that need to be taken into account during the parallelization of some routines in exact linear algebra:

- **Recursion:** In parallel numerical linear algebra, routines are mainly iterative algorithms [7] with fine-grain parallelization. This induces invariable block size with fixed cutting according to the matrix dimension, which makes it easier for the programmer to map and schedule tasks or threads manually.

  In exact linear algebra, one can benefit from asymptotically faster complexities by using Strassen [83] and Strassen-Winograd [29] variants for matrix multiplication, especially when applied on sufficiently large blocks. This implies using recursive algorithms that insures coarser grain cutting than iterative algorithms.

- **Unbalanced load and communication:** In parallel computation, the role of the scheduler of a runtime system is to assign jobs on available processors in order to optimize some criteria: maximizing the average workload and minimizing the overall completion time.

  In the case of numerical Gaussian elimination, the selection of pivots takes a significant amount of time in complete or full pivoting. In practice this is rarely done, because the improvement in stability is marginal. As a result, partial pivoting is used in practice. Thus, input matrix does not have a generic rank profile i.e. all it's leading principal minors are non zero. Therefore the splitting of the matrix can be done statically according to a granularity parameter. Whereas

triangular decomposition over finite fields often discovers rank deficiencies upon computation, thus generating tasks of unbalanced workloads. Hence, the runtime system used to parallelize exact linear algebra algorithms needs to have a scheduler that handles this issue, namely a work-stealing based scheduler. The more the scheduler is advanced the better it is, including dataflow dependency-based task scheduling. This allows to have finer synchronizations between tasks.

- **Routine composition:** In numerical linear algebra, the use of parallel iterative algorithms implies often a single level of parallelism. Thus, the scheduling of tasks can be done manually by the programmer.

  Since recursive algorithms are used in parallel exact linear algebra, parallel routines are called in each level of recursion. The composition of parallel routines implies many levels of parallelism. Task dataflow parallel programming languages rely on runtime schedulers that are aware of dependencies between tasks. By detecting these dependencies, resources utilization can be improved for composed tasks.

We thus want to avoid an API with low-level management and instead use runtime systems with dataflow-based synchronizations for the parallelization of exact linear algebra libraries. Consequently, a high level description of parallelism is required as the one used for instance in OpenMP [9], TBB [61] and xKaapi [46] parallel programming environments.

To solve exact linear algebra problems in parallel, the user needs a high-level library where genericity, performance and portability are the main concern. The goal of such a library should be :

- to allow the user to work at a high level of abstraction, thus avoiding some complications arising from the use of native threading
- to hide many details specific to parallel programming,
- to take into account large range of machine architectures.

2.3 PALADIn language

We present here PALADIn which is a domain specific language dedicated to parallel computations in exact linear algebra. It is included in the `FFLAS-FFPACK` library [14], but it can be used in any C++ linear algebra library. It supports OpenMP, TBB and xKaapi parallel environments and also allows the execution of the program in sequential.

For the sake of simplicity, lightness and portability, no precompilation phase is needed to use the PALADIn library. Indeed, domain specific languages can use precompilers to generate C/C++ programs. The PALADIn library does not need to use any precompiler, it can be compiled with any C/C++ compiler (g++, clang++, ...). By using g++ compiler with the release version 4.9, or Clang++ version 3.7, or newer versions, the user can benefit from a dataflow parallelization.

A domain specific language can be implemented in C++ by using either C/C++ macros or by C++ template meta-programming. Many aspects led us to implement a macros-based language:

- By adding macros, no important modifications are to be done to the original program.
- Macros can be used also for C-based libraries.
- Simpler for the programmer and the user.
- No function call runtime overhead when using macros.

Nevertheless, using a macro-based language implies some challenges:

- To give a simple interface to the user, some macros need to be overloaded. C++ allows to specify more than one definition for a function name in the same scope, which is called function overloading. When an overloaded function is called, the compiler determines the most appropriate definition to use by comparing the argument types used to call the function with the parameter types specified in the definitions. However, the compiler cannot detect overloaded macros in the preprocessing step as it would do with functions.
- To give the user more freedom, the PALADIn library allows to give a variable number of parameter. This leads to treat with variadic macros that make it more complex to iterate over arguments or count the list of arguments.

PALADIn focuses on four mains aspects:

1. Give an optimized parallel interface for exact linear algebra computation.
2. Be able to use sequential C++ and parallel implementation using different runtime systems with a unique syntax.
3. Provide the user the choice of different range cut strategies.
4. Allow switching between a dataflow model and an explicit task synchronization model with one implementation.

2.3.1 Implementation examples

By using the PALAD-Interface, the user can benefit from data or task parallelism. We do not intend to explain the PALADIn keywords here. Further explanation on the grammar and description are given in sections 2.3.2 to 2.3.5. In this section, we only give two code examples using PALADIn to illustrate its syntax. The first example shows the parallel loop syntax, and the second example illustrates the task syntax.

- **Parallel loop:**
  Let us consider three arrays `T`, `T1` and `T2`. The arrays can be any C/C++ structure. In this example, we sum `T1` with `T2` and store the result in `T` component-wise. A simple C++ code performing this operation is:

Listing 2.1: Loop summing of two arrays in C++

```
for(size_t i = 0 ; i < n ; ++i){
    T[i] = T1[i] + T2[i];
}
```

The translation of the above code in the PALALDIn syntax is:

Listing 2.2: Loop summing of two arrays with PALADIn

```
PARFOR1D(i, n, SPLITTER(),
    T[i] = T1[i]+T2[i];);
```

Using PALADIn parallel for, the user can set the keyword `SPLITTER()` to specify the desired strategy to cut chunks of the loop range iterated with `it`. In this example, since no arguments are given to the `SPLITTER` keyword the cutting strategy used is the default. More details on setting this keyword and the cutting strategies can be found in the PALADIn description in sections 2.3.4 and 2.3.5.3.

- **Task parallelism:**
  In this example we illustrate the PALADIn task syntax. Let us consider a free function `axpy` that uses three parameters `a`, `x` and `y` and computes `y+= ax`.

Listing 2.3: Task call with PALADIn

```
void axpy(const Element a, const Element b, Element y){
    y += a*x;
}

TASK(MODE(READ(a,x) READWRITE(y)),
    axpy(a,x,y));
```

The `READ` macro specifies that the arguments `a` and `x` are only read in the task execution. The `READWRITE` macro indicates that the variable `y` is in read and write mode during the task execution.

2.3.2 PALADIn grammar

PALADIn extends the instruction set of C++ with new instructions; the following grammar defines the sequences of those instructions (traces) that are considered not only valid syntactically but also at execution. In particular, it doesn't allow not only incorrect syntax constructions but also incorrect in term of the performance of executions, as for instance preventing nesting of `PARFOR` or `PAR_BLOCK`.

Any trace (full instruction stream) of a valid PALADIn program is an instance of PALADIN_INSTR. In this grammar, SEQ_INSTR denotes any sequential instruction (at any level of trace) resulting from the execution of a standard C++ block of instruction, excluding the new PALADIn keywords.

We extend this grammar by adding new set of instructions (lexicographic units are in bold):

PALADIN_INSTR ⟶ SEQ_INSTR
| `PAR_BLOCK`{SYNCH_INSTR}
| `PARFOR1D`(INTERVAL1D, SYNCH_INSTR)
| `PARFOR2D`(INTERVAL2D, SYNCH_INSTR)
| `PARFORBLOCK1D`(INTERVAL1D, SYNCH_INSTR)
| `PARFORBLOCK2D`(INTERVAL2D, SYNCH_INSTR)
| (PALADIN_INSTR;)*

SYNCH_INSTR ⟶ SEQ_INSTR
| `SYNCH_GROUP`(ASYNCH_INSTR)
| (SYNCH_INSTR;)*

ASYNCH_INSTR ⟶ SYNCH_INSTR
| `TASK`(DEPENDENCIES, ASYNCH_INSTR)
| `FOR1D`(INTERVAL1D, ASYNCH_INSTR)
| `FOR2D`(INTERVAL2D, ASYNCH_INSTR)
| `FORBLOCK1D`(INTERVAL1D, ASYNCH_INSTR)
| `FORBLOCK2D`(INTERVAL2D, ASYNCH_INSTR)
| `CHECK_DEPENDENCIES`
| (ASYNCH-INSTR;)*

DEPENDENCIES ⟶ MODE((CONSTREF_STATE)?
| (REF_STATE)?
| (READ_STATE)?
| (WRITE_STATE)?
| (READWRITE_STATE)?)

INTERVAL1D ⟶ IDF, INT_EXPR, SPLITTER

INTERVAL2D ⟶ IDF, INT_EXPR, INT_EXPR, SPLITTER

CONSTREF_STATE ⟶ ϵ | `CONSTREFERENCE`(VAR)+

REF_STATE ⟶ ϵ | `REFERENCE`(VAR)+

READ_STATE ⟶ ϵ | `READ`(VAR)+

WRITE_STATE ⟶ ϵ | `WRITE`(VAR)+

READWRITE_STATE ⟶ ϵ | `READWRITE`(VAR)+

SPLITTER ⟶ `SPLITTER`(INT_EXPR*, CUTTING_STRATEGY*, STRATEGY_PARAMETER*)
| `NOSPLIT()`

CUTTING_STRATEGY ⟶ `SINGLE`
| `ROW`
| `COLUMN`
| `BLOCK`
| `RECURSIVE`

STRATEGY_PARAMETER ⟶ `THREADS`
| `FIXED`
| `GRAIN`
| `TWO_D`
| `TWO_D_ADAPT`
| `THREE_D`
| `THREE_D_INPLACE`
| `THREE_D_ADAPT`

2.3.3 PALADIn description

The PALADIn extends the C++ language with new keywords that enables two complementary parallel programming paradigms:

- Data parallelism (*i.e.* SPMD *Single Program Multiple Data*), thanks to parallel regions defined by `PARFOR` keywords.
- Task parallelism:
  - serial-parallel computations (*i.e.* fork-join) with `PAR_BLOCK` and `SYNCH_GROUP` keywords;
  - asynchronous task parallelism (*i.e.* tasks which synchronizations are defined by data dependency instead) inside the `TASK` keyword with `READ`, `WRITE` and `READWRITE` keywords and also between dependent tasks by `CHECK_DEPENDENCIES` keyword.

PARFOR

To enable parallelism in the main sequential stream of instructions, `PARFOR1D(`$i, f, l,$ `SPLITTER,` I`)` declares a new parallel loop where the variable i ranges the interval $[f, l[$ to execute the body I. At each step, the interval is split (eventually recursively), thanks to `SPLITTER` methods, in sub-intervals that are concurrently computed. The `PARFOR1D` is terminated when all sub intervals are computed. Note that, like in conventional SPMD programming, I may contain branching according to the current iteration value i. The `PAR_BLOCK{`I`}` is the special case where the interval contains only one element.

`PARFOR1D`, `PARFOR2D` or `PAR_BLOCK` define a new parallel region. Like in OpenMP, parallel region shall not be nested within another parallel region.

TASK

`SYNCH_GROUP(`I`)` – with I denoting a block of instructions – enables to declare a new synchronization point (*i.e.* local barrier) : at execution, the `SYNCH_GROUP(`I`)` instruction is passed only after completion of all parallel computations forked by I.

Indeed, within a `SYNCH_GROUP`, the instruction `TASK(`D`,` I`)` forks the execution of the instruction I. The default synchronization (local barrier) after I is at the end of the `SYNCH_GROUP`. Moreover, D defines additional synchronizations from expressing dataflow dependencies; indeed D optionally defines the access mode to objects through four lists of variables:

- `REFERENCE`(*variables list*) : those variables are passed by reference to I (by default, variables are passed by value, similarly to mod `firstprivate` in OpenMP);
- `READ`(*variables list*) : those variables are read by I, but not modified;
- `WRITE`(*variables list*) : those variables are written, but not read;
- `READWRITE`(*variables list*) : those variables are read then written (update).

Note that `MODE` enables to describe any DAG of tasks in a sequential way; but only those explicit dependencies define synchronizations within a group, before the local barrier at the end of the group.

While OpenMP4 and xKaapi supports data dependencies, environments like OpenMP3 or TBB do not. Also, to guarantee synchronizations related to data dependencies in a language with no support of it, we have defined a new instruction `CHECK_DEPENDENCIES` that forces the dependencies previously defined in the current group. This implementation may be pessimistic but ensures PALADIn independence from the underlying parallel environment. In our OpenMP4 and xKaapi implementations, since dependencies are ensured at task creation, `CHECK_DEPENDENCIES` has no effect. But in OpenMP3 and TBB it is compiled as a synchronization barrier within the group.

2.3.4 Cutting strategies

The `SPLITTER` keyword in the previous grammar gives the range cut strategy used to execute the corresponding program inside the loop. It uses three parameters to define explicitly each strategy: (nt, *CUTTING_STRATEGY*, *STRATEGY_PARAMETER*). The first parameter nt is an integer value that refers to the number of threads to be used. We present here all the cutting strategies defined by the parameters *CUTTING_STRATEGY* and *STRATEGY_PARAMETER* that are used in the following macros: `PARFOR1D PARFOR2D`, `FOR1D` and `FOR2D`.

2.3.4.1 Iterative split strategies

PALADIn implements 3 cutting strategies that cuts over one dimension or two dimensions of the output matrices. The keywords `ROW` or `COLUMN` defines cutting strategies that cuts respectively over the rows or columns of the output matrix. The `BLOCK` cutting strategy cuts over the two dimensions. Each cutting strategy has a strategy parameter that specifies how the grain size of the cutting is set.

Thus, PALADIn implements an overall of 9 different matrix cutting strategies for iterative algorithms set by the `SPLITTER()` keyword and are grouped in two categories:

- The one dimension cutting where the cutting strategy is enabled on only one dimension of the output matrix. The `SPLITTER` keyword can be defined in 6 different ways:

  `SPLITTER(`nt`, ROW, THREADS)`: This cutting take into account the number of processors nt, and splits the rows of the matrix into exactly nt row slabs.

  `SPLITTER(`nt`, ROW, FIXED)`: This cutting strategy cuts the rows of the matrix with a fixed grain size. The latter could be optimized during the installation of the library. It is set to 256 (optimized value on many systems).

`SPLITTER(`gr`, ROW, GRAIN)`: This cutting strategy cuts the rows of the matrix with a fixed grain size that can be set by the user. The gr parameter in this case plays the role of the grain size set by the user. If gr is set to 256 using the `GRAIN` strategy parameter, this cuts the matrix into row blocks of size 256.

`SPLITTER(`nt`, COLUMN, THREADS)`: As the `(ROW,THREADS)` strategy, this cutting take into account the number of processors nt but splits the columns of the matrix into exactly nt column slabs.

`SPLITTER(`nt`, COLUMN, FIXED)`: This cutting strategy cuts the columns of the matrix with a fixed grain size set by default to 256.

`SPLITTER(`gr`, COLUMN, GRAIN)`: This cutting strategy cuts the columns of the matrix with a fixed grain size that can be set by the user. As the `SPLITTER(`gr`, ROW, GRAIN)` cutting strategy, here the user sets the grain size in the gr parameter.

- The two dimension cutting

  `SPLITTER(`nt`, BLOCK, THREADS)`: This cutting strategy cuts the two dimensions of the output matrix. When performing the operation $C \leftarrow A \times B$, it splits A in s row slabs and B in t column slabs, and thus splits the matrix C in $s \times t$ tiles. The values for s and t are chosen such that their product equals the number of threads given by nt. When no argument is given to `SPLITTER()`, this cutting strategy is set as the default.

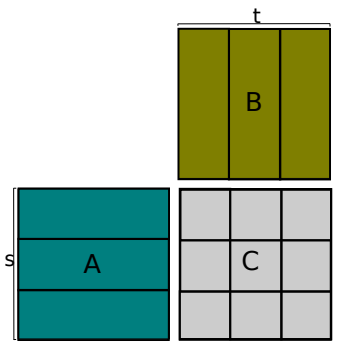

Figure 2.1: (nt, `BLOCK, THREADS)` cutting strategy for iterative matrix-matrix multiplication algorithm

`SPLITTER(`nt`, BLOCK, FIXED)`: This cutting strategy cuts the two dimensions of the matrix C as the previous cutting strategy but with a fixed grain size set to 256. This gives tiles of size 256×256.

`SPLITTER(`gr`, BLOCK, GRAIN)`: This strategy allows the user to give a block size gr. Thus, tiles of the output matrix are of size $gr \times gr$.

The `SPLITTER(`nt`, SINGLE, THREADS)` strategy is the strategy that does not cut the matrices and thus allows to execute a task sequentially inside a parallel program. This allows the user to call sequential tasks over small dimensions while preserving the dependencies settings inside the parallel program. The `NOSPLIT()` keyword provides a standard sequential behavior of the loop with no task creation.

2.3.4.2 Recursive split strategies

In the case of recursive matrix multiplication algorithms that involve splitting of three dimensions, one can use 5 different recursive cuttings provided by PALADIn. We show here these cutting strategies, that are dedicated for the parallel general matrix multiplication (`pfgemm`) operation: computing $C \leftarrow \alpha A \times B + \beta C$, where A, B and C are dense matrices with dimensions respectively (m, k), (k, n) and (m, n). These recursive cutting strategies can be applied to other linear algebra routines involving splittings over three dimensions. Recursive splits can be done on three dimensions: inner, outer and third dimensions. Cuttings corresponds to the number of recursive calls.

`SPLITTER(`nt`, RECURSIVE, TWO_D)`: The 2D recursive partitioning performs a 2×2 splitting of the matrix C at each level of recursion. Each recursive call is then allocated a quarter of the number of threads available. This constrains the total number of tasks created to be a power of 4 and the splitting will work best when the number of threads is also a power of 4. The `TWO_D` cuts the inner and outer dimensions

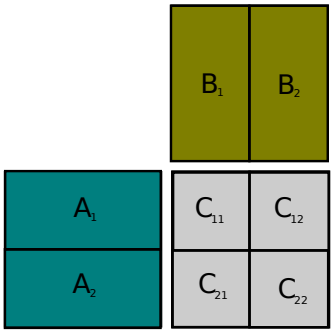

Figure 2.2: `TWO_D` cutting strategy for recursive matrix-matrix multiplication algorithm

`SPLITTER(`nt`, RECURSIVE, TWO_D_ADAPT)`: The 2D recursive adaptive partitioning cuts the largest dimension between m and n, at each level of recursion, creating two independent recursive calls. The number of threads is then divided by two and allocated for each separate call (with a discrepancy of allocated threads of at most one). This splitting better adapts to an arbitrary number of threads provided.

The 3D strategy splits the three dimensions m, n and k.

`SPLITTER(`nt`, RECURSIVE, THREE_D_INPLACE)`: The 3D in-place recursive cutting strategy performs 4 multiply calls, waits until blocks elements are computed and then performs 4 multiply and accumulation. This variant is called *inplace* since blocks of matrix C are computed in place.

`SPLITTER(`nt`, RECURSIVE, THREE_D)`: performs 8 multiply calls in parallel and then performs the add at the end. To perform 8 multiplications in parallel we need to store the block results of 4 multiplications in temporary matrices. As in the previous routine, each task calls recursively the routine.

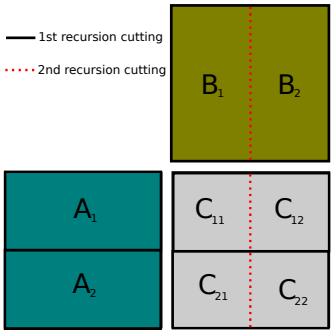

Figure 2.3: `TWO_D_ADAPT` cutting strategy for recursive matrix-matrix multiplication algorithm

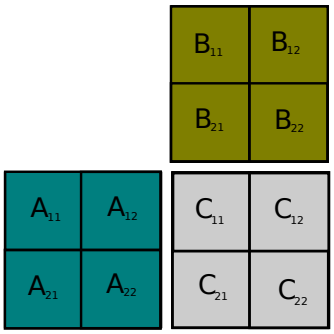

Figure 2.4: The 3D cutting strategy for recursive matrix-matrix multiplication algorithm

`SPLITTER(`nt`, RECURSIVE, THREE_D_ADAPT)`: The 3D recursive adaptive cutting strategy cuts the largest of the three dimensions in halves. When the dimension k is split, a temporary is allocated to perform the two products in parallel. As the split of the k dimension introduces some overhead, one can introduce a weighted penalty system to only split this dimension when it is largely greater than the other dimensions: with a penalty factor of p, the dimension k is split only when $\max(m, n) < pk$.

2.3.5 Implementation of PALADIn language

The PALADIn language is implemented by macro definitions with implementations provided for sequential C++ programs and several target parallel environments. Currently for the C++ language the libraries OpenMP, TBB and xKaapi are targeted. Thus syntax of C++ is not modified, enabling to use PALADIn for any program written in C++.

2.3.5.1 Macro definitions

We list below all implemented macros in the PALADIn language and give their specification and usage:

- `NUM_THREADS` : gives the number of current threads set by default.

- `MAX_THREADS`: gives the number of maximum threads in a parallel region (gives the max number of threads of the machines if not in a parallel region).
- `PAR_BLOCK{...}`: Defines a parallel region creating a team of threads that are launched in parallel (code inside will be executed by a single thread in the team).
- `SYNCH_GROUP(`I`)`: This macro is used to synchronize a group of tasks. It adds a synchronization point at the end of the block of instructions I. `SYNCH_GROUP` macro is effective when the block of instructions I is a block of tasks, each defined by the macro TASK. This macro can be used only once inside a routine to get the maximum parallel performance using TBB. `CHECK_DEPENDENCIES` can be used to add explicit synchronizations between tasks.
- `CHECK_DEPENDENCIES`: In an explicit synchronization environnement behaves as a local synchronization (waits for all children of a current task), does nothing in a dataflow environnement.
- `TASK(MODE(...),` I`)`: creates a task for the block of instructions I.
- `MODE(`D_i`)`: defines access mode for variables. Five access modes are defined for the MODE macro, ($0 \leq i \leq 5$):
  - `READ(variables)` access mode sets the variables that are read
  - `WRITE(variables)` access mode sets the variables that are written
  - `READWRITE(variables)` access mode sets the variables that are read and written
  - `CONSTREFERENCE(variables) and REFERENCE(variables)` access mode sets the variables that are captured by reference
  - all variables are captured by value if macros `CONSTREFERENCE` and `REFERENCE` are not specified inside the MODE macro.
- `FOR1D(i, dim, SPLITTER(), I)` : Cutting matrix in blocks over one dimension (`dim`). The cutting strategy is defined by the `SPLITTER` parameter. The splitting strategy is applied on the block of instruction I. The user doesn't have access to internal chunk dimensions and to the internal iterator used to range each chunk. Instructions inside the `FOR1D` have the same syntax as if inside a for loop. This is not the case for the `FORBLOCK1D`.
- `FORBLOCK1D(iterator, dim, SPLITTER(), I)` : Cutting matrix in blocks over one dimension (`dim`). The cutting strategy is defined by the `SPLITTER` parameter. The splitting strategy is applied on the block of instruction I. On the contrary to the `FOR1D`, the user has access to internal chunk dimensions and to the internal iterator used to range each chunk(using `iterator.begin()` and `iterator.end()`).
- `FOR2D(iterator, dim1, dim2, SPLITTER(), I)`: Cutting matrix in blocks over two dimensions (`dim1` and `dim2`). The cutting strategy is defined by the `SPLITTER` parameter. The splitting strategy is applied on the block of instruction I. As the `FOR1D`, the `FOR2D` does not give access to the user to the internal iterator.

- `FORBLOCK2D(iterator, dim1, dim2, SPLITTER(), I)`: Cutting matrix in blocks over two dimensions (`dim1` and `dim2`). The cutting strategy is defined by the `SPLITTER` parameter. The splitting strategy is applied on the block of instruction I. Internal iterator has two dimensions that can be accessed thanks to `iterator.ibegin()` and `iterator.iend()` for the row dimension, and `iterator.jbegin()` and `iterator.jend()` for the column dimension.

- `PARFOR1D(i, dim, SPLITTER(), I)`: As the `FOR1D` but the `PARFOR1D` opens a parallel region.

- `PARFORBLOCK1D(iterator, dim, SPLITTER(), I)`: As the `FORBLOCK1D` but the `PARFORBLOCK1D` opens a parallel region.

- `PARFOR2D(iterator, dim1, dim2, SPLITTER(), I)`: As the `FOR2D` but the `PARFOR2D` opens a parallel region.

- `PARFORBLOCK2D(iterator, dim1, dim2, SPLITTER(), I)`: As the `FORBLOCK2D` but the `PARFORBLOCK2D` opens a parallel region.

2.3.5.2 Complementary class functions

Inside a for loop (sequential or parallel loop) the PALADIn interface sets a cutting strategy and iterates over dimensions. Thus we need to store the iterators which cannot be done using C++ macros. Therefore, we added complementary class functions to the PALADIn language. Using template C++ functions we implemented the different cutting strategies and stored the iterators.

The `SPLITTER` macro that is used in the `FOR1D`, `FOR2D`, `PARFOR1D`, `PARFOR2D`, `FORBLOCK1D`, `FORBLOCK2D`, `PARFORBLOCK1D` and the `PARFORBLOCK2D` gives the number of threads or the grain size and the cutting strategy to be used. It can be set for sequential or for parallel executions.

We show here the implementation of the `SPLITTER` cutting strategies using c++ struct functions. This is implemented in the *blockcuts.inl* file given in appendix D. Inside the namespace `FFLAS` we define a Helper that specifies whether the program will be executed in parallel or in sequential. This is implemented inside the namespace `ParSeqHelper`. There two struct functions are defined:

- The `Parallel(size_t n, CuttingStrategy m, StrategyParameter p)` struct function contains three members:
  - The first member is an unsigned integer. It is used to set the number of threads or the grain size depending on the strategy used.
  - The `CuttingStrategy` member is an $enum - specifier$ that declares the `BLOCK`, `ROW`, `COLUMN`, `SINGLE` and the `RECURSIVE` enumerators inside an unscoped enumeration.

  The `StrategyParameter` member is also a $enum - specifier$ that declares inside an unscoped enumeration the `THREADS`, `FIXED`, `GRAIN`, `TWO_D`, `TWO_D_ADAPT`, `THREE_D`, `THREE_D_INPLACE` and the `THREE_D_ADAPT` enumerators.

- The `Sequential()` struct function that sets the number of threads to 1 using no cutting strategy (the `SINGLE` keyword).

The `NOSPLIT()` keyword is defined by `FFLAS::ParSeqHelper::Sequential()` and used for sequential execution. For parallel execution the user can give more information on the method of cutting of the for loops using the `SPLITTER`(t, *meth*, *strat*) macro. The latter is defined by `FFLAS::ParSeqHelper::Parallel`(t, *meth*, *strat*) where *meth* is the method of cutting the matrix (i.e. `BLOCK`, `ROW`, `COLUMN`, `SINGLE` or `RECURSIVE`), *strat* is the strategy parameter (i.e. `THREADS`, `FIXED`, `GRAIN`, `TWO_D`, `TWO_D_ADAPT`, `THREE_D`, `THREE_D_INPLACE` or `THREE_D_ADAPT`) and t is the number of threads or the grain size that is taken into account for the given cutting strategy.

2.3.5.3 Code examples

We show here the PALADIn semantics and its equivalence in OpenMP and TBB on the *axpy* example given in section 2.3.3. The task that performs this operation is invoked by :

```
void axpy(const Element a, const Element b, Element y){
    y += a*x;
}

SYNCH_GROUP(
    TASK( MODE( READ(x, y) READWRITE(y)),
        axpy(a, x, y)););
```

We show its equivalent implementation with OpenMP 3 syntax:

```
#pragma omp task
    axpy(a, x, y);
#pragma omp taskwait
```

With OpenMP 4 syntax using the ”depend” clause:

```
#pragma omp task depend(in:a,x) depend(inout:y)
    axpy(a, x, y);
#pragma omp taskwait
```

Using lambda function, the syntax with tbb becomes:

```
tbb::task_group g;
g.run([&y, a, x](){axpy(a, x, y);});
g.wait();
```

The `SYNCH_GROUP` macro ensures that a local synchronization is set at the end of the *axpy* task.

xKaapi tasks are created using the OpenMP3.1 and OpenMP4.0 task when using the `libkomp` runtime library. The user can thus benefit from different programming environments and also xKaapi parallel library when using task parallelism. But since data parallelism (i.e. parallel loops) is not supported in the `libkomp` runtime library, the

user can only benefit from OpenMP and TBB implementations when using `PARFOR1D` and `PARFOR2D` macros.

Below we illustrate two examples using the PALADIn syntax, and show, in section 2.4, how the cutting strategy can have an impact on the parallel performance.

Example 1: Matrix addition

The first example depicts three different implementations to write a parallel loop of a C++ program: one using the OpenMP parallel loop syntax, Listing 2.4, the other one using TBB `parallel_for`, Listing 2.5, and the PALADIn syntax, Listing 2.6, for the parallelization of the same loop by using different cutting strategies. In this example we attempt to perform the operation $C \leftarrow A + B$, where the matrices A, B and C are stored in a row major manner. The `pfadd` routine processes this operation on several pairs of operands simultaneously which allows each thread to execute a vectorized add operation.

Listing 2.4: *parallel fadd* with OpenMP parallel loop

```
1 void pfadd(const Field & F, const Element *A, const Element *B,
    Element *C, size_t n){
2     #pragma omp parallel for
3     for(size_t i = 0 ; i < n ; ++i){
4         FFLAS::fadd(F, 1, n, A+i*n, n, B+i*n, n, C+i*n, n);
5     }
6 }
```

Listing 2.5: *parallel fadd* with TBB

```
1 void pfadd(const Field & F, const Element *A, const Element *B,
    Element *C, size_t n){
2   parallel_for(blocked_range<size_t>(0, n),
3   [&](blocked_range<size_t> & r){
4     for(size_t i = r.begin() ; i < r.end() ; ++i){
5       FFLAS::fadd(F, 1, n, A+i*n, n, B+i*n, n, C+i*n, n);
6     }
7   });
8 }
```

Listing 2.6: *parallel fadd* with PALADIn

```
1   void pfadd(const Field & F, const Element *A, const Element *B,
        Element *C, size_t n){
2     PARFORBLOCK1D(it, n, SPLITTER(32, ROW, THREADS),
3     FFLAS::fadd(F, it.end()-it.begin(), n, A+it.begin()*n, n, B+
          it.begin()*n, n, C+it.begin()*n, n););
4   }
```

The `SPLITTER` parameter, in Listing 2.6, can be defined as a parallel or a sequential helper. In the sequential case, no cutting strategy will be used. In this example it is set as a parallel helper with `FFLAS::ParSeqHelper::Parallel` and takes three arguments to specify a cutting strategy: the number of threads, the strategy method of splitting and the strategy parameter (i.e. in this example the `(32, ROW, THREADS)` strategy that cuts rows of the matrix into 32 slabs).

Example 2: The sparse matrix-vector product

As a second example, we use the sparse matrix-vector product over a finite field. In this operation, the matrix is stored in the classical Compress Sparse Rows (CSR) format [16], see Figure 2.5. The CSR format is composed of 3 arrays: the first one to store the value of non zeros, the second one to store the column indices of the non zeros elements, and a third one containing pointer of where the ith rows start in the two previous arrays. Hence the CSR save some memory which increase performance as the SpMV operation is memory bound.

The OpenMP implementation is shown in Listing 2.7, the TBB implementation in Listing 2.8 and the PALADIn implementation in Listing 2.9. In the latter implementation the `SPLITTER()` keyword is set with no arguments, this will choose the default cutting strategy which is (nt, `BLOCK, THREADS`), where nt here is the number of processors of the machine.

The performance behavior of this operation and its implementations are explained in section 2.4.

$$M = \begin{bmatrix} 1 & & & & & 7 & & \\ & & 2 & & 5 & & & 8 \\ & & & 4 & & & 6 & \\ 1 & -1 & & & & & & \\ & & & & 9 & & & \\ & & & & & & & \\ & & & 1 & 1 & & & \\ 4 & & & & & 6 & & \\ & & & & & & 7 & 9 \end{bmatrix} \tag{2.1}$$

Val	1	7	2	5	8	4	6	1	-1	9	1	1	4	6	7	9
Col	1	6	3	5	9	4	7	1	2	5	4	5	1	6	7	9
RowPtr	1	3	6	9	11	12	12	13	15	17						

Figure 2.5: CSR storage of the matrix M.

We illustrate, in this example, the implementation of a simple parallel loop with OpenMP and TBB on a sparse matrix-vector multiplication operation.

Listing 2.7: Parallel implementation of SpMV with OpenMP

```
void spmv(const Field & F, const CSRMat & A, const Element *x,
    Element *y){
    #pragma omp parallel for
    for(size_t i = 0 ; i < n ; ++i){
        size_t start = M.rowPtr[i], stop = M.rowPtr[i+1];
        for(size_t j = start ; j < stop ; ++j){
            y[i] += M.val[j] * x[M.col[j]];
        }
    }
}
```

Listing 2.8: Parallel implementation of SpMV with TBB

```
void spmv(const Field & F, const CSRMat & A, const Element *x,
    Element *y){
    parallel_for(blocked_range<size_t>(0, A.m),
        [&](blocked_range<size_t> & r){
            for(size_t i = r.begin() ; i < r.end() ; ++i){
                size_t start = M.rowPtr[i], stop = M.rowPtr[i+1];
                for(size_t j = start ; j < stop ; ++j){
                    y[i] += M.val[j] * x[M.col[j]];
                }
            }
        });
}
```

We show below the PALADIn syntax to implementing the same sparse matrix-vector operation.

Listing 2.9: Parallel implementation of SpMV with PALADIn

```
void spmv(const Field & F, const CSRMat & A, const Element *x,
    Element *y){
    PARFORBLOCK1D(it, 0, A.m, SPLITTER(),
            for(size_t i = it.begin() ; i < it.end() ; ++i){
                size_t start = M.rowPtr[i], stop = M.rowPtr[i+1];
                for(size_t j = start ; j < stop ; ++j){
                    y[i] += M.val[j] * x[M.col[j]];
                }
            }
}
```

2.4 Performances of linear algebra routines using the PALADIn language

In this section we show the performance of the PALADIn library for the data parallelism programming style (i.e. using parallel loops). Performance of task parallelism will be presented in chapter 3 and chapter 6.

Parallelizing loops with OpenMP can be very simple using `#pragma omp parallel for`. This lets the scheduler of OpenMP to choose the default mode for cutting loop iterations in chunks and distribute them on available resources. The user can set the strategy for the scheduler to specify the size of chunks that can be executed statically or dynamically. Using the PALADIn cutting strategies one can have better performance without important modification of the program.

We show here the performance of the two examples described in the previous section.

2.4.1 Matrix Addition

Table 2.1 shows the performance of Listing 2.4, Listing 2.5 and Listing 2.6 described before. For the PALADIn implementation two cutting strategies are used, `(ROW, THREADS)` and `(ROW, FIXED)`, according to one dimension to show that for a simple parallel loop one can achieve better performance using the PALADIn cutting strategy than the default cutting strategies given by a standard parallel model such as OpenMP and TBB. We execute the PALADIn cutting strategies with OpenMP and TBB. Table 2.1 demonstrates that the best performance is obtained with the `(ROW, THREADS)` cutting strategy for the parallel dense matrix-matrix addition operation. Even when using the same cutting strategy `(ROW, THREADS)` we can see that TBB is slower than OpenMP. Since the executions are done on 32 cores of a NUMA machine architecture, TBB does not choose obviously the best cutting strategy adapted to the machine hierarchy in this case.

Matrix dimension	1000	2000	3000	4000
omp parallel for	1.7	4.2	8.4	15.0
omp `PARFORBLOCK1D(ROW, THREADS)`	**1.2**	**3.6**	**8.2**	**14.0**
omp `PARFORBLOCK1D(ROW, FIXED)`	1.9	5.8	9.8	17.0
TBB parallel_for	5.0	16.0	28.0	160.0
TBB `PARFORBLOCK1D(ROW, THREADS)`	**1.4**	**6.6**	**15.0**	**30.0**
TBB `PARFORBLOCK1D(ROW, FIXED)`	2.6	11.0	23.0	34.0

Table 2.1: Timings in milliseconds of the PALADIn language using two different cutting strategies compared to openmp "parallel for" for the fadd operation of two square matrices on 32 cores of the HPAC machine

2.4.2 SpMV Operation

For the experiments we used two matrices:

- ffs619 of dimensions 653365×653365 with an average of 100 non zeros elements by row
- ff809 of dimensions 3602667×3602667 with an average of 110 non zeros elements by row

The non zeros elements of the matrices are not uniformly distributed, more than 90 % of the non zeros elements are in the first thousand rows and the last rows have at most 3 elements. The computation is done over the finite field $\mathbb{Z}/524309\mathbb{Z}$, using 8 cores.

Results are reported in table 2.2. The SpMV operation using OpenMP implementation with default loop strategy does not perform well because the scheduling strategy cannot deal with unbalanced workload caused by the particular distribution of the non zero elements. The SpMV operation using TBB default loop strategy performs better because TBB uses task parallelism with a work stealing strategy when computing chunks of the loop. This allows balancing the workloads more efficiently over the cores. However, by using default loop strategy, the TBB tasks are composed of at most two rows. This means that for the sparsest part of the matrix, a task only computes 8 multiplications and 8 additions. Hence, the overhead of TBB task management greatly impacts the performance. With the PALADIn implementation, the `(ROW, THREADS)` using OpenMP and TBB cutting strategy produces only 8 tasks from the loop range on 8 cores but only the first task does the majority of the computation. This strategy is not adapted for the structure of the input matrices, since all arithmetic operations are grouped in the first rows.

The `(ROW, FIXED)` cutting strategy splits the loop into a fixed number of iterations (256 in these benchmarks) allowing the scheduler to efficiently balance the workload over the cores and the tasks are big enough to cover the management overhead.

Matrix	ffs619	ff809
OpenMP	0.49	0.26
omp `FORBLOCK1D(ROW, THREADS)`	0.40	0.24
omp `FORBLOCK1D(ROW, FIXED)`	**2.00**	**0.95**
TBB	0.95	0.43
tbb `FORBLOCK1D(ROW, THREADS)`	0.44	0.26
tbb `FORBLOCK1D(ROW, FIXED)`	**1.99**	**0.90**

Table 2.2: Performance in Gfops of PALADIn compared to OpenMP and TBB "parallel for" for the CSR spmv operation of two sparses matrices arising in the discrete logarithm problem [5] on the HPAC machine.

We can see clearly, in this table, that using the cutting strategy `(ROW, FIXED)` one can achieve at least a speed-up of 2 to perform a sparse matrix-vector multiplication operation.

The performances of fork-join parallelism and dataflow synchronizations using the PALADIn language will be detailed in chapters 3 and 6.

2.5 PALADIn in `FFLAS-FFPACK`

The PALADIn is implemented inside the `FFLAS-FFPACK` library in the folder "paladin" of the "trunk/fflas-ffpack" repository. Hereafter we list the files existing in this folder and their contents:

- The template class functions:
  - pfgemm_variants.inl contains all recursive cuttings strategies.
  - blockcuts.inl contains all iterative cutting strategies.
- Macros definitions are all in the *parallel.h* file.

- Auxiliary files: kaapi_routines.inl is an attempt to write some functions with the xKaapi syntax. This work is postponed since the implementation of the libkomp [18] runtime system: xKaapi tasks can be simply written in OpenMP tasks via the libkomp runtime system.

In this manuscript, the parallel implementation of all routines will be given using the PALADIn syntax to allow testing with the different supported runtime systems. In chapter 3, mainly the `FORBLOCK` keywords are used in the implementation of the iterative algorithms. Task parallelism is then used in all other recursive routines, using `TASK` keyword, in chapters 3 and 6.

2.6 Conclusion

We presented in this Chapter, the PALADIn interface that allows the user, using mainly C macros, to write C++ code and benefit from sequential and parallel executions on shared memory architectures. We have shown three parallel environment libraries: OpenMP, TBB and xKaapi, that are supported in this domain specific language. This interface provides data parallelism and task parallelization. Hence, depending on the runtime system used, the task parallelization can be performed either by using explicit synchronizations or using data-dependency based synchronizations.

We have proved that, comparing to OpenMP and TBB parallel for, the diversity of matrix cutting strategies provided in this language, helps the user to obtain always better performance.

The PALADIn interface can be used in any C++ library for linear algebra computation and gets the best parallel performance from three supported runtime systems.

Further extensions of the PALADIn library can be implemented, especially when detecting dependencies between tasks. For now, data dependencies are detected thanks to the pointer passed in parameters. The computation of data dependencies could be affected and the result could be incorrect when treating with overlapping blocks. In the latter case, the range of the blocks can be passed in parameter in the macros `READ`, `WRITE` and `READWRITE`. This feature relies on the latest versions of OpenMP where dependencies can be expressed by specifying the range of blocks. Still not all structure types are supported.

The PALADIn syntax will be used in the next chapters to be able to switch and compare between OpenMP, xKaapi and TBB runtime systems and to get the best performances from each.

Chapter 3

Parallel building blocks in exact computation

This chapter focuses on the parallelization of building block kernels in exact linear algebra. These kernel routines are matrix multiplication and triangular solving matrix routines on top of which efficient parallel Gaussian elimination implementations are built in chapter 6. We look into parallel implementations of these routines in order to compose them into higher level parallel algorithms.

Thus, we studied classical and fast-variants for the computation of the matrix multiplication operation. We denote by *classical* algorithms the cubic time complexity algorithms performing a standard matrix multiplication. We also denote by *fast* variants the algorithms that have sub-cubic time complexity ($O(n^\omega)$, with $\omega < 3$) for the same computation such as Strassen [83] algorithm. Classical matrix multiplication has attained a great maturity in numerical linear algebra and resulted in the implementation of the `dgemm` routine that has been intensively studied [88, 50, 59]. The parallel implementations of fast-variants have been also studied in numerical linear algebra. For shared memory architectures, Strassen's fast matrix multiplication has been explored in parallel and implemented for instance in [24, 68, 6, 4, 29]. Optimization tools were presented for finding many fast algorithms based on factoring bilinear forms and were used by [6] to automatically translate a fast matrix multiplication algorithm to high performance sequential and parallel implementations. For their parallel algorithms, they use the ideas of breadth-first and depth first traversals of the recursion trees, which were first considered by [68] and [4] for minimizing memory footprint and communication.

First, we examine the ingredients for the design of parallel kernels in exact linear algebra. We show the impact of the grain size, the runtime system used and the data mapping strategy used. We also propose in this section a new data mapping strategy on NUMA architecture. Then, we study the most common existing parallel implementations and algorithms to compute general matrix multiplication. We also propose new implementations that are adapted for the computation in exact linear algebra using the cutting strategies provided by the PALADIn language. Different variants and algorithms are investigated:

- Classical algorithms that compute the matrix product with cubic complexity. Here iterative and recursive algorithms are studied with different implementations to pick the best parallel implementation of classical matrix multiplication operation on shared memory machines.

- Fast algorithms that compute the matrix product with sub-cubic complexities are then studied. We restrict ourselves to the parallelization of the Strassen-Winograd algorithm [90]. Here we looked into the trade-off of using asymptotically faster algorithms but that introduce more synchronizations between tasks and managed to find a parallel implementation of Strassen-Winograd algorithm that surpasses the performance of reference parallel numerical libraries (such as the MKL and the PLASMA-Quark libraries).

- Parallel hybrid variants that combine classical and fast algorithms are also explored. Here we focus on mixing our best implementations of classical and fast variants to find the best threshold above which algorithms can be switched. The Strassen-Winograd algorithm parallel implementation switches to the classical algorithm on modest size base cases and give the best performance between all our implemented variants.

In Section 3.1 we present the ingredients that need to be considered in exact linear algebra. In section 3.2 we first show a panorama of parallel classical matrix multiplication algorithms in § 3.2.1 where we propose recursive and iterative implementations. In § 3.2.2, we present a parallel implementation of a Strassen-Winograd variant and attempt to mix the algorithms together to obtain the best parallel implementation in terms of performance and parallel speed-up. We then compare with the existing state of the art libraries in § 3.2.3.

Last, iterative, recursive and hybrid algorithm implementations are explored for the computation of triangular solving matrix of [38]. The parallel implementations of these algorithms are studied and are compared with the state of the art numerical libraries. We prove that the iterative parallel variant has a better parallel speed-up than the recursive variant. Moreover, the hybrid variant improves over the iterative variant when the column dimension of the right hand side is small.

3.1 Ingredients for the design of parallel kernels in Exact linear algebra

Over a finite field, while some aspects are similar to numerical computation there remain some specificities that are different. We list concisely these specificities on which parallel efficiency of exact algorithms relies.

3.1.1 Impact of modular reductions

Computations over prime finite fields such as $\mathbb{Z}/p\mathbb{Z}$, with p less than 23 bits, are done in [36], first, by embedding finite field elements in integers stored as floating point numbers. Then, modular reduction operations are applied to convert back elements over the finite field. To minimize the number of modular reductions in these algorithms, the technique is to accumulate several multiplications before reducing while keeping the result exact. Moreover, for further improvement we consider block algorithms that have better cache efficiency as mentioned in section 1.2. This approach is only valid as long as integer computations do not exceed the size of the mantissa. For instance in the multiplication of $A \times B$ over $\mathbb{Z}/p\mathbb{Z}$, with n the common dimension and elements are in $[\![0..p-1]\!]$, no overflow occurs if $n(p-1)^2 < 2^{mantissa}$. Further explanation of the

impact of modular reductions is given in the case of sequential block LU factorization in the next chapter (section 4.5).

3.1.2 Fast variants for matrix multiplication

As numerical stability is not an issue over a finite field, asymptotically fast matrix multiplication algorithms, like Winograd's variant of Strassen algorithm [90] can be systematically used on top of the BLAS.

Table 3.1 shows the impact on sequential performance of fast variants compared to standard matrix multiplication. Here the fast variant corresponds to the Strassen-Winograd implementation of [17].

	1024	2048	4096	8192	16384
sgemm OpenBLAS	**27.30**	**28.16**	28.80	29.01	29.17
$O(n^3)$-fgemm Mod 37	21.90	24.93	26.93	28.10	28.62
$O(n^{2.81})$-fgemm Mod 37	22.32	27.40	**32.32**	**37.75**	**43.66**
dgemm OpenBLAS	15.31	16.01	16.27	16.36	16.40
$O(n^3)$-fgemm Mod 131071	15.69	16.20	16.40	16.43	16.47
$O(n^{2.81})$-fgemm Mod 131071	**16.17**	**18.05**	**20.28**	**22.87**	**25.81**

Table 3.1: Effective Gfops ($2n^3/time/10^9$) of matrix multiplications: fgemm vs OpenBLAS d/sgemm on one core of the HPAC machine

In this table we compare the sequential speed obtained by the classical `fgemm` algorithm of the `FFLAS-FFPACK` library. The efficiency of the `fgemm` routine rely on the efficiency of the BLAS. We compile our codes linking with OpenBLAS. In table 3.1 computations are done over a small finite field, a large finite field and without modular reductions, to see the impact of modular reductions compared to openBLAS dgemm sequential execution.

We can also enable or disable the option to allow the use of fast variants in `fgemm` Mod 37 and `fgemm` Mod 131071. The difference between the small and large moduli is that the routine realizes that for a small modulus, it can use floats instead of doubles: thus over small moduli (here modulo 37), field elements are stored in single precision floating point numbers. Table 3.1 shows that in both cases, single or double precision, a speed-up of more than 40% can easily be attained when using fast variants.

We of course can also benefit from Strassen-Winograd algorithms in parallel versions of matrix multiplication. In practice, we will for now restrict ourselves to a parallel cutting of blocks that uses the classical algorithm, but when it degenerates to a sequential call, then it can use Strassen-Winograd variants. In the following, we thus mainly study the trade-off between having fine grain parallelization for load and communication balancing and the best size of blocks suited for the fast variants.

3.1.3 The impact of the grain size

The granularity is the block dimension (or the dimension of the smallest blocks in recursive splittings). Matrices with dimensions below this threshold are treated by a base-case variant. It is an important parameter for optimizing parallel efficiency: a finer grain allows more flexibility in the scheduling when running on numerous cores, but it also challenges the efficiency of the scheduler and can increase the bus traffic.

Over a finite field, we showed that using fast variants improves greatly the computation performance for matrix multiplication, when called on sufficiently large blocks. With fixed number of resources, rather than fixing a small grain, we fix the number of threads to be executed i.e. having fixed cutting for iterative variants and fixed number of recursion for recursive variants. If the matrix dimension gets larger, with fixed number of threads, the granularity is larger. Calling fast variants on these large blocks gives even better performance than considering small granularity and counting on an optimized runtime system that executes more tasks efficiently.

3.1.4 The impact of the runtime system and dataflow parallelism

Generating a large number of tasks causes overheads that severely impacts parallel execution, if the runtime does not handle it efficiently. This penalizes the use of fine-grain parallelization. Based on the xKaapi library, the `libkomp` runtime [18] system comes with fast task creation and small scheduling overheads and implements recursive tasks in a very efficient way. In table 3.2 we show the overhead of using `libkomp` and `libgomp` runtime systems on one core compared to a sequential execution of a block algorithm. We use for this comparison the best recursive algorithm for matrix multiplication, the `TWO_D_ADAPT` recursive variant, that is presented in §2.3.4.2 , with eight recursive calls. But even if we use optimized runtime systems for OpenMP tasks, such as `libkomp`, the cost of creating tasks should not be neglected. Indeed, the performance results using the `libkomp` runtime system on 1 core are nonetheless slower than the sequential execution as shown by the fourth column of table 3.2.

matrix dimension	block sequential	1 core `libgomp`	1 core `libkomp`
2000	**13.87**	13.58	13.67
4000	**15.10**	14.63	14.68
6000	**15.50**	15.44	15.47

Table 3.2: Execution speed(Gfops) on 1 core: overhead of using runtime systems on block algorithms (using 128 tasks).

Using the version 4.9 of gcc compiler we can also benefit from the dataflow dependencies model implemented in OpenMP-4.0. In our experiments we use the `depend` clause of OpenMP-4.0 to express dependencies between data produced and/or consumed by tasks which makes it possible to construct the DAG (directed acyclic graph) of dependencies between tasks before execution. This feature helps to reduce the idle time of resources by removing unnecessary synchronizations. We will see in the next sections

the impact of dataflow parallelization using the `libkomp` runtime that also implements the latest norm of OpenMP-4.0.

3.1.5 Distance-aware mapping policy optimization on NUMA architecture

The efficiency of computations on a NUMA machine architecture can be disrupted due to remote accesses between different NUMA nodes as explained in § 1.3.3. This led us to focus on data placement strategies to reduce as much as possible distant memory accesses.

In our experiments data are allocated, initialized and then computed. Recall that the mapping of data to a specific node is only determined at their initialization. Hence in order to experiment with different mapping strategies, it suffices to choose how the initialization phase is done. Data is initialized with two for loops. Each iteration is incremented with a fixed chunk size. We show in Listing 3.1 the algorithm implemented using the PALADIn language to dispatch chunks on available processors in a round and robin manner. Hence, a `FORBLOCK2D` is used to set the cutting strategy of the nested for loops and the initialization of each block of the matrix is done inside a PALADIn task.

Listing 3.1: Parallel data mapping on NUMA architecture

```
1
2   BS=std::max(BS, (size_t)Protected::WinogradThreshold(F) );
3   SYNCH_GROUP(
4         FORBLOCK2D(iter, m, n, SPLITTER(BS, Block, Grain),
5                     TASK(MODE(CONSTREFERENCE(F)),
6                          fzero(F,
7                                iter.iend()-iter.ibegin(),
8                                iter.jend()-iter.jbegin(),
9                                C+iter.ibegin()*n+iter.jbegin(), n)
                                    ;
10                         );
11                  );
12          );
```

To see the impact of remote accesses, we conducted experiments on the matrix-matrix multiplication operation with different mapping strategies of matrices A, B and C. First we map all the data on a single NUMA node, and execute the program on all processors. Then we conduct the same experiment by mapping on two, three and then all four NUMA nodes while the execution is still launched on all processors.

For the sake of clarity and simplicity we show only the different mapping strategies for one variant of matrix multiplication `TWO_D_ADAPT`, with four levels of recursion, in Table 3.3. By placing data on a single node, computation speed is affected by the time data are accessed from distant NUMA nodes whereas by dispatching all matrices on different NUMA nodes execution time is faster. Moreover, when 32 threads try to access data on the same NUMA node, contentions on data access also degrade execution performance.

matrix dimension	1 node	2 nodes	3 nodes	all nodes	numactl -i all
4000	233.99	275.97	291.18	**307.68**	295.60
6000	247.10	303.44	329.05	**347.21**	310.119
8000	265.66	292.02	342.85	**350.72**	310.147

Table 3.3: Execution speed(Gfops): with different data mapping.

The NUMA control policy (numactl) runs processes with a specific NUMA scheduling or memory placement policy. Used with "–interleave" policy setting, memory will be allocated using round robin on specified nodes. In Table 3.3 the policy is set on all nodes to compare with our mapping strategy. The overall performance of our mapping policy is better. This is explained by the size of the mapped chunks. Using the numactl policy data are mapped by page in each processor. Our mapping policy has the same strategy but maps larger blocks on each core. This means that data contiguous chunks in memory are larger using our mapping strategy.

3.2 Parallel matrix multiplication over finite fields

In this section we explore various variants of classical and fast algorithms and combine them together to pick out the best in terms of parallel speed-up. We thus switch between parallel and sequential routines for cubic and sub-cubic algorithms. We consider that a parallel version of matrix multiplication can be performed within two levels:

- a first level of parallelism where we call a parallel variant (Classical or fast variant),
- then a second level where the parallel algorithm switches on smaller blocks to sequential variant or another parallel variant.

With this assumption, we refine our search to only 6 different variants.

level 1	level 2
parallel classical variant	sequential fast sequential classical parallel fast
parallel fast variant	sequential classical sequential fast parallel classical

3.2.1 Classical parallel algorithms

In this section we study the classical parallel implementations as the first level of parallelism. The second level of parallelism is considered as a black box. This allows us to focus on the cutting strategies of the classical parallel algorithms regardless of what algorithm is used as a kernel routine.

Here we recall the classical algorithms for the computation of matrix multiplication presented in Chapter 2. These algorithms are divided in two categories: iterative

algorithms and recursive algorithms. Recursive algorithms are known to be more adaptive but could imply a harder control on threads/tasks binding in a parallel context. Whereas iterative algorithms have static cutting of tasks, thus one can have a better control on threads or task stack scheduling.

In the case of classical parallel block matrix multiplication, we explore a variety of well known 1D, 2D and 3D cutting strategies, with their iterative and recursive variants. The parallel classical MM implemented routines perform the same operation as the sequential `fgemm` routine implemented in the `FFLAS-FFPACK` library which is: $C \leftarrow A \times B$, where A, B and C are dense matrices with dimensions respectively (m, k), (k, n) and (m, n).

3.2.1.1 Iterative algorithms

In the iterative scheme, the matrices are split over the dimension m and/or n. We recall that the 1D cutting is when the splitting occur over one dimension (m or n), and the 2D cutting when both dimensions are split.

Figure 3.1 summarizes the 1D and the 2D cutting strategies. In the case of the 1D cutting, only one dimension is split into p blocks, where p is the number of threads. In figure 3.1 (left) we show only the 1D cutting over rows. The figure on the right shows the 2D cutting over the dimensions m and n. Here we cut the rows of A in s blocks and the columns of B in t blocks such that $s \times t$ equals p the number of available threads. Using these cutting strategies the number of blocks in the output matrix equals the number of processors. As mentioned in section 3.1.3, over a finite field, the grain size should be as large as possible to reduce modular reductions and to benefit from fast variants if used as kernel routines. However, using the PALADIn language, one can also set the desired grain size for his cutting strategy.

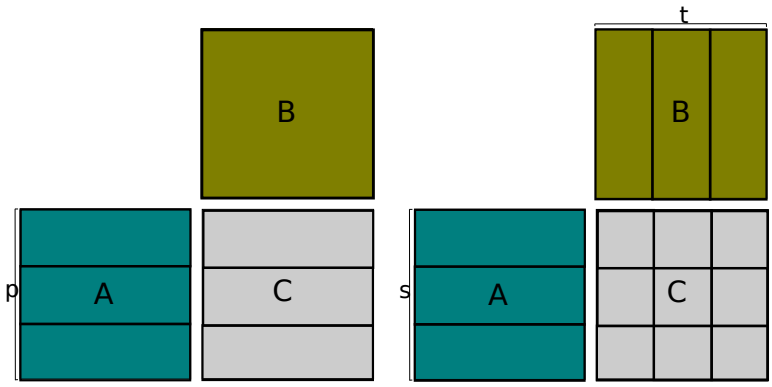

Figure 3.1: 1D and 2D iterative cutting

The 3D iterative cutting strategy splits over the three dimensions. Cutting over the dimension k creates more tasks but also more synchronizations. The 3D cutting scheme in figure 3.2 shows that cutting over the dimension k involves synchronizations between tasks. This scheme is thus similar to the 2D cutting strategy. One can execute these `fgemm` calls in parallel by introducing temporaries, but one needs to synchronize at the end to perform the fadd operation. This addition phase can be done in two

ways: adding all temporaries in a sequential way one after the other, or adding blocks two by two following a binary tree scheme. The second method is actually the recursive scheme of the 3D cutting strategy. It is detailed in § 3.2.1.2.

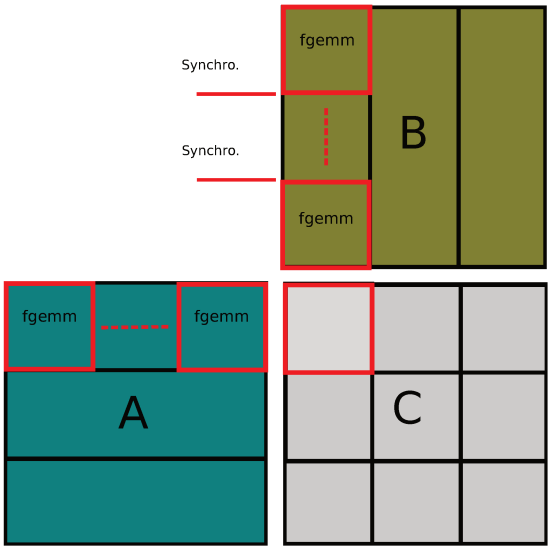

Figure 3.2: The 3D iterative cutting

3.2.1.2 Recursive algorithms

Recursive algorithms have the same cutting strategies as the iterative algorithms. We cut over one, two or three dimensions. However, we add an adaptive variant for the 2D and the 3D cutting strategies. These adaptive variants cut only the largest dimension at each recursion. Figures 3.3 and 3.4 illustrate these cuttings strategies.

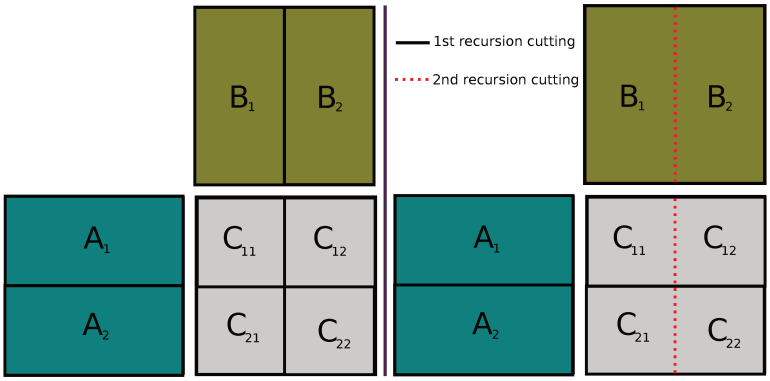

Figure 3.3: 2D recursive cutting

Fast variants that are efficient in practice and that can be composed with other routines are all recursive algorithms. More precisely, fast variants such as Strassen or Strassen-Winograd algorithms cut over three dimensions in each recursion and have a similar pattern as the 3D recursive cutting of the classical algorithm. This motivated

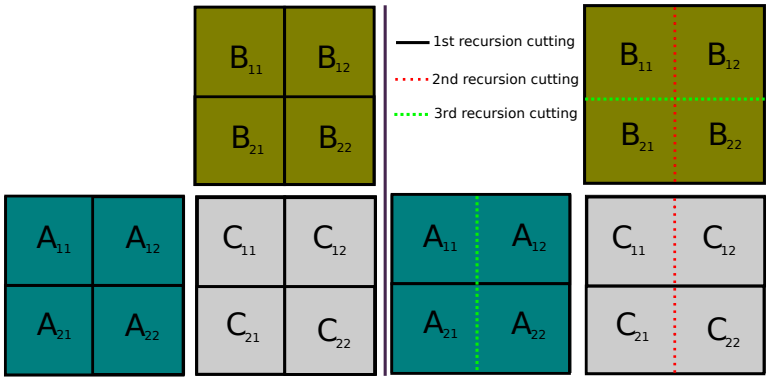

Figure 3.4: 3D recursive cutting

us to look into various 3D cutting strategies for the classical algorithm. This allows to better analyze the overhead of replacing one matrix multiplication call by several matrix additions in a parallel scheme. Three variants have been implemented in the 3D recursive cutting. Figure 3.4 summarizes the 3D cutting strategy. The cutting can occur over the three dimensions or over the maximum dimension at each level of the recursion. When the three dimensions are cut simultaneously at each recursion, 8 recursive `fgemm` calls are generated.

3.2.1.3 Performance of classical algorithms

In this section we show performance of parallel exact linear algebra routines using tasks with explicit synchronizations. All parallel implementations are done using the PALADIn language. We compare here execution speed of different cutting strategies for the matrix product operation using the implementation of OpenMP in the GNU gcc compiler (via the `libgomp` runtime library) and xKaapi (using `libkomp` runtime library).

In our experiments for classical algorithms we will focus on the design of a parallel matrix multiplication routine, based on Strassen's $O(n^{2.81})$ sequential algorithm. In order to parallelize the computation at the coarsest grain, our approach is to first apply a classical block algorithm generating a prescribed number of independent tasks, each of which will then use the sequential Strassen-Winograd algorithm.

Figures 3.6, 3.5 and 3.7, for sake of simplicity, show the behavior of the best 5 different cutting strategies for the parallel matrix multiplication operation. Experiments are conducted on square matrices with dimensions between 1000 and 15000 and elements are over the finite field $\mathbb{Z}/131071\mathbb{Z}$, using 32 cores.

With the `libgomp` runtime, the iterative strategy using `SPLITTER(`nt`, BLOCK, THREADS)` cutting is much faster, as recursive tasks seem to be poorly handled. Thanks to its efficient management of recursive tasks, the `libkomp` runtime behaves better for the recursive variants. Using TBB tasks, in figure 3.6, all cutting strategies have almost the same behavior when matrix dimensions gets bigger. On smaller dimensions the 2D recursive variants have the best behavior using TBB or xKaapi (libkomp).

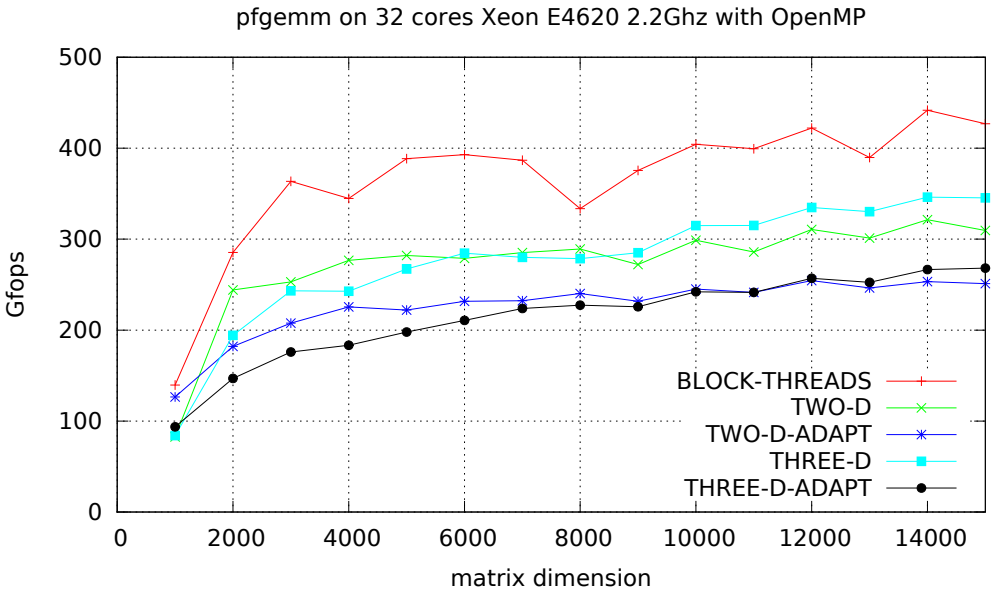

Figure 3.5: Speed of different matrix multiplication cutting strategies using OpenMP tasks

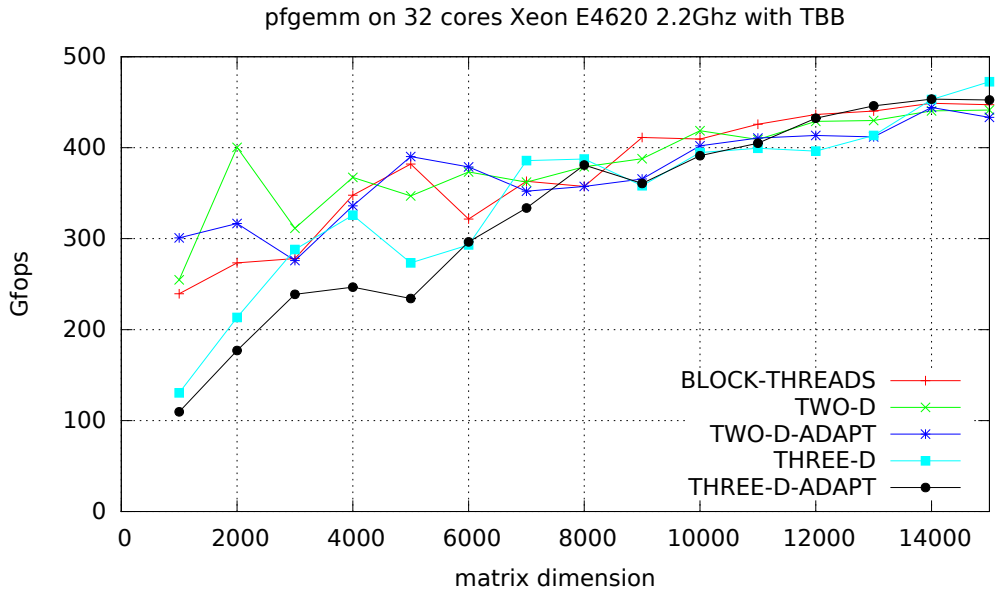

Figure 3.6: Speed of different matrix multiplication cutting strategies using TBB tasks (run on the HPAC machine)

3.2.2 Parallel fast variants

We focus now on the parallelization of the Strassen-Winograd algorithm for the matrix multiplication operation. We will first describe the sequential scheme of the Strassen-Winograd algorithm [17] implemented in the `FFLAS-FFPACK` library. Then, we will detail the parallel variant using the PALADIn language. We thus focus at the first

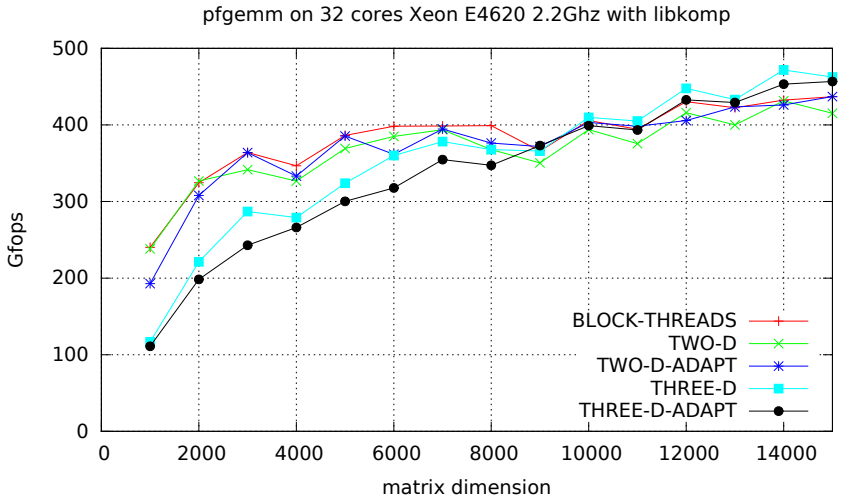

Figure 3.7: Speed of different matrix multiplication cutting strategies using xKaapi tasks

level of parallelism in this case regardless of what are the base case routines used. In § 3.2.2.2 we show the performance of this variant for each chosen base case routine.

3.2.2.1 Parallelization of the Strassen-Winograd algorithm

We first review the Strassen-Winograd algorithm. We consider the following recursive block operation:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

This algorithm performs the following 7 multiplications and 15 additions. We use the same notations as in [17].

- A series of 8 preliminary additions:
  $S_1 \leftarrow A_{21} + A_{22}$; $S_2 \leftarrow S_1 - A_{11}$
  $S_3 \leftarrow A_{11} + A_{21}$; $S_4 \leftarrow A_{12} + S_2$
  $T_1 \leftarrow B_{12} + B_{11}$; $T_2 \leftarrow B_{22} + T_1$
  $T_3 \leftarrow B_{22} + B_{12}$; $T_4 \leftarrow T_2 + B_{21}$.

- The 7 recursive multiplications:
  $P_1 \leftarrow A_{11} \times B_{11}$; $P_2 \leftarrow A_{12} \times B_{21}$
  $P_3 \leftarrow S_4 \times B_{22}$; $P_4 \leftarrow A_{22} \times T_4$
  $P_5 \leftarrow S_1 \times T_1$; $P_6 \leftarrow S_2 \times T_2$
  $P_7 \leftarrow S_3 \times T_3$.

- The 7 final additions:
  $U_1 \leftarrow P_1 + P_2$; $U_2 \leftarrow P_1 - P_6$
  $U_3 \leftarrow U_2 + P_7$; $U_4 \leftarrow U_2 + P_5$
  $U_5 \leftarrow U_4 + P_3$; $U_6 \leftarrow U_3 - P_4$
  $U_7 \leftarrow U_3 + P_5$.

The output matrix C is constructed with blocks U_1, U_5, U_6 and U_7 as follows:
$C = \begin{bmatrix} U_1 & U_5 \\ U_6 & U_7 \end{bmatrix}$

The schedule of [17] used for the Strassen-Winograd sequential algorithm aims to reduce memory allocation by allocating only two temporaries. However, one needs to use more temporaries to allow the execution of the 7 multiplications in parallel. All 8 pre-additions are performed in 8 temporary blocks that are required for the computation of the 7 multiplications. Then to reduce the number of additional temporary blocks we write 4 of the multiplication in the 4 blocks of the output matrix C and thus require only 3 additional temporary blocks to perform the overall 7 multiplications in parallel. We thus propose an implementation with 11 temporaries that manages to execute the 7 multiplications in parallel once the 8 additions are computed for the Strassen-Winograd algorithm. We call it the `WinoPar` variant of the pfgemm routine.

3.2.2.2 Performance of fast variants

As the `WinoPar` variant is set at the first level of parallelism, we can switch between three base case algorithms that are: the classical sequential, the fast sequential or the classical parallel base case algorithms.

Let p be the number of threads given to the `WinoPar` algorithm. The 7 multiplications are run in parallel that is each of the recursive call executed on $nt = \frac{p}{7^l}$ threads concurrently where l is the current level of the recursion.

Thus, with a fixed number of threads, the number of recursive level in the `WinoPar` implementation differs depending on the base case algorithm used. If the base case algorithm used is the classical parallel matrix multiplication, l becomes smaller. This allows to have a sufficient number of threads for the execution of the parallel base case routine used.

Figure 3.8 shows the computation speed of our best parallel implementations on all the 32 cores of the HPAC machine. In our experiments we compute experimentally the number of recursive levels for the `WinoPar` algorithm depending on the matrix dimensions and on the base case algorithms used:

- In figure 3.8 the number of recusive levels l for the `WinoPar`→`ClassicSeq` variant is set manually in order to have the maximum performance for this variant for large matrix dimensions. It is set as follows:
  - For matrix dimensions between 1000 and 4000, $l = 1$.
  - For matrix dimensions between 5000 and 8000, $l = 2$.
  - For matrix dimensions between 9000 and 16000, $l = 3$.
  - For matrix dimensions between 16000 and 32000, $l = 4$.
- In the case of the `WinoPar`→`ClassicPar` variant, experiments shows that our implementation cannot achieve better performance than the `WinoPar`→`ClassicSeq` variant. In figure 3.8, the `WinoPar`→`ClassicPar` variant switches at the base case level to the `TWO_D_ADAPT` variant of the `pfgemm` routine. We thus give here a smaller number of recursive levels l to the `WinoPar` algorithm but with a sufficient

number nt of threads that execute the `TWO_D_ADAPT` base case calls concurrently. These parameters are set as follows:

- For matrix dimensions between 1000 and 16000, $l = 1$ and at the base case level $nt = 32$. This allows to execute the base case `pfgemm` routine on all available processors. Figure 3.8 shows that the `WinoPar`→`ClassicSeq` routine (red curve) starts to have better performance than the pfgemm (`ClasssicPar`→`WinoSeq`) routine (blue curve) when matrix dimension gets larger than 9000. Therefore we increment the number of recursive levels l only when $9000 \leq \frac{m}{l+1}$, with m the matrix dimension. This gives $l = 2$ only when matrix dimension gets larger than 18000.
- For matrix dimensions between 18000 and 32000, $l = 2$ and $nt = 32$.

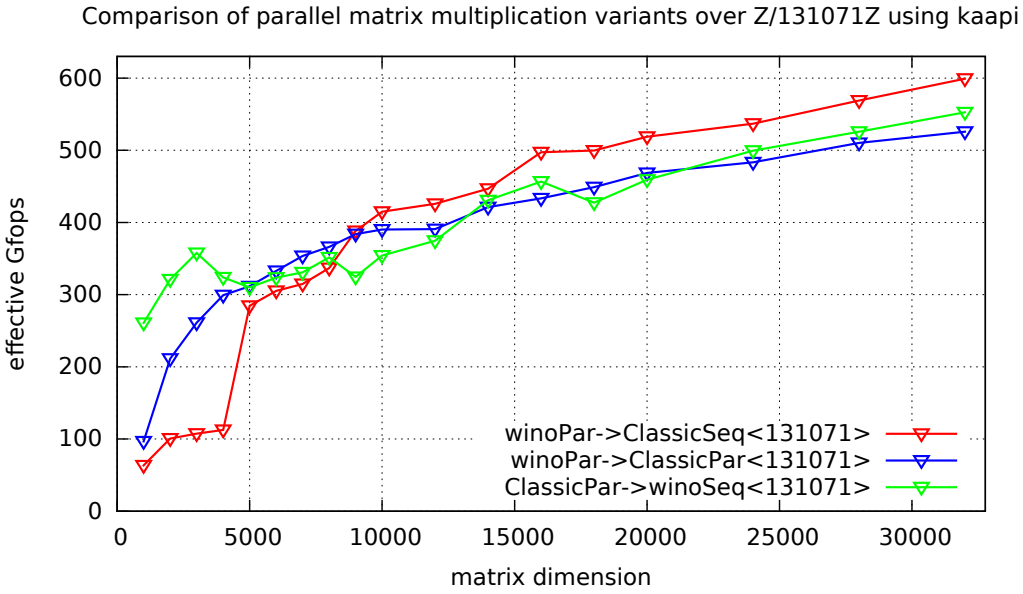

Figure 3.8: Speed of fast matrix multiplication variants (run on the HPAC machine)

Ideally, the number of recursive levels l should be computed automatically during the configuration of the library to be able to pick the best performance depending on the machine hierarchy and on the number of available processors. Thus this figure 3.8 demonstrates first that our parallel implementation `WinoPar` of the fast variant of the Strassen-Winograd algorithm has the best performance for larger matrix dimensions on the HPAC machine. Second, our implementation using the classical parallel matrix multiplication that switches to Winograd's algorithm has the best performance on smaller dimensions.

Figure 3.9 shows the speed-up of our best parallel implementation of matrix multiplication which is the `WinoPar`→`ClassicSeq` variant for matrix dimension 24000 and 32000. This figure demonstrates that we can attain a speed-up of 23 on the HPAC machine. We compare with our best sequential implementation of the `fgemm` routine

that attained 26.7531 Gfops on one processor. Our parallel fast variant speed, executed on 1 thread, is 23.4619 for matrix dimension 32000.

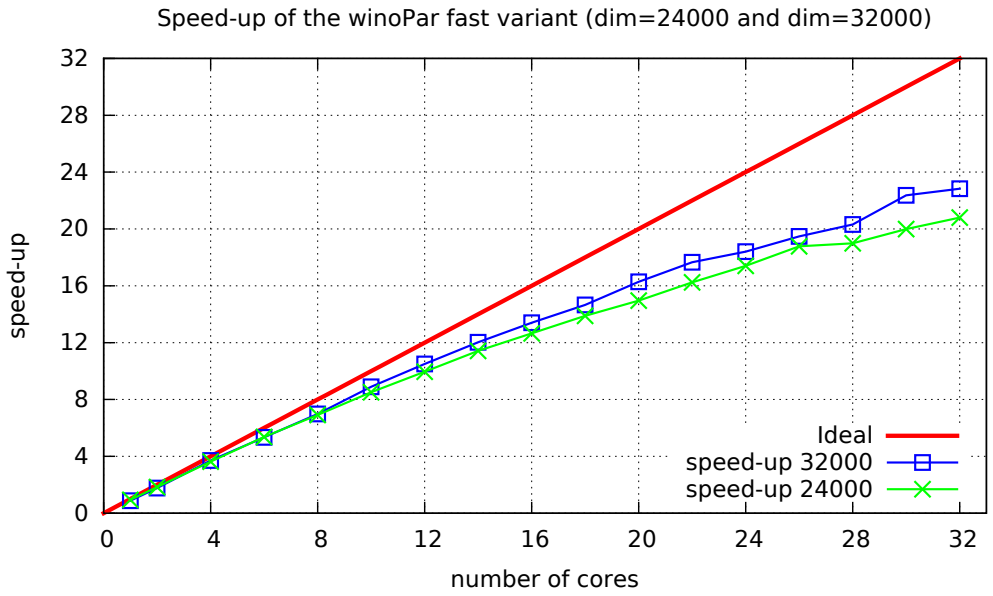

Figure 3.9: Speed-up of our best parallel fast matrix multiplication (WinoPar) (run on the HPAC machine)

3.2.3 Comparison with state of the art in parallel numerical linear algebra

We conducted experiments comparing with the state of the art numerical libraries: Intel MKL, OpenBLAS, Plasma Quark and the Strassen implementation of [6] that we called `BensonBallard`. The latter is a recursive implementation that cuts by halves matrix dimensions at each recursion.

Figure 3.10 compares our parallel implementations with these state of the art numerical libraries. This figure demonstrates that our `WinoPar`→`ClassicSeq` variant achieves the best performance for large matrix dimensions. This variant, for the matrix dimension 32000, is 30% faster than the MKL dgemm routine and the parallel OpenBLAS, 45% faster than the PLASMA-QUARK dgemm routine and 53% faster than the `BensonBallard` routine. Compared to all other variants, figure 3.10 shows that our `WinoPar`→`ClassicSeq` and `ClasssicPar`→`WinoSeq` variants are asymptotically faster on large dimensions thanks to the use of the sub-cubic fast algorithms. Indeed, the `ClasssicPar`→`WinoSeq` variant that uses the `TWO_D_ADAPT` recursive implementation switches to the sequential Winograd as a base case algorithm if the base case block size is bigger than a fixed threshold. However, on small dimensions, between 1000 and 3000, the base case block size is smaller than the Winograd threshold and therefore the classical algorithm for matrix multiplication is used on leafs on the recursion tree. Thus, at these dimensions, the `ClasssicPar`→`WinoSeq` have the same performance as the MKL dgemm since we rely on the same version of the sequential

OpenBLAS. For the matrix dimension 4000, the base case Winograd algorithm is triggered on blocks with dimensions 500×1000. This means that the configuration of the sequential Winograd threshold is not suited for parallel variants. This explains the drop in performance of the `ClasssicPar→WinoSeq` compared to the MKL dgemm for small dimensions higher than 4000. The parallel OpenBLAS dgemm that uses pthreads has almost the same behavior as the MKL dgemm on large matrix dimensions. However, on small dimensions, with no data mapping strategy the OpenBLAS dgemm performance are slower. The `BensonBallard` variant has almost the same speed as the `WinoPar→ClassicSeq` for dimensions between 1000 and 4000 where we set only one level of recursion for the Winograd parallel algorithm. Thus, it is possible that in the Strassen implementation of the `BensonBallard` variant the number of recursion levels is set to 1 for all dimensions. This explains the low performance of this variant.

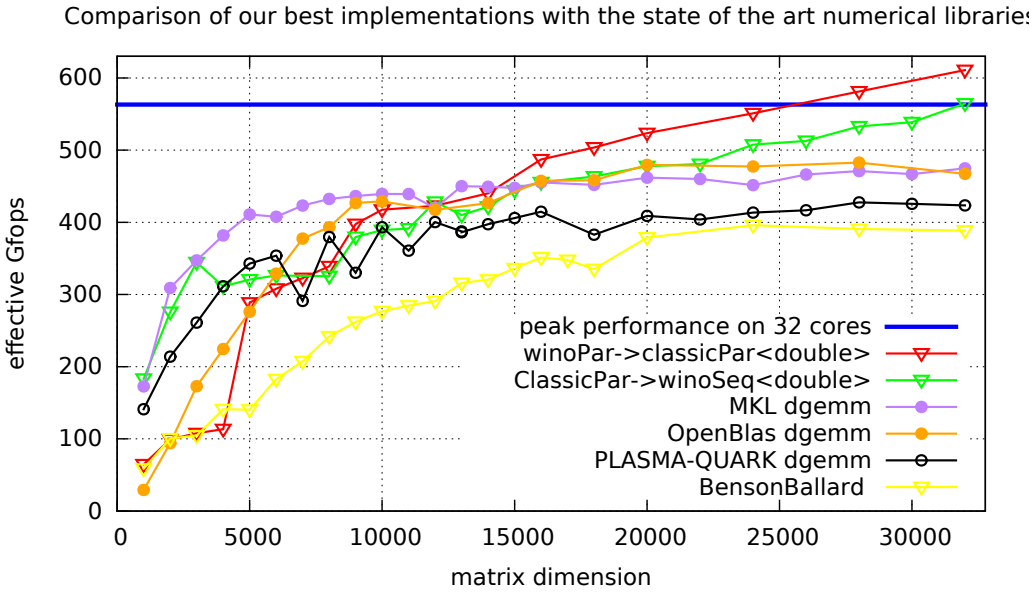

Figure 3.10: Comparison with state of the art numerical libraries (run on the HPAC machine)

3.3 Parallel triangular solving matrix: TRSM

In this section we study various cutting strategies for the computation of the `ftrsm` routine in parallel. We call this parallel routine the `pftrsm` routine. We identify three different types of parallelization: the block iterative, the block recursive and a hybrid variant combining both. The latter proves to deliver the best efficiency in practice, in particular when the unknown rectangular matrix is very skinny. We will consider here, without loss of generality, the lower left case of the `ftrsm` operation: computing $X \leftarrow L^{-1}B$.

3.3.1 Iterative variant

In the iterative variant (Algorithm 1), the parallelization is obtained by splitting the outer dimension of the matrices X and B:

$$\left[\ X_1 \ \middle|\ \dots \ \middle|\ X_k\ \right] \leftarrow L^{-1} \left[\ B_1 \ \middle|\ \dots \ \middle|\ B_k\ \right].$$

The computation of each $X_i \leftarrow L^{-1}B_i$ is independent from the others. Hence the algorithm consists in a length k parallel iteration creating k sequential `ftrsm` tasks. The cost of these sequential `ftrsm` is not associative, and one needs to maximize the computational size of each of these tasks. Hence the number of blocks k is set as the number of available threads.

Algorithm 1 Iterative `pftrsm`

Split $\left[\ X_1 \ \middle|\ \dots \ \middle|\ X_k\ \right] = L^{-1} \left[\ B_1 \ \middle|\ \dots \ \middle|\ B_k\ \right]$
for $i = 1 \dots k$ **do**
  $X_i \leftarrow L^{-1}B_i$
end for

3.3.2 Recursive variant

This variant is simply based on the block recursive algorithm (Algorithm 2) where each matrix multiplication is performed by the parallel matrix multiplication `pfgemm` of section 3.2.1. The three tasks in Algorithm 2 can not be executed concurrently.

Algorithm 2 Recursive `pftrsm`

Split $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} L_1 & \\ L_2 & L_3 \end{bmatrix}^{-1} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
$X_1 \leftarrow L_1^{-1}B_1$
$X_2 \leftarrow B_2 - L_2X_1$
$X_2 \leftarrow L_3^{-1}BX_2$

3.3.3 Hybrid variant

Lastly, we propose to combine the two above variants into a hybrid algorithm. The motivation is to handle the case when the column dimension of B is rather small: the cutting of Algorithm 1 produces slices that may become too thin, and reduce the efficiency of each of the sequential TRSM. Instead, the hybrid variant applies the iterative algorithm with the restriction that the column dimension of the slices X_i and B_i remains above a given threshold. Consequently, this splitting may create fewer tasks than the number of available threads. Each of them then runs the parallel recursive variant using an equal part of the unused remaining threads. More precisely, the parameters are set so that the number of threads given to the recursive variant, and henceforth to the matrix multiplications, is always a power of 2, in order to better benefit from the adaptive recursive splitting.

Let T be the threshold, p the number of threads provided and n, the column dimension of B. Let $\ell = \min\{\ell \in \mathbb{Z}_{\geq 0} : \frac{p}{2^\ell}T < n\}$. Then each recursive TRSM task is allocated 2^ℓ threads and the iterative TRSM splits X and B in $k = p/2^\ell$ slices.

3.3.4 Experiments on parallel `ftrsm`

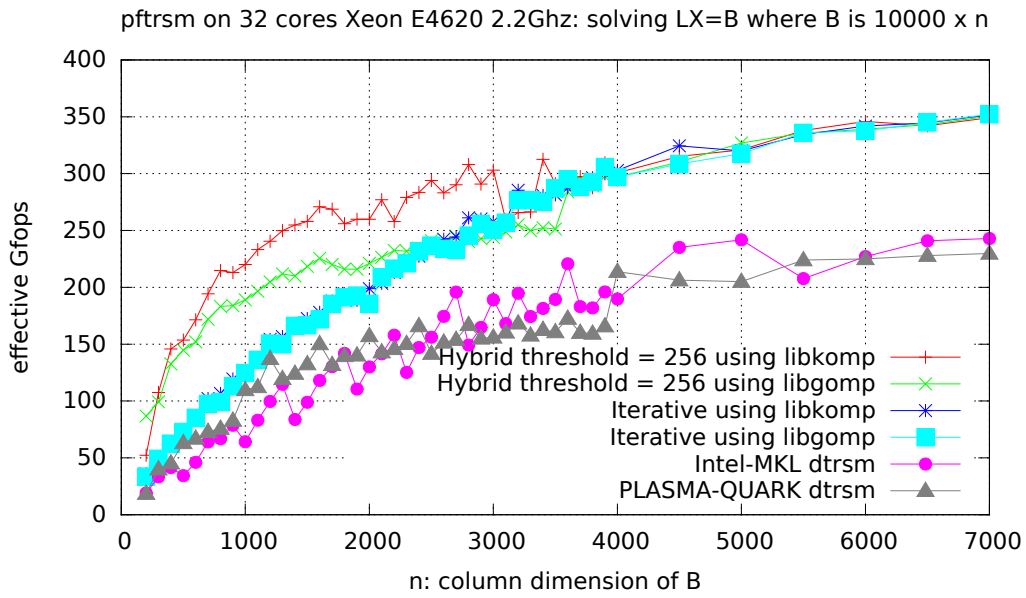

Figure 3.11: Comparing the Iterative and the Hybrid variants for parallel `ftrsm` using `libkomp` and `libgomp`. Only the outer dimension varies: B and X are $10000 \times n$.

Figure 3.11 compares the computation speed of the iterative and the hybrid version of the parallel `ftrsm`, on triangular systems of dimension 10000 but with right hand side of varying column dimension. The hybrid variant clearly improves over the iterative variant up to $n = 3000$. Moreover, the `libkomp` and `libgomp` runtimes perform similarly on the iterative algorithm (which is essentially a parallel for loop), but for the hybrid variant, `libkomp` reaches a higher efficiency because it handles more efficiently recursive tasks. Lastly, the performances of the numerical `dtrsm` routines of `Intel-MKL` and `Plasma-Quark` are shown, and appear to be consistently slower.

3.4 Conclusion

In this chapter we studied the implementation of efficient parallel block kernels in exact linear algebra. At first we focused on the matrix multiplication operation and showed a variety of algorithms that switch between classical/fast variants and parallel/sequential implementations. We picked the best parallel variant of our library which is an efficient parallel implementation of the Strassen-Winograd algorithm that switches to classical algorithm at the base case. The performance of this algorithm surpasses the performance of state of the art numerical libraries. Then, we studied the parallel implementation of triangular solving matrix routines. We presented iterative, recursive and hybrid block variants. We proved that the iterative parallel algorithm has a better speed-up then the recursive parallel algorithm that adds more synchronizations in each recursion.

In the next chapters we will focus on the Gaussian elimination algorithm in sequential and in parallel. Chapter 4 focuses on the design of block Gaussian elimination where modular reductions are involved, presents a new state of the art recursive Gaussian elimination algorithm and gives a panorama of existing block algorithms. Using the latter algorithms chapter 5 introduces a new matrix invariant and shows how to compute it. The parallel performance of those Gaussian elimination algorithms are studied in chapter 6 where we show that their parallel speed-up depends greatly on the parallelization of the kernel routines `pfgemm` and `pftrsm`.

Chapter 4

Exact Gaussian elimination

The design of efficient dense linear algebra routines, as mentioned in section 1.2, is done by using block algorithms that gather arithmetic operations in matrix-matrix multiplications. In chapter 3 we showed that exact linear algebra routines involve modular reduction operations that need to be delayed. These aspects, together with the use of fast variants for matrix multiplication, imposes having coarse grain parallelization. We thus need to take into account the blocking structure of Gaussian elimination algorithm variants by considering sufficiently large blocks. This condition should not compromise the parallelism of the block algorithm used. In this chapter we study the impact of using block algorithms for the computation of Gaussian elimination over a finite field, where modular reductions are involved in a delayed design. More precisely, we explore a panorama of block algorithms that takes into account the arising conditions in chapter 3. We study existing algorithms, present a new recursive algorithm for block Gaussian elimination and give the implementations of the most important of them. We then study the cost of modular reduction for these algorithms in terms of arithmetic complexity.

We exploit these algorithms for any rank matrices, i.e. invertible matrices or matrices having arbitrary rank profile. However, computing the rank profile or equivalently the echelon form of the matrix can have an impact on the execution behavior and the algorithm pattern. We will not study here properties related to the echelon form of the matrix. These aspects will be detailed in the next chapter, where the focus is on how to recover the rank profile or equivalently the echelon form of the matrix.

4.1 Gaussian elimination Algorithm block variants

Several schemes are used to design block linear algebra algorithms: the splitting can occur on one dimension only, producing row or column slabs [67], or both dimensions, producing tiles [21]. Note that, here, we denote by tiles a partition of the matrix into sub-matrices in the mathematical sense regardless what the underlying data storage is.

Algorithms processing blocks can be either iterative or recursive. Most numerical dense Gaussian elimination algorithms use iterative algorithms and block (slab or tile) iterative algorithms because it allows to have better control on data in memory. In [21] they use tile iterative block algorithms. In [28] the classic tile iterative algorithm

is combined with a slab recursive one for the panel elimination. In numerical linear algebra, the arithmetic cost function is associative: any choice of block decomposition yields the same amount of operations (the product $A[B_1 B_2]$ takes essentially the same time as the two products $A[B_1]$ and $A[B_2]$ done in sequence) [67]. The block iterative algorithms are preferred for their better control over the block sizes, used for cache aware algorithms.

In dense linear algebra over a finite field, the use of not only sub-cubic algorithms such as Strassen's [83], but also of delayed modular reductions make the cost no longer associative: the larger the matrix product, the more efficient the computation. Over exact domains, recursive algorithms are therefore preferred to benefit from fast matrix arithmetic.

Figure 4.1 summarizes some the four main block splittings obtained by combining these two aspects.

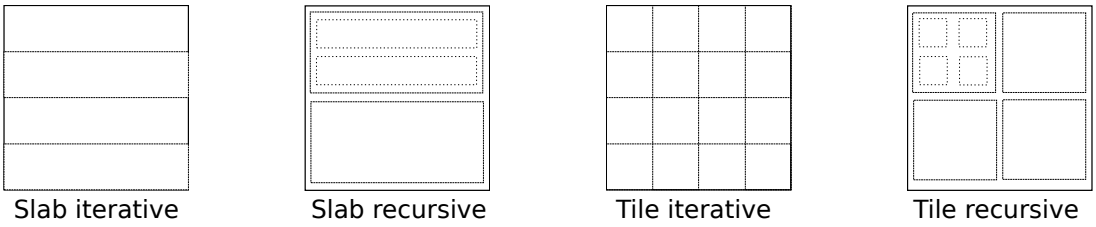

Figure 4.1: Main types of block splitting

4.2 Slab algorithms for PLUQ factorization

Slab algorithms are the most common block algorithms in exact linear algebra. This is because until quite recently it was the only way to design block algorithms that compute the echelon forms and the rank profiles. Algorithms computing the column echelon form (or equivalently the row rank profile) used to share a common pivoting strategy: to search for pivots in a row-major fashion and consider the next row only if no non-zero pivot was found (see [62] and references therein). In this section, we give the algorithm implementation of the slab recursive variant of [62] and then propose the corresponding slab iterative variant.

4.2.1 The slab recursive algorithm

Most algorithms computing rank profiles are slab recursive [65, 82, 58, 62]. We present here the CUP decomposition algorithm 3 of [62]. This slab recursive algorithm splits the initial matrix in halves over rows. It works in place allowing for instance the computation of the inverse of a matrix on the same storage as the input matrix.

Algorithm 3 Slab recursive CUP decomposition

Input: $A = (a_{ij})$ a $m \times n$ matrix over a field
Output: r: the rank of A
Output: P: $n \times n$ permutation matrix
Output: $A \leftarrow \begin{bmatrix} C\backslash U & V \\ M & 0 \end{bmatrix}$ where C is $r \times r$ unit lower triangular, U is $r \times r$ upper triangular, and

$$A = \begin{bmatrix} C \\ M \end{bmatrix} \begin{bmatrix} U & V \end{bmatrix} P.$$

if m=1 **then**
  if $A = \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}$ **then**
    $i \leftarrow$ the col. index of the first non zero entry of A
    $P \leftarrow T_{0,i}$ the transposition of indices 0 and i, $r \leftarrow 1$
    $A \leftarrow AP$
    for i from 2 to n **do** $A_{1,i} \leftarrow A_{1,i}A_{1,1}^{-1}$
    end for
    Return $(1, P, A)$
  end if
  Return $(0, I_n, A)$
end if
$k \leftarrow \lfloor \frac{m}{2} \rfloor$ ▷ Splitting $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$ where A_1 is $k \times n$.
Decompose $A_1 = C_1[U_1V_1]P_1$ ▷ $(r_1, P_1, A_1) \leftarrow CUP(A_1)$
if $r_1 = 0$ **then**
  Decompose $A_2 = C_2[U_2V_2]P_2$
  Return (r_2, P_2, A_2)
end if
$A_2 \leftarrow A_2P_1^T$ ▷ $[A'_{21}A'_{22}] \leftarrow [A_{21}A_{22}]P_1^T$
Here $A = \begin{bmatrix} C_1\backslash U_1 & V_1 \\ D_1 \ 0 & 0 \\ A'_{21} & A'_{22} \end{bmatrix}$ with $\begin{bmatrix} C_1\backslash U_1 \\ D_1 \ 0 \end{bmatrix}$ $k \times r_1$ and V_1 $r_1 \times (n - r_1)$
$G \leftarrow A'_{21}U_1^{-1}$ ▷ `ftrsm`(A'_{21}, U_1)
$H \leftarrow A'_{22} - GV_1$ ▷ `fgemm`(A'_{22}, G, V_1)
Decompose $H = C_2[U_2V_2]P_2$ ▷ $(r_2, P_2, H) \leftarrow CUP(H)$
$V'_1 \leftarrow V_1P_2^T$ ▷ PermC(V_1, P_2^T)
Here $A = \begin{bmatrix} C_1\backslash U_1 & V'_1 & \\ D_1 \ 0 & 0 & \\ G & C_2\backslash U_2 & V_2 \\ & D_2 \ 0 & 0 \end{bmatrix}$.
Move U_2, V_2 up next to V'_1 in A.
Here $A = \begin{bmatrix} C_1\backslash U_1 & V'_1 & \\ D_1 \ 0 & \backslash U_2 & V_2 \\ G & C_2\backslash 0 & 0 \\ & D_2 \, 0 & 0 \end{bmatrix}$.
$P \leftarrow Diag(I_{r_1}, P_2)P_1$
Return $(r_1 + r_2, P, A)$

The first recursive call is done on the upper slab followed by a series of updates. Then a second recursive call is made on the bottom slab. The matrix U is then reconstructed by performing block permutations as shown in figure 4.2.

This algorithm computes the column echelon form of the input matrix. It can be adapted to compute the row echelon form of the matrix by splitting into column slabs

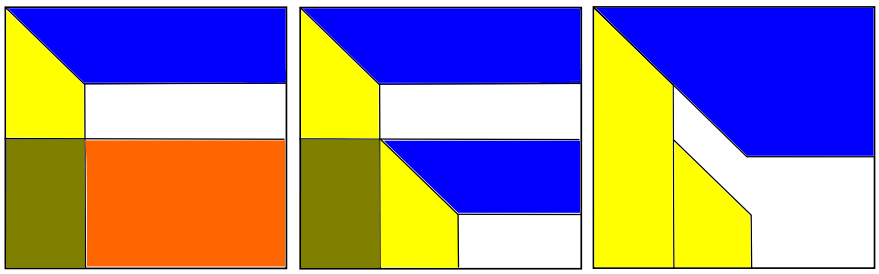

Figure 4.2: Slab recursive CUP decomposition and final block permutation.

and performing the pivot search in each column. This generates a `PLE` decomposition.

4.2.2 The slab iterative algorithm

The slab recursive algorithm can be translated into a slab iterative algorithm that splits the row dimension creating slabs as shown in in Figure 6.4. Elimination in each slab, called also panel factorization, is performed using a sequential algorithm. Then at each iteration, it follows the same scheme as the slab recursive algorithm using the same set of updates. Updates (using `ftrsm` and `fgemm` routines) can be done in slabs or in tiles as shown in Figure 6.4. Using tiling to perform the update phase at each iteration allows to fit more blocks into the cache memory and optimize the overall performance by reducing distant memory accesses.

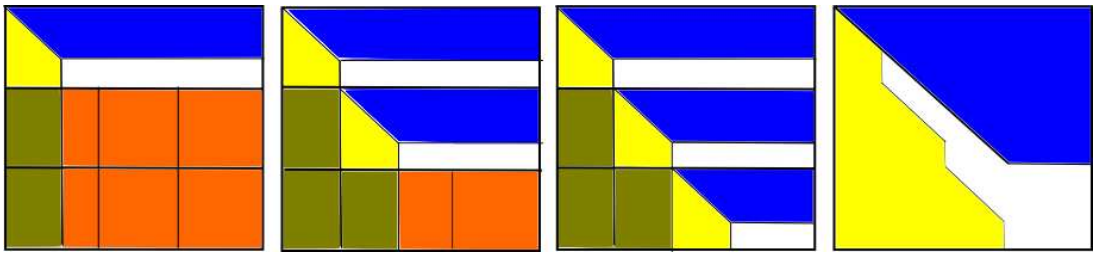

Figure 4.3: Slab iterative factorization of a matrix with rank deficiencies, with final reconstruction of the upper triangular factor

4.3 Tile algorithms for Gaussian elimination

Tile algorithms are more adapted to cache-like memory architectures and are preferred on slab algorithms. This helps reducing the dependency on the bus speed by reducing the number of distant memory loads into cache levels. We will first present our new tile recursive state of the art algorithm for computing a PLUQ decomposition. Then we present the tile iterative algorithm with its three variants: the right-looking, the crout and left-looking variants.

4.3.1 The Tile recursive algorithm

Our tile recursive PLUQ algorithm 4 is the first algorithm that has the same constant as the iterative algorithms for any rank. Moreover, compared to existing recursive algorithms such as the TURBO algorithm [42], our new algorithm is proved to have a rank sensitive complexity. The TURBO algorithm does not compute the lower triangular matrix L and performs five recursive calls. It therefore implies an arithmetic overhead compared to classic Gaussian elimination.

Our algorithm 4, computing a PLUQ decomposition, is based on a splitting of the matrix in four quadrants. A first recursive call is done on the upper left quadrant followed by a series of updates. Then two recursive calls can be made on the anti-diagonal quadrants if the first quadrant exposed some rank deficiency. After a last series of updates, a fourth recursive call is done on the bottom right quadrant.

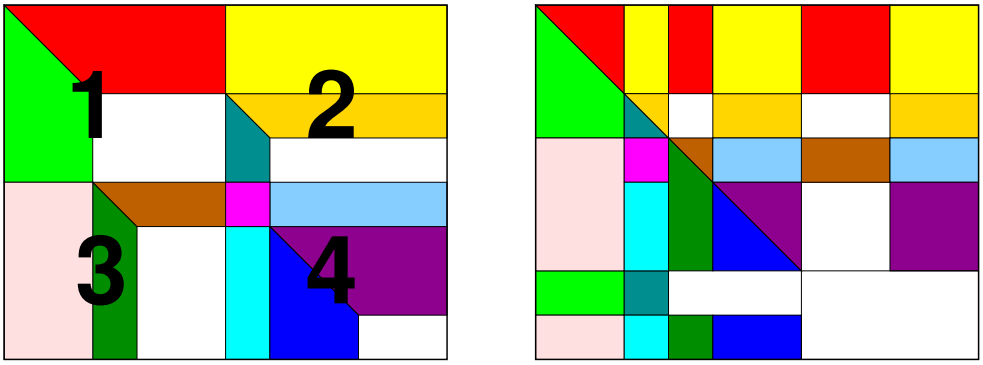

Figure 4.4: Tile recursive PLUQ decomposition and final block permutation.

Figure 4.4 illustrates the position of the blocks computed in the course of Algorithm 4, before and after the final permutation with matrices S and T.

This framework differs from the one in [42] by the order in which the quadrants are treated, leading to only four recursive calls in this case instead of five in [42]. The correctness of Algorithm 4 is proven in Appendix A.

Remark 1: Algorithm 4 is in-place (as defined in [62, Definition 1]): all operations of the `ftrsm`, `fgemm`, `PermC`, `PermR` subroutines work with $O(1)$ extra memory allocations except possibly in the course of fast matrix multiplications. The only constraint is for the computation of $J \leftarrow L_3^{-1} I$ which would overwrite the matrix I that should be kept for the final output. Hence a copy of I has to be stored for the computation of J. The matrix I has dimension $r_3 \times r_2$ and can be stored transposed in the zero block of the upper left quadrant (of dimension $(\frac{m}{2} - r_1) \times (\frac{n}{2} - r_1)$, as shown on Figure 4.4).

Algorithm 4 Tile recursive PLUQ decomposition

Input: $A = (a_{ij})$ a $m \times n$ matrix over a field
Output: P, Q: $m \times m$ and $n \times n$ permutation matrices
Output: r: the rank of A
Output: $A \leftarrow \begin{bmatrix} L\backslash U & V \\ M & 0 \end{bmatrix}$ where L is $r \times r$ unit lower triangular, U is $r \times r$ upper triangular, and

$$A = P \begin{bmatrix} L \\ M \end{bmatrix} \begin{bmatrix} U & V \end{bmatrix} Q.$$

if m=1 **then**
  if $A = \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}$ **then** $P \leftarrow \begin{bmatrix} 1 \end{bmatrix}, Q \leftarrow I_n, r \leftarrow 0$
  else
    $i \leftarrow$ the col. index of the first non zero elt. of A
    $P \leftarrow \begin{bmatrix} 1 \end{bmatrix}; Q \leftarrow T_{1,i}, r \leftarrow 1$
    Swap $a_{1,i}$ and $a_{1,1}$
  end if
  Return (P, Q, r, A)
end if
if n=1 **then**
  if $A = \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T$ **then** $P \leftarrow I_m; Q \leftarrow \begin{bmatrix} 1 \end{bmatrix}, r \leftarrow 0$
  else
    $i \leftarrow$ the row index of the first non zero elt. of A
    $P \leftarrow \begin{bmatrix} 1 \end{bmatrix}, Q \leftarrow T_{1,i}, r \leftarrow 1$
    Swap $a_{i,1}$ and $a_{1,1}$
    for j from $i+1$ to m **do** $a_{j,1} \leftarrow a_{j,1} a_{1,1}^{-1}$
    end for
  end if
  Return (P, Q, r, A)
end if

▷ Splitting $A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}$ where A_1 is $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$.

Decompose $A_1 = P_1 \begin{bmatrix} L_1 \\ M_1 \end{bmatrix} \begin{bmatrix} U_1 & V_1 \end{bmatrix} Q_1$ ▷ PLUQ(A_1)

$\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \leftarrow P_1^T A_2$ ▷ `PermR`(A_2, P_1^T)

$\begin{bmatrix} C_1 & C_2 \end{bmatrix} \leftarrow A_3 Q_1^T$ ▷ `PermC`(A_3, Q_1^T)

Here $A = \left[\begin{array}{cc|c} L_1\backslash U_1 & V_1 & B_1 \\ M_1 & 0 & B_2 \\ \hline C_1 & C_2 & A_4 \end{array}\right]$.

$D \leftarrow L_1^{-1} B_1$ ▷ `ftrsm`(L_1, B_1)
$E \leftarrow C_1 U_1^{-1}$ ▷ `ftrsm`(C_1, U_1)
$F \leftarrow B_2 - M_1 D$ ▷ `fgemm`(B_2, M_1, D)
$G \leftarrow C_2 - E V_1$ ▷ `fgemm`(C_2, E, V_1)
$H \leftarrow A_4 - E D$ ▷ `fgemm`(A_4, E, D)

Here $A = \left[\begin{array}{cc|c} L_1\backslash U_1 & V_1 & D \\ M_1 & 0 & F \\ \hline E & G & H \end{array}\right]$.

Decompose $F = P_2 \begin{bmatrix} L_2 \\ M_2 \end{bmatrix} \begin{bmatrix} U_2 & V_2 \end{bmatrix} Q_2$ ▷ PLUQ(F)

Decompose $G = P_3 \begin{bmatrix} L_3 \\ M_3 \end{bmatrix} \begin{bmatrix} U_3 & V_3 \end{bmatrix} Q_3$ ▷ PLUQ(G)

$\begin{bmatrix} H_1 & H_2 \\ H_3 & H_4 \end{bmatrix} \leftarrow P_3^T H Q_2^T$ ▷ `PermR`(H, P_3^T); `PermC`(H, Q_2^T)

$\begin{bmatrix} E_1 \\ E_2 \end{bmatrix} \leftarrow P_3^T E$ ▷ $\texttt{PermR}(E, P_3^T)$

$\begin{bmatrix} M_{11} \\ M_{12} \end{bmatrix} \leftarrow P_2^T M_1$ ▷ $\texttt{PermR}(M_1, P_2^T)$

$\begin{bmatrix} D_1 & D_2 \end{bmatrix} \leftarrow DQ_2^T$ ▷ $\texttt{PermR}(D, Q_2^T)$

$\begin{bmatrix} V_{11} & V_{12} \end{bmatrix} \leftarrow V_1 Q_3^T$ ▷ $\texttt{PermR}(V_1, Q_3^T)$

Here $A = \left[\begin{array}{ccc|cc} L_1\backslash U_1 & V_{11} & V_{12} & D_1 & D_2 \\ M_{11} & 0 & 0 & L_2\backslash U_2 & V_2 \\ M_{12} & 0 & 0 & M_2 & 0 \\ \hline E_1 & L_3\backslash U_3 & V_3 & H_1 & H_2 \\ E_2 & M_3 & 0 & H_3 & H_4 \end{array}\right]$.

$I \leftarrow H_1 U_2^{-1}$ ▷ $\texttt{ftrsm}(H_1, U_2)$

$J \leftarrow L_3^{-1} I$ ▷ $\texttt{ftrsm}(L_3, I)$

$K \leftarrow H_3 U_2^{-1}$ ▷ $\texttt{ftrsm}(H_3, U_2)$

$N \leftarrow L_3^{-1} H_2$ ▷ $\texttt{ftrsm}(L_3, H_2)$

$O \leftarrow N - JV_2$ ▷ $\texttt{fgemm}(N, J, V_2)$

$R \leftarrow H_4 - KV_2 - M_3 O$ ▷ $\texttt{fgemm}(H_4, K, V_2); \texttt{fgemm}(H_4, M_3, O)$

Decompose $R = P_4 \begin{bmatrix} L_4 \\ M_4 \end{bmatrix} \begin{bmatrix} U_4 & V_4 \end{bmatrix} Q_4$ ▷ $\text{PLUQ}(R)$

$\begin{bmatrix} E_{21} & M_{31} & 0 & K_1 \\ E_{22} & M_{32} & 0 & K_2 \end{bmatrix} \leftarrow P_4^T \begin{bmatrix} E_2 & M_3 & 0 & K \end{bmatrix}$ ▷ $\texttt{PermR}$

$\begin{bmatrix} D_{21} & D_{22} \\ V_{21} & V_{22} \\ 0 & 0 \\ O_1 & O_2 \end{bmatrix} \leftarrow \begin{bmatrix} D_2 \\ V_2 \\ 0 \\ O \end{bmatrix} Q_4^T$ ▷ $\texttt{PermC}$

Here $A = \left[\begin{array}{ccc|ccc} L_1\backslash U_1 & V_{11} & V_{12} & D_1 & D_{21} & D_{22} \\ M_{11} & 0 & 0 & L_2\backslash U_2 & V_{21} & V_{22} \\ M_{12} & 0 & 0 & M_2 & 0 & 0 \\ \hline E_1 & L_3\backslash U_3 & V_3 & I & O_1 & O_2 \\ E_{21} & M_{31} & 0 & K_1 & L_4\backslash U_4 & V_4 \\ E_{22} & M_{32} & 0 & K_2 & M_4 & 0 \end{array}\right]$.

$S \leftarrow \begin{bmatrix} I_{r_1+r_2} & & & \\ & I_{k-r_1-r_2} & & \\ & & I_{r_3+r_4} & \\ & & & I_{m-k-r_3-r_4} \end{bmatrix}$

$T \leftarrow \begin{bmatrix} I_{r_1} & & & & \\ & & I_{r_2} & & \\ & I_{r_3} & & & \\ & & & I_{r_4} & \\ & & I_{k-r_1-r_3} & & \\ & & & & I_{n-k-r_2-r_4} \end{bmatrix}$

$P \leftarrow \text{Diag}(P_1 \begin{bmatrix} I_{r_1} & \\ & P_2 \end{bmatrix}, P_3 \begin{bmatrix} I_{r_3} & \\ & P_4 \end{bmatrix}) S$

$Q \leftarrow T\text{Diag}(\begin{bmatrix} I_{r_1} & \\ & Q_3 \end{bmatrix} Q_1, \begin{bmatrix} I_{r_2} & \\ & Q_4 \end{bmatrix} Q_2)$

$A \leftarrow S^T A T^T$ ▷ $\texttt{PermR}(A, S^T); \texttt{PermC}(A, T^T)$

Here $A = \begin{bmatrix} L_1\backslash U_1 & D_1 & V_{11} & D_{21} & V_{12} & D_{22} \\ M_{11} & L_2\backslash U_2 & 0 & V_{21} & 0 & V_{22} \\ E_1 & I & L_3\backslash U_3 & O_1 & V_3 & O_2 \\ E_{21} & K_1 & M_{31} & L_4\backslash U_4 & 0 & V_4 \\ M_{12} & M_2 & 0 & 0 & 0 & 0 \\ E_{22} & K_2 & M_{32} & M_4 & 0 & 0 \end{bmatrix}$

Return $(P, Q, r_1 + r_2 + r_3 + r_4, A)$

4.3.2 The tile iterative variants

The tile iterative elimination consists in cutting each of the two dimensions of the matrix in t parts. This implies that potentially an elimination may happen in any of the t^2 tiles depending on the rank deficiencies discovered. Thus, the matrix multiplication updates generate a quartic number of `fgemm` calls most of which have dimension 0. Instead we will call tile iterative, an adaptation of the above slab iterative algorithm, where the panel factorization is performed in column tiles.

This splitting limits the search space on the column dimension to keep operations more local. We will see later under which condition it is still possible to compute echelon forms and rank profiles. Now with this splitting, the operations remain more local and updates can be parallelized. This approach shares similarities with the computation of the panel elimination described in [28]. Figure 6.5 illustrates this tile iterative factorization obtained by the combination of a row-slab iterative algorithm, and a column-slab iterative panel factorization.

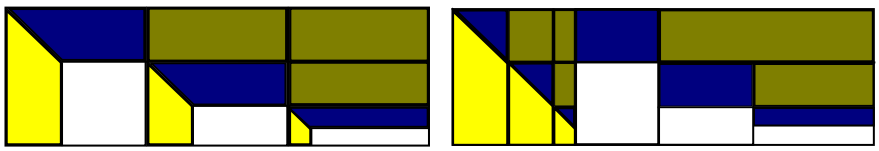

Figure 4.5: Panel PLUQ factorization: tiled sub-calls inside a single slab and final reconstruction

From now on, for the sake of simplicity and to focus on the problem of reducing the modular reduction count, we assume that no rank deficiency occurs. This means that in the sequential algorithm execution we will always find a invertible pivot during the elimination and in a block algorithm execution no rank deficiency occurs in any of the diagonal block. This hypothesis is satisfied by matrices with generic rank profile (i.e. having all their leading principal minor non zero).

We will present here different variants of iterative Gaussian elimination. More precisely, tile iterative algorithms range in three categories: the right-looking, left-looking and the Crout variant [27, §5.4]. They correspond to three ways of scheduling the block operations: the panel LU decomposition and the corresponding updates using triangular system solve (`ftrsm` and `futrsm`), and matrix products (`fgemm`). Table 4.1 sketches the different shapes of the associated routine calls in the main loop of each variant. We denote by:

- `futrsm` (k,k,n) is the operation that solves $X = A^{-1}B$ where A is a unit diagonal triangular $k \times k$ matrix and B is a dense $k \times n$ matrix. B can be on left or right hand side
- `ftrsm` (k,k,n) is the operation that solves $X = A^{-1}B$ where A is a triangular $k \times k$ matrix and B is a dense $k \times n$ matrix. B can be on left or right hand side
- `fgemm` (m,k,n) denotes the multiplication of an m×k matrix A by an k×n matrix B in C: $C \leftarrow \alpha A \times B + \beta C$

Left looking	Crout	Right looking

for i=1 to n/k **do**
`futrsm` ((i-1)k,(i-1)k,k)
`fgemm` (n-(i-1)k,(i-1)k,k)
`pluq` (k,k)
`ftrsm` (k,k,n-ik)
end for
	for i=1 to n/k **do**
	`fgemm` (n-(i-1)k,(i-1)k,k)
	`fgemm` (k,(i-1)k,n-ik)
	`pluq` (k,k)
	`futrsm` (k,k,n-ik)
	`ftrsm` (k,k,n-ik)
	end for
		for i=1 to n/k **do**
		`pluq` (k,k)
		`futrsm` (k,k,n-ik)
		`ftrsm` (k,k,n-ik)
		`fgemm` (n-ik,k,n-ik)
		end for

Table 4.1: Main loops of the Left looking, Crout and Right looking tile iterative block LU factorization, n and k are respectively matrix and block dimensions (see [27, Chapter 5])

The tile iterative right looking, left looking and Crout variants of LU decomposition consist on cutting the input matrix into tiles of fixed size. In the following, we present in details the block scheme of each variant. We will then detail the computation of the number of modular reductions of these variants in the section 4.5.

4.3.2.1 The tile iterative Right Looking variant

The scheme of the tile iterative right looking variant is very similar to the numerical tile iterative versions of the state of the art numerical libraries such as PLASMA-Quark library [77]. In the first iteration, as shown in equation 4.1, LU factorization routine is called on the upper left block, and then updates using `ftrsm` and `fgemm` are performed on the remaining blocks.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_1\backslash & 0 & 0 \\ A'_{21} & Id & 0 \\ A'_{31} & 0 & Id \end{bmatrix} \begin{bmatrix} \backslash U_1 & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix} \tag{4.1}$$

We recall the operations that are performed during this computation:

- LU decomposition on first block : $A_{11} = L_1 . U_1$
- `ftrsm` update:
  - $A'_{21} = A_{21}.U_1^{-1}$; $A'_{31} = A_{31}.U_1^{-1}$;
  - $A'_{12} = L_1^{-1}.A_{12}$; $A'_{13} = L_1^{-1}.A_{13}$;
- `fgemm` update:
  - $A'_{22} = A_{22} - A'_{21}.A'_{12}$; $A'_{23} = A_{23} - A'_{21}.A'_{13}$
  - $A'_{32} = A_{32} - A'_{31}.A'_{12}$; $A'_{33} = A_{33} - A'_{31}.A'_{13}$

Since LU decomposition is done in-place, matrices L and U are stored in A during the computation as shown in figures 4.6, 4.7 and 4.8. In each iteration, LU factorization is called on the upper left block and then update tasks such as `ftrsm` and `fgemm` routines are executed on the remaining blocks. However, at the end of the call of each update routine on a block, modular reductions are applied on the elements of the block. The

number of modular reductions could be costly and should be delayed. Indeed, in this example, the input matrix is splitted in 3×3 blocks and thus performing three iterations for the computation of the final L and U matrices. On the last (bottom right) block `fgemm` routine is called 2 times during the algorithm for 3×3 splitting of the matrix. If the number of iterations is k, this gives $k-1$ calls of `fgemm` task on this (bottom right) block and thus performs unnecessary modular reduction operations that could be avoided.

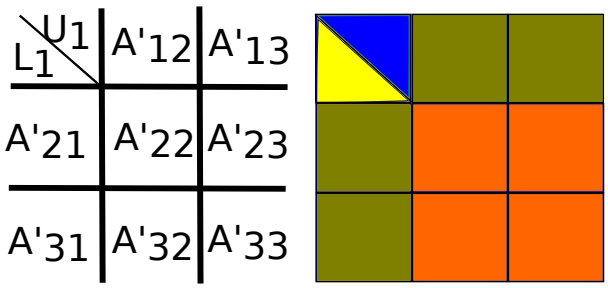

Figure 4.6: Tiled LU right looking variant: first iteration on blocks in place

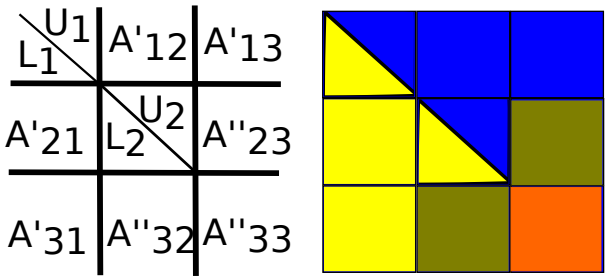

Figure 4.7: Tiled LU right looking variant: second iteration on blocks in place

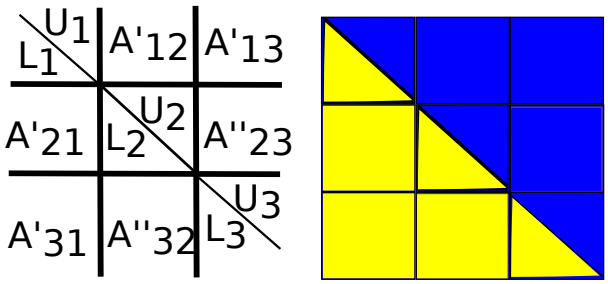

Figure 4.8: Tiled LU right looking variant: last iteration on blocks in place

This led us to investigate other tile variants such as the left-looking and the Crout variant that has fewer modular reduction complexity. These variants allows to accumulate several multiplications before reducing.

4.3.2.2 The tile iterative CROUT variant

The tile iterative Crout variant delays the modular reductions for the `fgemm` calls. we explain here-under the steps of this tile Crout variant for LU decomposition.

In each iteration, `fgemm` tasks are not called on all blocks, but only on adjacent tiles on which depends the critical path of the tile LU decomposition (figure 4.9). Then LU task is called on the current diagonal block. And then the series of `ftrsm` calls are performed as in the right looking variant as shown in figure 4.10. This tile scheme not only privileges tasks of the critical path but also insures that `fgemm` task is called only once on each block during the overall computation. In this example, the matrix is splitted in 4×4 blocks. For every block `fgemm` routine is called only once in all iterations.

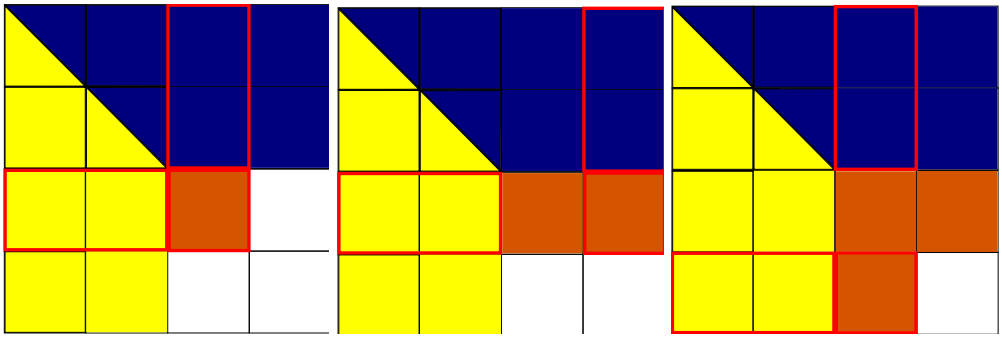

Figure 4.9: Tiled LU Crout variant in place (1)

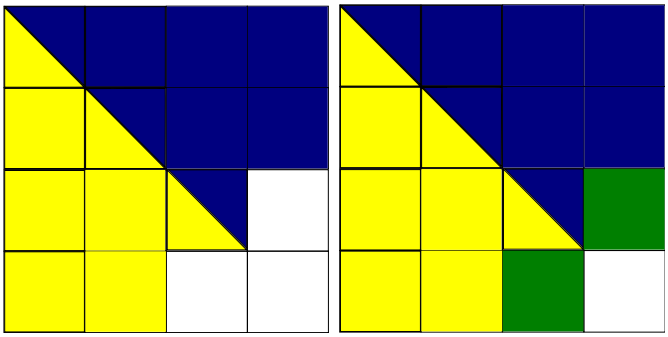

Figure 4.10: Tiled LU Crout variant in place (2)

This variant performs less modular reductions than the right looking variant and has a better performance behavior in a sequential execution.

4.3.2.3 The tile iterative Left Looking variant

The tile iterative left looking variant is similar to the Crout variant as it delays updates but it privileges most left-hand side task execution: the iteration begins with the update `ftrsm` task that is called on the first adjacent block on the right of the last LU task as shown in figure 4.11. This allows to delay modular reductions also for

the `futrsm` tasks. Indeed, in each iteration only one sequential task will perform the `futrsm` call. Then, a series of `fgemm` tasks are performed only on the lower blocks. Figure 4.12 shows the LU task followed by the last `ftrsm` call on the lower blocks.

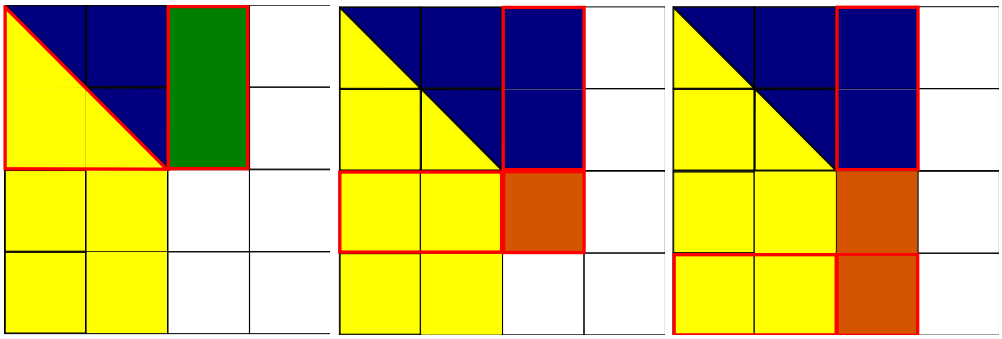

Figure 4.11: Tiled left-looking variant in place (1)

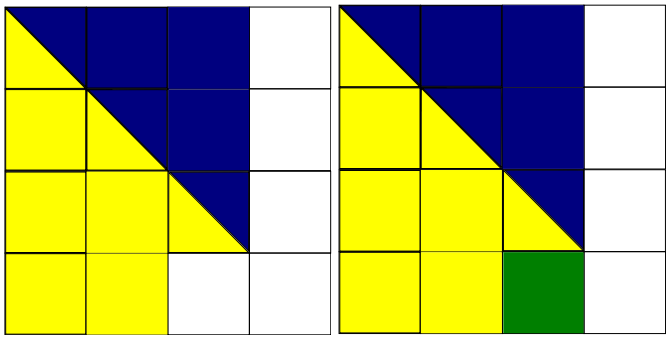

Figure 4.12: Tiled left-looking variant in place (2)

Ideally tiles of a block algorithm should fit into the cache memory to reduce as much as possible the dependency on the bus speed. At the cost of reducing modular reductions, this tile left looking variant of LU decomposition compromises this aspect of cache efficient tile algorithms. Indeed, the sequential `ftrsm` task gets bigger with each iteration.

All three variants explained above aim to reduce modular operations during LU decomposition. In a parallel execution context, the performance behavior of each variant could differ since the size of tiles and the number of independent tasks generated in each iteration are essential to get the best parallel speed-up. The parallel aspects of these variants will be discussed in Chapter 6.

4.4 Complexity analysis of the new tile recursive algorithm

We study here the time complexity of Algorithm 4 by counting the number of field operations. For the sake of simplicity, we will assume here that the dimensions m and n are powers of two. The analysis can be extended to the general case for arbitrary m and n.

For $i = 1, 2, 3, 4$ we denote by T_i the cost of the i-th recursive call to PLUQ, on a $\frac{m}{2} \times \frac{n}{2}$ matrix of rank r_i. We also denote by $T_{\texttt{ftrsm}}(m, n)$ the cost of a call `ftrsm` on a rectangular matrix of dimensions $m \times n$, and by $T_{\texttt{fgemm}}(m, k, n)$ the cost of multiplying an $m \times k$ by an $k \times n$ matrix.

Theorem 1: Algorithm 4, run on an $m \times n$ matrix of rank r, performs $O\left(mnr^{\omega-2}\right)$ field operations.

Proof: Let $T = T_{\texttt{pluq}}(m, n, r)$ be the cost of Algorithm 4 run on a $m \times n$ matrix of rank r. From the complexities of the subroutines given, e.g., in [36] and the recursive calls in Algorithm 4, we have:

$$\begin{aligned}
T &= T_1 + T_2 + T_3 + T_4 + T_{\texttt{ftrsm}}(r_1, \frac{m}{2}) + T_{\texttt{ftrsm}}(r_1, \frac{n}{2}) + T_{\texttt{ftrsm}}(r_2, \frac{m}{2}) \\
&\quad + T_{\texttt{ftrsm}}(r_3, \frac{n}{2}) + T_{\texttt{fgemm}}(\frac{m}{2} - r_1, r_1, \frac{n}{2}) + T_{\texttt{fgemm}}(\frac{m}{2}, r_1, \frac{n}{2} - r_1) \\
&\quad + T_{\texttt{fgemm}}(\frac{m}{2}, r_1, \frac{n}{2}) + T_{\texttt{fgemm}}(r_3, r_2, \frac{n}{2} - r_2) + T_{\texttt{fgemm}}(\frac{m}{2} - r_3, r_2, \frac{n}{2} - r_2 - r_4) \\
&\quad + T_{\texttt{fgemm}}(\frac{m}{2} - r_3, r_3, \frac{n}{2} - r_2 - r_4) \\
&\leq T_1 + T_2 + T_3 + T_4 + K(\frac{m}{2}(r_1^{\omega-1} + r_2^{\omega-1}) + \frac{n}{2}(r_1^{\omega-1} + r_3^{\omega-1}) + \frac{m}{2}\frac{n}{2}r_1^{\omega-2} \\
&\quad + \frac{m}{2}\frac{n}{2}r_2^{\omega-2} + \frac{m}{2}\frac{n}{2}r_3^{\omega-2}) \\
&\leq T_1 + T_2 + T_3 + T_4 + K'mnr^{\omega-2}
\end{aligned}$$

for some constants K and K' (we recall that $a^{\omega-2} + b^{\omega-2} \leq 2^{3-\omega}(a+b)^{\omega-2}$ for $2 \leq \omega \leq 3$).

Let $C = max\{\frac{K'}{1-2^{4-2\omega}}; 1\}$.Then we can prove by a simultaneous induction on m and n that $T \leq Cmnr^{\omega-2}$.

Indeed, if $(r = 1, m = 1, n \geq m)$ or $(r = 1, n = 1, m \geq n)$ then $T \leq m - 1 \leq Cmnr^{\omega-2}$. Now if it is true for $m = 2^j, n = 2^i$, then for $m = 2^{j+1}, n = 2^{i+1}$, we have

$$\begin{aligned}
T &\leq \frac{C}{4}mn(r_1^{\omega-2} + r_2^{\omega-2} + r_3^{\omega-2} + r_4^{\omega-2}) + K'mnr^{\omega-2} \\
&\leq \frac{C(2^{3-\omega})^2}{4}mnr^{\omega-2} + K'mnr^{\omega-2} \\
&\leq K'\frac{2^{4-2\omega}}{1-2^{4-2\omega}}mnr^{\omega-2} + K'mnr^{\omega-2} \leq Cmnr^{\omega-2}.
\end{aligned}$$

□

In order to compare this algorithm with usual Gaussian elimination algorithms, we now refine the analysis to compare the leading constant of the time complexity in the special case where the matrix is square and has a generic rank profile: $r_1 = \frac{m}{2} = \frac{n}{2}, r_2 = 0, r_3 = 0$ and $r_4 = \frac{m}{2} = \frac{n}{2}$ at each recursive step.

Hence, with C_ω the constant of matrix multiplication, we have

$$\begin{aligned}
T_{\texttt{pluq}} &= 2T_{\texttt{pluq}}(\frac{n}{2}, \frac{n}{2}, \frac{n}{2}) + 2T_{\texttt{ftrsm}}(\frac{n}{2}, \frac{n}{2}) + T_{\texttt{fgemm}}(\frac{n}{2}, \frac{n}{2}, \frac{n}{2}) \\
&= 2T_{\texttt{pluq}}(\frac{n}{2}, \frac{n}{2}, \frac{n}{2}) + 2\frac{C_\omega}{2^{\omega-1}-2}\left(\frac{n}{2}\right)^\omega + C_\omega\left(\frac{n}{2}\right)^\omega
\end{aligned}$$

Writing $T_{\texttt{pluq}}(n,n,n) = \alpha n^\omega$, the constant α satisfies:

$$\alpha = C_\omega \frac{1}{(2^\omega - 2)} \left(\frac{1}{2^{\omega-2} - 1} + 1 \right) = C_\omega \frac{2^{\omega-2}}{(2^\omega - 2)(2^{\omega-2} - 1)}.$$

which is equal to the constant of the CUP and LUP decompositions [62, Table 1]. In particular, it equals $2/3$ when $\omega = 3, C_\omega = 2$, matching the constant of the classical Gaussian elimination.

The algorithm 4 is thus the first tile recursive algorithm after [42] but with rank sensitive $O\left(mnr^{\omega-2}\right)$ complexity.

4.5 Modular reductions

When computing over a finite field, it is of paramount importance to reduce the number of modular reductions in the course of linear algebra algorithms. The classical technique is to accumulate several multiplications before reducing, namely replacing $\sum_{i=1}^n (a_i b_i \mod p)$ with $(\sum_{i=1}^n a_i b_i)$ while keeping the result exact. If a_i and b_i are integers between 0 and $p-1$ this is possible with integer or floating point units if the result does not overflow, or in other words if $n(p-1)^2 < 2^{\text{mantissa}}$, see, e.g., [36] for more details.

This induces a splitting of matrices in blocks of size the largest n^* satisfying the latter condition. Now the use of block algorithms in parallel, introduces a second blocking parameter that interferes in counting modular reductions. We will therefore compare the number of modular reductions of the three variants presented in section 4.3.2 of the tile iterative algorithm (left-looking, right-looking and Crout), the slab recursive algorithm 3 of [36], and the tile recursive algorithm 4.

For the sake of simplicity, we will assume that the block dimensions in the parallel algorithms are always below n^*. In other words operations are done with full delayed reduction for a single multiplication and any number of additions: operations of the form $\sum a_i b_i$ are reduced modulo p only once at the end of the addition, but $a \cdot b \cdot c$ requires two reductions. For instance, with this model, the number of reductions required by a classic multiplication of matrices of size $m \times k$ by $k \times n$ is simply: $R_{\texttt{fgemm}}(m,k,n) = mn$. This extends also for triangular solving with an $m \times n$ unknown matrix.

Theorem 2: Over a prime field modulo p, the number of reductions modulo p required by `ftrsm` (m,n) with full delayed reduction is:

$$\begin{array}{lcll} R_{\texttt{futrsm}}(m,n) & = & (m-1)n & \text{if the triangular matrix has a unit diagonal,} \\ R_{\texttt{ftrsm}}(m,n) & = & (2m-1)n & \text{in general.} \end{array}$$

Proof: If the matrix has unit diagonal, then a fully delayed reduction is required only once after the update of each row of the result. In the general case, we invert each diagonal element first and multiply each element of the right hand side by this inverse diagonal element, prior to the update of each row of the result. This gives mn extra reductions. Actually, with unit diagonal, the computation of the last row of the solution of $UX = B$ requires no modular reduction as it is just a division by 1, we will therefore rather use $R_{\texttt{futrsm}}(m,m,n) = (m-1)n$. With this refinement, this also reduces to $R_{\texttt{ftrsm}}(m,m,n) = (2m-1)n$. □

$k=1$	Iterative Right looking	$\frac{1}{3}n^3 - \frac{1}{3}n$
	Iterative Left Looking	$\frac{3}{2}n^2 - \frac{5}{2}n + 1$
	Iterative Crout	$\frac{3}{2}n^2 - \frac{5}{2}n + 1$
$k>1$	Tile Iterative Right looking	$\frac{1}{3k}n^3 + \left(1 - \frac{1}{k}\right)n^2 + \left(\frac{1}{6}k - \frac{3}{2} + \frac{1}{k}\right)n$
	Tile Iterative Left looking	$\left(2 - \frac{1}{2k}\right)n^2 - \frac{5}{2}kn + 2k^2 - 2k + 1$
	Tile Iterative Crout	$\left(\frac{5}{2} - \frac{1}{k}\right)n^2 + \left(-2k - \frac{3}{2} + \frac{1}{k}\right)n + k^2$
	Tile Recursive	$2n^2 - n\log_2 n - 2n$
	Slab Recursive	$(1 + \frac{1}{4}\log_2(n))n^2 - \frac{1}{2}n\log_2 n$

Table 4.2: Counting modular reductions in full rank block LU factorization of an $n \times n$ matrix modulo p when $np(p-1) < 2^{\text{mantissa}}$, for a block size of k dividing n.

In table 4.2 The first three rows are obtained by setting $k = 1$ in the following block versions. The next three rows are obtained via the following analysis where the base case (i.e. the $k \times k$ factorization) always uses the best unblocked version, that is the Left Looking variant. The last two rows of the table corresponds to the tile recursive and slab recursive algorithms.

Theorem 3: Table 4.2 is correct.

Proof: We will first detail the computation of the number of modular reductions for each variant of the first three rows of table 4.2. Following Table 4.1, we have for k=1:

Right looking	Amount of modular reductions
for i=1 to n **do**	
`pluq` (1,1)	0
`futrsm` (1,1,n-i)	0
`ftrsm` (1,1,n-i)	$n-i$
`fgemm` (n-i,1,n-i)	$(n-i)^2$
end for	

Table 4.3: Number of modular reduction at each iteration of the Right looking iterative block LU factorization.

Table 4.3 gives the amount of modular reductions for the Right looking variant at each step of the iteration. Thus:

$$R_{RightLooking} = \sum_{i=1}^{n}(n-i) + \sum_{i=1}^{n}(n-i)^2 = \frac{1}{3}n^3 - \frac{n}{3}.$$

Left looking	Amount of modular reductions
`pluq` (1,1)	0
`ftrsm` (1,1,n-1)	$n-1$
for i=2 to n **do**	
`futrsm` ((i-1),(i-1),1)	$i-2$
`fgemm` (n-(i-1),(i-1),1)	$n-i+1$
`pluq` (1,1)	0
`ftrsm` (1,1,n-i)	$n-i$
end for	

Table 4.4: Number of modular reduction at each iteration of the Left looking iterative block LU factorization.

Table 4.4 gives the amount of modular reductions for the Left looking variant at each step of the iteration. Thus:

$$R_{LeftLooking} = n-1+\sum_{i=2}^{n}(i-2)+\sum_{i=2}^{n}(n-i)+\sum_{i=2}^{n} i = \frac{3}{2}n^2-\frac{5}{2}n+1.$$

Crout	Amount of modular reductions
`pluq` (1,1)	0
`futrsm` (1,1,n-1)	0
`ftrsm` (1,1,n-1)	n-1
for i=2 to n **do**	
`fgemm` (n-(i-1),i-1,1)	$n-i+1$
`fgemm` (1,i-1,n-i)	$n-i$
`pluq` (1,1)	0
`futrsm` (1,1,n-i)	0
`ftrsm` (1,1,n-i)	$n-i$
end for	

Table 4.5: Number of modular reduction at each iteration of the Crout iterative block LU factorization.

The amount of modular reductions for the Crout variant is thus:

$$R_{Crout} = n-1+\sum_{i=2}^{n}(n-i+1)+\sum_{i=2}^{n}(n-i)+\sum_{i=2}^{n}(n-i) = \frac{3}{2}n^2-\frac{5}{2}n+1.$$

Now, for $k \geq 1$:
The right looking variant performs $\frac{n}{k}$ such $k \times k$ base cases, $\texttt{pluq}(k,k)$, then, at iteration i, $(\frac{n}{k}-i)(\texttt{futrsm}(k,k,k)+\texttt{ftrsm}(k,k,k))$, and $(\frac{n}{k}-i)^2$ `fgemm` (k,k,k), for a total of $\frac{n}{k}(\frac{3}{2}n^2-\frac{5}{2}n+1)+\sum_{i=1}^{\frac{n}{k}}(n-ik)\left((3k-2)+(\frac{n}{k}-i)k\right) = \frac{1}{3k}n^3+\left(1-\frac{1}{k}\right)n^2+\left(\frac{1}{6}k-\frac{3}{2}+\frac{1}{k}\right)n$.

The Crout variant requires,

- at each step, except the first one, to compute $R_{\texttt{fgemm}}(n-ik, ik, k)$ reductions for the pivot and below and $R_{\texttt{fgemm}}(k, ik, n-(i-1)k)$ for the other block;

- at each step, to perform one base case for the pivot block, to solve unitary triangular systems, to the left, below the pivot, using $(\frac{n}{k}-i)R_{\texttt{futrsm}}(k,k,k)$ reductions and to solve triangular systems to the right, using $(\frac{n}{k}-i)R_{\texttt{ftrsm}}(k,k,k)$ reductions.

Similarly, the Left looking variant requires $R_{\texttt{fgemm}}(n-ik, ik, k) + R_{\texttt{pluq}}(k)$ $+R_{\texttt{futrsm}}(ik, ik, k) + R_{\texttt{ftrsm}}(k, k, n-ik)$ reductions in the main loop.

PLE and CUP are the slab recursive algorithms of [62] and `pluq` is the tile recursive algorithm 4 explained in section 4.3.1.

Computation of the number of modular reductions for tile recursive `pluq` algorithm: If the top left square block is full rank then `pluq` reduces to one recursive call, two square `ftrsm` (one unitary, one generic) one square matrix multiplication and a final recursive call. In terms of modular reductions, this gives: $R_{\texttt{pluq}}(n) = 2R_{\texttt{pluq}}(\frac{n}{2}) + R_{\texttt{utrsm}}(\frac{n}{2},\frac{n}{2}) + R_{\texttt{ftrsm}}(\frac{n}{2},\frac{n}{2}) + R_{\texttt{fgemm}}(\frac{n}{2},\frac{n}{2},\frac{n}{2})$. Therefore, using Theorem 2, the number of reductions within `pluq` satisfies $T(n) = 2T(\frac{n}{2}) + n^2$ so that it is $R_{\texttt{pluq}}(n,n) = 2n^2 - 2n$ if n is a power of two.

Computation of the number of modular reductions for PLE and CUP algorithms: For row or column oriented elimination this situation is more complicated since the recursive calls will always be rectangular even if the intermediate matrices are full-rank.

$$R_{\text{PLE}}(m,n) = R_{\text{PLE}}(\frac{m}{2}, n) + R_{\text{PLE}}(\frac{m}{2}, n-\frac{m}{2}) + R_{\texttt{ftrsm}}(\frac{m}{2},\frac{m}{2}) + R_{\texttt{fgemm}}(\frac{m}{2},\frac{m}{2}, n-\frac{m}{2}) \quad (4.2)$$

From equation 4.2, the number of modular reductions of the PLE algorithm depends only on the number of modular reductions caused by the `ftrsm` routine calls and by the `fgemm` routine calls. Let $N_{\texttt{ftrsm}}$ and $N_{\texttt{fgemm}}$ be the number of modular reductions introduced respectively by the $R_{\texttt{ftrsm}}$ and the $R_{\texttt{fgemm}}$ in all the recursion tree of the slab recursive PLE algorithm. Thus, $R_{\text{PLE}}(m,n) = N_{\texttt{ftrsm}} + N_{\texttt{fgemm}}$.

In equation 4.2, there is two recursive calls of R_{PLE}. Thus, each of the $R_{\texttt{ftrsm}}$ and $R_{\texttt{fgemm}}$ are called two times at each level of the recursion except for the first recursion. This gives a total number of $\sum_{i=1}^{\log_2(m)} 2^{i-1}$ times of $R_{\texttt{ftrsm}}$ calls.

Now from theorem 2, $R_{\texttt{ftrsm}}(\frac{m}{2},\frac{m}{2}) = (2\frac{m}{2}-1)\frac{m}{2} = 2(\frac{m}{2})^2 - \frac{m}{2}$ and $R_{\texttt{ftrsm}}(\frac{m}{4},\frac{m}{4}) = 2(\frac{m}{4})^2 - \frac{m}{4}$, and so on.

Thus, $N_{\texttt{ftrsm}} = 2\sum_{i=1}^{\log_2(m)} 2^{i-1}(\frac{m}{2^i})^2 - \sum_{i=1}^{\log_2(m)} 2^{i-1}(\frac{m}{2^i})$.

In the case of the `fgemm` routine calls, we perform a total number of modular reductions in the recursive PLE algorithm:

$$
\begin{aligned}
N_{\texttt{fgemm}} &= \sum_{i=1}^{\log_2(m)} \sum_{j=1}^{2^{i-1}} R_{\texttt{fgemm}}(\frac{m}{2^i}, \frac{m}{2^i}, n-(2j-1)\frac{m}{2^i}). \\
&= \sum_{i=1}^{\log_2(m)} \sum_{j=1}^{2^{i-1}} \frac{m}{2^i}(n-(2j-1)\frac{m}{2^i}) \\
&= \sum_{i=1}^{\log_2(m)} (2^{i-1}n\frac{m}{2^i} - \frac{m^2}{2^{2i}} \sum_{j=1}^{2^{i-1}}(2j-1)) \\
&= \sum_{i=1}^{\log_2(m)} (n\frac{m}{2} - \frac{m^2}{2^2}) = \sum_{i=1}^{\log_2(m)} \frac{m}{2}(n-\frac{m}{2})
\end{aligned}
$$

We thus obtain:

$$
\begin{aligned}
R_{\text{PLE}}(m,n) &= 2\sum_{i=1}^{\log_2(m)} 2^{i-1}(\frac{m}{2^i})^2 - \sum_{i=1}^{\log_2(m)} 2^{i-1}(\frac{m}{2^i}) + \sum_{i=1}^{\log_2(m)} \frac{m}{2}(n-\frac{m}{2}) \\
&= m^2 - m - \frac{m}{2}\log_2(m) + \frac{mn}{2}\log_2(m) + \frac{m}{2}(n-\frac{m}{2})\log_2(m)
\end{aligned}
$$

Thus if $m = n$, $R_{\text{PLE}}(n,n) = (1+\frac{1}{4}\log_2(n))n^2 - \frac{1}{2}n\log_2 n - n$
□

This shows that the tile recursive algorithm (algorithm 4) requires fewer modular reductions, as soon as m is larger than 32. Over finite fields, since reductions can be much more expensive than multiplications or additions by elements of the field, this is a non negligible advantage. We show in Section 4.6 that this participates to the better practical performance of the `pluq` algorithm.

In Table 4.2 we see that the left looking variant always performs less modular reductions. Then the tile recursive performs less modular reductions than the Crout variant as soon as $2 \leq k \leq \frac{n}{2+\sqrt{2}}$. Finally the right looking variant clearly performs more modular reductions. This explains the respective performance of the algorithms shown on Table 4.6 (except for larger dimensions where fast matrix multiplication comes into play). Also, we see that even when the number of modular reductions is an order of magnitude lower than that of the integer operations the cost of the divisions is nonetheless not negligible. Moreover, the best algorithms here may not perform well in parallel, as will be shown in chapter 6.

4.6 Experiments

In the following experiments, we measured the real time of the computation averaged over 10 instances (100 for $n < 500$) of $n \times n$ matrices with rank $r = n/2$ for any even integer value of n between 20 and 700. In order to ensure that the row and column rank profiles of these matrices are uniformly random, we construct them as the product $A = L\mathcal{R}U$, where L and U are random non-singular lower and upper

	$k = 212$			$k = \frac{n}{3}$			Recursive	
	Right	Crout	Left	Right	Crout	Left	Tile	Slab
n=3000	3.02	2.10	**2.05**	2.97	2.15	2.10	2.16	2.26
n=5000	11.37	8.55	8.43	9.24	8.35	8.21	**7.98**	8.36
n=7000	29.06	22.19	21.82	22.56	22.02	21.73	**20.81**	21.66

Table 4.6: Timings (in seconds) of sequential LU factorization variants on one core

triangular matrices and $\mathcal{R}$ is an $m \times n$ matrix with zeros except on $r = n/2$ positions, chosen uniformly at random, set to one. The $\mathcal{R}$ matrix will be defined in the next chapter. The effective speed is obtained by dividing an estimate of the arithmetic cost $(2mnr + 2/3r^3 - r^2(m + n))$ by the computation time.

During the Gaussian elimination algorithm the search of a pivot and the permutation strategies is the key the recover the echelon form and even more information on the rank profile of the matrix. These aspects are explained in chapter 5 where we propose an implementation combining our tile recursive algorithm with an iterative base case.We present in chapter 5 a new base case (algorithm 6) algorithm and its implementation over a finite field that is written in the `FFLAS-FFPACK` library[1]. It is based on a lexicographic order search and row and column rotations. Moreover, the schedule of the update operations is that of a Crout elimination, for it reduces the number of modular reductions, as shown in § 4.5. Figure 4.13 shows its computation speed (3), compared to that of the pure recursive algorithm (6), and to the base case algorithm 6, using a product order search, and either a left-looking (4) or a right-looking (5) schedule. At $n = 200$, the left-looking variant (4) improves over the right looking variant (5) by a factor of about 2.14 as it performs fewer modular reductions. Then, the Crout variant (3) again improves variant (4) by a factor of about 3.15. Lastly we also show the speed of the final implementation, formed by the tile recursive algorithm cascading to either the Crout base case (1) or the left-looking one (2). The threshold where the cascading to the base case occurs is experimentally set to its optimum value, i.e. 200 for variant (1) and 70 for variant (2). This illustrates that the gain on the base case efficiency leads to a higher threshold, and improves the efficiency of the cascade implementation (by an additive gain of about 2.2 effective Gfops in the range of dimensions considered). One can execute the benchmark-pluq binary in the benchmarks folder of the `FFLAS-FFPACK` library to reproduce the curve (1).

Algorithm 4 combined with the the best base case algorithm has been implemented in the `FFLAS-FFPACK` library[2] . We present here experiments comparing its efficiency with the implementation of the CUP/PLE decomposition (the slab recursive algorithm), called `LUdivine` in this same library.

Table 4.7 shows the cache misses reported by the callgrind tool (valgrind emulator version 3.8.1). We also report in the last column the corresponding computation time (without emulator). We used the same matrices as in Figure 4.13, with rank half the

[1] FFLAS-FFPACK revision 1193, http://linalg.org/projects/fflas-ffpack, linked against OpenBLAS-v0.2.8.
[2] http://linalg.org/projects/fflas-ffpack

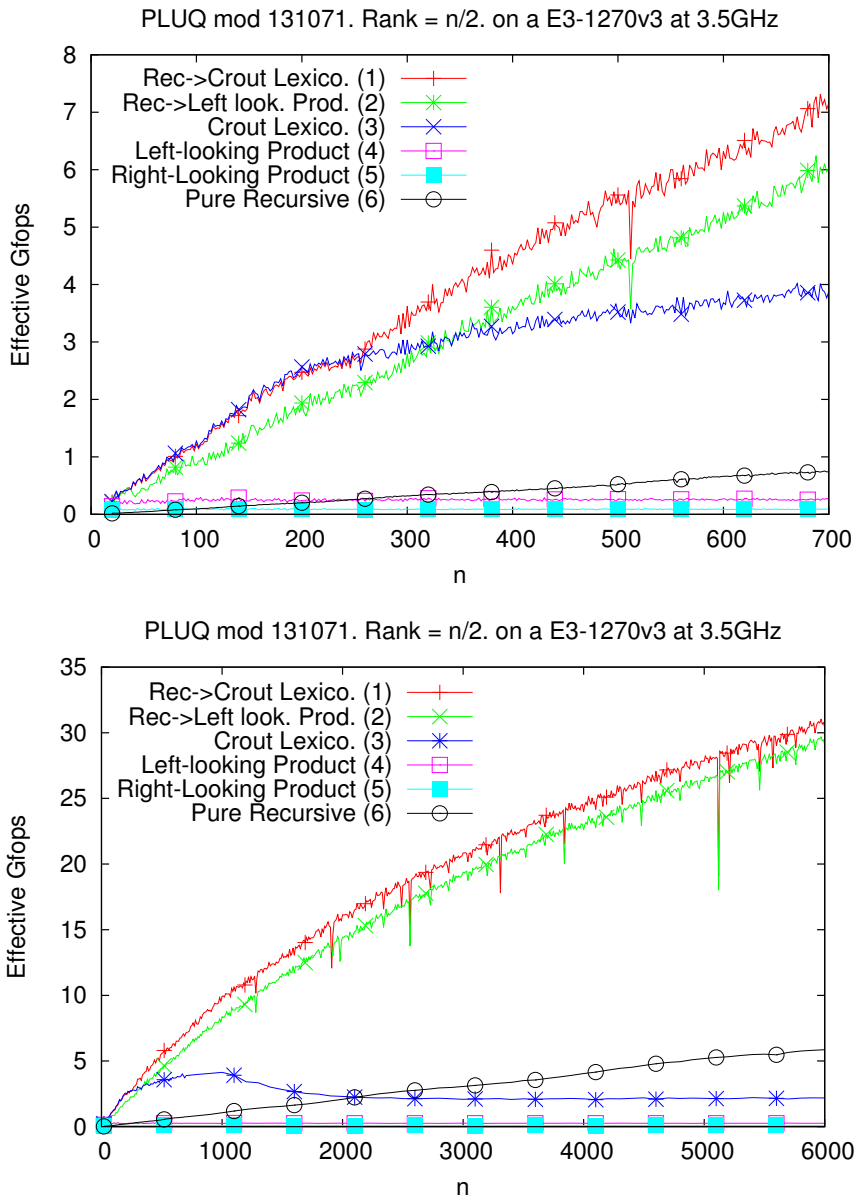

Figure 4.13: Computation speed of PLUQ decomposition base cases.

dimension. We first notice the impact of the base case on the tile recursive PLUQ algorithm: although it does not change the number of cache misses, it strongly reduces the total number of memory accesses (fewer permutations), thus improving the computation time. Now as the dimension grows, the amount of memory accesses and of cache misses plays in favor of the tile recursive PLUQ which becomes faster than `LUdivine`.

Matrix	Algorithm	Accesses	L1 Misses	L3 Misses	L3/Accesses	Timing (s)
A4K	`LUdivine`	1.529E+10	1.246E+09	2.435E+07	.159	**2.31**
	Tile-rec-no-base-case	1.319E+10	**7.411E+08**	1.523E+07	**.115**	5.82
	Tile-rec-base-case	**8.105E+09**	7.467E+08	**1.517E+07**	.187	2.48
A8K	`LUdivine`	7.555E+10	9.693E+09	2.205E+08	.292	15.2
	Tile-rec-no-base-case	6.150E+10	**5.679E+09**	1.305E+08	**.212**	28.4
	Tile-rec-base-case	**4.067E+10**	5.686E+09	**1.303E+08**	.321	**15.1**
A12K	`LUdivine`	2.003E+11	3.141E+10	7.943E+08	.396	46.5
	Tile-rec-no-base-case	1.575E+11	**1.911E+10**	4.691E+08	**.298**	73.9
	Tile-rec-base-case	**1.111E+11**	1.913E+10	**4.687E+08**	.422	**45.5**
A16K	`LUdivine`	4.117E+11	7.391E+10	1.863E+09	.452	103
	Tile-rec-no-base-case	3.142E+11	4.459E+10	1.092E+09	**.347**	150
	Tile-rec-base-case	**2.299+11**	**4.458E+10**	**1.088E+09**	.473	**98.8**

Table 4.7: Cache misses for dense matrices with rank equal half of the dimension

4.7 Conclusion

The tile recursive PLUQ algorithm that we propose introduces a finer treatment of rank deficiency that reduces the number of arithmetic operations, makes the time complexity rank sensitive and allows to perform the computation in-place. It also performs fewer modular reductions when computing over a finite field. Overall the new algorithm is also faster in practice than previous implementations with large enough matrices.

Second, three base cases variants were studied for the tile recursive PLUQ algorithm: the left-looking, the right-looking and the Crout variants. Modular reductions over finite fields has an impact on each variant making the left-looking or the Crout variants the best suited for a base case algorithm for the tile recursive algorithm. The right looking variant with its cubic number of modular reductions is to be excluded as a base case algorithm.

The new tile recursive algorithm computes the row and column rank profiles of the matrix and of all of its leading sub-matrices. In Chapter 5 we define a new matrix invariant, the rank profile matrix, that can be revealed by this algorithm. In chapter 6 we study the parallelization aspects of the tile/slab iterative and recursive algorithms and show that the tile recursive algorithm has the best speed-up.

Chapter 5

Computation of echelon forms

In the previous chapter, we proposed a first Gaussian elimination algorithm, with a recursive splitting of both row and column dimensions, which computes a PLUQ decomposition with L a lower triangular matrix and U an upper unit triangular matrix while preserving the sub-cubic rank-sensitive time complexity and keeping the computation in-place. This algorithm also computes simultaneously the row and column rank profiles. Consequently, we analyze in this chapter the conditions on the pivoting that reveal the rank profiles and introduce a new matrix invariant, the rank profile matrix. This normal form contains the row and column rank profile information of the matrix and that of all its leading sub-matrices.

This normal form is closely related to a permutation matrix appearing in the Bruhat decomposition [19] and in related variants [25, 51, 13, 73, 74]. Still, no connection to the rank profiles were made. In another setting, the construction of matrix Schubert varieties in [75, Ch. 15] defines a similar invariant, but presents neither a matrix decomposition nor any computational aspects.

More precisely, in this chapter we gather the following key contributions:

- we define a new matrix invariant over a field, the rank profile matrix, summarizing all information on the row and column rank profiles of all the leading sub-matrices;
- we study the conditions for a Gaussian elimination algorithm to compute all or part of this invariant, through the corresponding PLUQ decomposition;
- as a consequence, we show that the classical iterative CUP decomposition algorithm can actually be adapted to compute the rank profile matrix.
- we also show that both the row and the column echelon forms of a matrix can be recovered from some PLUQ decompositions thanks to an elementary post-processing algorithm.

We first recall the definitions of the row and column rank profiles and the matrix factorizations that reveal them. We then decompose, in section 5.2, the pivoting strategy of any PLUQ algorithm into two types of operations: the search of the pivot and the permutation used to move it to the main diagonal. We propose a new search and a new permutation strategy. Afterwards, we introduce in section 5.3 the rank profile matrix, a normal form summarizing all rank profile information of a matrix and of all its leading

sub-matrices. We show how algorithms can reveal this normal form. In particular we show three new pivoting strategy combinations that compute the rank profile matrix and use one of them, an iterative Crout CUP with rotations, as a base case for the tile recursive Gaussian elimination algorithm. Finally, we show that preserving both the row and column rank profiles, together with ensuring a monotonicity of the associated permutations, allows us to compute faster several other matrix decomposition, such as the LEU and Bruhat decompositions, and echelon forms.

5.1 Rank profile

5.1.1 The row and column rank profiles

The *row rank profile* (resp. *column rank profile*) of an $m \times n$ matrix A with rank r, denoted by RowRP(A) (resp. ColRP(A)), is the lexicographically smallest sequence of r indices of linearly independent rows (resp. columns) of A. An $m \times n$ matrix has generic row (resp. column) rank profile if its row (resp. column) rank profile is $(1, .., r)$. Lastly, an $m \times n$ matrix has generic rank profile if its r first leading principal minors are non-zero. Note that if a matrix has generic rank profile, then its row and column rank profiles are generic, but the converse is false: the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ does not have generic rank profile even if its row and column rank profiles are generic. The row support (resp. column support) of a matrix A, denoted by RowSupp(A) (resp. ColSupp(A)), is the subset of indices of its non-zero rows (resp. columns).

We recall that the row echelon form of an $m \times n$ matrix A is an upper triangular matrix $E = TA$, for a non-singular matrix T, with the zero rows of E at the bottom and the non-zero rows in stair-case shape: $\min\{j : a_{i,j} \neq 0\} < \min\{j : a_{i+1,j} \neq 0\}$. As T is non singular, the column rank profile of A is that of E, and therefore corresponds to the column indices of the leading elements in the staircase. Similarly the row rank profile of A is composed of the row indices of the leading elements in the staircase of the column echelon form of A.

5.1.2 Rank profile and triangular matrix decompositions

The rank profiles of a matrix and the triangular matrix decomposition obtained by Gaussian elimination are strongly related. Any $m \times n$ matrix A of rank r with generic rank profile has a unique LU decomposition: $A = LU$ for L an $m \times m$ unit lower triangular matrix with last $m - r$ columns those of the identity, and U an $m \times n$ upper triangular matrix with last $m - r$ rows equal to zero. If A has generic row rank profile, then it only takes a column permutation to produce a matrix AP^T that has a generic rank profile, with P a permutation matrix. It leads to an LUP decomposition: $A = LUP$, where $AP^T = LU$ is the LU decomposition of AP^T. Allowing row and column permutations generalizes the LU decomposition to any matrix, with no restriction on its rank profile: there always exist a pair of permutation matrices P, Q such that $P^T AQ^T$ has generic rank profile, thus leading to the PLUQ decomposition $A = PLUQ$. These relations, summarized in Table 5.1, suggest that the permutation matrices P and Q may carry information on the rank profiles.

Note that various matrix decompositions exist that reveal row or column rank profiles. The reader may refer to [62] for a detailed treatment of how most of them are

equivalent up to permutations. Hence we will stick to the PLUQ decomposition in the remaining of the manuscript.

Decomposition	Exists for	Unique
A=LU	Generic rank profile	Y
A=LUP	Generic row rank profile	N
A=PLU	Generic col rank profile	N
A=PLUQ	Any matrix	N

Table 5.1: Triangular decompositions for given rank profiles

The elimination of matrices with arbitrary rank profiles gives rise to several matrix factorizations and many algorithmic variants. In numerical linear algebra one often uses the PLUQ decomposition, with P and Q permutation matrices, L a lower unit triangular matrix and U an upper triangular matrix. The LSP and LQUP variants of [58] are used to reduce the complexity rank deficient Gaussian elimination to that of matrix multiplication. Many other algorithmic decompositions exist allowing fraction free computations [63], in-place computations [36, 62] or sub-cubic rank-sensitive time complexity [82, 62]. In section 4.3.1 we proposed a Gaussian elimination algorithm with a recursive splitting of both row and column dimensions, and replacing row and column transpositions by rotations. This elimination can compute simultaneously the row and column rank profile while preserving the sub-cubic rank-sensitive time complexity and keeping the computation in-place.

A common strategy in computer algebra to compute the row rank profile is to search for pivots in a row-major fashion: exploring each row in order, moving to the next row only when the current row is all zeros. Such a $\bar{P}LU\bar{Q}$ decomposition can be transformed into a CUP decomposition (where $P = \bar{Q}$ and $C = \bar{P}L$ is in column echelon form) and the first r values of the permutation associated to $\bar{P}$ are exactly the row rank profile. A block recursive algorithm can be derived from this scheme by splitting the row dimension in halves. Similarly, the column rank profile can be obtained in a column major search: exploring the current column, and moving to the next column only if the current one is zero. The $\bar{P}LU\bar{Q}$ decomposition can be transformed into a PLE decomposition (where $P = \bar{P}$ and $E = U\bar{Q}$ is in row echelon form) and the first r values of $\bar{Q}$ are exactly the column rank profile [62]. The corresponding block recursive algorithm uses a splitting of the column dimension.

This splitting in only one dimension results in operations with rectangular matrices of unbalanced dimensions. This is a major cause of inefficiency due to poor locality of the memory accesses. This problem is not specific to recursive algorithms, it also arises in block iterative algorithms: to the best of our knowledge, no algorithm using a splitting of both row and column dimension exists that computes echelon forms or rank profiles.

Recursive elimination algorithms splitting both row and column dimensions include the TURBO algorithm [42] and the LEU decomposition [73]. No connection is made to the computation of the rank profiles in any of these algorithms. The TURBO algorithm does not compute the lower triangular matrix L and performs five recursive calls. It therefore implies an arithmetic overhead compared to classic Gaussian elimination.

The LEU avoids permutations but at the expense of many additional matrix products. As a consequence its time complexity is not rank-sensitive.

5.2 Ingredients of a PLUQ decomposition algorithm

Over a field, the LU decomposition generalizes to matrices with arbitrary rank profiles, using row and column permutations (in some cases such as the CUP, or LSP decompositions, the row permutation is embedded in the structure of the C or S matrices). However such PLUQ decompositions are not unique and not all of them will necessarily reveal rank profiles and echelon forms. We will characterize the conditions for a PLUQ decomposition algorithm to reveal the row or column rank profile or the rank profile matrix.

We consider the four types of operations of a Gaussian elimination algorithm in the processing of the k-th pivot:

Pivot search: finding an element to be used as a pivot,

Pivot permutation: moving the pivot in diagonal position (k, k) by column and/or row permutations,

Update: applying the elimination at position (i, j):
$a_{i,j} \leftarrow a_{i,j} - a_{i,k}a_{k,k}^{-1}a_{k,j}$,

Normalization: dividing the k-th row (resp. column) by the pivot.

Choosing how each of these operation is done, and when they are scheduled results in an elimination algorithm. Conversely, any Gaussian elimination algorithm computing a PLUQ decomposition can be viewed as a set of specializations of each of these operations together with a scheduling.

The choice of doing the normalization on rows or columns only determines which of U or L will be unit triangular. The scheduling of the updates vary depending on the type of algorithm used: iterative, recursive, slab or tiled block splitting, with right-looking, left-looking or Crout variants.Neither the normalization nor the update impact the capacity to reveal rank profiles and we will thus now focus on the pivot search and the permutations.

Choosing a search and a permutation strategy fixes the matrices P and Q of the PLUQ decomposition obtained and, as we will see, determines the ability to recover information on the rank profiles. Once these matrices are fixed, the L and the U factors are unique. We introduce the pivoting matrix.

Definition 1: The pivoting matrix of a PLUQ decomposition $A = PLUQ$ of rank r is the r-sub-permutation matrix

$$\Pi_{P,Q} = P \begin{bmatrix} I_r & \\ & 0_{(m-r)\times(n-r)} \end{bmatrix} Q.$$

The r non-zero elements of $\Pi_{P,Q}$ are located at the initial positions of the pivots in the matrix A. Thus $\Pi_{P,Q}$ summarizes the choices made in the search and permutation operations.

5.2.1 Pivot search

The search operation vastly differs depending on the field of application. In numerical dense linear algebra, numerical stability is the main criterion for the selection of the pivot. In sparse linear algebra, the pivot is chosen so as to reduce the fill-in produced by the update operation. In order to reveal some information on the rank profiles, a notion of precedence has to be used: a usual way to compute the row rank profile, as already mentioned, is to search in a given row for a pivot and only move to the next row if the current row was found to be all zeros. This guarantees that each pivot will be on the first linearly independent row, and therefore the row support of $\Pi_{P,Q}$ will be the row rank profile. The precedence here is that the pivot's coordinates must minimize the order for the first coordinate (the row index). As a generalization, we consider the most common preorders of the cartesian product $\{1, \dots m\} \times \{1, \dots n\}$ inherited from the natural orders of each of its components and describe the corresponding search strategies, minimizing this preorder:

Row order: $(i_1, j_1) \preceq_{\text{row}} (i_2, j_2)$ iff $i_1 \leq i_2$: *search for any invertible element in the first non-zero row.*

Column order: $(i_1, j_1) \preceq_{\text{col}} (i_2, j_2)$ iff $j_1 \leq j_2$. *search for any invertible element in the first non-zero column.*

Lexicographic order: $(i_1, j_1) \preceq_{\text{lex}} (i_2, j_2)$ iff $i_1 < i_2$ or $i_1 = i_2$ and $j_1 \leq j_2$: *search for the leftmost non-zero element of the first non-zero row.*

Reverse lexicographic order: $(i_1, j_1) \preceq_{\text{revlex}} (i_2, j_2)$ iff $j_1 < j_2$ or $j_1 = j_2$ and $i_1 \leq i_2$: *search for the topmost non-zero element of the first non-zero column.*

Product order: $(i_1, j_1) \preceq_{\text{prod}} (i_2, j_2)$ iff $i_1 \leq i_2$ and $j_1 \leq j_2$: *search for any non-zero element at position (i, j) being the only non-zero of the leading (i, j) sub-matrix.*

Example 1: Consider the matrix $\begin{bmatrix} 0 & 0 & 0 & a & b \\ 0 & c & d & e & f \\ g & h & i & j & k \\ l & m & n & o & p \end{bmatrix}$, where each literal is a non-zero element. The minimum non-zero elements for each preorder are the following:

Row order	a, b
Column order	g, l
Lexicographic order	a
Reverse lexic. order	g
Product order	a, c, g

5.2.2 Pivot permutation

The pivot permutation moves a pivot from its initial position to the leading diagonal. Besides this constraint all possible choices are left for the remaining values of the permutation. Most often, it is done by row or column transpositions, as it clearly involves a small amount of data movement. However, these transpositions can break the precedence relations in the set of rows or columns, and can therefore prevent the recovery of the rank profile information. A pivot permutation that leaves the precedence relations unchanged will be called k-monotonically increasing.

Definition 2: A permutation of $\sigma \in \mathcal{S}_n$ is called k-monotonically increasing if its last $n - k$ values form a monotonically increasing sequence.

In particular, the last $n-k$ rows of the associated row-permutation matrix P_σ are in row echelon form. For example, the cyclic shift between indices k and i, with $k < i$ defined as $R_{k,i} = (1, \ldots, k-1, i, k, k+1, \ldots, i-1, i+1, \ldots, n)$, that we will call a (k,i)-rotation, is an elementary k-monotonically increasing permutation.

Example 2: The $(1,4)$-rotation $R_{1,4} = (4, 1, 2, 3)$ is a 1-monotonically increasing permutation. Its row permutation matrix is $\begin{bmatrix} 0 & & & 1 \\ 1 & & & \\ & 1 & & \\ & & 1 & 0 \end{bmatrix}$. In fact, any (k,i)-rotation is a k-monotonically increasing permutation.

Monotonically increasing permutations can be composed as stated in Lemma 1.

Lemma 1: If $\sigma_1 \in \mathcal{S}_n$ is a k_1-monotonically increasing permutation and $\sigma_2 \in \mathcal{S}_{k_1} \times \mathcal{S}_{n-k_1}$ a k_2-monotonically increasing permutation with $k_1 < k_2$ then the permutation $\sigma_2 \circ \sigma_1$ is a k_2-monotonically increasing permutation.

Proof: The last $n-k_2$ values of $\sigma_2 \circ \sigma_1$ are the image of a sub-sequence of $n-k_2$ values from the last $n-k_1$ values of σ_1 through the monotonically increasing function σ_2. □

Therefore an iterative algorithm, using rotations as elementary pivot permutations, maintains the property that the permutation matrices P and Q at any step k are k-monotonically increasing. A similar property also applies with recursive algorithms.

5.3 The rank profile matrix

We start by introducing in Theorem 4 the rank profile matrix, that we will use throughout this document to summarize all information on the rank profiles of a matrix. From now on, matrices are over a field $\mathbf{K}$ and a valid pivot is a non-zero element.

Definition 3: An r-sub-permutation matrix is a matrix of rank r with only r non-zero entries equal to one.

Lemma 2: An $m \times n$ r-sub-permutation matrix has at most one non-zero entry per row and per column, and can be written $P \begin{bmatrix} I_r & \\ & 0_{(m-r)\times(n-r)} \end{bmatrix} Q$ where P and Q are permutation matrices.

Theorem 4: Let $A \in \mathrm{K}^{m\times n}$. There exists a unique $m \times n$ $\mathrm{rank}(A)$-sub-permutation matrix $\mathcal{R}_A$ of which every leading sub-matrix has the same rank as the corresponding leading sub-matrix of A. This sub-permutation matrix is called the *rank profile matrix of A*.

Proof: We prove existence by induction on the row dimension of the leading submatrices.

If $A_{1,1..n} = 0_{1\times n}$, setting $\mathcal{R}^{(1)} = 0_{1\times n}$ satisfies the defining condition. Otherwise, let j be the index of the leftmost invertible element in $A_{1,1..n}$ and set $\mathcal{R}^{(1)} = e_j^T$ the j-th n-dimensional row canonical vector, which satisfies the defining condition.

Now for a given $i \in \{1, \ldots, m\}$, suppose that there is a unique $i \times n$ rank profile matrix $\mathcal{R}^{(i)}$ such that $\mathrm{rank}(A_{1..i,1..j}) = \mathrm{rank}(\mathcal{R}_{1..i,1..j})$ for every $j \in \{1..n\}$. If $\mathrm{rank}(A_{1..i+1,1..n}) = \mathrm{rank}(A_{1..i,1..n})$, then $\mathcal{R}^{(i+1)} = \begin{bmatrix} \mathcal{R}^{(i)} \\ 0_{1\times n} \end{bmatrix}$. Otherwise, consider k, the

smallest column index such that $\text{rank}(A_{1..i+1,1..k}) = \text{rank}(A_{1..i,1..k})+1$ and set $\mathcal{R}^{(i+1)} = \begin{bmatrix} \mathcal{R}^{(i)} \\ e_k^T \end{bmatrix}$. Any leading sub-matrix of $\mathcal{R}^{(i+1)}$ has the same rank as the corresponding leading sub-matrix of A: first, for any leading subset of rows and columns with less than i rows, the case is covered by the induction; second define $\begin{bmatrix} B & u \\ v^T & x \end{bmatrix} = A_{1..i+1,1..k}$, where u, v are vectors and x is a scalar. From the definition of k, v is linearly dependent with B and thus any leading sub-matrix of $\begin{bmatrix} B \\ v^T \end{bmatrix}$ has the same rank as the corresponding sub-matrix of $\mathcal{R}^{(i+1)}$. Similarly, from the definition of k, the same reasoning works when considering more than k columns, with a rank increment by 1.
Lastly we show that $\mathcal{R}^{(i+1)}$ is a r_{i+1}-sub-permutation matrix. Indeed, u is linearly dependent with the columns of B: otherwise, $\text{rank}(\begin{bmatrix} B & u \end{bmatrix}) = \text{rank}(B) + 1$. From the definition of k we then have $\text{rank}(\begin{bmatrix} B & u \\ v^T & x \end{bmatrix}) = \textit{rank}(\begin{bmatrix} B & u \end{bmatrix}) + 1 = \text{rank}(B) + 2 = \text{rank}(\begin{bmatrix} B \\ v^T \end{bmatrix}) + 2$ which is a contradiction. Consequently, the k-th column of $\mathcal{R}^{(i)}$ is all zero, and $\mathcal{R}^{(i+1)}$ is a r-sub-permutation matrix.

To prove uniqueness, suppose there exist two distinct rank profile matrices $\mathcal{R}^{(1)}$ and $\mathcal{R}^{(2)}$ for a given matrix A and let (i, j) be some coordinates where $\mathcal{R}^{(1)}_{1..i,1..j} \neq \mathcal{R}^{(2)}_{1..i,1..j}$ and $\mathcal{R}^{(1)}_{1..i-1,1..j-1} = \mathcal{R}^{(2)}_{1..i-1,1..j-1}$. Then, $\text{rank}(A_{1..i,1..j}) = \text{rank}(\mathcal{R}^{(1)}_{1..i,1..j}) \neq \text{rank}(\mathcal{R}^{(2)}_{1..i,1..j}) = \text{rank}(A_{1..i,1..j})$ which is a contradiction. □

Example 3: $A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix}$ has $\mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$ for rank profile matrix over $\mathbb{Q}$.

Remark 2: The matrix E introduced in Malaschonok's LEU decomposition [73, Theorem 1], is in fact the rank profile matrix. There, the existence of this decomposition was only shown for $m = n = 2^k$, and no connection was made to the relation with ranks and rank profiles. Finally, after proving its uniqueness here, we propose this definition as a new matrix normal form.

The rank profile matrix has the following properties:

Lemma 3: Let A be a matrix.
1. $\mathcal{R}_A$ is *diagonal* if A has *generic rank profile*.
2. $\mathcal{R}_A$ is a *permutation* matrix if A is *invertible*
3. $\text{RowRP}(A) = \text{RowSupp}(\mathcal{R}_A)$; $\text{ColRP}(A) = \text{ColSupp}(\mathcal{R}_A)$.

Moreover, for all $1 \leq i \leq m$ and $1 \leq j \leq n$, we have:
4. $\text{RowRP}(A_{1..i,1..j}) = \text{RowSupp}((\mathcal{R}_A)_{1..i,1..j})$
5. $\text{ColRP}(A_{1..i,1..j}) = \text{ColSupp}((\mathcal{R}_A)_{1..i,1..j})$,

These properties show how to recover the row and column rank profiles of A and of any of its leading sub-matrix.

5.4 Algorithms that reveal the Rank Profile Matrix

5.4.1 How to reveal rank profiles

A PLUQ decomposition reveals the row (resp. column) rank profile if it can be read from the first r values of the permutation matrix P (resp. Q). Equivalently, by

Lemma 3, this means that the row (resp. column) support of the pivoting matrix $\Pi_{P,Q}$ equals that of the rank profile matrix.

Definition 4: The decomposition $A = PLUQ$ reveals:
1. the row rank profile if $\text{RowSupp}(\Pi_{P,Q}) = \text{RowSupp}(\mathcal{R}_A)$,
2. the col. rank profile if $\text{ColSupp}(\Pi_{P,Q}) = \text{ColSupp}(\mathcal{R}_A)$,
3. the rank profile matrix if $\Pi_{P,Q} = \mathcal{R}_A$.

Example 4: $A = \begin{bmatrix} 2&0&3&0\\ 1&0&0&0\\ 0&0&4&0\\ 0&2&0&1 \end{bmatrix}$ has $\mathcal{R}_A = \begin{bmatrix} 1&0&0&0\\ 0&0&1&0\\ 0&0&0&0\\ 0&1&0&0 \end{bmatrix}$ for rank profile matrix over $\mathbb{Q}$. Now the pivoting matrix obtained from a PLUQ decomposition with a pivot search operation following the row order (any column, first non-zero row) could be the matrix $\Pi_{P,Q} = \begin{bmatrix} 0&0&1&0\\ 1&0&0&0\\ 0&0&0&0\\ 0&1&0&0 \end{bmatrix}$. As these matrices share the same row support, the matrix $\Pi_{P,Q}$ reveals the row rank profile of A.

Remark 3: Example 4, suggests that a pivot search strategy minimizing row and column indices could be a sufficient condition to recover both row and column rank profiles at the same time, regardless the pivot permutation. However, this is unfortunately not the case. Consider for example a search based on the lexicographic order (first non-zero column of the first non-zero row) with transposition permutations, run on the matrix: $A = \begin{bmatrix} 0&0&1\\ 2&3&0 \end{bmatrix}$. Its rank profile matrix is $\mathcal{R}_A = \begin{bmatrix} 0&0&1\\ 1&0&0 \end{bmatrix}$ whereas the pivoting matrix could be $\Pi_{P,Q} = \begin{bmatrix} 0&0&1\\ 0&1&0 \end{bmatrix}$, which does not reveal the column rank profile. This is due to the fact that the column transposition performed for the first pivot changes the order in which the columns will be inspected in the search for the second pivot.

We will show that if the pivot permutations preserve the order in which the still unprocessed columns or rows appear, then the pivoting matrix will equal the rank profile matrix. This is achieved by the monotonically increasing permutations.

Theorem 5 shows how the ability of a PLUQ decomposition algorithm to recover the rank profile information relates to the use of monotonically increasing permutations. More precisely, it considers an arbitrary step in a PLUQ decomposition where k pivots have been found in the elimination of an $\ell \times p$ leading sub-matrix A_1 of the input matrix A.

Theorem 5: Consider a partial PLUQ decomposition of an $m \times n$ matrix A:

$$A = P_1 \begin{bmatrix} L_1 & \\ M_1 & I_{m-k} \end{bmatrix} \begin{bmatrix} U_1 & V_1 \\ & H \end{bmatrix} Q_1$$

where $\begin{bmatrix} L_1 \\ M_1 \end{bmatrix}$ is $m \times k$ lower triangular and $\begin{bmatrix} U_1 & V_1 \end{bmatrix}$ is $k \times n$ upper triangular, and let A_1 be some $\ell \times p$ leading sub-matrix of A, for $\ell, p \geq k$. Let $H = P_2 L_2 U_2 Q_2$ be a PLUQ decomposition of H. Consider the PLUQ decomposition

$$A = \underbrace{P_1 \begin{bmatrix} I_k & \\ & P_2 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} L_1 & \\ P_2^T M_1 & L_2 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} U_1 & V_1 Q_2^T \\ & U_2 \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} I_k & \\ & Q_2 \end{bmatrix} Q_1}_{Q}.$$

Consider the following clauses:

(i) $\mathrm{RowRP}(A_1) = \mathrm{RowSupp}(\Pi_{P_1,Q_1})$
(ii) $\mathrm{ColRP}(A_1) = \mathrm{ColSupp}(\Pi_{P_1,Q_1})$
(iii) $\mathcal{R}_{A_1} = \Pi_{P_1,Q_1}$
(iv) $\mathrm{RowRP}(H) = \mathrm{RowSupp}(\Pi_{P_2,Q_2})$
(v) $\mathrm{ColRP}(H) = \mathrm{ColSupp}(\Pi_{P_2,Q_2})$
(vi) $\mathcal{R}_H = \Pi_{P_2,Q_2}$
(vii) P_1^T is k-monotonically increasing or (P_1^T is ℓ-monotonically increasing and $p = n$)
(viii) Q_1^T is k-monotonically increasing or (Q_1^T is p-monotonically increasing and $\ell = m$)

Then,

(a) if (i) or (ii) or (iii) then $H = \begin{bmatrix} 0_{(\ell-k)\times(p-k)} & * \\ * & * \end{bmatrix}$
(b) if (vii) then ((i) and (iv)) $\Rightarrow \mathrm{RowRP}(A) = \mathrm{RowSupp}(\Pi_{P,Q})$;
(c) if (viii) then ((ii) and (v)) $\Rightarrow \mathrm{ColRP}(A) = \mathrm{ColSupp}(\Pi_{P,Q})$;
(d) if (vii) and (viii) then (iii) and (vi) $\Rightarrow \mathcal{R}_A = \Pi_{P,Q}$.

Proof: Let $P_1 = \begin{bmatrix} P_{11} & E_1 \end{bmatrix}$ and $Q_1 = \begin{bmatrix} Q_{11} \\ F_1 \end{bmatrix}$ where E_1 is $m \times (m-k)$ and F_1 is $(n-k) \times n$. On one hand we have

$$A = \underbrace{\begin{bmatrix} P_{11} & E_1 \end{bmatrix} \begin{bmatrix} L_1 \\ M_1 \end{bmatrix} \begin{bmatrix} U_1 & V_1 \end{bmatrix} \begin{bmatrix} Q_{11} \\ F_1 \end{bmatrix}}_{B} + E_1 H F_1. \tag{5.1}$$

On the other hand,

$$\begin{aligned} \Pi_{P,Q} &= P_1 \begin{bmatrix} I_k & \\ & P_2 \end{bmatrix} \begin{bmatrix} I_r & \\ & 0_{(m-r)\times(n-r)} \end{bmatrix} \begin{bmatrix} I_k & \\ & Q_2 \end{bmatrix} Q_1 \\ &= P_1 \begin{bmatrix} I_k & \\ & \Pi_{P_2,Q_2} \end{bmatrix} Q_1 = \Pi_{P_1,Q_1} + E_1 \Pi_{P_2,Q_2} F_1. \end{aligned}$$

Let $\overline{A}_1 = \begin{bmatrix} A_1 & 0 \\ 0 & 0_{(m-\ell)\times(n-p)} \end{bmatrix}$ and denote by B_1 the $\ell \times p$ leading sub-matrix of B.

(a) The clause (i) or (ii) or (iii) implies that all k pivots of the partial elimination were found within the $\ell \times p$ sub-matrix A_1. Hence $\mathrm{rank}(A_1) = k$ and we can write $P_1 = \begin{bmatrix} P_{11} & E_1 \\ 0_{(m-\ell)\times k} & \end{bmatrix}$ and $Q_1 = \begin{bmatrix} Q_{11} & 0_{k\times(n-p)} \\ & F_1 \end{bmatrix}$, and the matrix A_1 writes

$$A_1 = \begin{bmatrix} I_\ell & 0 \end{bmatrix} A \begin{bmatrix} I_p \\ 0 \end{bmatrix} = B_1 + \begin{bmatrix} I_\ell & 0 \end{bmatrix} E_1 H F_1 \begin{bmatrix} I_p \\ 0 \end{bmatrix}. \tag{5.2}$$

Now $\mathrm{rank}(B_1) = k$ as a sub-matrix of B of rank k and since

$$\begin{aligned} B_1 &= \begin{bmatrix} P_{11} & \begin{bmatrix} I_\ell & 0 \end{bmatrix} \cdot E_1 \end{bmatrix} \begin{bmatrix} L_1 \\ M_1 \end{bmatrix} \begin{bmatrix} U_1 & V_1 \end{bmatrix} \begin{bmatrix} Q_{11} \\ F_1 \cdot \begin{bmatrix} I_p \\ 0 \end{bmatrix} \end{bmatrix} \\ &= P_{11} L_1 U_1 Q_{11} + \begin{bmatrix} I_\ell & 0 \end{bmatrix} E_1 M_1 \begin{bmatrix} U_1 & V_1 \end{bmatrix} Q_1 \begin{bmatrix} I_p \\ 0 \end{bmatrix} \end{aligned}$$

where the first term, $P_{11} L_1 U_1 Q_{11}$, has rank k and the second term has a disjoint row support.

Finally, consider the term $\begin{bmatrix} I_\ell & 0 \end{bmatrix} E_1 H F_1 \begin{bmatrix} I_p \\ 0 \end{bmatrix}$ of equation (5.2). As its row support is disjoint with that of the pivot rows of B_1, it has to be composed of rows linearly dependent with the pivot rows of B_1 to ensure that $\text{rank}(A_1) = k$. As its column support is disjoint with that of the pivot columns of B_1, we conclude that it must be the zero matrix. Therefore the leading $(\ell - k) \times (p - k)$ sub-matrix of $E_1 H F_1$ is zero.

(b) From (a) we know that $A_1 = B_1$. Thus $\text{RowRP}(B) = \text{RowRP}(A_1)$. Recall that $A = B + E_1 H F_1$. No pivot row of B can be made linearly dependent by adding rows of $E_1 H F_1$, as the column position of the pivot is always zero in the latter matrix. For the same reason, no pivot row of $E_1 H F_1$ can be made linearly dependent by adding rows of B. From (i), the set of pivot rows of B is $\text{RowRP}(A_1)$, which shows that

$$\text{RowRP}(A) = \text{RowRP}(A_1) \cup \text{RowRP}(E_1 H F_1). \tag{5.3}$$

Let $\sigma_{E_1} : \{1..m-k\} \to \{1..m\}$ be the map representing the sub-permutation E_1 (i.e. such that $E_1[\sigma_{E_1}(i), i] = 1 \; \forall i$). If P_1^T is k-monotonically increasing, the matrix E_1 has full column rank and is in column echelon form, which implies that

$$\begin{aligned} \text{RowRP}(E_1 H F_1) &= \sigma_{E_1}(\text{RowRP}(H F_1)) \\ &= \sigma_{E_1}(\text{RowRP}(H)), \end{aligned} \tag{5.4}$$

since F_1 has full row rank. If P_1^T is ℓ monotonically increasing, we can write $E_1 = \begin{bmatrix} E_{11} & E_{12} \end{bmatrix}$, where the $m \times (m - \ell)$ matrix E_{12} is in column echelon form. If $p = n$, the matrix H writes $H = \begin{bmatrix} 0_{(\ell-k)\times(n-k)} \\ H_2 \end{bmatrix}$. Hence we have $E_1 H F_1 = E_{12} H_2 F_1$ which also implies

$$\text{RowRP}(E_1 H F_1) = \sigma_{E_1}(\text{RowRP}(H)).$$

From equation (5.2), the row support of $\Pi_{P,Q}$ is that of $\Pi_{P_1,Q_1} + E_1 \Pi_{P_2,Q_2} F_1$, which is the union of the row support of these two terms as they are disjoint. Under the conditions of point (b), this row support is the union of $\text{RowRP}(A_1)$ and $\sigma_{E_1}(\text{RowRP}(H))$, which is, from (5.4) and (5.3), $\text{RowRP}(A)$.

(c) Similarly as for point (b).

(d) From (a) we have still $A_1 = B_1$. Now since $\text{rank}(B) = \text{rank}(B_1) = \text{rank}(A_1) = k$, there is no other non-zero element in $\mathcal{R}_B$ than those in $\mathcal{R}_{\overline{A}_1}$ and $\mathcal{R}_B = \mathcal{R}_{\overline{A}_1}$. The row and column support of $\mathcal{R}_B$ and that of $E_1 H F_1$ are disjoint. Hence

$$\mathcal{R}_A = \mathcal{R}_{\overline{A}_1} + \mathcal{R}_{E_1 H F_1}. \tag{5.5}$$

If both P_1^T and Q_1^T are k-monotonically increasing, the matrix E_1 is in column echelon form and the matrix F_1 in row echelon form. Consequently, the matrix $E_1 H F_1$ is a copy of the matrix H with k zero-rows and k zero-columns interleaved, which does not impact the linear dependency relations between the non-zero rows and columns. As a consequence

$$\mathcal{R}_{E_1 H F_1} = E_1 \mathcal{R}_H F_1. \tag{5.6}$$

Now if Q_1^T is k-monotonically increasing, P_1^T is ℓ-monotonically increasing and $p = n$, then, using notations of point (b), $E_1 H F_1 = E_{12} H_2 F_1$ where E_{12} is in column echelon form. Thus $\mathcal{R}_{E_1 H F_1} = E_1 \mathcal{R}_H F_1$ for the same reason. The symmetric case where Q_1^T is p-monotonically increasing and $\ell = m$ works similarly. Combining equations (5.2), (5.5) and (5.6) gives $\mathcal{R}_A = \Pi_{P,Q}$.

□

5.4.2 Algorithms for rank profiles

Using Theorem 5, we deduce what rank profile information is revealed by a PLUQ algorithm by the way the Search and the Permutation operations are done. Table 5.2 summarizes these results, and points to instances known in the literature, implementing the corresponding type of elimination. More precisely, we first distinguish in this table the ability to compute the row or column rank profile or the rank profile matrix, but we also indicate whether the resulting PLUQ decomposition preserves the monotonicity of the rows or columns. Indeed some algorithm may compute the rank profile matrix, but break the precedence relation between the linearly dependent rows or columns, making it unusable as a base case for a block algorithm of higher level.

Search	Row Perm.	Col. Perm.	Reveals	Monotonicity	Instance
Row order	Transposition	Transposition	RowRP		[58, 62]
Col. order	Transposition	Transposition	ColRP		[65, 62]
Lexicographic	Transposition	Transposition	RowRP		[82]
	Transposition	Rotation	RowRP, ColRP, $\mathcal{R}$	Col.	here, [40]
	Rotation	Rotation	RowRP, ColRP, $\mathcal{R}$	Row, Col.	here, [40]
Rev. lexico.	Transposition	Transposition	ColRP		[82]
	Rotation	Transposition	RowRP, ColRP, $\mathcal{R}$	Row	here, [40]
	Rotation	Rotation	RowRP, ColRP, $\mathcal{R}$	Row, Col.	here, [40]
Product	Rotation	Transposition	RowRP	Row	here, [40]
	Transposition	Rotation	ColRP	Col	here, [40]
	Rotation	Rotation	RowRP, ColRP, $\mathcal{R}$	Row, Col.	here, [39]

Table 5.2: Pivoting Strategies revealing rank profiles

5.4.2.1 Iterative algorithms

We start with iterative algorithms, where each iteration handles one pivot at a time. Here Theorem 5 is applied with $k = 1$, and the partial elimination represents how one pivot is being treated. The elimination of H is done by induction.

Row and Column order Search

The row order pivot search operation is of the form: *any non-zero element in the first non-zero row*. Each row is inspected in order, and a new row is considered only when the previous row is all zeros. With the notations of Theorem 5, this means that A_1 is the leading $\ell \times n$ sub-matrix of A, where ℓ is the index of the first non-zero row of A. When permutations P_1 and Q_1, moving the pivot from position (ℓ, j) to (k, k) are transpositions, the matrix Π_{P_1,Q_1} is the element $E_{\ell,j}$ of the canonical basis.

Its row rank profile is (ℓ) which is that of the $\ell \times n$ leading sub-matrix A_1. Finally, the permutation P_1 is ℓ-monotonically increasing, and Theorem 5 case (b) can be applied to prove by induction that any such algorithm will reveal the row rank profile: $\text{RowRP}(A) = \text{RowSupp}(\Pi_{P,Q})$. The case of the column order search is similar.

Lexicographic order based pivot search

In this case the Pivot Search operation is of the form: *first non-zero element in the first non-zero row*. The lexicographic order being compatible with the row order, the above results hold when transpositions are used and the row rank profile is revealed. If in addition column rotations are used, $Q_1 = R_{1,j}$ which is 1-monotonically increasing. Now $\Pi_{P_1,Q_1} = E_{\ell,j}$ which is the rank profile matrix of the $\ell \times n$ leading sub-matrix A_1 of A. Theorem 5 case (d) can be applied to prove by induction that any such algorithm will reveal the rank profile matrix: $\mathcal{R}_A = \Pi_{P,Q}$. Lastly, the use of row rotations, ensures that the order of the linearly dependent rows will be preserved as well. Algorithm 5 is an instance of Gaussian elimination with a lexicographic order search and rotations for row and column permutations.

The case of the reverse lexicographic order search is similar. As an example, the algorithm in [82, Algorithm 2.14] is based on a reverse lexicographic order search but with transpositions for the row permutations. Hence it only reveals the column rank profile.

The analysis of sections 5.4.1 and 5.4.2 shows that other pivoting strategies can be used to compute the rank profile matrix, and preserve the monotonicity. We present here a new base case algorithm and its implementation over a finite field that we wrote in the `FFLAS-FFPACK` library[1]. It is based on a lexicographic order search and row and column rotations. Moreover, the schedule of the update operations is that of a Crout elimination, for it reduces the number of modular reductions, as shown in 4.5. Algorithm 5 summarizes this variant.

Algorithm 5 Crout variant of PLUQ with lexicographic search and column rotations

$k \leftarrow 1$
for $i = 1 \dots m$ **do**
  $A_{i,k..n} \leftarrow A_{i,k..n} - A_{i,1..k-1} \times A_{1..k-1,k..n}$
  if $A_{i,k..n} = 0$ **then**
    Loop to next iteration
  end if
  Let $A_{i,s}$ be the left-most non-zero element of row i.
  $A_{i+1..m,s} \leftarrow A_{i+1..m,s} - A_{i+1..m,1..k-1} \times A_{1..k-1,s}$
  $A_{i+1..m,s} \leftarrow A_{i+1..m,s}/A_{i,s}$
  Bring $A_{*,s}$ to $A_{*,k}$ by column rotation
  Bring $A_{i,*}$ to $A_{k,*}$ by row rotation
  $k \leftarrow k + 1$
end for

[1] FFLAS-FFPACK revision 1193, http://linalg.org/projects/fflas-ffpack, linked against OpenBLAS-v0.2.8.

Product order based pivot search

The search here consists in finding any non-zero element $A_{\ell,p}$ such that the $\ell \times p$ leading sub-matrix A_1 of A is all zeros except this coefficient. If the row and column permutations are the rotations $R_{1,\ell}$ and $R_{1,p}$, we have $\Pi_{P_1,Q_1} = E_{\ell,p} = \mathcal{R}_{A_1}$. Theorem 5 case (d) can be applied to prove by induction that any such algorithm will reveal the rank profile matrix: $\mathcal{R}_A = \Pi_{P,Q}$. An instance of such an algorithm is given in algorithm 6. If P_1 (resp. Q_1) is a transposition, then Theorem 5 case (c) (resp. case (b)) applies to show by induction that the columns (resp. row) rank profile is revealed.

Unlike the common Gaussian elimination, where pivots are searched in the whole current row or column, the strategy is here to proceed with an incrementally growing leading sub-matrix. This implies a Z-curve type search scheme, as shown on Figure 5.1. This search strategy is meant to ensure the properties on the rank profile that have been presented in Section 5.3.

In order to perform the correct updates on the remaining parts, when a pivot is found its whole row and column have to be permuted to the current diagonal location, see Figure 5.1. But then, in order to preserve the row and column rank profiles, all the rows and column in between have to be shifted by 1 position.

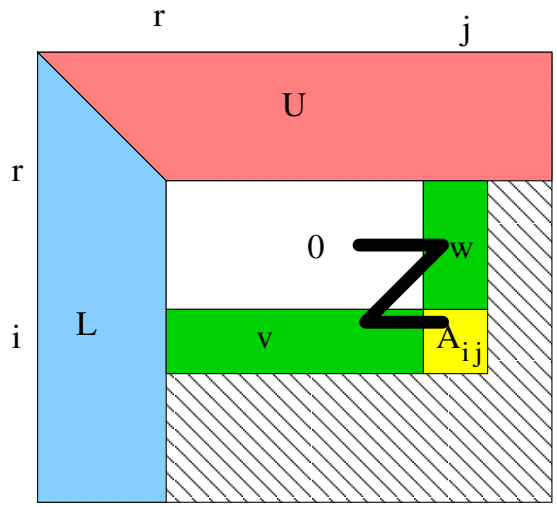

Figure 5.1: Iterative base case PLUQ decomposition

Therefore after the elimination step, the rows and columns of the matrix, as well as the rows of the left permutation matrix and the columns of the right permutation matrix have to be cyclically shifted accordingly. This is presented in the last steps of Algorithm 6, where the notation $A_{*,i>>>_1 j}$ means that in matrix A, columns i through j, both inclusive, have to be shifted by 1 position, cyclically to the right.

Remark 4: Applying the cyclic permutations in steps 22 to 25 may cost in worst case a cubic number of operations. Instead one can delay these permutations and leave the pivots at the position where they were found. These positions are then used to form

Algorithm 6 PLUQ iterative base case

Input: A a $m \times n$ matrix over a field
Output: P, Q: $m \times m$ and $n \times n$ permutation matrices
Output: r: the rank of A
Output: $A \leftarrow \begin{bmatrix} L\backslash U & V \\ M & 0 \end{bmatrix}$ where L is $r \times r$ unit lower triang., U is $r \times r$ upper triang. and such that $A = P \begin{bmatrix} L \\ M \end{bmatrix} \begin{bmatrix} U & V \end{bmatrix} Q$.

1: $r \leftarrow 0; i \leftarrow 0; j \leftarrow 0$
2: **while** $i < m$ or $j < n$ **do**
3: ▷ Let $v = \begin{bmatrix} A_{i,r} & \dots & A_{i,j-1} \end{bmatrix}$ and $w = \begin{bmatrix} A_{r,j} & \dots & A_{i-1,r} \end{bmatrix}^T$
4: **if** $j < n$ and $w \neq 0$ **then**
5: $p \leftarrow$ row index of the first non zero entry in w
6: $q \leftarrow j; j \leftarrow \max(j+1, n)$
7: **else if** $i < m$ and $v \neq 0$ **then**
8: $q \leftarrow$ column index of the first non zero entry in v
9: $p \leftarrow i; i \leftarrow \max(i+1, m)$
10: **else if** $i < m$ and $j < n$ and $A_{i,j} \neq 0$ **then**
11: $(p, q) \leftarrow (i, j)$
12: $i \leftarrow \max(i+1, m); j \leftarrow \max(j+1, n)$
13: **else**
14: $i \leftarrow \max(i+1, m); j \leftarrow \max(j+1, n)$
15: continue
16: **end if** ▷ At this stage, $A_{p,q}$ is a pivot
17: **for** $k = p+1 : n$ **do**
18: $A_{k,q} \leftarrow A_{k,p} A_{p,q}^{-1}$
19: $A_{k,q+1:n} \leftarrow A_{k,q+1:n} - A_{k,q} A_{p,q+1:n}$
20: **end for**
21: ▷ Cyclic shifts of pivot column and row
22: $A_{0:m,r:q} \leftarrow A_{0:m,r>>>_1 q}$
23: $A_{r:p,0:n} \leftarrow A_{r>>>_1 p,0:n}$
24: $P \leftarrow P_{r>>>_1 p,*}$;
25: $Q \leftarrow Q_{*,r>>>_1 q}$
26: $r \leftarrow r+1$
27: **end while**

the matrices P and Q, only after the end of the `while` loop. Then applying these permutations to the current matrix gives the final decomposition $\begin{bmatrix} L\backslash U & V \\ M & 0 \end{bmatrix}$.

Remark 5: In order to further improve the data locality, this iterative algorithm can be transformed into a left-looking variant [27]. Over a finite field, this variant performs fewer modular operations: Step 19 of Algorithm 6 requires a modular reduction after each multiplication while a left-looking variant will delay these reductions within block operations.

Updating Algorihtm 6 with Remarks 4 and 5 would be too technical to be presented here, but this is how we implemented the base case used for the experiments of Section 4.6.

5.4.2.2 Recursive algorithms

A recursive Gaussian elimination algorithm can either split one of the row or column dimension, cutting the matrix in wide or tall rectangular slabs, or split both dimensions, leading to a decomposition into tiles.

Slab recursive algorihtms

When the row dimension is split, this means that the search space for pivots is the whole set of columns, and Theorem 5 applies with $p = n$. This corresponds to either a row or a lexicographic order. From case(b), one shows that, with transpositions, the algorithm recovers the row rank profile, provided that the base case does. If in addition, the elementary column permutations are rotations, then case (d) applies and the rank profile matrix is recovered. Finally, if rows are also permuted by monotonically increasing permutations, then the PLUQ decomposition also respects the monotonicity of the linearly dependent rows and columns. The same reasoning holds when splitting the column dimension.

Tile recursive algorithms

Tile recursive Gaussian elimination algorithms [39, 73, 42] are more involved, especially when dealing with rank deficiencies.

In algorithm 4, the search area A_1 has arbitrary dimensions $\ell \times p$, often specialized as $m/2 \times n/2$. As a consequence, the pivot search can not satisfy neither row, column, lexicographic or reverse lexicographic orders. Now, if the pivots selected in the elimination of A_1 minimizes the product order, then they necessarily also respect this order as pivots of the whole matrix A. Now, from (a), the remaining matrix H writes $H = \begin{bmatrix} 0_{(\ell-k)\times(p-k)} & H_{12} \\ H_{21} & H_{22} \end{bmatrix}$ and its elimination is done by two independent eliminations on the blocks H_{12} and H_{21}, followed by some update of H_{22} and a last elimination on it. Here again, pivots minimizing the row order on H_{21} and H_{12} are also pivots minimizing this order for H, and so are those of the fourth elimination.

Now the block row and column permutations used in Algorithm 4 to form the PLUQ decomposition are r-monotonically increasing. Hence, from case (d), the algorithm computes the rank profile matrix and preserves the monotonicity. If only one of the row or column permutations are rotations, then case (b) or (c) applies to show that either the row or the column rank profile is computed.

5.5 Application of the Rank Profile Matrix

5.5.1 Rank profile matrix based triangularizations

5.5.1.1 LEU decomposition

The LEU decomposition introduced in [73] involves a lower triangular matrix L, an upper triangular matrix U and a r-sub-permutation matrix E.

Theorem 6: Let $A = PLUQ$ be a PLUQ decomposition revealing the rank profile matrix ($\Pi_{P,Q} = \mathcal{R}_A$). Then an LEU decomposition of A with $E = \mathcal{R}_A$ is obtained as

follows (only using row and column permutations):

$$A = \underbrace{P \begin{bmatrix} L & 0_{m\times(n-r)} \end{bmatrix} P^T}_{\overline{L}} \underbrace{P \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} Q}_{E} \underbrace{Q^T \begin{bmatrix} U \\ 0_{(n-r)\times n} \end{bmatrix} Q}_{\overline{U}} \tag{5.7}$$

Proof: First $E = P \begin{bmatrix} I_r \\ 0 \end{bmatrix} Q = \Pi_{P,Q} = \mathcal{R}_A$. Then there only needs to show that $\overline{L}$ is lower triangular and $\overline{U}$ is upper triangular. Suppose that $\overline{L}$ is not lower triangular, let i be the first row index such that $\overline{L}_{i,j} \neq 0$ for some $i < j$. First $j \in \text{RowRP}(A)$ since the non-zero columns in $\overline{L}$ are placed according to the first r values of P. Remarking that $A = P \begin{bmatrix} L & 0_{m\times(n-r)} \end{bmatrix} \begin{bmatrix} U \\ 0 & I_{n-r} \end{bmatrix} Q$, and since right multiplication by a non-singular matrix does not change row rank profiles, we deduce that $\text{RowRP}(\Pi_{P,Q}) = \text{RowRP}(A) = \text{RowRP}(\overline{L})$. If $i \notin \text{RowRP}(A)$, then the i-th row of $\overline{L}$ is linearly dependent with the previous rows, but none of them has a non-zero element in column $j > i$. Hence $i \in \text{RowRP}(A)$.

Let (a, b) be the position of the coefficient $\overline{L}_{i,j}$ in L, that is $a = \sigma_P^{-1}(i), b = \sigma_P^{-1}(j)$. Let also $s = \sigma_Q(a)$ and $t = \sigma_Q(b)$ so that the pivots at diagonal position a and b in L respectively correspond to ones in $\mathcal{R}_A$ at positions (i, s) and (j, t). Consider the $\ell \times p$ leading sub-matrices A_1 of A where $\ell = \max_{x=1..a-1}(\sigma_P(x))$ and $p = \max_{x=1..a-1}(\sigma_Q(x))$. On one hand (j, t) is an index position in A_1 but not (i, s), since otherwise $\text{rank}(A_1) = b$. Therefore, $(i, s) \not\prec_{prod} (j, t)$, and $s > t$ as $i < j$. As coefficients (j, t) and (i, s) are pivots in $\mathcal{R}_A$ and $i < j$ and $t < s$, there can not be a non-zero element above (j, t) at row i when it is chosen as a pivot. Hence $\overline{L}_{i,j} = 0$ and $\overline{L}$ is lower triangular. The same reasoning applies to show that $\overline{U}$ is upper triangular. □

Remark 6: Note that the LEU decomposition with $E = \mathcal{R}_A$ is not unique, even for invertible matrices. As a counter-example, the following decomposition holds for any $a \in \text{K}$:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & -a \\ 0 & 1 \end{bmatrix} \tag{5.8}$$

5.5.1.2 Bruhat decomposition

The Bruhat decomposition, that has inspired Malaschonok's LEU decomposition [73], is another decomposition with a central permutation matrix [13, 51].

Theorem 7 ([13]): Any invertible matrix A can be written as $A = VPU$ for V and U uppper triangular invertible matrices and P a permutation matrix. The latter decomposition is called the *Bruhat decomposition* of A.

It was then naturally extended to singular square matrices by [51]. Corollary 1 generalizes it to matrices with arbitrary dimensions, and relates it to the PLUQ decomposition.

Corollary 1: Any $m \times n$ matrix of rank r has a VPU decomposition, where V and U are upper triangular matrices, and P is a r-sub-permutation matrix.

Proof: Let J_n be the unit anti-diagonal matrix. From the LEU decomposition of J_nA, we have $A = \underbrace{J_nLJ_n}_{V}\underbrace{J_nE}_{P}U$ where V is upper triangular.□

5.5.1.3 Relation to LUP and PLU decompositions

The LUP decomposition $A = LUP$ only exists for matrices with generic row rank profile (including matrices with full row rank). Corollary 2 shows upon which condition the permutation matrix P equals the rank profile matrix $\mathcal{R}_A$. Note that although the rank profile A is trivial in such cases, the matrix $\mathcal{R}_A$ still carries important information on the row and column rank profiles of all leading sub-matrices of A.

Corollary 2: Let A be an $m \times n$ matrix.

If A has generic column rank profile, then any PLU decomposition $A = PLU$ computed using reverse lexicographic order search and row rotations is such that $\mathcal{R}_A = P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$. In particular, $P = \mathcal{R}_A$ if $r = m$.

If A has generic row rank profile, then any LUP decomposition $A = LUP$ computed using lexicographic order search and column rotations is such that $\mathcal{R}_A = \begin{bmatrix} I_r & 0 \end{bmatrix} P$. In particular, $P = \mathcal{R}_A$ if $r = n$.

Proof: Consider A has generic column rank profile. From table 5.2, any `pluq` decomposition algorithm with a reverse lexicographic order based search and rotation based row permutation is such that $\Pi_{P,Q} = P \begin{bmatrix} I_r \end{bmatrix} Q = \mathcal{R}_A$. Since the search follows the reverse lexicographic order and the matrix has generic column rank profile, no column will be permuted in this elimination, and therefore $Q = I_n$. The same reasoning hold for when A has generic row rank profile.□

Note that the L and U factors in a PLU decomposition are uniquely determined by the permutation P. Hence, when the matrix has full row rank, $P = \mathcal{R}_A$ and the decomposition $A = \mathcal{R}_A LU$ is unique. Similarly the decomposition $A = LU\mathcal{R}_A$ is unique when the matrix has full column rank. Now when the matrix is rank deficient with generic row rank profile, there is no longer a unique PLU decomposition revealing the rank profile matrix: any permutation applied to the last $m - r$ columns of P and the last $m - r$ rows of L yields a PLU decomposition where $\mathcal{R}_A = P \begin{bmatrix} I_r \end{bmatrix}$.

Lastly, we remark that the only situation where the rank profile matrix $\mathcal{R}_A$ can be read directly as a sub-matrix of P or Q is as in corollary 2, when the matrix A has generic row or column rank profile. Consider a `pluq` decomposition $A = PLUQ$ revealing the rank profile matrix ($\mathcal{R}_A = P \begin{bmatrix} I_r \end{bmatrix} Q$) such that $\mathcal{R}_A$ is a sub-matrix of P. This means that $P = \mathcal{R}_A + S$ where S has disjoint row and column support with $\mathcal{R}_A$. We have $\mathcal{R}_A = (\mathcal{R}_A + S) \begin{bmatrix} I_r \end{bmatrix} Q = (\mathcal{R}_A + S) \begin{bmatrix} Q_1 \\ 0_{(n-r)\times n} \end{bmatrix}$. Hence $\mathcal{R}_A(I_n - \begin{bmatrix} Q_1 \\ 0_{(n-r)\times n} \end{bmatrix}) = S \begin{bmatrix} Q_1 \\ 0_{(n-r)\times n} \end{bmatrix}$ but the row support of these matrices are disjoint, hence $\mathcal{R}_A \begin{bmatrix} 0 \\ I_{n-r} \end{bmatrix} = 0$ which implies that A has generic column rank profile. Similarly, one shows that $\mathcal{R}_A$ can be a sub-matrix of Q only if A has a generic row rank profile.

5.5.2 Computing Echelon forms

Usual algorithms computing an echelon form [82, 62] use a slab block decomposition (with row or lexicographic order search), which implies that pivots appear in the order

of the echelon form. The column echelon form is simply obtained as $C = PL$ from the PLUQ decomposition. Using product order search, this is no longer true, and the order of the columns in L may not be that of the echelon form. Algorithm 7 shows how to recover the echelon form in such cases. Note that both the row and the column

Algorithm 7 Echelon form from a PLUQ decomposition

Input: P, L, U, Q, a PLUQ decomp. of A with $\mathcal{R}_A = \Pi_{P,Q}$
Output: C: the column echelon form of A

1: $C \leftarrow PL$
2: $(p_1, .., p_r) = \text{Sort}(\sigma_P(1), .., \sigma_P(r))$
3: **for** $i = 1..r$ **do**
4: $\quad \tau = (\sigma_P^{-1}(p_1), .., \sigma_P^{-1}(p_r), r+1, .., m)$
5: **end for**
6: $C \leftarrow CP_\tau$

echelon forms can thus be computed from the same PLUQ decomposition. Lastly, the column echelon form of the $i \times j$ leading sub-matrix, is computed by removing rows of PL below index i and filtering out the pivots of column index greater than j. The latter is achieved by replacing line 2 by $(p_1, .., p_s) = \text{Sort}(\{\sigma_P(i) : \sigma_Q(i) \leq j\})$.

5.6 Conclusion and perspectives

We showed the first reduction to matrix multiplication of the problem of computing both row and column rank profiles of all leading sub-matrices of an input matrix. Second, we introduced a new normal form, the rank profile matrix, and showed how it can be computed from a PLUQ decomposition. As a consequence, our tile recursive PLUQ algorithm gives more information on the matrix while having the best time-complexity compared to sequential state of the art algorithms. It is also faster in practice than previous implementations with large enough matrices. Lastly, it also exhibits more parallelism than classical Gaussian elimination since the recursive calls in step 2 and 3 are independent. The parallel aspects will be detailed in chapter 6 where we prove that this algorithm is also faster than state of the art algorithms in parallel.

Preliminary works are to be done on how to extend the computation of the new normal form on rings and on proving its unicity.

Chapter 6

Parallel computation of Gaussian elimination in exact linear algebra: Synthesis

We now investigate the composition of the finite field linear algebra routines in parallel. We report in this chapter the conclusions of our experience in parallelizing exact LU decomposition on shared memory computers. Moreover, we prove here that since all exact linear algebra problems are reduced to matrix multiplication, we are able to maintain sub-cubic complexities in parallel. For instance, in this chapter we show that LU decomposition reduced to Strassen-Winograd parallel variant of matrix multiplication is asymptotically faster than the state of the art numerical libraries on NUMA-like shared memory architectures. It also maintains high efficiency on any multi-core architecture.

This chapter synthesizes the contributions of this thesis by combining all previous aspects together. Thanks to chapters 4 and 5 we have a tile PLUQ decomposition that computes the rank profile matrix in sub-cubic complexity. Chapter 2 presents the PALADIn language that allows to implement efficiently this sub-cubic factorization in parallel using different runtime systems. Chapter 3 presents optimized parallel building blocks that are the kernel routines of the tile PLUQ factorization and thus helps to make a step forward towards better performance of parallel PLUQ factorization.

This chapter also demonstrates that recursive algorithms are preferred over iterative algorithms when it comes to code composition over exact domains. More precisely, in the case of PLUQ decomposition over finite fields tile recursive variants have better performance than slab recursive variants in parallel.

Section 6.1 details the constraints of composing parallel routines and the parallel models used for this issue. Section 6.2 deals with the parallelization of Gaussian elimination recursive and iterative algorithms. We show there the parallel performance and the parallel implementation for our tile recursive variant and for the recursive slab variant of [62] using the PALADIn language. Lastly section 6.2.3 presents our state of the art benchmarks of parallel PLUQ factorization and gives a comparison with existing state of the art numerical libraries.

6.1 Parallel constraints and code composition

6.1.1 Code composition

Different ways exist to express parallelism for a sequential routine. One can use data parallelism or task parallelism. However, when it comes to library parallelization that involves the composition of parallel routines, task parallelism allows to have a better handling of modules execution and scheduling. Nonetheless, the relative ease of expressing composed parallel routines and nested parallelism using tasks does not always simplify solving performance problems. Exposing enough task parallelism to maximize memory hierarchy utilization while simultaneously minimizing parallelization overheads is crucial for performance. Moreover, over finite fields, tasks need to be as large as possible to reduce the number of modular reductions, as shown in section 4.5. Thus, one can use task parallelism for better handling of code composition but should take into account the trade-off between using sufficiently small tasks to make the best use of available resources and sufficiently large blocks to tackle issues related to computation over finite fields. Task parallelism models are divided following two models: the dataflow model and the fork-join model.

6.1.2 Dataflow model vs fork-join model

The fork-join model sets up parallel sections where execution are launched in parallel at designated points and merge at a subsequent point in the program. The general principal of a dataflow program is that every variable denotes a single value during the execution. This property allows to represent a program with a graph where the nodes correspond to expressions and the edges represent the dependencies between expressions. An expression can be evaluated when all its arguments have been computed.

Here, we compare the tasks execution behavior of the LU factorization using the dataflow model and the fork-join model. Figure 6.1 shows the execution of tasks for the tile iterative LU decomposition. It illustrates the task queue execution of this algorithm with a 3×3 splitting of the matrix. We can see clearly that with the fork-join model we add unnecessary synchronizations. Indeed, the task that computes the LU factorization of the block $A22$ depends only on the task that has computed `fgemm` on this same block. Thus this task can be launched even if the task that executes the `fgemm` routine, for instance on the block A33, hasn't been terminated.

Figure 6.2 shows the dataflow model execution of the tasks during the LU decomposition. The availability of the data triggers the execution of the tasks. Hence execution with dataflow scheduling allows to retrieve the real theoretical critical path, which is the LU decomposition on the diagonal block. With this feature, the number of idle cores is reduced. This helps increasing performances of iterative algorithms since tasks are executed as soon as their data are ready regardless of what the number of iteration is. In recursive algorithms dataflow scheduling is not yet supported between levels of the recursion. Using OpenMP or xKaapi parallel environments, an explicit synchronization is mandatory at the end of a recursive program to ensure that all data are ready before the end of the current recursion level.

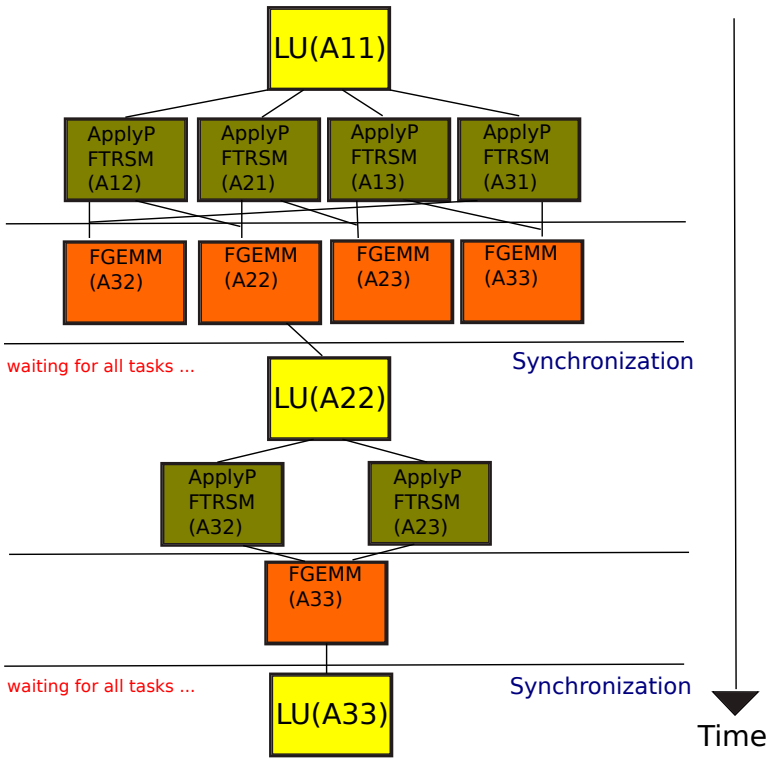

Figure 6.1: Tasks execution using a fork-join model on tile LU factorization with a 3×3 splitting of the matrix.

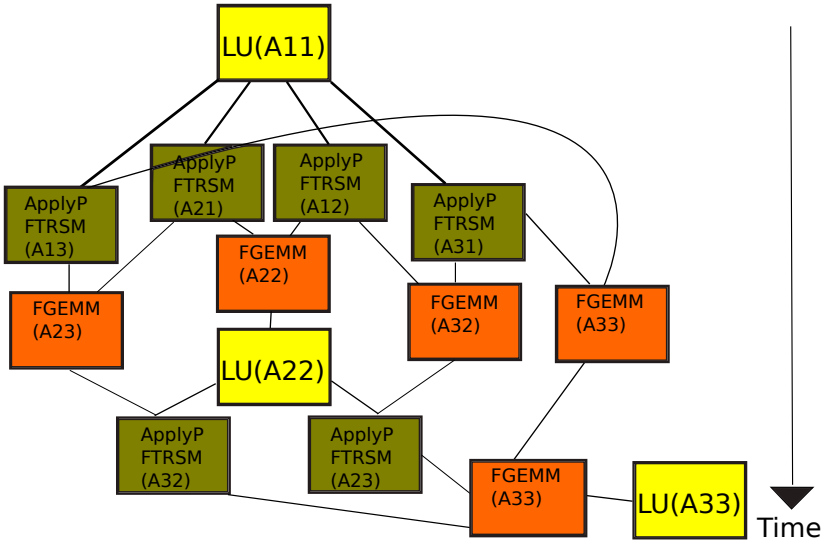

Figure 6.2: Tasks execution using a data-flow model on tile LU factorization with a 3×3 splitting of the matrix.

6.2 Parallel versions of Gaussian elimination algorithms

We focus in this section on the parallelization of the Gaussian elimination iterative and recursive variants. We study first the iterative right looking, left looking and crout block variants in parallel. Then we will focus on tile and slab iterative and recursive algorithms that handle rank deficiencies and compute that rank profile of the matrix in parallel.

6.2.1 Iterative variants

In chapter 4 we studied the cost of modular reductions of the right looking, the left looking and the crout variants. We also presented the block scheme of each variant and showed that the left looking and the crout variants are preferred in sequential execution since they perform fewer modular reductions than the right looking variant. We now study the parallel aspects of each variant.

The tile right looking variant scheme (§4.3.2.1) exhibits more parallelism than the other variants: At each iteration the right looking variant generates more tasks. The latter are concurrent tasks that are dispatched on all available processors. This allows to maximize resource utilization in comparison with the left looking and the crout variant where fewer tasks are to be executed in parallel at each iteration.

We show in figure 6.3 the performance of each variant on full rank matrices and demonstrate that the best variant in sequential is not necessarily the best in parallel. As it happens, the left looking variant that reduces the most modular reductions is seemingly the slowest in parallel. This is explained by the large sequential `ftrsm` calls and few `fgemm` tasks that are executed in parallel in each iteration. The latter reason is shared with the tile crout variant, whereas the tile right looking variant generates more independent `fgemm` tasks and thus provides more tasks to be executed concurrently.

We now consider the general case of matrices with arbitrary rank profile, that can lead to rank deficiencies in the panel eliminations. The slab recursive algorithm of [36] can be translated into a slab iterative algorithm. We present here the parallel aspects of the latter algorithm.

The slab iterative algorithm shown in Figure 6.4 consists in cutting the matrix according to one dimension creating slabs. We recall that the elimination in each slab, called panel factorization, is performed using a sequential algorithm that computes the CUP factorization. In this scheme the sequential CUP tasks constitute the critical path of the task execution queue. In each iteration all tasks become ready to be executed once the CUP task is terminated. At this level only one processor is active and all the others are idle waiting for data to be produced from the CUP call. This imposes a choice of a fine granularity to reduce the size of the sequential CUP tasks. However, during the factorization, the finer is the grain the more modular reductions are performed and the further the benefit of using fast variant is reduced.

Moreover, during the elimination of a rank deficient matrix, the rank obtained on each slab from the sequential CUP call is smaller than the row dimension, as shown in figure 6.4. This adds another difficulty since the starting column position of each panel is determined by the rank of the slabs computed so far. It can only be determined dynamically upon the execution. This implies in particular that no data-storage by

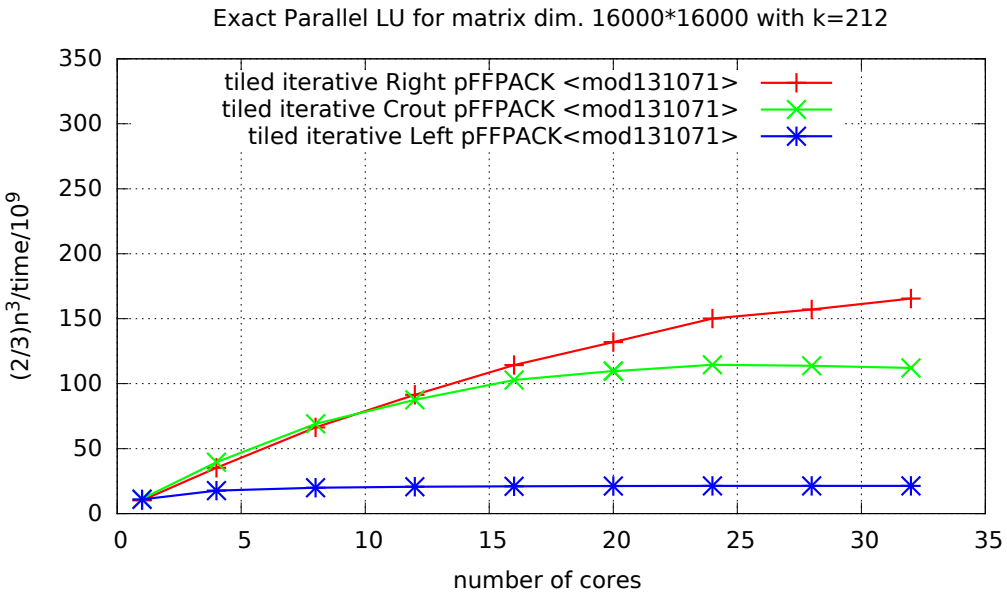

Figure 6.3: Parallel LU factorization on full rank matrices with modular operations

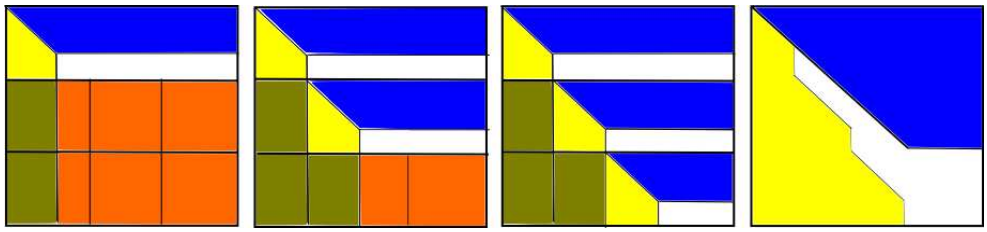

Figure 6.4: Slab iterative factorization of a matrix with rank deficiencies, with final reconstruction of the upper triangular factor

contiguous tiles is possible here. Moreover, the workload of each block operation may strongly vary, depending on the rank of the corresponding slab. Such heterogeneous tasks lead us to opt for work-stealing based runtimes instead of static thread management. So to optimize the performance of this algorithm we need to parallelize the big sequential factorization tasks while preserving the echelon form.

Thanks to the pivoting strategy of algorithm 4, it is still possible to split the panel factorization into column tiles and recover the rank profiles afterwards. Now with this splitting, the operations remain more local and updates can be parallelized. This approach shares similarities with the recursive computation of the panel described in [28]. Figure 6.5 illustrates this tile iterative factorization obtained by the combination of a row-slab iterative algorithm, and a column-slab iterative panel factorization.

This optimization used in the computation of the slab factorization improved the computation speed by a factor of 2, to achieve a speed-up of 6.5 on 32 cores with

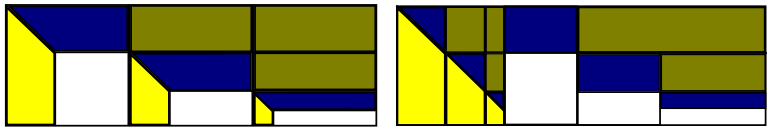

Figure 6.5: Panel PLUQ factorization: tiled sub-calls inside a single slab and final reconstruction

`libkomp`.

6.2.2 Recursive variants

Recursive algorithms in dense linear algebra are a natural choice for hierarchical memory systems [84]. For large problems, the geometric nature of the recursion implies that the total area of operands for recursive algorithms is less than that of iterative algorithms [54]. However, recursive algorithms take less advantage of dataflow models because of the explicit synchronizations that needs to be added at the end of a recursive program. This guarantees that the returned value is computed and ready before the end of the recursion. Nowadays, there is no runtime system that supports the dataflow feature on recursive tasks implementations.

Slab recursive algorithm

The slab recursive variant splits the row dimension or the column dimension in halves in each recursion. As the iterative variant the CUP calls are done in sequential. Figure 6.6 shows the scheme of this algorithm and the corresponding DAG (Directed Acyclic Graph). This variant shares the same issues as the slab iterative variant. This variant of [62] is implemented in the `FFLAS-FFPACK` library as the `LUdivine` routine. We present in appendix B the parallel implementation (`pLUdivine`) of this variant using the PALADIn language.

Tile recursive algorithm

The recursive splitting is done in four quadrants. Pivoting is done first recursively inside each quadrant and then between quadrants. It has the interesting feature that if the top-left tile is rank deficient, then the elimination of the bottom-left and top-right tiles can be executed in parallel as shown on Figure 6.7. This figure shows the DAG of the task queue execution of the tile recursive algorithm. After the first `PPLUQ` recursive call two `pftrsm` calls are executed concurrently and are mapped fairly on available processors. Then three `pfgemm` tasks are executed in parallel on $\frac{p}{3}$ processors each, where p is the total number of disposable processors. Once `pfgemm1` and `pfgemm2` tasks produced their data, respectively `PPLUQ2` and `PPLUQ3` are immediately launched even if the big `pfgemm3` hasn't been terminated.

As an illustration, Algorithm 8 shows the first half of this algorithm implemented with OpenMP tasks with dataflow dependencies. The implementation of all the algorithm using the PALADIn syntax is presented in appendix C.

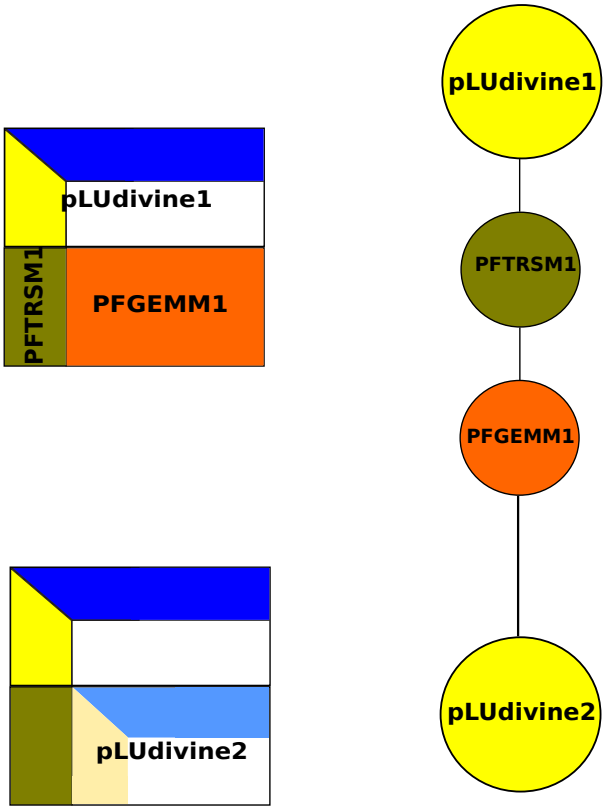

Figure 6.6: Graph of dataflow dependencies inside the tile recursive PLUQ recursion

6.2.3 Parallel experiments on full rank matrices

This section summarizes the parallel contributions of this thesis. All parallel kernel routines implemented in previous chapters are composed and tested within the parallel Gaussian elimination application using the libkomp runtime system. We thus compare all `pfgemm` variants within our parallel tile PLUQ factorization. In our experiments we denote by `explicit synchronization` the classical fork-join model, where synchronizations are explicitly defined by the programmer, e.g. by a `# pragma omp taskwait` instruction. We denote by `dataflow synchronization` the task model where synchronizations are automatically inferred by the scheduler thanks to data dependencies specified by the programmer.

Figure 6.8 shows the parallel performance with `explicit synchronization` of the tile recursive `ppluq` routine using the `libkomp` runtime system with all different `pfgemm` variants of chapter 3. This figure demonstrates that the best performance are obtained using the `TWO_D_ADAPT` variant of the `pfgemm` routine. The number of threads given to all kernel routines is set to the number of available processors (i.e. 32 on the HPAC machine). Note that depending on the number of runs for each variant, variations of 5 to 10 Gfops explain irregularities.

Figure 6.9 compares the parallel behavior of the iterative and recursive `ppluq` routine using `explicit synchronizations` and `dataflow synchronizations`, over fi-

Algorithm 8 `ppluq`(A) tile recursive algorithm

if $\min(m,n) < T$ **then**
  Base Case done by an iterative PLUQ decomposition
end if
Split $A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}$ where A_1 is $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$.

`pluq`(A_1) ▷ Decompose $A_1 = P_1 \begin{bmatrix} L_1 \\ M_1 \end{bmatrix} \begin{bmatrix} U_1 & V_1 \end{bmatrix} Q_1$

`#pragma omp task shared`(A_2, P_1) `depend(in:`P_1`) depend(inout:`A_2`)`

`laswp`(A_2, P_1^T) ▷ $\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \leftarrow P_1^T A_2$

`#pragma omp task shared`(A_3, Q_1) `depend(in:`Q_1`) depend(inout:`A_3`)`
`laswp`(A_3, Q_1^T) ▷ $\begin{bmatrix} C_1 & C_2 \end{bmatrix} \leftarrow A_3 Q_1^T$

Here $A = \left[\begin{array}{cc|c} L_1 \backslash U_1 & V_1 & B_1 \\ M_1 & 0 & B_2 \\ \hline C_1 & C_2 & A_4 \end{array}\right]$.

`#pragma omp task shared`(L_1, B_1) `depend(in:`L_1`) depend(inout:`B_1`)`
`trsm`(L_1, B_1) ▷ $D \leftarrow L_1^{-1} B_1$

`#pragma omp task shared`(U_1, C_1) `depend(in:`U_1`) depend(inout:`C_1`)`
`trsm`(C_1, U_1) ▷ $E \leftarrow C_1 U_1^{-1}$

`#pragma omp task shared`(B_2, M_1, D) `depend(in:`M_1, D`) depend(inout:`B_2`)`
`MM`(B_2, M_1, D) ▷ $F \leftarrow B_2 - M_1 D$

`#pragma omp task shared`(C_2, E, V_1) `depend(in:`E, V_1`) depend(inout:`C_2`)`
`MM`(C_2, E, V_1) ▷ $G \leftarrow BC_2 - EV_1$

`#pragma omp task shared`(A_4, E, D) `depend(in:`E, D`) depend(inout:`A_4`)`
`MM`(A_4, E, D) ▷ $H \leftarrow A_4 - ED$

Here $A = \left[\begin{array}{cc|c} L_1 \backslash U_1 & V_1 & D \\ M_1 & 0 & F \\ \hline E & G & H \end{array}\right]$.

`#pragma omp task shared`(F, P_2, Q_2) `depend(out:`P_2, Q_2`) depend(inout:`F`)`

`pluq`(F) ▷ Decompose $F = P_2 \begin{bmatrix} L_2 \\ M_2 \end{bmatrix} \begin{bmatrix} U_2 & V_2 \end{bmatrix} Q_2$

`#pragma omp task shared`(G, P_3, Q_3) `depend(out:`P_3, Q_3`) depend(inout:`G`)`

`pluq`(G) ▷ Decompose $G = P_3 \begin{bmatrix} L_3 \\ M_3 \end{bmatrix} \begin{bmatrix} U_3 & V_3 \end{bmatrix} Q_3$

`#pragma omp task shared`(P_3, Q_2, H) `depend(in:`P_3, Q_2`) depend(inout:`H`)`

`laswp`(H, P_3^T); `laswp`(H, Q_2^T) ▷ $\begin{bmatrix} H_1 & H_2 \\ H_3 & H_4 \end{bmatrix} \leftarrow P_3^T H Q_2^T$

`#pragma omp task shared`(P_3, E) `depend(in:`P_3`) depend(inout:`E`)`

`laswp`(E, P_3^T) ▷ $\begin{bmatrix} E_1 \\ E_2 \end{bmatrix} \leftarrow P_3^T E$

`#pragma omp task shared`(P_2, M_1) `depend(in:`P_2`) depend(inout:`M_1`)`

`laswp`(M_1, P_2^T) ▷ $\begin{bmatrix} M_{11} \\ M_{12} \end{bmatrix} \leftarrow P_2^T M_1$

`#pragma omp task shared`(D, Q_2) `depend(in:`Q_2`) depend(inout:`D`)`
`laswp`(D, Q_2^T) ▷ $\begin{bmatrix} D_1 & D_2 \end{bmatrix} \leftarrow D Q_2^T$

nite field $\mathbb{Z}/131071\mathbb{Z}$. It first shows how the tile recursive variants performs faster than the tile iterative variants, mostly for their fewer number of modular reductions, and the asymptotic speed-up of Strassen-Winograd algorithm. Now the tile recursive algorithm does not seem to take advantage of the use of tasks with data-flow dependencies, probably because each recursive level has to do an explicit synchronization termination, thus limiting the gain of this approach, whereas the overhead of the task dependency calculation slows down the computation. The tile iterative variants perform slower, but allow for a better use of tasks with data-flow dependency, which perform slightly better there.

Figure 6.10 shows that our tile recursive `ppluq` implementation, without modular reduction, behaves better than the plasma quark `getrf` and is close to the performance of the state of the art MKL `getrf`. The MKL `getrf` is 5% faster than our `ppluq` routine on matrix dimension 32000. Note that in figures 6.10 and 6.9 the performance of our tile `ppluq` is similar with and without modular reductions.

Our highly competitive performance is mainly due to the bi-dimensional cutting

`#pragma omp task shared(`V_1, Q_3`) depend(in:`Q_3`) depend(inout:`V_1`)`
`laswp(`V_1, Q_3^T`)` ▷ $\begin{bmatrix} V_{11} & V_{12} \end{bmatrix} \leftarrow V_1 Q_3^T$

Here $A = \left[\begin{array}{ccc|cc} L_1\backslash U_1 & V_{11} & V_{12} & D_1 & D_2 \\ M_{11} & 0 & 0 & L_2\backslash U_2 & V_2 \\ M_{12} & 0 & 0 & M_2 & 0 \\ \hline E_1 & L_3\backslash U_3 & V_3 & H_1 & H_2 \\ E_2 & M_3 & 0 & H_3 & H_4 \end{array}\right]$.

... (continue the elimination of H following Algorithm 4)

`#pragma omp taskwait`

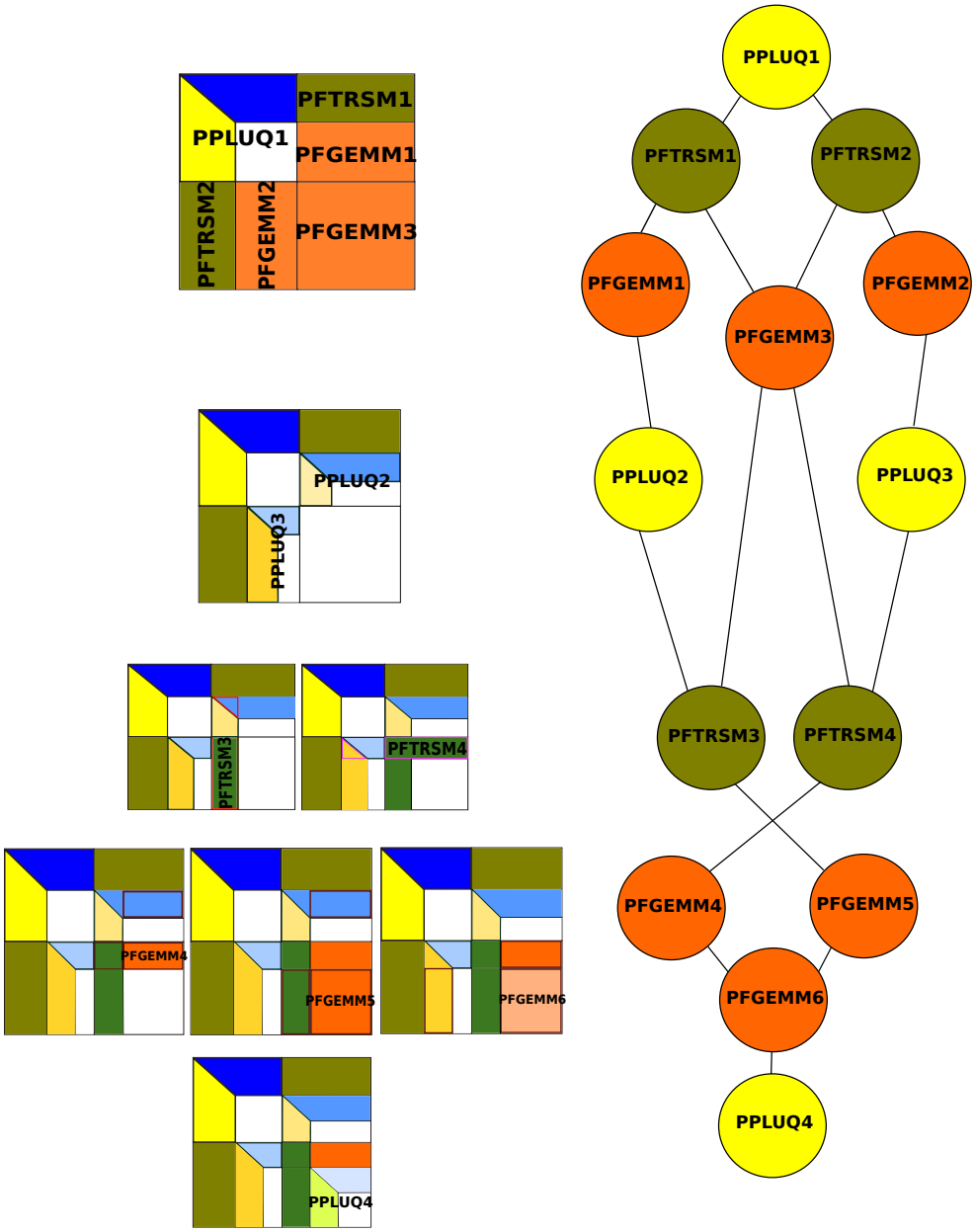

Figure 6.7: Graph of dataflow dependencies inside the tile recursive PLUQ recursion

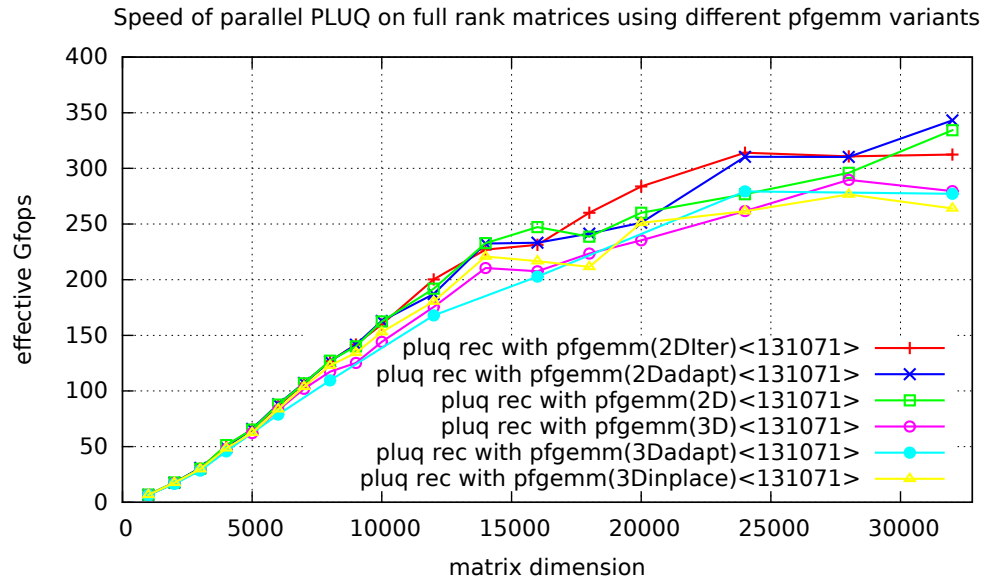

Figure 6.8: Parallel tile recursive PLUQ over $\mathbb{Z}/131071\mathbb{Z}$ on full rank matrices using different pfgemm variants with explicit synchronizations (run on the HPAC machine)

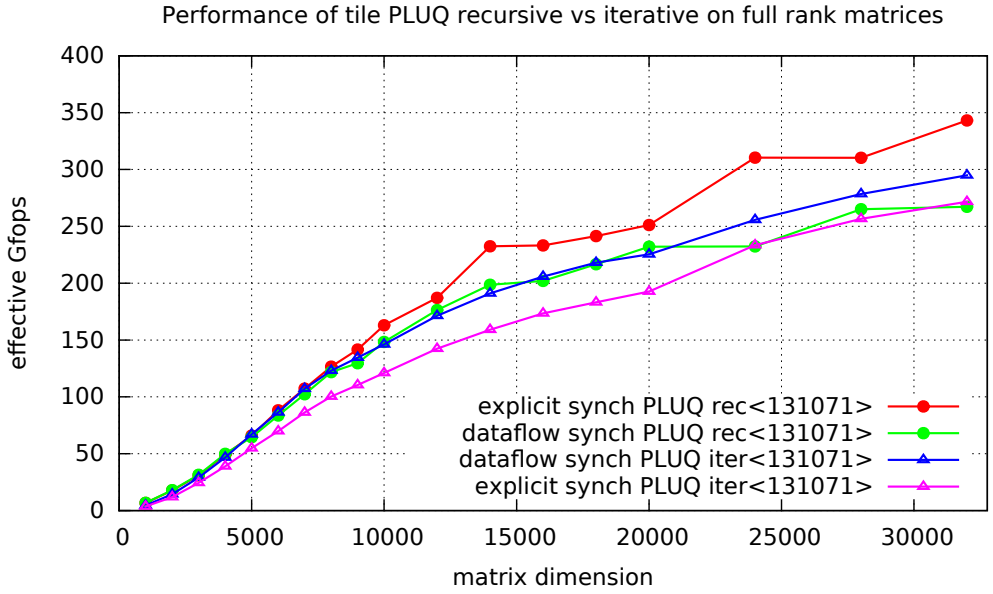

Figure 6.9: Parallel tile recursive and iterative PLUQ over $\mathbb{Z}/131071\mathbb{Z}$ on full rank matrices on 32 cores with and without dataflow synchronizations (run on the HPAC machine)

which allows for a faster panel elimination, parallel iterative/hybrid `pftrsm` kernels, more balanced and adaptive `pfgemm` kernels and some use of Strassen-Winograd algorithm. The use of the latter speeds up computation when the matrix dimension gets larger. The `pfgemm` kernel used here is the classical matrix multiplication algorithm

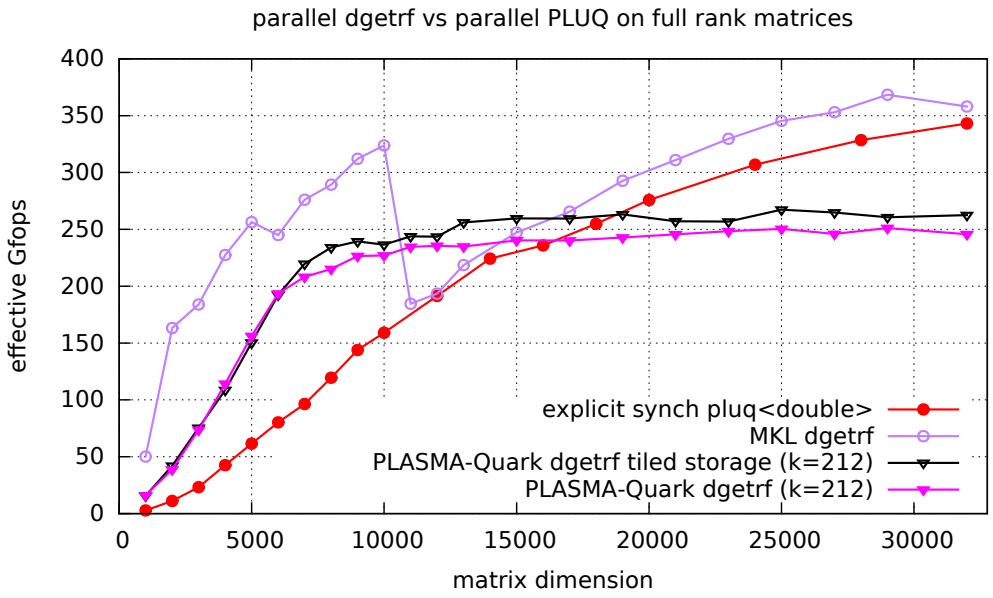

Figure 6.10: Effective Gfops of numerical parallel LU factorization on full rank matrices (run on the HPAC machine)

using the `TWO_D_ADAPT` strategy that switches to a sequential base case algorithm that uses the Strassen-Winograd implementation when the dimension is sufficiently large.

Our tile recursive `ppluq` implementation can attain a speed-up of 18 with matrix dimension 32000 and 14 with matrix dimension 16000 on 32 cores(i.e. on all NUMA nodes) of the HPAC machine), as shown in figure 6.11. We used a threads binding strategy that maps threads in a round and robin manner on all 32 processors. Moreover, the distant memory accesses when using different NUMA nodes impacts the performance of the tile recursive PLUQ algorithm.

To compute our parallel speed-up, we compare with the best sequential implementation of the `pluq` routine that attained 17.3519 Gfops for matrix dimension 32000. The computation speed of our parallel tile PLUQ on 1 core is 17.0638 for matrix dimension 32000.

6.2.4 Parallel experiments on rank deficient matrices

Figure 6.12 shows the execution speed of the parallel PLUQ variants on matrices with rank equal to half their dimension. The tile recursive PLUQ with explicit synchronizations is faster than the tile iterative PLUQ using dataflow synchronizations. Our tile recursive PLUQ algorithm exhibits more parallelism on the rank deficient matrices with two concurrent recursive calls. Note that the speed here is computed using $\frac{\frac{2}{3}r^3+2mnr-r^2}{time}$ where $r = \frac{m}{2}$. Thus in figure 6.12 the speed obtained with matrix dimension 32000 is 300 effective Gfops which corresponds to 63.675 seconds. In figure 6.10, where experiments are conducted on full rank matrices, we achieved a speed of almost 350 effective Gfops but that corresponds to 63.6695 seconds. We thus loose 50

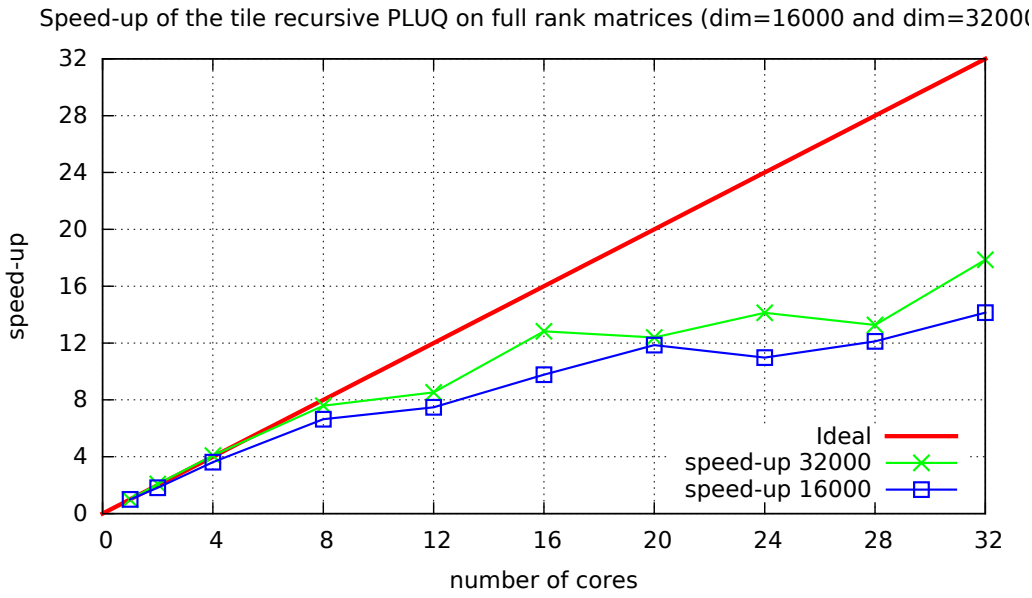

Figure 6.11: Parallel speed-up of the tile recursive PLUQ for matrix dimension = 16000 and 32000 (run on the HPAC machine)

Gfops when the input matrix is rank deficient because we perform more permutations. However, by looking to the timings in seconds of the two figures, the overhead of additional permutations introduced by the rank deficiency is compensated by the reduced amount of arithmetic operations to be performed. Thus, overall the computation speed remains of the same order of magnitude. This time again, the variant with explicit synchronizations performs best.

Using recursive variants linked against the `libkomp` library with explicit synchronizations, mapping data on different NUMA nodes that help reduce dependency on bus speed and fixing large granularity to benefit from the use of fast variants and to reduce modular reductions impact, we manage to obtain high performance for our tile recursive PLUQ factorization.

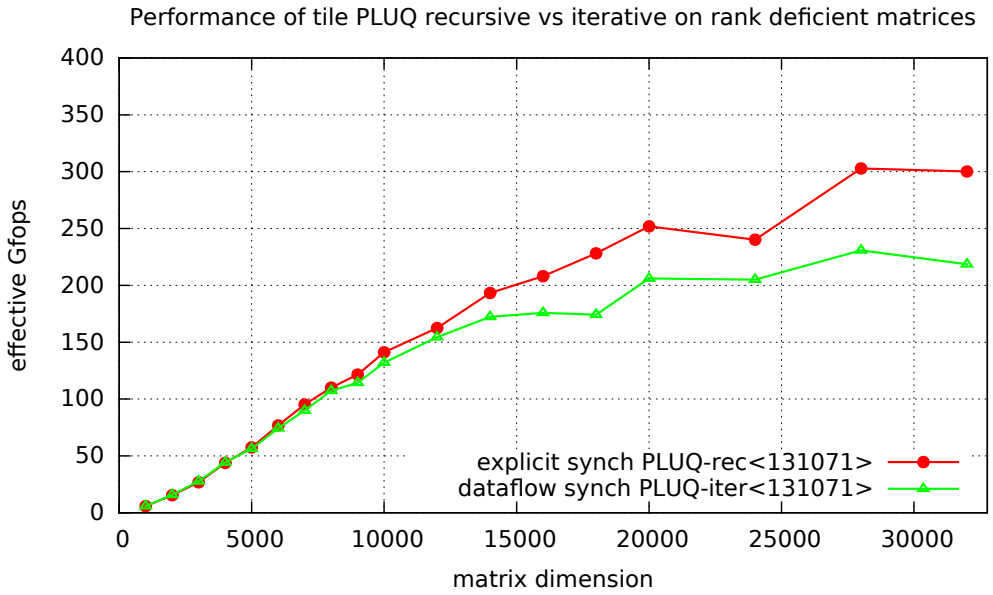

Figure 6.12: Performance of tile recursive PLUQ on 32 cores. Matrices rank is equal to half their dimensions.

Chapter 7

Conclusion

In this thesis, we focused on many aspects related to the design of high performance exact linear algebra software. This study resulted in practical and theoretical contributions. In the following we conclude on these contributions, first by summarizing the results achieved and then by proposing several possible research directions.

7.1 Contributions

In this manuscript, we first presented a domain specific language dedicated to the parallelization of exact linear algebra routines. With a unique syntax the PALADIn interface allows to implement and test our parallel implementations using different parallel environments (OpenMP, TBB and XKaapi). It also provided different matrix cutting strategies over one or two dimensions using iterative and recursive implementations. This interface proved that a generic parallel linear algebra library can be developed using runtime systems as plugins.

This language then helped to study the parallelization of different layers of a parallel exact linear algebra software using different runtime systems. A panorama of block algorithms is thus explored for the parallelization of building blocks routines and for the parallelization of Gaussian elimination routines. Several implementations of the subroutines used by the parallel Gaussian elimination routine (`ppluq`) have been investigated. We identified that the best recursive variant for matrix multiplication is the parallel Strassen-Winograd variant that switches to a classical sequential algorithm. The use of sub-cubic matrix multiplication requires to use a coarse grain parallelization scheme. Hence the data placement strategy need to be adapted consequently and the granularity has to be tuned as close as possible to the available resources. We also proposed a hybrid iterative and recursive parallelization for the triangular system solve with matrix right-hand side, performing efficiently even with unbalanced dimensions.

We then looked into a set of known block algorithms for the computation of Gaussian elimination and proposed a new tile recursive PLUQ factorization. There, we studied the impact of using block algorithms for the computation of Gaussian elimination over a finite field, where modular reductions are involved in a delayed design. We also proposed a new normal form, the rank profile matrix, summarizing all information on the row and column rank profiles of all the leading sub-matrices. We then explored the conditions for a Gaussian elimination algorithm to compute all or part of this invariant,

through the corresponding PLUQ decomposition. These new insights on the relation between pivoting and the ability to recover rank profiles, helped design new algorithms performing more efficiently in sequential and in parallel.

Our parallel building blocks, combined in our new tile recursive PLUQ algorithm deliver a high computing efficiency. The best performance is obtained with the parallel recursive `ppluq` variant using the *2D recursive adaptive* variant for matrix multiplication algorithm and the hybrid parallel `pftrsm` variant. As expected, the use of recursion challenges the runtime, and light-weight task implementations, such as the one in XKaapi happen to be crucial there. Dataflow task dependencies also help slightly improve performances. However, it seems to work best with numerous tasks using iterative algorithms, which in the other hand, implies a finer grain, and therefore a lesser improvement of the sub-cubic matrix multiplication algorithms. Finally, our experimental parallel results performance matches that of the reference numerical state of the art libraries while tackling the various aspects of computing over exact domains in parallel.

Overall, we proved in this thesis that sub-cubic exact linear algebra algorithms scale up in parallel. Moreover, recursive algorithms are good candidates to maintain scalability of sub-cubic algorithms in exact linear algebra. When using appropriate runtime systems, they scale better than iterative algorithms.

7.2 Future work

The `WinoPar`→`ClassicSeq` variant computes $C \leftarrow \alpha A \times B$. However the `ppluq` routine requires an implementation of the $C \leftarrow \beta C + \alpha A \times B$ operation which is not yet implemented for the Strassen-Winograd parallel variant. The number of temporaries in the parallel implementation of the latter operation can increase since no temporary storage is possible in blocks of the matrix C.

Further improvements can be made to the parallelization of the recursive steps of fast matrix multiplication algorithms. The focus will be on the scheduling heuristics that will reduce as much as possible task dependencies, while keeping the memory footprint contained. These scheduling heuristics, inspired by the results of [17], could take into account the constraints of having the critical path as short as possible. Moreover, the Strassen algorithm scheme reveals fewer dependencies than the Winograd variant scheme. This will allow to reduce the number of temporaries but at the cost of more additions. The parallel fast variants could improve further the performance of the `ppluq` routine.

7.3 Perspective

Several research directions can be studied in the continuation of this thesis. We will focus here on four axes:

- The PALADIn language is adapted to express multithreaded parallelism intrinsic to exact linear algebra problems. However, to support large scale exact computations on distributed memory architectures the language need to be re-adapted. This will allow PALADIn to support new parallel environments such as MPI.

Nonetheless, this implies new challenges on keeping the unique syntax feature of PALADIn. A first step can be done by implementing new keywords in PALADIn to express message passing parallelization. Then depending on the application and on the machine architecture the user can set a variable to enable the multithreading mode (for parallelization on shared memory architectures) or the message passing mode (for parallelization on distributed memory architectures) for each PALADIn task.

- The distant data accesses has an impact on the overall performance. Adapting the communication avoiding techniques of [53] in our framework of exact Gaussian elimination is highly relevant. In numerical linear algebra, tournament pivoting strategy has the property that the communication for computing the panel factorization in parallel depends only on the number of processors. A pre-processing step splits the panel in p blocks where p is the number of processors, and computes the LU decomposition on each block to find the best row pivots. These pivots are then permuted into the first positions, and thus the LU factorization of the entire panel can be performed with no pivoting. Over a finite field, tournament pivoting can be adapted to extract the rank profile information on each slab in the framework of our recursive and iterative algorithms. This strategy could compute the union of the non-zero pivots found in concurrent eliminations and reveals the row rank profile or the column rank profile. However, it is still unclear whether the rank profile matrix can still be revealed with such an algorithm.

- The PALADIn language is adapted to parallel computation over dense linear algebra where parallel algorithms efficiency rely on computationally intensive portions. Thus, our language can also be adapted to support computation on hybrid architectures and accelerators. OpenMP standard now includes a set of device constructs to support heterogeneous systems like GPUs. However, for now, only a limited number of GPUs is supported. One can include the OpenCL or/and the CUDA languages to have a rich set of supported devices. However, over a finite field, the best parallel performance are obtained using recursive implementations that involve several levels of parallelism. In the context of GPU programming this implies kernel composition that is only supported via the CUDA dynamic parallelism given that the GPU device supports it. If not, tile iterative algorithms can be exported on available GPU device. Thus, the challenge is on how to automate this choice inside the PALADIn language depending on the available accelerator device feature.

- We proved in this thesis that efficient implementations of parallel building blocks in exact linear algebra exist. But, based on our experience for higher level applications, such as the PLUQ factorization, code composition makes it more difficult to maintain high parallel efficiency. Efforts are to be done on both algorithmic and implementation sides. For instance, the composition of several parallel codes for the computation of Euclidean lattice reduction [76] can be investigated. The latter state of the art algorithms [76, 92] are based on the LLL algorithm where iterations of the QR factorization are involved. Relying on floating point approximations, the length of the iterative sequence of QR factorizations depends on the

precision used. It would be interesting to study how exact rational arithmetic, based on finite field and RNS systems, can be applied. Indeed, this will increase the total arithmetic work but computations are more independent and can be parallelized. Thus, an hybrid version combining numeric and exact computation of the QR factorization using RNS representations for rational multi-precision can be considered. This makes it possible to benefit from parallelization on the precision and on the problem dimension.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012037. URL: http://stacks.iop.org/1742-6596/180/i=1/a=012037.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. H. Bischof, and D. C. Sorensen. "LAPACK: a portable linear algebra library for high-performance computers". In: *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*. 1990, pp. 2–11. DOI: 10.1007/978-3-662-26811-7_3.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*. Vol. 9. Siam, 1999. URL: http://www.netlib.org/lapack/lug/.

[4] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *CoRR* abs/1202.3173 (2012). DOI: 10.1145/2312005.2312044.

[5] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. "Discrete Logarithm in GF(2809) with FFS". In: *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*. 2014, pp. 221–238. DOI: 10.1007/978-3-642-54631-0_13.

[6] A. R. Benson and G. Ballard. "A framework for practical parallel fast matrix multiplication". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. 2015, pp. 42–53. DOI: 10.1145/2688500.2688513.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997. URL: http://www.netlib.org/scalapack/slug/.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: Efficient Multithreaded Computing". PhD thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1996. DOI: 10.1006/jpdc.1996.0107.

[9] O. A. R. Board. *OpenMP Application Program Interface version 3*. 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[10] O. A. R. Board. *OpenMP Application Program Interface version 4*. 2013. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[11] W. Bosma, J. J. Cannon, and C. Playoust. "The Magma Algebra System I: The User Language". In: *J. Symb. Comput.* 24.3/4 (1997), pp. 235–265. DOI: 10.1006/jsco.1996.0125.

[12] W. Bosma, J. Cannon, and C. Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265. DOI: 10.1006/jsco.1996.0125.

[13] N. Bourbaki. *Groupes et Algègres de Lie*. Elements of mathematics Chapters 4–6. Springer, 2008. DOI: 10.1007/978-3-540-34393-6.

[14] B. Boyer, A. Breust, J.-G. Dumas, P. Giorgi, C. Pernet, Z. Sultan, and B. Vialla. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*. v2.1.0. https://github.com/linbox-team/fflas-ffpack. 2014.

[15] B. Boyer, J.-G. Dumas, C. Pernet, P. Giorgi, and B. D. Saunders. *LinBox-1.4.0: Exact Computational Linear Algebra*. URL: https://github.com/linbox-team/linbox.

[16] B. Boyer, J. Dumas, and P. Giorgi. "Exact sparse matrix-vector multiplication on GPU's and multicore architectures". In: *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France*. 2010, pp. 80–88. DOI: 10.1145/1837210.1837224.

[17] B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou. "Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm". In: *34th International Symposium on Symbolic and Algebraic Computation (ISSAC'09)*. Seoul, Republic of Korea: ACM, 2009, pp. 55–62. DOI: 10.1145/1576702.1576713.

[18] F. Broquedis, T. Gautier, and V. Danjean. "libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms". In: *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*. 2012, pp. 102–115. DOI: 10.1007/978-3-642-30961-8_8.

[19] F. Bruhat. "Sur les représentations induites des groupes de Lie". In: *Bulletin de la Société Mathématique de France* 84 (1956), pp. 97–205. URL: http://eudml.org/doc/86911.

[20] J. R. Bunch and J. E. Hopcroft. "Triangular factorization and inversion by fast matrix multiplication". In: *Mathematics of Computation* 28.125 (Jan. 1974), pp. 231–236. DOI: 10.1090/S0025-5718-1974-0331751-8.

[21] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Computing* 35.1 (2009), pp. 38–53. DOI: 10.1016/j.parco.2008.10.002.

[22] Z. Chen and A. Storjohann. "A BLAS Based C Library for Exact Linear Algebra on Integer Matrices". In: *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*. ISSAC '05. Beijing, China: ACM, 2005, pp. 92–99. DOI: 10.1145/1073884.1073899.

[23] F. Dahlgren and J. Torrellas. "Cache-only memory architectures". In: *Computer* 32.6 (June 1999), pp. 72–79. DOI: 10.1109/2.769448.

[24] P. D'alberto, M. Bodrato, and A. Nicolau. "Exploiting Parallelism in Matrix-computation Kernels for Symmetric Multiprocessor Systems: Matrix-multiplication and Matrix-addition Algorithm Optimizations by Software Pipelining and Threads Allocation". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 2:1–2:30. DOI: 10.1145/2049662.2049664.

[25] J. Della Dora. "Sur quelques algorithmes de recherche de valeurs propres". Université : Université scientifique et Médicale de Grenoble. Theses. Université Joseph-Fourier – Grenoble I, July 1973. URL: https://tel.archives-ouvertes.fr/tel-00010274.

[26] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki. "A survey of recent developments in parallel implementations of Gaussian elimination". In: *Concurrency and Computation: Practice and Experience* 27.5 (2015), pp. 1292–1309. DOI: 10.1002/cpe.3306.

[27] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, 1998. DOI: 10.1137/1.9780898719611.

[28] J. J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. "Achieving Numerical Accuracy and High Performance using Recursive Tile LU Factorization". In: *Concurrency and Computation: Practice and Experience* 26.7 (2014), pp. 1408–1431. DOI: 10.1002/cpe.3110. URL: http://hal.inria.fr/hal-00809765.

[29] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix–Matrix Multiply Algorithm". In: *J. Comput. Phys.* 110 (1994), pp. 1–10. DOI: 10.1006/jcph.1994.1001.

[30] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[31] J.-G. Dumas, L. Fousse, and B. Salvy. "Simultaneous modular reduction and Kronecker substitution for small finite fields". In: *Journal of Symbolic Computation* 46.7 (2011). Special Issue in Honour of Keith Geddes on his 60th Birthday, pp. 823–840. DOI: http://dx.doi.org/10.1016/j.jsc.2010.08.015.

[32] J.-G. Dumas, T. Gautier, P. Giorgi, J.-L. Roch, and G. Villard. *Givaro-4.0.1: une bibliothèque C++ pour le Calcul Formel*. Software IMAG-LMC. Oct. 2004. URL: https://github.com/linbox-team/givaro.

[33] J.-G. Dumas, T. Gautier, C. Pernet, J.-L. Roch, and Z. Sultan. "Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination". In: *Parallel Computing* (2015), pages. DOI: http://dx.doi.org/10.1016/j.parco.2015.10.003.

[34] J. Dumas, T. Gautier, C. Pernet, and Z. Sultan. "Parallel Computation of Echelon Forms". In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. 2014, pp. 499–510. DOI: 10.1007/978-3-319-09873-9_42.

[35] J.-G. Dumas, P. Giorgi, and C. Pernet. "FFPACK: Finite Field Linear Algebra Package". In: *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*. ISSAC '04. Santander, Spain: ACM, 2004, pp. 119–126. DOI: 10.1145/1005285.1005304.

[36] J.-G. Dumas, P. Giorgi, and C. Pernet. "Dense Linear Algebra over Word-Size Prime Fields: The FFLAS and FFPACK Packages". In: *ACM Trans. Math. Softw.* 35.3 (Oct. 2008), 19:1–19:42. DOI: 10.1145/1391989.1391992.

[37] J.-G. Dumas and C. Pernet. "Computational linear algebra over finite fields". In: *Handbook of Finite Fields*. Ed. by D. Panario and G. L. Mullen. Chapman & Hall/CRC, 2013. URL: http://hal.archives-ouvertes.fr/hal-00688254.

[38] J.-G. Dumas, C. Pernet, and J.-L. Roch. "Adaptive Triangular System Solving". In: *Challenges in Symbolic Computation Software*. Ed. by W. Decker, M. Dewar, E. Kaltofen, and S. Watt. Dagstuhl Seminar Proceedings 06271. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. URL: http://drops.dagstuhl.de/opus/volltexte/2006/770.

[39] J.-G. Dumas, C. Pernet, and Z. Sultan. "Simultaneous computation of the row and column rank profiles". In: *Proc. ISSAC'13*. Ed. by M. Kauers. ACM Press, 2013. DOI: 10.1145/2465506.2465517.

[40] J.-G. Dumas, C. Pernet, and Z. Sultan. "Computing the Rank Profile Matrix". In: *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*. ISSAC '15. Bath, United Kingdom: ACM, 2015, pp. 149–156. DOI: 10.1145/2755996.2756682.

[41] J.-G. Dumas, C. Pernet, and Z. Wan. "Efficient Computation of the Characteristic Polynomial". In: *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*. ISSAC '05. Beijing, China: ACM, 2005, pp. 140–147. DOI: 10.1145/1073884.1073905.

[42] J.-G. Dumas and J.-L. Roch. "On parallel block algorithms for exact triangularizations". In: *Parallel Computing* 28.11 (Nov. 2002), pp. 1531–1548. DOI: 10.1016/S0167-8191(02)00161-8.

[43] J.-G. Dumas. "LinBox". In: *International Congress on Mathematical Software, ICMS'2006*. Castro Urdiales, Spain, 2006. URL: https://hal.archives-ouvertes.fr/hal-00104044.

[44] J.-C. Faugère. "A new efficient algorithm for computing Gröbner bases (F4)". In: *Journal of Pure and Applied Algebra* 139.1–3 (June 1999), pp. 61–88. DOI: 10.1016/S0022-4049(99)00005-5.

[45] J. v. Gathen and J. Gerhard. *Modern Computer Algebra*. New York, NY, USA: Cambridge University Press, 1999.

[46] T. Gautier, X. Besseron, and L. Pigeon. "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors". In: *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*. 2007, pp. 15–23. DOI: 10.1145/1278177.1278182.

[47] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures". In: *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. 2013, pp. 1299–1308. DOI: 10.1109/IPDPS.2013.66.

[48] T. Gautier, J.-L. Roch, Z. Sultan, and B. Vialla. "Parallel Algebraic Linear Algebra Dedicated Interface". In: *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*. PASCO '15. Bath, United Kingdom: ACM, 2015, pp. 34–43. DOI: 10.1145/2790282.2790286.

[49] G. Golub and C. Van Loan. *Matrix Computations*. third. The Johns Hopkins University Press, 1996.

[50] K. Goto and R. A. v. d. Geijn. "Anatomy of High-performance Matrix Multiplication". In: *ACM Trans. Math. Softw.* 34.3 (May 2008), 12:1–12:25. DOI: 10.1145/1356052.1356053.

[51] D. Y. Grigor'ev. "Analogy of Bruhat decomposition for the closure of a cone of Chevalley group of a classical serie". In: *Soviet Mathematics Doklady* 23.2 (1981), pp. 393–397.

[52] D. Y. Grigor'ev. "Additive complexity in directed computations". In: *Theoretical Computer Science* 19 (1982), pp. 39–67. DOI: http://dx.doi.org/10.1016/0304-3975(82)90014-7.

[53] L. Grigori, J. W. Demmel, and H. Xiang. "CALU: a communication optimal LU factorization algorithm". In: *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011), pp. 1317–1350. DOI: 10.1137/080731992.

[54] F. G. Gustavson. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." In: *IBM Journal of Research and Development* 41.6 (1997), pp. 737–756. DOI: 10.1147/rd.416.0737.

[55] F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kagstrom, and P. Ling. "Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms." In: *PARA*. Ed. by B. Kagstrom, J. Dongarra, E. Elmroth, and J. Wasniewski. Vol. 1541. Lecture Notes in Computer Science. Springer, 1998, pp. 195–206. DOI: 10.1007/BFb0095337.

[56] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. Version 2.4.0, http://flintlib.org. 2013.

[57] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al. "An overview of the Trilinos project". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 397–423.

[58] O. H. Ibarra, S. Moran, and R. Hui. "A Generalization of the Fast LUP Matrix Decomposition Algorithm and Applications". In: *J. of Algorithms* 3.1 (1982), pp. 45–56. DOI: 10.1016/0196-6774(82)90007-4.

[59] Intel. *Intel Math Kernel Library*. 2007. URL: http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[60] *Intel Math Kernel Library. Reference Manual*. ISBN 630813-054US. Santa Clara, USA: Intel Corporation, 2009. URL: https://software.intel.com/en-us/mkl_11.2_ref_pdf.

[61] *Intel Threading Building Blocks*. Santa Clara, USA: Intel Corporation, 2008. URL: https://www.threadingbuildingblocks.org/.

[62] C.-P. Jeannerod, C. Pernet, and A. Storjohann. "Rank-profile revealing Gaussian elimination and the {CUP} matrix decomposition". In: *Journal of Symbolic Computation* 56 (2013), pp. 46–68. DOI: 10.1016/j.jsc.2013.04.004.

[63] D. J. Jeffrey. "LU factoring of non-invertible matrices". In: *ACM Comm. Comp. Algebra* 44.1/2 (July 2010), pp. 1–8. DOI: 10.1145/1838599.1838602.

[64] J. Jelinek and *et al. The GNU OpenMP implementation*. 2014. URL: https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[65] W. Keller-Gehrig. "Fast algorithms for the characteristic polynomial". In: *Th. Comp. Science* 36 (1985), pp. 309–317. DOI: 10.1016/0304-3975(85)90049-0.

[66] T. KelleyC. "Iterative MethodsforLinearand Nonlinear Equations". In: *RaleighN. C.: NorthCarolinaStateUniversity* (1995).

[67] K. Klimkowski and R. A. van de Geijn. "Anatomy of a Parallel Out-of-Core Dense Linear Solver". In: *ICPP*. Vol. 3. Urbana Champaign, Illinois, USA: CRC Press, Aug. 1995, pp. 29–33.

[68] B. Kumar, C.-H. Huang, R. Johnson, and P. Sadayappan. "A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction". In: *Parallel Processing Symposium, 1993., Proceedings of Seventh International*. Apr. 1993, pp. 582–588. DOI: 10.1109/IPPS.1993.262814.

[69] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. "Scheduling dense linear algebra operations on multicore processors". In: *Concurrency and Computation: Practice and Experience* 22.1 (2010), pp. 15–44. DOI: 10.1002/cpe.1467.

[70] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, J. Dongarra, J. Dongarra, M. Faverge, T. Herault, et al. "Multithreading in the PLASMA Library". In: *Multicore Computing: Algorithms, Architectures, and Applications* (2013), p. 119.

[71] F. Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. ISSAC '14. Kobe, Japan: ACM, 2014, pp. 296–303. DOI: 10.1145/2608628.2608664.

[72] *MAGMA library*. URL: http://icl.cs.utk.edu/magma/index.html.

[73] G. I. Malaschonok. "Fast generalized Bruhat decomposition". In: *CASC'10*. Vol. 6244. LNCS. Tsakhkadzor, Armenia: Springer-Verlag, 2010, pp. 194–202. DOI: 10.1007/978-3-642-15274-0_16.

[74] W. Manthey and U. Helmke. "Bruhat canonical form for linear systems". In: *Linear Algebra and its Applications* 425.2–3 (2007). Special Issue in honor of Paul Fuhrmann, pp. 261–282. DOI: 10.1016/j.laa.2007.01.022.

[75] E. Miller and B. Sturmfels. *Combinatorial commutative algebra*. Vol. 227. Springer, 2005. DOI: 10.1007/b138602.

[76] A. Novocin, D. Stehlé, and G. Villard. "An LLL-reduction Algorithm with Quasi-linear Time Complexity: Extended Abstract". In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*. STOC '11. San Jose, California, USA: ACM, 2011, pp. 403–412. DOI: 10.1145/1993636.1993691.

[77] *PLASMA library*. URL: http://icl.cs.utk.edu/plasma/index.html.

[78] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Trans. Math. Softw.* 39.2 (Feb. 2013), 13:1–13:24. DOI: 10.1145/2427023.2427030.

[79] V. Shoup. *A library for Number Theory*. URL: http://www.shoup.net/ntl.

[80] J. G. Siek and A. Lumsdaine. "The matrix template library: A generic programming approach to high performance numerical linear algebra". In: *Computing in Object-Oriented Parallel Environments*. Springer, 1998, pp. 59–70.

[81] W. Stein. *Modular forms, a computational approach*. Graduate studies in mathematics. AMS, 2007. URL: http://wstein.org/books/modform/modform.

[82] A. Storjohann. "Algorithms for Matrix Canonical Forms". PhD thesis. ETH-Zentrum, Zürich, Switzerland, Nov. 2000. DOI: 10.3929/ethz-a-004141007.

[83] V. Strassen. "Gaussian elimination is not optimal". In: *Numer. Math.* 13 (1969), pp. 354–356. DOI: 10.1007/BF02165411.

[84] S. Toledo. "Locality of Reference in LU Decomposition with Partial Pivoting". In: *SIAM J. Matrix Anal. Appl.* 18.4 (Oct. 1997), pp. 1065–1081. DOI: 10.1137/S0895479896297744.

[85] G. Villard. "Calcul formel et parallélisme: résolution de systèmes linéaires". PhD thesis. Institut National Polytechnique de Grenoble-INPG, 1988.

[86] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. "Intel math kernel library". In: *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188. DOI: 10.1007/978-3-319-06486-4.

[87] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 25:1–25:12. DOI: 10.1145/2503210.2503219.

[88] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated empirical optimizations of software and the {ATLAS} project". In: *Parallel Computing* 27.1-2 (2001). New Trends in High Performance Computing, pp. 3–35. DOI: 10.1016/S0167-8191(00)00087-9.

[89] V. V. Williams. "Multiplying matrices faster than Coppersmith-Winograd". In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*. New York, New York, USA: ACM, 2012, pp. 887–898. DOI: 10.1145/2213977.2214056.

[90] S. Winograd. "On multiplication of 2 x 2 matrices". In: *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388. DOI: 10.1016/0024-3795(71)90009-7.

[91] Z. Xianyi, W. Qian, and Z. Yunquan. "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor". In: *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*. ICPADS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97.

[92] C. X.-W., D. Stehlé, and G. Villard. "Perturbation analysis of the QR Factor R in the context of LLL lattice basis reduction." In: (). DOI: http://dx.doi.org/10.1090/S0025-5718-2012-02545-2.

Appendices

A Correctness of Algorithm 4

First note that $S\begin{bmatrix} L \\ M \end{bmatrix} = \left[\begin{array}{ccccc} L_1 & & & & \\ M_{11} & L_2 & & & \\ M_{12} & M_2 & 0 & & \\ \hline E_1 & I & L_3 & & \\ E_{21} & K_1 & M_{31} & L_4 & \\ E_{22} & K_2 & M_{32} & M_4 & 0\,0 \end{array}\right]$

Hence $P\begin{bmatrix} L \\ M \end{bmatrix} = \begin{bmatrix} P_1 & \\ & P_3 \end{bmatrix} \left[\begin{array}{ccc} L_1 & & \\ M_1 & P_2 \begin{bmatrix} L_2 \\ M_2 \end{bmatrix} & \\ \hline E_1 & I & L_3 \\ E_2 & K & M_3 \; P_4 \begin{bmatrix} L_4 \\ M_4 \end{bmatrix} \end{array}\right]$

Similarly, $[U\,V]\,T = \left[\begin{array}{ccc|ccc} U_1 & V_{11} & V_{12} & D_1 & D_{21} & D_{22} \\ & 0 & 0 & U_2 & V_{21} & V_{22} \\ & U_3 & V_3 & 0 & O_1 & O_2 \\ & & & & U_4 & V_4 \\ & & & & & 0 \end{array}\right]$

and $[U\,V]\,Q = \left[\begin{array}{ccc|cc} U_1 & & V_1 & D_1 & D_2 \\ & & 0 & U_2 & V_2 \\ & [U_3\,V_3]\,Q_3 & & 0 & O \\ & & & & [U_4\,V_4]\,Q_4 \end{array}\right] \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}.$

Now as $H_1 = IU_2, H_2 = IV_2 + L_3O, H_3 = KU_2$

and $H_4 = KV_2 + M_3O + P_4 \begin{bmatrix} L_4 \\ M_4 \end{bmatrix} [U_4\, V_4]\, Q_4$ we have

$$
\begin{aligned}
P \begin{bmatrix} L \\ M \end{bmatrix} [U\, V]\, Q &= \begin{bmatrix} P_1 & \\ & P_3 \end{bmatrix} \left[\begin{array}{ccc} L_1 & & \\ M_1\, P_2 \begin{bmatrix} L_2 \\ M_2 \end{bmatrix} & & \\ \hline E_1 & I & L_3 \\ E_2 & K & M_3\, P_4 \begin{bmatrix} L_4 \\ M_4 \end{bmatrix} \end{array}\right] \\
&\quad \left[\begin{array}{ccc|cc} U_1 & & V_1 & D_1 & D_2 \\ & 0 & & U_2 & V_2 \\ & [U_3\, V_3]\, Q_3 & & 0 & O \\ & & & & [U_4\, V_4]\, Q_4 \end{array}\right] \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \\
&= \begin{bmatrix} P_1 & \\ & P_3 \end{bmatrix} \left[\begin{array}{ccc} L_1 & & \\ M_1\, P_2 \begin{bmatrix} L_2 \\ M_2 \end{bmatrix} & & \\ \hline E_1 & & I_{r_3} \\ E_2 & & & I_{m-k-r_3} \end{array}\right] \\
&\quad \left[\begin{array}{ccc|cc} U_1 & & V_1 & D_1 & D_2 \\ & 0 & & U_2 & V_2 \\ & L_3\, [U_3\, V_3]\, Q_3 & & H_1 & H_2 \\ & M_3\, [U_3\, V_3]\, Q_3 & & H_3 & H_4 \end{array}\right] \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \\
&= \begin{bmatrix} P_1 & \\ & I_{m-k} \end{bmatrix} \left[\begin{array}{cc} L_1 & \\ M_1 & \\ \hline E & 0\, I_{m-k} \end{array}\right] \left[\begin{array}{cc|c} U_1 & V_1 & D \\ & 0 & F \\ & G & H \end{array}\right] \\
&\quad \begin{bmatrix} Q_1 & \\ & I_{n-k} \end{bmatrix} \\
&= \begin{bmatrix} P_1 & \\ & I_{m-k} \end{bmatrix} \left[\begin{array}{cc|c} L_1U_1 & L_1V_1 & B_1 \\ M_1U_1 & M_1V_1 & B_2 \\ C_1 & C_2 & A_4 \end{array}\right] \begin{bmatrix} Q_1 & \\ & I_{n-k} \end{bmatrix} \\
&= A
\end{aligned}
$$

B Parallel implementation of the LUdivine Algorithm 3

Listing 1: pLUdivine parallel implementation using PALADIn

```
1
2
3   SYNCH_GROUP(MAX_THREADS,
4   {
5     if (trans == FFLAS::FflasTrans){
6       R = pLUdivine (F, Diag, trans, colDim, Nup, A, lda, P, Q,
7       LuTag, cutoff, nt/2);
8       typename Field::Element_ptr Ar = A + Nup*incRow;    // SW
9       typename Field::Element_ptr Ac = A + R*incCol;      // NE
10      typename Field::Element_ptr An = Ar+ R*incCol;      // SE
11      if (!R){
12        if (LuTag == FFPACK::FfpackSingular )
13        return 0;
14      }
15      else {
16        TASK(MODE(READ(P,R)
17                  CONSTREFERENCE(F, P, Ar)
18                  READWRITE(Ar[0]) ),
19             FFPACK::applyP (F, FFLAS::FflasLeft, FFLAS::
                   FflasNoTrans,Ndown, 0,(int) R, Ar, lda, P);
20        );
21        CHECK_DEPENDENCIES;
22        // Ar <- L1^-1 Ar
23        TASK(MODE(READ(A[0])
24                  CONSTREFERENCE(F, A, Ar)
25                  READWRITE(Ar[0])),
26             FFLAS::ftrsm( F, FFLAS::FflasLeft, FFLAS::FflasLower
                   , FFLAS::FflasNoTrans, Diag, R, Ndown, F.one, A,
                    lda, Ar, lda, PH);
27        );
28        CHECK_DEPENDENCIES;
29        // An <- An - Ac*Ar
30        if (colDim>R)
31        TASK(MODE(READ(Ac[0], Ar[0])
32                  CONSTREFERENCE(F)
33                  READWRITE(An[0])),
34             fgemm( F, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans,
                   colDim-R, Ndown, R, F.mOne, Ac, lda, Ar, lda, F.
                   one, An, lda, pWH);
35        );
36        CHECK_DEPENDENCIES;
37      }
38      // Recursive call on SE
39      TASK(MODE(READ(lda)
40                CONSTREFERENCE(F, P, Q, R2, An)
```

```
41                  READWRITE(An[0], P, Q)
42                  WRITE(R2, x1[0])),
43              R2 = pLUdivine (F, Diag, trans, colDim-R, Ndown, An,
                    lda, P + R, Q + Nup, LuTag, cutoff, nt/2);
44        );
45        for (size_t i = R; i < R + R2; ++i)
46        P[i] += R;
47        if (R2) {
48          // An <- An.P2
49          TASK(MODE(READ(P, R, R2, x1[0])
50                    CONSTREFERENCE(F, A, P)
51                    READWRITE(A[0])),
52                FFPACK::applyP (F, FFLAS::FflasLeft, FFLAS::
                      FflasNoTrans,
53          Nup,(int) R, (int)(R+R2), A, lda, P);
54          );
55        }
56        else {
57          if (LuTag == FFPACK::FfpackSingular)
58          return 0;
59        }
60      }
61      else { // trans == FFLAS::FflasNoTrans
62
63        R = pLUdivine (F, Diag, trans, Nup, colDim, A, lda, P, Q,
             LuTag, cutoff, nt/2);
64        typename Field::Element_ptr Ar = A + Nup*incRow;    // SW
65        typename Field::Element_ptr Ac = A + R*incCol;      // NE
66        typename Field::Element_ptr An = Ar+ R*incCol;      // SE
67        if (!R){
68          if (LuTag == FFPACK::FfpackSingular )
69          return 0;
70        }
71        else { /*  R>0 */
72          // Ar <- Ar.P
73          TASK(MODE(READ(P, R) CONSTREFERENCE(F, P, R, Ar)
                READWRITE(Ar[0])),
74          FFPACK::applyP (F, FFLAS::FflasRight, FFLAS::FflasTrans,
75          Ndown, 0,(int) R, Ar, lda, P);
76          );
77          CHECK_DEPENDENCIES;
78          // Ar <- Ar.U1^-1
79          TASK(MODE(READ(A[0]) CONSTREFERENCE(F, A, Ar, R)
                READWRITE(Ar[0])),
80          ftrsm( F, FFLAS::FflasRight, FFLAS::FflasUpper,
81          FFLAS::FflasNoTrans, Diag, Ndown, R,
82          F.one, A, lda, Ar, lda, PH);
83          );
84          CHECK_DEPENDENCIES;
```

```
85          // An <- An - Ar*Ac
86          if (colDim>R)
87          TASK(MODE(READ(Ac[0], Ar[0]) CONSTREFERENCE(F) READWRITE(
                An[0])),
88          fgemm( F, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, Ndown
                , colDim-R, R,
89          F.mOne, Ar, lda, Ac, lda, F.one, An, lda , pWH);
90          );
91          CHECK_DEPENDENCIES;
92
93        }
94        // Recursive call on SE
95        TASK(MODE(READ(lda, R, Nup) CONSTREFERENCE(F, P, Q, R2, An)
              READWRITE(An[0], P, Q) WRITE(R2, x1[0])),
96        R2=pLUdivine (F, Diag, trans, Ndown, N-R, An, lda, P+R, Q+
              Nup, LuTag, cutoff, nt/2);
97        for (size_t i = R; i < R + R2; ++i)
98        P[i] += R;
99        );
100       CHECK_DEPENDENCIES;
101
102
103       if (R2){
104
105         // An <- An.P2
106         FFPACK::applyP (F, FFLAS::FflasRight, FFLAS::FflasTrans,
107         Nup,(int) R, (int)(R+R2), A, lda, P);
108
109       }
110       else{
111         if (LuTag == FFPACK::FfpackSingular)
112         return 0;
113       }
114
115     }
116
117     // Non zero row permutations
118     for (size_t i = Nup; i < Nup + R2; i++)
119     Q[i] += Nup;
120     if (R < Nup){
121       // Permutation of the 0 rows
122       if (Diag == FFLAS::FflasNonUnit){
123         for ( size_t i = Nup, j = R ; i < Nup + R2; ++i, ++j){
124           //    TASK(MODE(READ(A, x2[0]) CONSTREFERENCE(F, Q, A)
                    READWRITE(Q) ),
125           FFLAS::fassign( F, colDim - j, A + i*incRow + j*incCol,
                  incCol, A + j * (lda + 1), incCol);
126
```

```
127             for (typename Field::Element_ptr Ai = A + i*incRow + j*
                    incCol;
128             Ai != A + i*incRow + colDim*incCol; Ai+=incCol)
129             F.assign (*Ai, F.zero);
130             size_t t = Q[j];
131             Q[j]=Q[i];
132             Q[i] = t;
133           }
134         }
135         else { // Diag == FFLAS::FflasUnit
136           for ( size_t i = Nup, j = R+1 ; i < Nup + R2; ++i, ++j){
137             //                          TASK(MODE(READ(A, x2[0])
                    CONSTREFERENCE(F, Q, A) READWRITE(Q) ),
138             FFLAS::fassign( F, colDim - j,
139             A + i*incRow + j*incCol, incCol,
140             A + (j-1)*incRow + j*incCol, incCol);
141
142             for (typename Field::Element_ptr Ai = A + i*incRow + j*
                    incCol;
143             Ai != A + i*incRow + colDim*incCol; Ai+=incCol)
144             F.assign (*Ai, F.zero);
145             size_t t = Q[j-1];
146             Q[j-1]=Q[i];
147             Q[i] = t;
148           }
149         }             }
150   });
151   return R + R2;
```

C Parallel implementation of the pluq Algorithm 4 using the PALADIn syntax

Listing 2: ppluq parallel implementation using PALADIn

```
1
2   FFLAS::FFLAS_DIAG OppDiag = (Diag == FFLAS::FflasUnit)? FFLAS::
        FflasNonUnit : FFLAS::FflasUnit;
3
4   size_t M2 = M >> 1;
5   size_t N2 = N >> 1;
6   size_t * P1 = FFLAS::fflas_new<size_t> (M2);
7   size_t * Q1 = FFLAS::fflas_new<size_t> (N2);
8   size_t* MathP = 0;
9   size_t* MathQ = 0;
10  size_t* P2,*P3,*Q2,*Q3,*P4,*Q4;
11  size_t R1,R2,R3,R4;
12
13  // A1 = P1 [ L1 ] [ U1 V1 ] Q1
14  //         [ M1 ]
15  R1 = pPLUQ (Fi, Diag, M2, N2, A, lda, P1, Q1,nt);
16
17  typename Field::Element * A2 = A + N2;
18  typename Field::Element * A3 = A + M2*lda;
19  typename Field::Element * A4 = A3 + N2;
20  typename Field::Element * F = A2 + R1*lda;
21  typename Field::Element * G = A3 + R1;
22
23  typedef FFLAS::StrategyParameter::TwoDAdaptive twoda;
24  typedef FFLAS::CuttingStrategy::Recursive rec;
25  // Helper for pfgemm calls
26  typename FFLAS::ParSeqHelper::Parallel<FFLAS::CuttingStrategy::
        Recursive,FFLAS::StrategyParameter::TwoDAdaptive> pWH (std
        ::max(nt,1));
27  // helper for pftrsm calls
28  typename FFLAS::ParSeqHelper::Parallel<FFLAS::CuttingStrategy::
        Block,FFLAS::StrategyParameter::Threads> PH (std::max(nt,1)
        );
29
30  SYNCH_GROUP(
31
32  // [ B1 ] <- P1^T A2
33  // [ B2 ]
34  TASK(MODE(READ(P1) CONSTREFERENCE(Fi, P1, A2) READWRITE(A2[0]))
        ,
35  { papplyP( Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, N-N2, 0,
        M2, A2, lda, P1); }
36  );
37  // [ C1 C2 ] <- A3 Q1^T
```

```
38    TASK(MODE(READ(Q1) CONSTREFERENCE(Fi, Q1, A3) READWRITE(A3[0]))
          ,
39    papplyP( Fi, FFLAS::FflasRight, FFLAS::FflasTrans, M-M2, 0, N2,
          A3, lda, Q1););
40
41    CHECK_DEPENDENCIES;
42    // D <- L1^-1 B1
43    TASK(MODE(READ(A[0], R1, PH) CONSTREFERENCE(Fi, PH, A2)
          READWRITE(A2[0])),
44    ftrsm( Fi, FFLAS::FflasLeft, FFLAS::FflasLower, FFLAS::
          FflasNoTrans, OppDiag, R1, N-N2, Fi.one, A, lda, A2, lda,
          PH));
45
46    // E <- C1 U1^-1
47    TASK(MODE(READ(R1, A[0], PH) CONSTREFERENCE(A3, Fi, M2, R1, PH)
          READWRITE(A3[0])),
48    ftrsm(Fi, FFLAS::FflasRight, FFLAS::FflasUpper, FFLAS::
          FflasNoTrans, Diag, M-M2, R1, Fi.one, A, lda, A3, lda,  PH)
          );
49
50    CHECK_DEPENDENCIES;
51
52    // F <- B2 - M1 D
53    TASK(MODE(READ(A2[0], A[R1*lda], pWH) READWRITE(F[0])
          CONSTREFERENCE(A, A2, F, pWH, Fi)),
54    fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M2-R1, N-
          N2, R1, Fi.mOne, A + R1*lda, lda, A2, lda, Fi.one, F, lda,
          pWH));
55
56    // G <- C2 - E V1
57    TASK(MODE(READ(R1, A[R1], A3[0], pWH) READWRITE(G[0])
          CONSTREFERENCE(Fi, A, A3, G, pWH)),
58    fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M-M2, N2-
          R1, R1, Fi.mOne, A3, lda, A+R1, lda, Fi.one, G, lda, pWH));
59
60    CHECK_DEPENDENCIES;
61
62    P2 = FFLAS::fflas_new<size_t>(M2-R1);
63    Q2 = FFLAS::fflas_new<size_t>(N-N2);
64
65    // F = P2 [ L2 ] [ U2 V2 ] Q2
66    //        [ M2 ]
67    TASK(MODE(CONSTREFERENCE(Fi, P2, Q2, F,/* A4R2,*/ R2) WRITE(R2
          /*, A4R2[0]*/) READWRITE(F[0], P2, Q2) ),
68    R2 = pPLUQ( Fi, Diag, M2-R1, N-N2, F, lda, P2, Q2,nt/2)
69    );
70
71    P3 = FFLAS::fflas_new<size_t>(M-M2);
72    Q3 = FFLAS::fflas_new<size_t>(N2-R1);
```

```
73  // G = P3 [ L3 ] [ U3 V3 ] Q3
74  //        [ M3 ]
75  TASK(MODE(CONSTREFERENCE(Fi, G, Q3, P3, R3) WRITE(R3, P3, Q3)
        READWRITE(G[0])),
76  R3 = pPLUQ( Fi, Diag, M-M2, N2-R1, G, lda, P3, Q3,nt/2));
77
78  // H <- A4 - ED
79  TASK(MODE(CONSTREFERENCE(Fi, A3, A2, A4, pWH) READ(M2, N2, R1,
        A3[0], A2[0]) READWRITE(A4[0])),
80  fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M-M2, N-N2
        , R1, Fi.mOne, A3, lda, A2, lda, Fi.one, A4, lda, pWH));
81
82  CHECK_DEPENDENCIES;
83
84  // [ H1 H2 ] <- P3^T H Q2^T
85  // [ H3 H4 ]
86  TASK(MODE(READ(P3, Q2) CONSTREFERENCE(Fi, A4, Q2, P3) READWRITE
        (A4[0])),
87  papplyP( Fi, FFLAS::FflasRight, FFLAS::FflasTrans, M-M2, 0, N-
        N2, A4, lda, Q2);
88  papplyP( Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, N-N2, 0, M-
        M2, A4, lda, P3););
89
90  CHECK_DEPENDENCIES;
91  // [ E1 ] <- P3^T E
92  // [ E2 ]
93  TASK(MODE(READ(P3) CONSTREFERENCE(Fi, P3, A3) READWRITE(A3[0]))
        ,
94  papplyP( Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, R1, 0, M-M2
        , A3, lda, P3));
95  // [ M11 ] <- P2^T M1
96  // [ M12 ]
97  TASK(MODE(READ(P2) CONSTREFERENCE(P2, A, Fi) READWRITE(A[R1*lda
        ])),
98  papplyP(Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, R1, 0, M2-R1
        , A+R1*lda, lda, P2));
99
100 // [ D1 D2 ] <- D Q2^T
101 TASK(MODE(READ(Q2) CONSTREFERENCE(Fi, Q2, A2) READWRITE(A2[0]))
        ,
102 papplyP( Fi, FFLAS::FflasRight, FFLAS::FflasTrans, R1, 0, N-N2,
         A2, lda, Q2));
103
104 // [ V1 V2 ] <- V1 Q3^T
105 TASK(MODE(READ(Q3) CONSTREFERENCE(Fi, Q3, A) READWRITE(A[R1])),
106 papplyP( Fi, FFLAS::FflasRight, FFLAS::FflasTrans, R1, 0, N2-R1
        , A+R1, lda, Q3));
107 // I <- H1 U2^-1
108 // K <- H3 U2^-1
```

```
109   TASK(MODE(READ(R2, F[0], P2) CONSTREFERENCE(Fi, A4, F, PH, R2)
          READWRITE(A4[0])),
110   ftrsm( Fi, FFLAS::FflasRight, FFLAS::FflasUpper, FFLAS::
          FflasNoTrans, Diag, M-M2, R2, Fi.one, F, lda, A4, lda, PH))
          ;
111   CHECK_DEPENDENCIES;
112   typename Field::Element_ptr temp = 0;
113
114   TASK(MODE(READ(A4[0], R3, P2) READWRITE(temp[0], R2)
          CONSTREFERENCE(Fi, A4, temp, R2, R3)),
115   temp = FFLAS::fflas_new (Fi, R3, R2);
116   FFLAS::fassign (Fi, R3, R2, A4, lda, temp, R2);
117   );
118   CHECK_DEPENDENCIES;
119
120   // J <- L3^-1 I (in a temp)
121   TASK(MODE(READ(R2, R3, G[0]) CONSTREFERENCE(Fi, G, temp, R2, R3
          , PH) READWRITE(temp[0])),
122   ftrsm( Fi, FFLAS::FflasLeft, FFLAS::FflasLower, FFLAS::
          FflasNoTrans, OppDiag, R3, R2, Fi.one, G, lda, temp, R2, PH
          ););
123
124   // N <- L3^-1 H2
125   TASK(MODE(READ(R3, R2, G[0]) CONSTREFERENCE(Fi, G, A4, R3, R2,
          PH) READWRITE(A4[R2])),
126   ftrsm(Fi, FFLAS::FflasLeft, FFLAS::FflasLower, FFLAS::
          FflasNoTrans, OppDiag, R3, N-N2-R2, Fi.one, G, lda, A4+R2,
          lda, PH));
127
128   CHECK_DEPENDENCIES;
129
130   // O <- N - J V2
131   TASK(MODE(READ(R2, F[R2]) CONSTREFERENCE(Fi, R2, A4, R3, temp,
          pWH) READWRITE(A4[R2], temp[0])),
132   fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, R3, N-N2-
          R2, R2, Fi.mOne, temp, R2, F+R2, lda, Fi.one, A4+R2, lda,
          pWH);
133   FFLAS::fflas_delete (temp);
134   temp=0;
135   );
136
137   typename Field::Element_ptr R = 0;
138   // R <- H4 - K V2
139   TASK(MODE(READ(R2, R3, M2, N2, A4[R3*lda], F[R2])
          CONSTREFERENCE(Fi, R, F, R2, R3, pWH) READWRITE(R[0])),
140   R = A4 + R2 + R3*lda;
141   fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M-M2-R3, N
          -N2-R2, R2, Fi.mOne, A4+R3*lda, lda, F+R2, lda, Fi.one, R,
          lda, pWH)
```

```
142   );
143   CHECK_DEPENDENCIES;
144
145   // R <- R - M3 0
146   TASK(MODE(READ(R3, R2, A4[R2], G[R3*lda]) CONSTREFERENCE(Fi, A4
          , R, R3, R2, G, pWH) READWRITE(R[0])),
147   fgemm( Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M-M2-R3, N
          -N2-R2, R3, Fi.mOne, G+R3*lda, lda, A4+R2, lda, Fi.one, R,
          lda, pWH));
148   CHECK_DEPENDENCIES;
149   // H4 = P4 [ L4 ] [ U4 V4 ] Q4
150   //         [ M4 ]
151   TASK(MODE(CONSTREFERENCE(Fi, R4, R, P4, Q4, R2, R3, M2, N2)
          READWRITE(R[0]) WRITE(R4, P4[0], Q4[0])),
152   P4 = FFLAS::fflas_new<size_t>(M-M2-R3);
153   Q4 = FFLAS::fflas_new<size_t>(N-N2-R2);
154   R4 = pPLUQ (Fi, Diag, M-M2-R3, N-N2-R2, R, lda, P4, Q4,nt);
155   );
156   CHECK_DEPENDENCIES;
157
158   // [ E21 M31 0 K1 ] <- P4^T [ E2 M3 0 K ]
159   // [ E22 M32 0 K2 ]
160   TASK(MODE(READ(P4[0], R2, R3, M2) CONSTREFERENCE(Fi, P4, A3, R2
          , R3) READWRITE(A3[R3*lda])),
161   papplyP(Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, N2+R2, 0, M-
          M2-R3, A3+R3*lda, lda, P4));
162
163   // [ D21 D22 ]      [ D2 ]
164   // [ V21 V22 ]  <-  [ V2 ] Q4^T
165   // [  0   0  ]      [  0 ]
166   // [ 01   02 ]      [  0 ]
167   TASK(MODE(READ(Q4[0], R2, N2, M2, R3) CONSTREFERENCE(Fi, Q4, A2
          , R2, R3) READWRITE(A2[R2])),
168   papplyP( Fi, FFLAS::FflasRight, FFLAS::FflasTrans, M2+R3, 0, N-
          N2-R2, A2+R2, lda, Q4));
169
170   // P <- Diag (P1 [ I_R1    ] , P3 [ I_R3    ])
171   //           [      P2 ]       [      P4 ]
172   WAIT;
173   MathP = FFLAS::fflas_new<size_t>(M);
174   composePermutationsP (MathP, P1, P2, R1, M2);
175   composePermutationsP (MathP+M2, P3, P4, R3, M-M2);
176   for (size_t i=M2; i<M; ++i)
177   MathP[i] += M2;
178
179   if (R1+R2 < M2){
180     // P <- P S
181     TASK(MODE(CONSTREFERENCE(R1, R2, R3, R4, MathP, M2) READ(R1,
            R2, R3, R4, M2) READWRITE(MathP[0])),
```

```
182     PermApplyS( MathP, 1,1, M2, R1, R2, R3, R4);
183     );
184
185     // A <-  S^T A
186     TASK(MODE(READ(R1, R2, R3, R4) CONSTREFERENCE(Fi, A, R1, R2,
            R3, R4) READWRITE(A[0])),
187     pMatrixApplyS( Fi, A, lda, N, M2, R1, R2, R3, R4));
188   }
189
190   // Q<- Diag ( [ I_R1    ] Q1,  [ I_R2    ] Q2 )
191   //            [      Q3 ]      [      P4 ]
192   MathQ = FFLAS::fflas_new<size_t>(N);
193   TASK(MODE(CONSTREFERENCE(Q1, Q2, Q3, Q4, R1, R2) READ(Q1[0], Q2
        [0], Q3[0], Q4[0], R1, R2) READWRITE(MathQ[0])),
194   composePermutationsQ (MathQ, Q1, Q3, R1, N2);
195   composePermutationsQ (MathQ+N2, Q2, Q4, R2, N-N2);
196   for (size_t i=N2; i<N; ++i)
197       MathQ[i] += N2;
198   );
199   CHECK_DEPENDENCIES;
200
201   if (R1 < N2){
202     // Q <- T Q
203     TASK(MODE(CONSTREFERENCE(R1, R2, R3, R4) READ(R1, R2, R3, R4)
            READWRITE(MathQ[0])),
204     PermApplyT (MathQ, 1,1,N2, R1, R2, R3, R4););
205
206     // A <-   A T^T
207     TASK(MODE(READ(R1, R2, R3, R4) CONSTREFERENCE(Fi, A, R1, R2,
            R3, R4) READWRITE(A[0])),
208     pMatrixApplyT(Fi, A, lda, M, N2, R1, R2, R3, R4));
209   }
210   CHECK_DEPENDENCIES;
211   TASK(MODE(CONSTREFERENCE(MathP, MathQ) READ(MathP[0], MathQ[0])
        READWRITE(P[0], Q[0])),
212   MathPerm2LAPACKPerm (Q, MathQ, N);
213   MathPerm2LAPACKPerm (P, MathP, M);
214   );
215   );
216   FFLAS::fflas_delete( MathQ);
217   FFLAS::fflas_delete( MathP);
218   FFLAS::fflas_delete( P1);
219   FFLAS::fflas_delete( P2);
220   FFLAS::fflas_delete( P3);
221   FFLAS::fflas_delete( P4);
222   FFLAS::fflas_delete( Q1);
223   FFLAS::fflas_delete( Q2);
224   FFLAS::fflas_delete( Q3);
225   FFLAS::fflas_delete( Q4);
```

```
226
227   return R1+R2+R3+R4;
```

D Implementation of the block cutting strategies: `blockcuts.inl`

Listing 3: ppluq parallel implementation using PALADIn

```
1
2
3
4
5
6  namespace FFLAS {
7      namespace CuttingStrategy{
8          struct Single{};
9          struct Row{};
10         struct Column{};
11         struct Block{};
12         struct Recursive{};
13     }
14
15     namespace StrategyParameter{
16         struct Fixed{};
17         struct Threads{};
18         struct Grain{};
19         struct TwoD{};
20         struct TwoDAdaptive{};
21         struct ThreeD{};
22         struct ThreeDInPlace{};
23         struct ThreeDAdaptive{};
24     }
25
26     /*! ParSeqHelper for both fgemm and ftrsm
27     */
28         /*! ParSeqHelper for both fgemm and ftrsm
29     */
30     namespace ParSeqHelper {
31         template <typename C=CuttingStrategy::Block, typename P=
               StrategyParameter::Threads>
32         struct Parallel{
33             typedef C Cut;
34             typedef P Param;
35
36             Parallel(size_t n=MAX_THREADS):_numthreads(n){}
37
38             friend std::ostream& operator<<(std::ostream& out,
                   const Parallel& p) {
39                  return out << "Parallel: " << p.numthreads();
40             }
41             size_t numthreads() const { return _numthreads; }
42             size_t& set_numthreads(size_t n) { return _numthreads
                   =n; }
```

```
43              // CuttingStrategy method() const { return _method; }
44              // StrategyParameter strategy() const { return _param
                    ; }
45          private:
46              size_t _numthreads;
47              // CuttingStrategy _method;
48              // StrategyParameter _param;
49
50          };
51          struct Sequential{
52              Sequential() {}
53              template<class Cut,class Param>
54              Sequential(Parallel<Cut,Param>& ) {}
55              friend std::ostream& operator<<(std::ostream& out,
                    const Sequential&) {
56                  return out << "Sequential";
57              }
58              size_t numthreads() const { return 1; }
59          };
60      }
61
62
63      template<class Cut=CuttingStrategy::Block, class Strat=
            StrategyParameter::Threads>
64      inline void BlockCuts(size_t& RBLOCKSIZE, size_t& CBLOCKSIZE,
65                      const size_t m, const size_t n,
66                      const size_t numthreads);
67
68      template<>
69      inline void BlockCuts<CuttingStrategy::Single,
            StrategyParameter::Threads>(size_t& RBLOCKSIZE,
70                                  size_t& CBLOCKSIZE,
71                                  const size_t m, const size_t n,
72                                  const size_t numthreads) {
73      assert(numthreads==1);
74          RBLOCKSIZE = std::max(m,(size_t)1);
75          CBLOCKSIZE = std::max(n,(size_t)1);
76      }
77
78
79      template<>
80      inline void BlockCuts<CuttingStrategy::Row,StrategyParameter
            ::Fixed>(size_t& RBLOCKSIZE,
81                                  size_t& CBLOCKSIZE,
82                                  const size_t m, const size_t n,
83                                  const size_t numthreads) {
84          RBLOCKSIZE = std::max(std::min(m,
                __FFLASFFPACK_MINBLOCKCUTS),(size_t)1);
85          CBLOCKSIZE = std::max(n,(size_t)1);
```

```
86      }
87
88
89      template<>
90      inline void BlockCuts<CuttingStrategy::Row,StrategyParameter
            ::Grain>(size_t& RBLOCKSIZE,
91                               size_t& CBLOCKSIZE,
92                               const size_t m, const size_t n,
93                               const size_t grainsize) {
94          RBLOCKSIZE = std::max(std::min(m,grainsize),(size_t)1);
95          CBLOCKSIZE = std::max(n,(size_t)1);
96      }
97
98      template<>
99      inline void BlockCuts<CuttingStrategy::Block,
            StrategyParameter::Grain>(size_t& RBLOCKSIZE,
100                              size_t& CBLOCKSIZE,
101                              const size_t m, const size_t n,
102                              const size_t grainsize) {
103         RBLOCKSIZE = std::max(std::min(m,grainsize),(size_t)1);
104         CBLOCKSIZE = std::max(std::min(n,grainsize),(size_t)1);
105     }
106
107
108     template<>
109     inline void BlockCuts<CuttingStrategy::Column,
            StrategyParameter::Fixed>(size_t& RBLOCKSIZE,
110                                 size_t& CBLOCKSIZE,
111                                 const size_t m, const size_t n,
112                                 const size_t numthreads) {
113         RBLOCKSIZE = std::max(m,(size_t)1);
114         CBLOCKSIZE = std::max(std::min(n,
                __FFLASFFPACK_MINBLOCKCUTS),(size_t)1);
115     }
116
117
118     template<>
119     inline void BlockCuts<CuttingStrategy::Column,
            StrategyParameter::Grain>(size_t& RBLOCKSIZE,
120                                 size_t& CBLOCKSIZE,
121                                 const size_t m, const size_t n,
122                                 const size_t grainsize) {
123         RBLOCKSIZE = std::max(m,(size_t)1);
124         CBLOCKSIZE = std::max(std::min(n,grainsize),(size_t)1);
125     }
126
127
128     template<>
```

```
129     inline void BlockCuts<CuttingStrategy::Block,
           StrategyParameter::Fixed>(size_t& RBLOCKSIZE,
130                               size_t& CBLOCKSIZE,
131                               const size_t m, const size_t n,
132                               const size_t numthreads) {
133          RBLOCKSIZE = std::max(std::min(m,
               __FFLASFFPACK_MINBLOCKCUTS),(size_t)1);
134          CBLOCKSIZE = std::max(std::min(n,
               __FFLASFFPACK_MINBLOCKCUTS),(size_t)1);
135     }
136
137     template<>
138     inline void BlockCuts<CuttingStrategy::Row,StrategyParameter
           ::Threads>(size_t& RBLOCKSIZE,
139                               size_t& CBLOCKSIZE,
140                               const size_t m, const size_t n,
141                               const size_t numthreads) {
142          RBLOCKSIZE = std::max(m/numthreads,(size_t)1);
143          CBLOCKSIZE = std::max(n,(size_t)1);
144     }
145
146
147     template<>
148     inline void BlockCuts<CuttingStrategy::Column,
           StrategyParameter::Threads>(size_t& RBLOCKSIZE,
149                                 size_t& CBLOCKSIZE,
150                                 const size_t m, const size_t n
                                       ,
151                                 const size_t numthreads) {
152          RBLOCKSIZE = std::max(m,(size_t)1);
153          CBLOCKSIZE = std::max(n/numthreads,(size_t)1);
154     }
155
156     template<>
157     inline void BlockCuts<CuttingStrategy::Block,
           StrategyParameter::Threads>(size_t& RBLOCKSIZE,
158                               size_t& CBLOCKSIZE,
159                               const size_t m, const size_t n,
160                               const size_t numthreads) {
161          if (numthreads<65) {
162               const short maxtc[64] =
163               {1,2,3,2,5,3,7,4,3,5,11,4,13,7,5,4,17,6,19,5,7,11,
164                23,6,5,13,9,7,29,6,31,8,11,17,7,6,37,19,13,8,41,
165                7,43,11,9,23,47,8,7,10,17,13,53,9,11,8,19,29,59,
166                10,61,31,9,8};
167               const short maxtr[64] =
168               {1,1,1,2,1,2,1,2,3,2,1,3,1,2,3,4,1,3,1,4,3,2,1,4,5,
169                2,3,4,1,5,1,4,3,2,5,6,1,2,3,5,1,6,1,4,5,2,1,6,7,5,
170                3,4,1,6,5,7,3,2,1,6,1,2,7,8};
```

```
171
172             RBLOCKSIZE=std::max(m/(size_t)maxtr[numthreads-1],(
                    size_t)1);
173             CBLOCKSIZE=std::max(n/(size_t)maxtc[numthreads-1],(
                    size_t)1);
174         } else {
175             const size_t maxt = (size_t)sqrt((double)numthreads);
176             size_t maxtr=maxt,maxtc=maxt;
177             for(size_t i=maxt; i>=1; --i) {
178                 size_t j=maxt;
179                 size_t newpr = i*j;
180                 for( ; newpr < numthreads; ++j, newpr+=i ) {}
181                 if (newpr == numthreads) {
182                     maxtc = j;
183                     maxtr = i;
184                     break;
185                 }
186             }
187             RBLOCKSIZE=std::max(m/maxtr,(size_t)1);
188             CBLOCKSIZE=std::max(n/maxtc,(size_t)1);
189         }
190     }
191
192
193     template<class Cut=CuttingStrategy::Block, class Param=
            StrategyParameter::Threads>
194     inline void BlockCuts(size_t& rowBlockSize, size_t&
            colBlockSize,
195                           size_t& lastRBS, size_t& lastCBS,
196                           size_t& changeRBS, size_t& changeCBS,
197                           size_t& numRowBlock, size_t& numColBlock,
198                           size_t m, size_t n,
199                           const size_t numthreads) {
200         BlockCuts<Cut,Param>(rowBlockSize, colBlockSize, m, n,
                numthreads);
201         numRowBlock = m/rowBlockSize;
202         numColBlock = n/colBlockSize;
203
204         changeRBS = m-rowBlockSize*numRowBlock;
205         lastRBS = rowBlockSize;
206         if (changeRBS) ++rowBlockSize;
207
208         changeCBS = n-colBlockSize*numColBlock;
209         lastCBS = colBlockSize;
210         if (changeCBS) ++colBlockSize;
211
212
213     }
214
```

```
215
216 }
217
218
219
220 namespace FFLAS {
221     template <typename blocksize_t=size_t, typename Cut=
        CuttingStrategy::Block, typename Param=StrategyParameter
        ::Threads>
222     struct ForStrategy1D {
223         ForStrategy1D(const blocksize_t n, const ParSeqHelper::
            Parallel<Cut,Param> H) {
224             build(n,H);
225         }
226         ForStrategy1D(const blocksize_t b, const blocksize_t e,
            const ParSeqHelper::Parallel<Cut,Param> H) {
227             build(e-b,H);
228         }
229
230         void build(const blocksize_t n, const ParSeqHelper::
            Parallel<Cut,Param> H) {
231
232             if ( Protected::AreEqual<Param, StrategyParameter::
                Threads>::value ) {
233                 numBlock = std::max((blocksize_t)(H.numthreads())
                    ,(blocksize_t)1);
234             } else if ( Protected::AreEqual<Param,
                StrategyParameter::Grain>::value ) {
235                 numBlock = std::max(n/ (blocksize_t)(H.numthreads
                    ()), (blocksize_t)1);
236             } else {
237                 numBlock = std::max(n/(blocksize_t)(
                    __FFLASFFPACK_MINBLOCKCUTS),(blocksize_t)1);
238             }
239             firstBlockSize = n/numBlock;
240             if (firstBlockSize<1) {
241                 firstBlockSize = (blocksize_t)1;
242                 numBlock = n;
243             }
244             changeBS = n - numBlock*firstBlockSize;
245             lastBlockSize = firstBlockSize;
246             if (changeBS) ++firstBlockSize;
247
248         }
249
250         blocksize_t initialize() {
251             ibeg = 0; iend = firstBlockSize;
252
253             return current = 0;
```

```
254         }
255         bool isTerminated() const { return current == numBlock; }
256 
257         blocksize_t begin() const { return ibeg; }
258         blocksize_t end() const { return iend; }
259 
260         blocksize_t blocksize() const { return firstBlockSize; }
261         blocksize_t numblocks() const { return numBlock; }
262 
263 
264         blocksize_t operator++() {
265             ibeg = iend;
266             iend += (++current<changeBS?firstBlockSize:
                        lastBlockSize);
267 
268             return current;
269         }
270 
271     protected:
272         blocksize_t ibeg, iend;
273 
274         blocksize_t current;
275         blocksize_t firstBlockSize,lastBlockSize;
276         blocksize_t changeBS;
277         blocksize_t numBlock;
278 
279     };
280 
281     template <typename blocksize_t=size_t, typename Cut=
            CuttingStrategy::Block, typename Param=StrategyParameter
            ::Threads>
282     struct ForStrategy2D {
283         ForStrategy2D(const blocksize_t m, const blocksize_t n,
                const ParSeqHelper::Parallel<Cut,Param> H)
284             {
285                 BlockCuts<Cut,Param>(rowBlockSize, colBlockSize,
286                                      lastRBS, lastCBS,
287                                      changeRBS, changeCBS,
288                                      numRowBlock, numColBlock,
289                                      m, n,
290                                      H.numthreads());
291 
292             BLOCKS = numRowBlock * numColBlock;
293         }
294 
295 
296         blocksize_t initialize() {
297             _ibeg = 0; _iend = rowBlockSize;
298             _jbeg = 0; _jend = colBlockSize;
```

```
299             return current = 0;
300         }
301         bool isTerminated() const { return current == BLOCKS; }
302 
303         blocksize_t ibegin() const { return _ibeg; }
304         blocksize_t jbegin() const { return _jbeg; }
305         blocksize_t iend() const { return _iend; }
306         blocksize_t jend() const { return _jend; }
307 
308 
309         blocksize_t operator++() {
310             ++current;
311             blocksize_t icurr = current/numColBlock;
312             blocksize_t jcurr = current%numColBlock;
313             if (jcurr) {
314                 _jbeg = _jend;
315                 _jend += (jcurr<changeCBS?colBlockSize:lastCBS);
316             } else {
317                 _ibeg = _iend;
318                 _iend += (icurr<changeRBS?rowBlockSize:lastRBS);
319                 _jbeg = 0;
320                 _jend = colBlockSize;
321             }
322             return current;
323         }
324 
325         friend std::ostream& operator<<(std::ostream& out, const
                ForStrategy2D& FS2D) {
326             out<<"RBLOCKSIZE: "<<FS2D.rowBlockSize<<std::endl;
327             out<<"CBLOCKSIZE: "<<FS2D.colBlockSize<<std::endl;
328             out<<"changeRBS  : "<<FS2D.changeRBS<<std::endl;
329             out<<"changeCBS  : "<<FS2D.changeCBS<<std::endl;
330             out<<"lastRBS    : "<<FS2D.lastRBS<<std::endl;
331             out<<"lastCBS    : "<<FS2D.lastCBS<<std::endl;
332             out<<"NrowBlocks: "<<FS2D.numRowBlock<<std::endl;
333             out<<"NcolBlocks: "<<FS2D.numColBlock<<std::endl;
334             out<<"curr: " << FS2D.current << '/' << FS2D.BLOCKS
                    << std::endl;
335             out<<"_ibeg: " << FS2D._ibeg << std::endl;
336             out<<"_iend: " << FS2D._iend << std::endl;
337             out<<"_jbeg: " << FS2D._jbeg << std::endl;
338             out<<"_jend: " << FS2D._jend << std::endl;
339             return out;
340         }
341 
342         blocksize_t rowblocksize() const { return rowBlockSize; }
343         blocksize_t rownumblocks() const { return numRowBlock; }
344         blocksize_t colblocksize() const { return colBlockSize; }
345         blocksize_t colnumblocks() const { return numColBlock; }
```

```
346
347
348     protected:
349         blocksize_t _ibeg, _iend, _jbeg, _jend;
350         blocksize_t rowBlockSize, colBlockSize;
351
352         blocksize_t current;
353         blocksize_t lastRBS; blocksize_t lastCBS;
354         blocksize_t changeRBS; blocksize_t changeCBS;
355         blocksize_t numRowBlock; blocksize_t numColBlock;
356         blocksize_t BLOCKS;
357
358     };
359
360 }
```

Abstract

Adaptive parallel generic exact Linear Algebra

Triangular matrix decompositions are fundamental building blocks in computational linear algebra. They are used to solve linear systems, compute the rank, the determinant, the null-space or the row and column rank profiles of a matrix. The project of my PhD thesis is to develop high performance shared memory parallel implementations of exact Gaussian elimination.

In order to abstract the computational code from the parallel programming environment, we developed a domain specific language, PALADIn: Parallel Algebraic Linear Algebra Dedicated Interface, that is based on $C/C++$ macros. This domain specific language allows the user to write $C++$ code and benefit from sequential and parallel executions on shared memory architectures using the standard OpenMP, TBB and Kaapi parallel runtime systems and thus providing data and task parallelism.

Several aspects of parallel exact linear algebra were studied. We incrementally build efficient parallel kernels, for matrix multiplication, triangular system solving, on top of which several variants of PLUQ decomposition algorithm are built. We study the parallelization of these kernels using several algorithmic variants: either iterative or recursive and using different splitting strategies.

We propose a recursive Gaussian elimination that can compute simultaneously the row and column rank profiles of a matrix as well as those of all of its leading submatrices, in the same time as state of the art Gaussian elimination algorithms. We also study the conditions making a Gaussian elimination algorithm reveal this information by defining a new matrix invariant, the rank profile matrix.