Parallel Computation of Echelon Forms*

Jean-Guillaume Dumas[1], Thierry Gautier[2], Clément Pernet[3], and Ziad Sultan[1,2]

[1] LJK-CASYS, UJF, CNRS, Inria, G'INP, UPMF, Grenoble, France
[2] LIG-MOAIS UJF, CNRS, Inria, G'INP, UPMF, Grenoble, France
[3] LIP-AriC UJF, CNRS, Inria, UCBL, ÉNS de Lyon, France

Abstract. We propose efficient parallel algorithms and implementations on shared memory architectures of LU factorization over a finite field. Compared to the corresponding numerical routines, we have identified three main specifities of linear algebra over finite fields. First, the arithmetic complexity could be dominated by modular reductions. Therefore, it is mandatory to delay as much as possible these reductions while mixing fine-grain parallelizations of tiled iterative and recursive algorithms. Second, fast linear algebra variants, e.g., using Strassen-Winograd algorithm, never suffer from instability and can thus be widely used in cascade with the classical algorithms. There, trade-offs are to be made between size of blocks well suited to those fast variants or to load and communication balancing. Third, many applications over finite fields require the rank profile of the matrix (quite often rank deficient) rather than the solution to a linear system. It is thus important to design parallel algorithms that preserve and compute this rank profile. 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. Experiments demonstrate that the tile recursive variant performs better and matches the performance of reference numerical software when no rank deficiency occurs. Furthermore, even in the most heterogeneous case, namely when all pivot blocks are rank deficient, we show that it is possbile to maintain a high efficiency.

1 Introduction

Triangular matrix factorization is a main building block in computational linear algebra. Driven by a large range of applications in computational sciences, parallel numerical dense LU factorization has been intensively studied since several decades which results in software of great maturity (e.g., LINPACK is used for benchmarking the efficiency of the top 500 supercomputers. More recently, efficient sequential exact linear algebra routines were developed [5]. They are used in algebraic cryptanalysis, computational number theory, or integer linear programming and they 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

* This work is partly funded by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013).

floating point numbers, and then rely on the efficiency of the floating point matrix multiplication `dgemm` of the BLAS. The conversion back to the finite field, done by costly modular reductions, is delayed as much as possible. 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. In order to exploit the power of multi-core and many-core architectures, we now investigate the parallelization of the finite field linear algebra routines. We report in this paper the conclusions of our experience in parallelizing exact LU decomposition for shared memory parallel computers. We try to emphasize which specificities of exact computation domains led us to use different approaches than that of numerical linear algebra. In short, we will illustrate that 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.

Types of block algorithms. Several schemes are used to design block linear algebra algorithms: the splitting can occur on one dimension only, producing row or column slabs [11], or both dimensions, producing tiles [2]. 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. Figure 1 summarizes some of the various existing block splitting obtained by combining these two aspects. Most numerical dense Gaussian elimination algorithms, like in [2], use tiled iterative block algorithms. In [4] the classic tiled iterative algorithm is combined with a slab recursive one for the panel elimination. Over exact domains, recursive algorithms are preferred to benefit from fast matrix arithmetic (see below). Slab recursive exact algorithms can be found in [10] and references therein and [6] presents a tiled recursive algorithm.

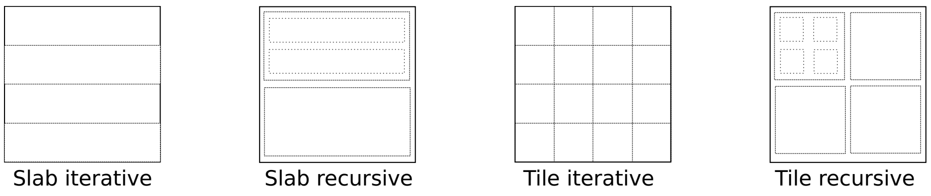

Fig. 1. Main types of block splitting

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 (often referred to as the panel factorization). It is an important parameter for optimizing efficiency: a finer grain allows more flexibility in the scheduling when running numerous cores, but it also challenges the efficiency of the scheduler and can increase the bus traffic.

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 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 3.1. Moreover, numerical stability is not an issue over a finite field, and asymptotically fast matrix multiplication algorithms, like Winograd's variant of Strassen algorithm [8, §12] can be used on top of the BLAS. Their speed-up increases with matrix dimension. The cost of sequential matrix multiplication over finite field is therefore not associative: a larger granularity delivers better sequential efficiency.

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 [10,6]. This is a key invariant used in many applications using exact Gaussian elimination, such as Gröbner basis computations [7] and computational number theory [13].

In the case of numerical LU factorization, quite often all panel blocks have full rank. Therfore the splitting can be done statically according to a granularity parameter. Over exact domains, on the contrary, the large blocks are almost always rank deficient. Thus, the tiles or slabs have unpredictable dimensions and the block splitting necessarily dynamic, as will be illustrated in Section 4.

Consequently the design of a parallel 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.

It also implies several requirements on the parallel run-time being used:

- the block splitting has to be dynamically computed;
- the computing load for each task is not known in advance (some panel blocks may have high rank deficiency), making the tasks very heterogeneous.

This motivated us to look into parallel execution runtimes using tasks with work-stealing based scheduling.

All experiments have been conducted on a 32 cores Intel Xeon E5-4620 2.2Ghz (Sandy Bridge) with L3 cache(16384 KB). The numerical BLAS is ATLAS v3.11.4, LAPACK v3.4.2 and PLASMA v2.5.0. We used X-KAAPI-2.1 version with last git commit: xkaapi_2.1-30-g263c19c638788249. The gcc compiler version used is gcc 4.8.2 that supports OpenMP 3.1.

We introduce in Section 2 the algorithmic building blocks on which our algorithms will rely and the parallel programming models and runtimes that we used in our experiments. In order to handle each problem separately, we focus in Section 3 on the simpler case where no rank deficiency occur. In particular Section 3.1 presents detailed analysis of the number of modular reductions required by various block algorithms including the tiled and slab recursive, the left-looking, right-looking and Crout variants of the tiled iterative algorithm. Lastly Section 4 deals with elimination with rank deficiencies. We there present and compare new slab iterative, tiled iterative and tiled recursive parallel algorithms that preserve rank profiles. We then show that the latter can match state of the art numerical routines, even when taking rank deficiencies into account.

2 Preliminaries

2.1 Auxiliary Sequential Routines

All block algorithms that we will describe rely on four types of operations that we denote using the BLAS/LAPACK naming convention:

`gemm`: general matrix multiplication, computing $C \leftarrow \alpha A \times B + \beta C$,
`trsm`: solving upper/lower triang. syst. with matrix right/left h.s $B \leftarrow BU^{-1}$.
`laswp`: permuting rows or columns by sequence of swaps.
`getrf`: computing (P, L, U, Q), L and U stored in place of A, s.t. $A = PLUQ$.

A first prefix letter `d` or `f` specifies if the routine works over double precision floating point numbers or finite field coefficients and an optional prefix `p` stands for parallel implementation. Our implementations use the sequential routines of the `fflas-ffpack` library[1] [5]. There, the elements of a finite $\mathbb{Z}/p\mathbb{Z}$ for a prime p of size about 20 bits are integers stored in a double precision floating point number. The sequential `fgemm` routine combines recursive steps of Winograd's algorithm calls to numerical BLAS `dgemm` and reductions modulo p when necessary. The `ftrsm` and `fgetrf` routines use block recursive algorithms to reduce most arithmetic operations to `fgemm`. More precisely `fgetrf` is either done by a slab recursive algorithm [5] or a tile recursive algorithm [6].

2.2 Parallel Programming Models

We base our implementation on the task based parallel features of the OpenMP standard. This is motivated by the use of recursive algorithms where tasks are

[1] `http://linalg.org/projects/fflas-ffpack`

mandatory. Now in tile iterative algorithms, loops with tasks happen to perform at least as well as parallel loops.

`libgomp` is the GNU implementation of the OpenMP API for multi-platform shared-memory parallel programming in C/C++ and Fortran. Alternatively, we also used `libkomp` [1], an optimized version of `libgomp`, based on the `XKaapi` runtime, that reduces the overhead of the OpenMP directives and handles more efficiently threads creation, synchronization and management. In the experiments of the next sections, we will compare efficiency of the same code linked against each of these two libraries.

2.3 Parallel Matrix Multiplication

In the iterative block algorithms, all matrix product tasks are sequential, whereas the recursive block algorithms must call parallel matrix products `pfgemm`, which we describe here. Operation `pfgemm` is of the form $C \leftarrow \alpha A \times B + \beta C$. In order to split the computation into independent tasks, only the row dimension of A and the column dimension of B only are split. The granularity of the split can be chosen in two different ways: either as a fixed value, or by a ratio of the input dimension (e.g. the total number of cores). We chose the second option that maximizes the size of the blocks while ensuring a large enough number of tasks for the computing resources. All our experiments showed that this option performs better than the first one. When used as a subroutine in a parallel factorization, it will create more tasks than the number of available cores, but this heuristic happens to be a good compromise in terms of efficiency.

Figure 2 shows the computation time on 32 cores of various matrix multiplications: the numerical `dgemm` implementation of `Plasma-Quark`, the implementation of `pfgemm` of `fflas-ffpack` using OpenMP tasks, linked against the `libkomp` library. This implementation is run over the finite field $\mathbb{Z}/131071\mathbb{Z}$ or over field of real double floating point numbers, with or without fast Strassen-Winograd's matrix product. One first notices that most routine perform very similarly. More precisely, `Plasma-Quark dgemm` is faster on small matrices but the effect of Strassen-Winograd's algorithm makes `pfgemm` faster on larger matrices, even on the finite field where additional modular reductions occur. In terms of speed-up, the `pfgemm` reaches a factor of approximately 27 (using 32 cores) whereas the numerical `dgemm` of `Plasma-Quark` reaches a factor of 29, but this mostly reflects the fact that `dgemm` has a less efficient sequential reference timing since it does not use Strassen-Winograd's algorithm.

Similarly, other basic routines used in the recursive block algorithms, such as `ftrsm` (solving matrix triangular systems) and `flaswp` (permuting rows or columns), have been parallelized by splitting a dimension into a constant number of blocks (typically the number of cores).

3 Eliminations with No Rank Deficiency

In this section, we make the assumption that no rank deficiency occurs during the elimination of any of the diagonal block. This hypothesis is satisfied by

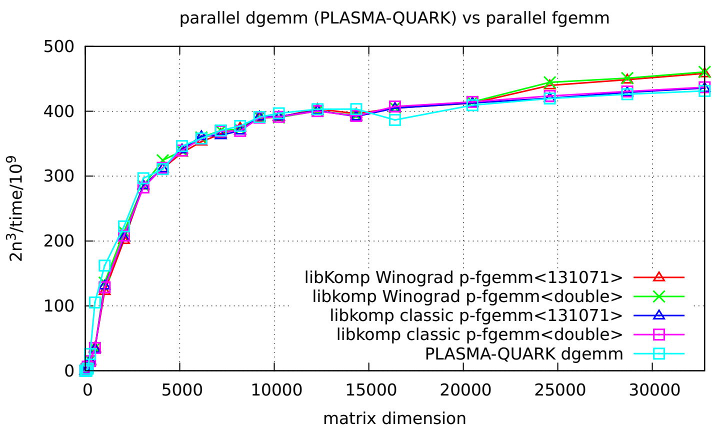

Fig. 2. Speed of exact and numerical matrix multiplication routines

matrices with generic rank profile (i.e. having all their leading principal minor non zero). This assumption allows us to focus on the problem of reducing the modular reduction count.

3.1 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., [5] 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 the number of reductions. We will therefore compare the number of modular reductions of three variants of the tile iterative algorithm (left-looking, right-looking and Crout, see [3]), the slab recursive algorithm of [5], and the tile recursive algorithm of [6]. 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: $R_{\texttt{gemm}}(m, k, n) = mn$. From [6, Theorem 3], this extends also for triangular solving with an $m \times n$ unknown matrix: with unit diagonal, $R_{\texttt{utrsm}}(m, m, n) = mn$ (actually the computation of the last row of the solution requires no modulo reduction as it is just a division by 1, we will therefore rather use $R_{\texttt{utrsm}}(m,m,n) = (m-1)n)$ and $R_{\texttt{trsm}}(m,m,n) = 2mn$ (with the previous refinement for $R_{\texttt{utrsm}}(m,m,n)$, this also reduces to $R_{\texttt{trsm}}(m,m,n) = (2m-1)n)$. Table 1 sketches the different shapes of the associated routine calls in the main loop of each variant.

Then the number of modular reductions required for these different LU factorization strategies is given in Table 2. The last two rows of the table corresponds to [6, Theorem 4] where $R_{\texttt{utrsm}}$ has been refined to $(m-1)n$ as mentioned above. 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 variant described above. Following Table 1, we thus have:

Table 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 [3, Chapter 5])

Left looking	Crout	Right looking

for i=1 to n/k **do**
    `utrsm` ((i-1)k,(i-1)k,k)
    `gemm` (n-(i-1)k,(i-1)k,k)
    `pluq` (k,k)
    `trsm` (k,k,n-ik)
	for i=1 to n/k **do**
	    `gemm` (n-(i-1)k,(i-1)k,k)
	    `gemm` (k,(i-1)k,n-ik)
	    `pluq` (k,k)
	    `utrsm` (k,k,n-ik)
	    `trsm` (k,k,n-ik)
		for i=1 to n/k **do**
		    `pluq` (k,k)
		    `utrsm` (k,k,n-ik)
		    `trsm` (k,k,n-ik)
		    `gemm` (n-ik,k,n-ik)

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

The Crout variant requires, at each step, except the first one, to compute $R_{\texttt{gemm}}(n-ik, ik, k)$ reductions for the pivot and below and $R_{\texttt{gemm}}(k, ik, n-(i-1)k)$ 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 $(\frac{n}{k}-i)R_{\texttt{utrsm}}(k,k,k)$ reductions and to solve triangular systems to the right, using $(\frac{n}{k}-i)R_{\texttt{trsm}}(k,k,k)$ reductions.

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

In Table 2 we see that the left looking variant always performs less modular reductions. Then the tiled recursive performs less modular reductions than the Crout variant as soon as $2 \le k \le \frac{n}{2+\sqrt{2}}$. Finally the right looking variant clearly performs more modular reductions. This explains the respective performance of the algorithms shown on Table 3 (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 next.

Table 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

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

Table 3. Timings (in seconds) of sequential LU factorization variants on one core

	$k = 212$			$k = \frac{n}{3}$			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

3.2 Parallel Experiments

In Figure 3 we compare the tiled iterative variants with the tiled recursive algorithm. The latter uses as a base case an iterative Crout algorithm too which performs fewer modular operations, The tiled recursive algorithm performs better than all other tiled iterative versions. This can be explained by a finer and more adaptive granularity and a better locality. The left looking variant performs poorly for it uses an expensive sequential `trsm` task. Although Crout and right-looking variant perform about the same number of matrix products, those of an iteration of the right-looking variant are independent, contrarily to those of the Crout variant, which explains a better performance despite a larger number of modular reductions.

Figure 4 shows the performance without modular reductions, of the tiled recursive parallel implementation on full rank matrices compared to `Plasma-Quark`. The best block size for the latter library was determined by hand for each matrix size. The two possible data-storage for `Plasma-Quark` are used: the collection of tiles or the row-major data-storage. Our tiled recursive parallel PLUQ implementation without modular reductions behaves better than the `Plasma-Quark` `getrf_tile`. This is mainly due to the bi-dimensional cutting which allows for a

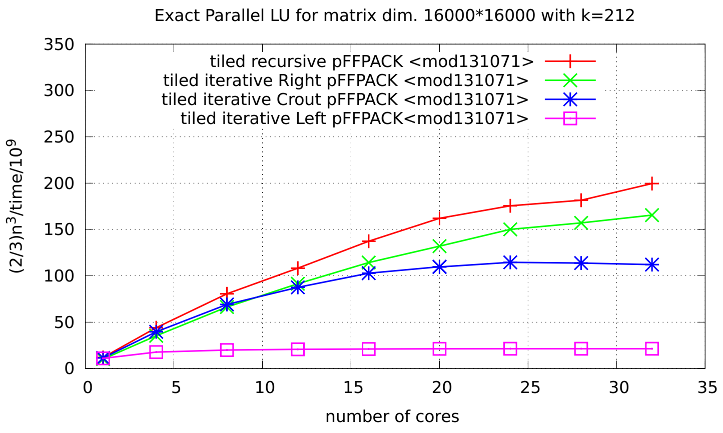

Fig. 3. Parallel LU factorization on full rank matrices with modular operations

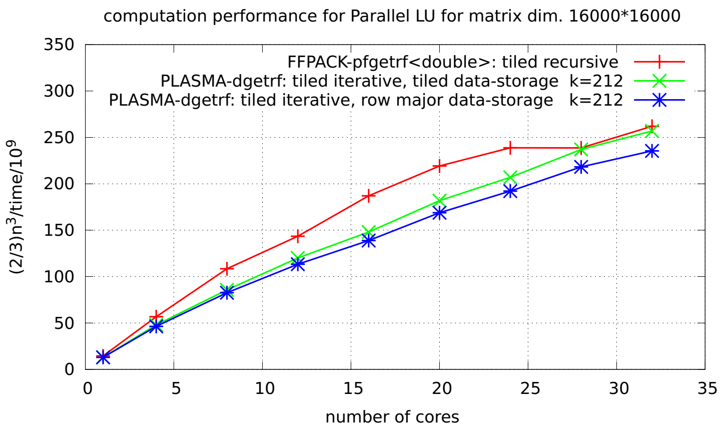

Fig. 4. Speed (normalized to $2/3n^3$ of parallel LU factorization on full rank matrices without modular operations

faster panel elimination, parallel `trsm` computations, more balanced `gemm` computations and some use of Strassen-Winograd's algorithm. This explains why performance become similar again on more than 24 cores: the size of the sequential blocks get below the threshold where this algorithm speeds up computations (typically 2400 on this machine).

4 Elimination with Rank Deficiencies

4.1 Pivoting Strategies

We now consider the general case of matrices with arbitrary rank profile, that can lead to rank deficiencies in the panel eliminations. Algorithms computing the row rank profile (or equivalently the column echelon form) used to share

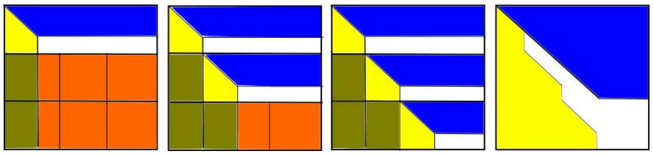

Fig. 5. Slab iterative factorization of a matrix with rank deficiencies, with final reconstruction of the upper triangular factor

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 [10] and references therein). Such an iterative algorithm can be translated into a slab recursive algorithm splitting the row dimension in halves (as implemented in sequential in [5]) or into a slab iterative algorithm. More recently, we presented in [6] a more flexible pivoting strategy that results in a tile recursive algorithm, cutting both dimensions simultaneously. As a by product, both row and column rank profiles are also computed simultaneously.

A slab iterative algorithm. In the slab iterative algorithm shown in Figure 5, each panel factorization has to be run by a sequential algorithm. This sequential task is costly and therefore imposes a choice of a fine granularity, which, as we saw, on the other hand implies more modular reductions and a lesser speed-up of Strassen-Winograd's algorithm.

Another difficulty is the fact that the starting column position of each panel is determined by the rank of the blocks computed so far. It can only be determined dynamically upon the execution. This implies in particular that no data-storage by tiles, that fit the tiles of the algorithm 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.

Tiled iterative elimination. In order to speed-up the panel computation, we can split it into column tiles. Thanks to the pivoting strategy of [6], it is still possible to 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 [4]. Figure 6 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 `libkomp`.

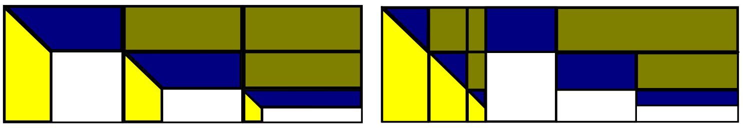

Fig. 6. Panel PLUQ factorization: tiled sub-calls inside a single slab and final reconstruction

Tiled recursive elimination. Recursive algorithms in dense linear algebra is a natural choice for hierarchical memory systems [14]. For large problems, the geometric nature of the recursion causes that the total area of operands for recursive algorithms is less compared to iterative algorithms [9]. We use the tile recursive algorithm described in [6]: 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 parallelized. Thus it can be run in parallel using recursive tasks and the `pfgemm`, `ftrsm` and `flaswp` routines.

Figure 7 shows performance obtained for the tiled recursive and the tiled iterative factorization. Both versions are tested using `libgomp` and `libkomp` libraries. The input `S16K` is a 16000×16000 matrix with low rank deficiency (rank is 15500). Linearly independent rows and columns of the generated matrix are uniformly distributed on the dimension.

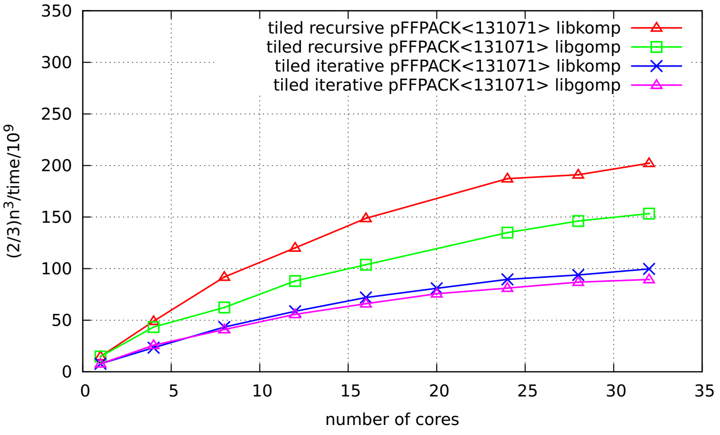

Fig. 7. Performance of tiled recursive and tiled iterative factorizations using `libgomp` and `libkomp`. Matrix dimension $n = 16000$ with rank 15500

The implementation with OpenMP of the tiled recursive LU maintained high efficiency in the case of rank deficient matrices. It attained a speed-up of 13.6 on 32 cores. Besides the fact that it benefits from Strassen-Winograd implementation, it is adapted to minimize memory accesses and optimize data placement. Using `libkomp` instead of `libgomp` library and numactl, for round and robin interleave memory placement, that helps reducing dependency on bus speed, we manage to obtain high performance for our tiled recursive LU factorization.

5 Conclusion

We analyzed five different algorithms for the computation of Gaussian elimination over a finite field. The granularity surely optimizes the parallelization of these algorithms but at the cost of more modular operations. Algorithms optimizing modular reductions are unfortunately not the most efficient in parallel. The best compromise is obtained with our recursive tiled algorithm that performs best in both aspects.

Perspective. Our future work focuses on two main issues. First, the use of specific allocators that can be used for a better mapping of data in memory and reduce distant accesses. Second, parallel programming frameworks for multicore processors [12] could be more effective than binding threads on each NUMA node. Dataflow based dependencies, like when using OpenMP 4.0 directives, can ensure more parallelism for recursive implementation using `libkomp` [1] library.

References

1. Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)
2. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35(1), 38–53 (2009), `http://dx.doi.org/10.1016/j.parco.2008.10.002`
3. Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V.: Numerical Linear Algebra for High Performance Computers. SIAM (1998)
4. Dongarra, J.J., Faverge, M., Ltaief, H., Luszczek, P.: Achieving numerical accuracy and high performance using recursive tile LU factorization. Concurrency and Computation: Practice and Experience 26(7), 1408–1431 (2014), `http://hal.inria.fr/hal-00809765`
5. Dumas, J.-G., Giorgi, P., Pernet, C.: Dense linear algebra over prime fields. ACM TOMS 35(3), 1–42 (2008), `http://arxiv.org/abs/cs/0601133`
6. Dumas, J.-G., Pernet, C., Sultan, Z.: Simultaneous computation of the row and column rank profiles. In: Kauers, M. (ed.) Proc. ISSAC 2013, Grenoble, France, pp. 181–188. ACM Press, New York (2013)
7. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra 139(1–3), 61–88 (1999)
8. Gathen, J.V., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, New York (1999)
9. Gustavson, F.G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development 41(6), 737–756 (1997)
10. Jeannerod, C.-P., Pernet, C., Storjohann, A.: Rank-profile revealing Gaussian elimination and the CUP matrix decomposition. J. Symb. Comp. 56, 46–68 (2013)
11. Klimkowski, K., van de Geijn, R.A.: Anatomy of a parallel out-of-core dense linear solver. In: ICPP, vol. 3, pp. 29–33. CRC Press (August 1995)
12. Kurzak, J., Ltaief, H., Dongarra, J., Badia, R.M.: Scheduling dense linear algebra operations on multicore processors. Concurrency and Computation: Practice and Experience 22(1), 15–44 (2010)
13. Stein, W.: Modular forms, a computational approach. Graduate studies in mathematics. AMS (2007), `http://wstein.org/books/modform/modform`
14. Toledo, S.: Locality of reference in lu decomposition with partial pivoting. SIAM Journal on Matrix Analysis and Applications 18(4), 1065–1081 (1997)