Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination

Jean-Guillaume Dumas, Thierry Gautier, Clément Pernet, Jean-Louis Roch, Ziad Sultan

Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination[☆]

Jean-Guillaume Dumas[a,], Thierry Gautier[b,], Clément Pernet[c,], Jean-Louis Roch[b,], Ziad Sultan[a,b,]

[a] *LJK-CASYS, UJF, CNRS, Inria, G'INP, UPMF*
[b] *LIG-MOAIS UJF, CNRS, Inria, G'INP, UPMF*
[c] *LIP-AriC UJF, CNRS, Inria, UCBL, ÉNS de Lyon*

Abstract

We present block algorithms and their implementation for the parallelization of Gaussian elimination over a finite field on shared memory architectures. Specificities of exact computations over a finite field include the use of subcubic matrix arithmetic and of costly modular reductions. As a consequence coarse grain block algorithms perform more efficiently than fine grain ones 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 a 32 cores NUMA architecture. 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.

Keywords: PLUQ decomposition, Parallel shared memory computation, Finite field, Dataflow task dependencies, NUMA architecture

1. Introduction

Dense Gaussian elimination over finite field 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

[☆]This work is partly funded by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013). Corresponding author: clement.pernet@imag.fr, tel: +33 4 37 28 74 75; fax : +33 4 72 72 80 80

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. As in numerical linear algebra, exact dense Gaussian elimination is a key building block for problems that are dense by nature but also for large sparse problems, for instance:

- When using sparse direct methods, blocks of new non-zero elements can arise (fill-in) and one needs to switch to optimized dense methods on these blocks;
- Sparse block iterative methods also induce dense elimination on blocks of iterated vectors, like in the block-Wiedemann or block-Lanczos algorithms.

Recently, efficient sequential exact linear algebra routines have been developed [6]. The kernel routines run over small finite fields and are usually lifted over $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{Z}[x]$. 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 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.

Block algorithms of dense linear algebra routines aggregate arithmetic operations in matrix multiplication. They thus rely on the efficiency of vector instructions and have a high computation per memory access rate.

The design of efficient exact Gaussian elimination differs from that in numerical linear algebra in many ways. As shown for instance in [5], numerical and exact approaches differ mainly in the pivoting strategies, in the cost of the underlying arithmetic and in the treatment of rank deficiencies.

Our focus is on parallel implementations using various pivoting strategies that will reveal the echelon form, i. e., the rank profile of the matrix [13, 7]. The latter is a key invariant used in many applications such as Gröbner basis computations [8] and computational number theory [17].

Another difference with numerical strategies is the systematic use of fast matrix multiplication variants, like Strassen and Strassen-Winograd algorithms [6]. They give better complexity for matrix multiplication and increase the performance of the latter when executed on sufficiently large blocks. The threshold from which fast variants take over classic variants can be automatically tuned.

As PLUQ decomposition reduces to matrix-matrix multiplication and triangular matrix solve, we thus study several variants of the latter sub-routines as single computations or composed in the higher level decomposition.

The sub-routines used for the computation of parallel PLUQ decomposition are mainly:

- the `fgemm` routine that stands for Finite field General Matrix Multiplication and computes: $C \leftarrow \beta C + \alpha A \times B$ where A, B, C are dense matrices.

- the `ftrsm` routine that stands for Finite field Triangular Solving Matrix and computes: $A \leftarrow BU^{-1}$ where U is an upper triangular matrix, and B a dense matrix (or $A \leftarrow L^{-1}B$ where L is a lower triangular matrix, and B a dense matrix).
- the `pluq` routine that computes the triangular factorization $P, L, U, Q = A$, where P and Q are permutation matrices, U is upper triangular and L is unit invertible lower triangular.

Several schemes are used to design block linear algebra algorithms: the splitting in blocks can occur on one dimension only, producing row or column slabs [15], or both dimensions, producing tiles [3].

Algorithms processing blocks can also be either iterative or recursive. Figure 1 summarizes some of the various existing block splitting obtained by combining these two aspects.

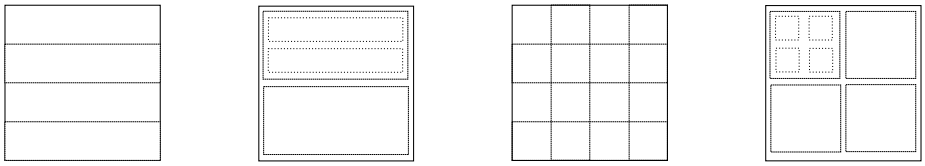

Figure 1: Main types of block splitting

Finally, we study the impact of these cutting strategies with the implementation of parallel versions of the `fgemm`, `ftrsm` and `pluq` sub-routines. We use the OpenMP library with task parallelization using two runtime implementations: `libgomp` [14], the GNU implementation of the OpenMP Application Programming Interface and `libkomp` an implementation of the OpenMP standard runtime based on the X-KAAPI library [10]. 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. Indeed, the X-KAAPI library handles better parallelization with fine-granularity as we show in section 2 by comparing the `libkomp` and `libgomp` runtime systems. However, over finite fields, with a fixed number of resources, we show that parallelization's priority is to focus on finding the best number of threads to be executed rather than fixing a fine-granularity.

1.1. Methodology of experiments

All experiments have been conducted on a 32 cores Intel Xeon E5-4620 2.2Ghz (Sandy Bridge) with L3 cache(16384 KB). All implemented routines are in the FFLAS-FFPACK library[1]. The numerical BLAS are ATLAS v3.11.4, OpenBLAS r0.2.9, MKL sp1.1.106, LAPACK v3.4.2 and PLASMA v2.5.0. We

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

used the X-KAAPI-2.1 version with last git commit: xkaapi_40ea2eb. The gcc compiler version is 4.9.1 (supporting OpenMP 4.0), 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 (consistant 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.

In section 2 we detail the main parameters that we consider. In section 3 we study different iterative and recursive variants and cutting strategies for the parallel matrix multiplication `pfgemm` and compare them with our best iterative standard parallel matrix multiplication [5]. In section 4 we show three parallel algorithms for the `pftrsm` routine: an iterative variant, a recursive variant and a hybrid combination. We then study the impact of these variants when they are composed in the PLUQ factorization, in section 5. Overall, our focus is on the computation of echelon forms in the case of rank deficient matrices. We show in this section that the performance of exact factorization can match that 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 possible to maintain a high efficiency.

2. Ingredients for the design of parallel kernels

The parallelization of standard versions of basic linear algebra routines has attained great maturity in numerical computation [16, 3]. Over finite fields, 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:

Impact of modular reductions. Computations over finite fields are done, first, by embedding finite field elements in integers stored as floating point numbers. Secondly, modular reduction operations are applied to convert back elements over finite fields. 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. The best implementations of floating point matrix multiplication are given by the BLAS that we thus use in our framework. This approach is only valid as long as integer computation does 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, we control overflowing if $n(p-1)^2 < 2^{mantissa}$. Furthermore, block algorithms of LU factorization

require eliminations on blocks. This induces a choice of the best block size k. Table 1 [5] shows the impact of block size for iterative and recursive algorithms on the number of modular reductions. This table demonstrates that the number of modular reduction is smaller in the case of tiled recursive LU factorization. This is one of the reasons that motivated us to look for tiled recursive variants over finite fields.

$k \wedge 1$	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	$\left(\frac{5}{2} - \frac{1}{k}\right) n^2 + \left(-2k - \frac{3}{2} + \frac{1}{k}\right) n + k^2$
	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 1: Counting modular reductions in full rank block LU factorization of an $n \times n$ matrix modulo p for a block size of k dividing n.

Fast variants for matrix multiplication. Numerical stability is not an issue over a finite field, and asymptotically fast matrix multiplication algorithms, like Winograd's variant of Strassen algorithm [9, §12] can be systematically used on top of the BLAS.

Table 2 shows the impact on performance of fast variants compared to standard matrix multiplication. In this table we compare the sequential speed ob-

	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 2: Effective Gfops ($2n^3/time/10^9$) of matrix multiplications: fgemm vs OpenBLAS d/sgemm on one core of a Xeon E5-4620 0 @ 2.20GHz

tained of 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 2 computations are done over small finite fields,

large finite fields and without modular reductions, to see the impact of modular reductions compared to openBLAS dgemm sequential execution.

We also can activate or deactivate the option that allows to use 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 and a small matrix dimension, 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 2 shows that in both cases, single or double precision, a speed-up of more than 40% can easily be attained.

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 naive 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.

Further experiments would have to be made in order to use fast parallel variants also. The cost of sequential matrix multiplication over finite field is therefore not associative: a larger granularity delivers better sequential efficiency [1]. So to take advantage of these fast variants in parallel standard matrix multiplication, we need to limit the cutting of matrices into sufficiently large blocks. In this case trade-offs are to be made between having fine-grain parallelization to generate more parallelism and having coarce-grain to take benefit from fast variants.

The impact of 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 (often referred to as the panel factorization, in the case of the PLUQ decompostion). It is an important parameter for optimizing 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. 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. The cost of sequential matrix multiplication over finite field is therefore not associative: a larger granularity delivers better sequential efficiency [1]. Over finite fields, 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.

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 X-KAAPI library, the `libkomp` runtime [2] system comes with very little task creation and scheduling overheads and implements recursive tasks in a very efficient way. In table 3 we show the overhead of using `libgomp` and `libkomp` runtime systems on one core compared to a sequential execution of block algorithm. We use for this comparison the best recursive algorithm for matrix multiplication, the *2D recursive adaptive*, that is detailed in section 3, with seven recursive calls. But even if we use optimized runtime systems for OpenMP tasks, the cost of creating tasks should not be neglected.

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

Using the latest version of gcc compiler we can also benefit from the dataflow features of 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. This allows to construct the DAG (directed acyclic graph) diagram that precomputes dependencies of all tasks before execution. This feature helps reducing the idle time of resources by removing unnecessary synchronizations. We see in next sections the impact of dataflow parallelization using the `libkomp` runtime that also implements the latest norms of OpenMP-4.0.

Data mapping on NUMA architecture. The efficiency of computations on a NUMA machine architecture can be disrupted due to remote accesses between different NUMA nodes. 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. Elements of the matrices are mapped on a specific core or node only during the initialization phase. There data mapping can be handled. Data are initialized with two parallel for loops. Each iteration is incremented with a fixed chunk size. To see the impact of remote accesses, we conducted experiments with different mapping strategies of matrices A, B and C in the case of matrix multiplication. First we map all the data on a single NUMA node, and execute the

program on all nodes. Then we conduct the same experiments by mapping on two, three and then all four NUMA nodes.

For the sake of clarity and simplicity we show only the different mapping strategies for one variant of matrix multiplication *2D recursive adaptive*, with four levels of recursion, in table 4. Experiments are done on 32 cores (4 NUMA nodes with 8 cores each). 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 4: Execution speed(Gfops): with different data mapping.

3. Parallel matrix multiplication

In this section, we study different variants of parallel matrix multiplication algorithms. We present various cutting strategies of block recursive and iterative algorithms for the `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). In this section, experiments are done with $\beta = 0$ and $\alpha = 1$.

3.1. Algorithmic variants

The 2D partitioning. This strategy splits the row dimension of A and the column dimension of B and thus generates independent tasks. More precisely, we distinguish an iterative and two recursive variants, as shown in Figure 2.

The 2D iterative partitioning splits A in s row slabs and B in t column slabs, and 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 available.

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 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.

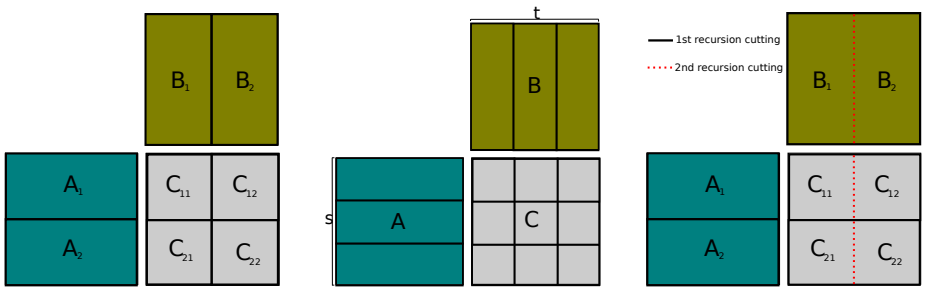

Figure 2: 2D partitioning: recursive (left), iterative (middle) and recursive adaptive (right) cutting.

The 3D partitioning. This strategy splits the three dimensions m, n and k. Again we present three variants:

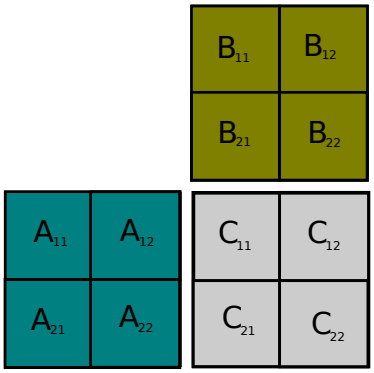

Figure 3: 3D partitioning : cutting A and B according to dimensions m, n and k.

The 3D in-place recursive variant 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. We show two implementations of this variant using OpenMP tasks and using dependencies.

In this 3D scheme we generate more tasks than in the 2D scheme. But with the *3D-Inplace recursive* variant we add synchronizations between tasks at each level of recursion. This can slow down the performance of this variant. The "#pragma omp taskwait" directive synchronizes all four

Implementation 1 *3D-Inplace recursive* with OpenMP tasks

Input: $A = (a_{ij})$ a $m \times k$ matrix over a field
$B = (b_{ij})$ a $k \times n$ matrix over a field
Output: C: $m \times n$ matrix over a field
#pragma omp task shared(C_{11}, A_{11}, B_{11})
$C_{11} = A_{11}.B_{11}$
#pragma omp task shared(C_{12}, A_{12}, B_{22})
$C_{12} = A_{12}.B_{22}$
#pragma omp task shared(C_{21}, A_{22}, B_{21})
$C_{21} = A_{22}.B_{21}$
#pragma omp task shared(C_{22}, A_{21}, B_{12})
$C_{22} = A_{21}.B_{12}$
#pragma omp taskwait
#pragma omp task shared(C_{11}, A_{12}, B_{21})
$C_{11} += A_{12}.B_{21}$
#pragma omp task shared(C_{12}, A_{11}, B_{12})
$C_{12} += A_{11}.B_{12}$
#pragma omp task shared(C_{21}, A_{21}, B_{11})
$C_{21} += A_{21}.B_{11}$
#pragma omp task shared(C_{22}, A_{22}, B_{22})
$C_{22} += A_{22}.B_{22}$
#pragma omp taskwait
Return (C)

Implementation 2 *3D-Inplace recursive* using OpenMP4.0 dependencies

Input: $A = (a_{ij})$ a $m \times k$ matrix over a field
$B = (b_{ij})$ a $k \times n$ matrix over a field
Output: C: $m \times n$ matrix over a field
#pragma omp task shared(C_{11}, A_{11}, B_{11}) depend(in:A_{11}, B_{11}) depend(out:C_{11})
$C_{11} = A_{11}.B_{11}$
#pragma omp task shared(C_{12}, A_{12}, B_{22}) depend(in:A_{12}, B_{22}) depend(out:C_{12})
$C_{12} = A_{12}.B_{22}$
#pragma omp task shared(C_{21}, A_{22}, B_{21}) depend(in:A_{22}, B_{21}) depend(out:C_{21})
$C_{21} = A_{22}.B_{21}$
#pragma omp task shared(C_{22}, A_{21}, B_{12}) depend(in:A_{21}, B_{12}) depend(out:C_{22})
$C_{22} = A_{21}.B_{12}$
#pragma omp task shared(C_{11}, A_{12}, B_{21}) depend(in:A_{12}, B_{21}) depend(inout:C_{11})
$C_{11} += A_{12}.B_{21}$
#pragma omp task shared(C_{12}, A_{11}, B_{12}) depend(in:A_{11}, B_{12}) depend(inout:C_{12})
$C_{12} += A_{11}.B_{12}$
#pragma omp task shared(C_{21}, A_{21}, B_{11}) depend(in:A_{21}, B_{11}) depend(inout:C_{21})
$C_{21} += A_{21}.B_{11}$
#pragma omp task shared(C_{22}, A_{22}, B_{22}) depend(in:A_{22}, B_{22}) depend(inout:C_{22})
$C_{22} += A_{22}.B_{22}$
#pragma omp taskwait
Return (C)

tasks created before. So, the second task that rewrites in the block C_{11}, for instance, needs to wait for all data of the matrix to be produced.

In this OpenMP Implementation 1, each task calls recursively the *3D-Inplace recursive* routine. Using OpenMP 4.0 directives helps specifying dependencies between tasks. This allows to start the computations on a

block once its data are produced. We show the OpenMP code of the *3D-Inplace recursive* routine using the clause "depend" in Implementation 2.

The 3D recursive variant 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.

The 3D recursive adaptive variant 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 the k dimension introduces some overhead, one can introduce a weighted penalty system to only split this dimension when it is largely great than the other dimensions.: with a penatly factor of p, the dimension k is split only when $\max(m, n) < pk$.

In all these recursive schemes, the recursion is stopped when the number of threads is less than or equal to one or when the matrix dimension becomes below a threshold (set to 220 in the experiments).

Even if the *3D recursive* variants suffers from an additional cost for temporary matrix allocation, we will show that it behaves better in parallel than the *3D-Inplace recursive*. Using dependencies provides better scheduling of the tasks. Each add task is launched immediately once its block elements are produced.

3.2. Experiments on square matrices

3.2.1. Comparison of all variants

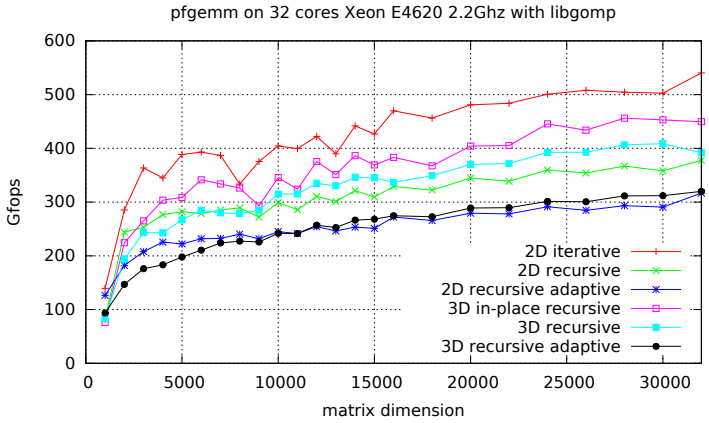

Figure 4: Speed of different matrix multiplication routines using libgomp

Experiments are conducted on square matrices with dimensions between 1000 and 32000 and elements are over the finite field $\mathbb{Z}/131071\mathbb{Z}$, using 32 cores. Figures 4 and 5 show the execution speed of all variants, using OpenMP4.0

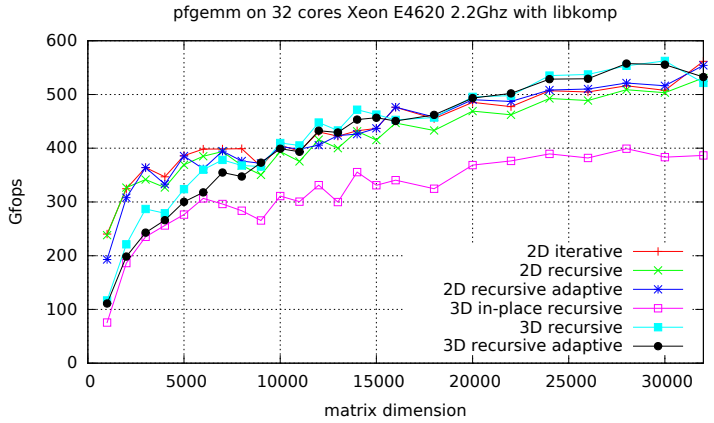

Figure 5: Speed of different matrix multiplication routines using libkomp

tasking model, linked with the two runtimes `libgomp` and `libkomp` respectively. With `libgomp`, the 2D iterative variant 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, except the 3D in-place recursive one, for a reason that we could not explain. The speed is now at least as good as that with `libkomp`.

In the next experiments we will therefore only show executions of implementations linked against `libkomp` library.

If the 3D recursive adaptive variant performs best on large matrices, the *2D recursive adaptive* algorithm is close to it on large instances (550 Gflops for $n = 32000$), but maintains a better efficiency with smaller matrices.

3.2.2. Comparison with the state of the art in numerical computation

Figure 6 shows the computation time of various matrix multiplications: the numerical `dgemm` implementation of `Plasma-Quark`, and `Intel-MKL` the implementation of `pfgemm` of `fflas-ffpack` using OpenMP-4.0 dataflow model. 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 routines perform very similarly. More precisely, `Intel-MKL 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.

3.3. Experiments on rectangular matrices

As shown in the previsou section, the 3D splitting variants perform similarly to the 2D variants for large matrices (with a slight improvement), but are less efficient on smaller matrices, due to more synchronizations and data copies.

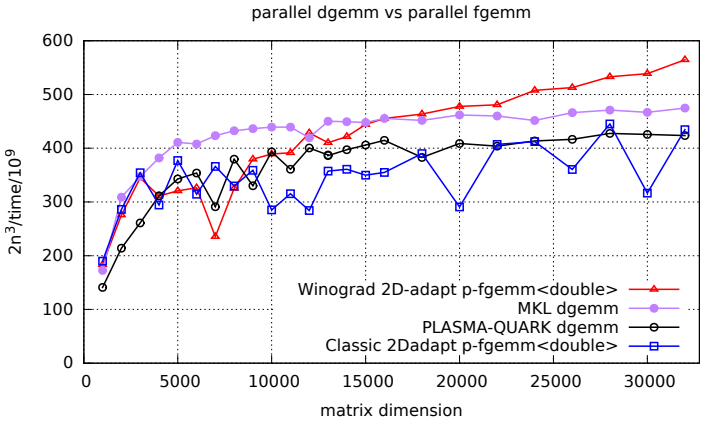

Figure 6: Speed of exact and numerical matrix multiplication routines

We now compare these variants in a situation supposed to be favorable for the 3D splitting: when the dimension k is large compared to m and n. Figure 7 reveals that indeed, the 3D recursive adaptive variant (with best penalty factor, found to be 12) outperforms the best 2D variants for very unbalanced cases: $m, n \leq 1000$ and $k = 20000$. However, the computation speeds gets quickly similar in all three variants.

This fact, combined with the results of figures 5 lead us to only consider the 2D splitting variants when calling `pfgemm` from other other routines. This has been confirmed by experiments on e.g. the PLUQ decomposition where 3D splitting of `pfgemm` always led to slower computations.

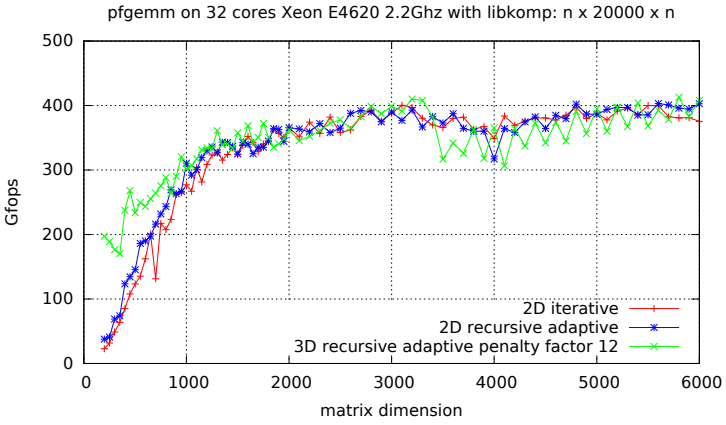

Figure 7: Comparing computation speed on rectangular matrices with large inner dimension k.

4. Parallel triangular solving matrix

In this section we study different cutting strategies for the computation the of parallel `pftrsm` routine. We identify three different types of parallelizations. The block iterative variant, block recursive variant and an hybrid variant that combines the iterative and the recursive variants. The latter proves to be deliver the best efficiency in practice, in particular when the hand-side is highly rectangular. In this subsection, we will consider, without loss of generality, the lower left case of the FTRSM operation: computing $X \leftarrow L^{-1}B$.

Iterative variant. In the iterative variant (Algorithm 1), the parallization is obtained by splitting the outer dimension of the right hand side matrices B and X:

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

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 need to maximize the computational size of each of these tasks. Hence the number of blocks k is set as the number of available threads.

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. The three tasks in Algorithm 2 can not be executed concurrently.

Algorithm 1 Iterative TRSM

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

Algorithm 2 Recursive TRSM

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

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 my become very 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.

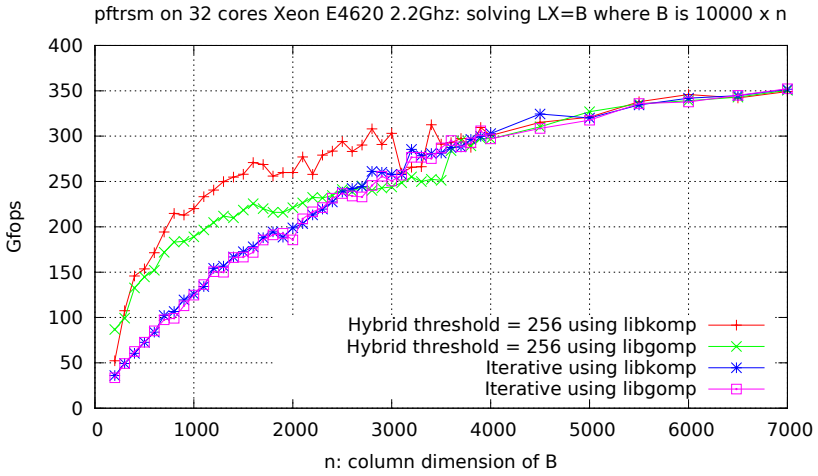

Figure 8: Comparing the Iterative and the Hybrid variants for parallel FTRSM using `libkomp` and `libgomp`. Only the outer dimension varies: B and X are $10000 \times n$.

Experiments on parallel FTRSM. Figure 8 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 for it handles more efficiently recursive tasks.

5. Parallel Gaussian elimination

In this section, we present the parallelization of two different algorithms for the computation of PLUQ decomposition: A tile iterative and a tile recursive algorithm. This computation is widely studied and has attained great ripeness in numerical computation. Over finite fields, strategies and interests differs from numerical computation in the case of Gaussian elimination. Moreover, the knowledge of the position of all pivots or equivalenty, the rank profile of the matrix[13, 7] that is revealed by performing a PLUQ decomposition is valuable and used in many applications. However, over finite field, only certain pivoting strategies can give this information on the input matrix.

In [5], parallel iterative and recursive implementations of exact PLUQ decompositions revealing the echelon form of the matrix are presented. It is there shown that the parallel recursive implementation behaves the best in terms of performance. We thus focus mainly on the optimization of this state of the art parallel recursive implementation in order to benefit from the optimized building kernels presented in the previous sections and some new tasking strategies using data dependencies. We then present parallel experiments in the case of full rank and rank deficient matrices.

5.1. Algorithmic variants for PLUQ factorization

We 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 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 [13] 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 [6]) or into a slab iterative algorithm. More recently, a more flexible pivoting strategy that results in a tile recursive algorithm, cutting both dimensions simultaneously was presented in [7]. As a by product, both row and column rank profiles are also computed simultaneously.

Tiled iterative algorithm. It has been showned in [5] that the slab iterative algorithm performing a PLUQ decomposition is slow due to large sequential tasks (see [5, § 4, Figure 5]). At each iteration a PLUQ decomposition is called sequentially on big slab blocks of size $k \times n$, where k is the size of iteration chunks and n is the column dimension of the input matrix. These sequential tasks are costly and therefore impose a choice of a fine granularity.

Over finite field, the only way to compute in parallel the rank profile of the input matrix using a PLUQ decomposition was by cutting according to one dimension. Thanks to the quad recursive algorithm presented in [7] the computation of rank profiles can be done by cutting according to two dimensions. By using this experience we managed to update the slab iterative algorithm into a tiled iterative algorithm for the computation of PLUQ decomposition in parallel. In order to speed-up the panel computation, we can split it into column tiles. Then the pivoting strategy of the latter paper makes it still possible to recover the rank profiles afterwards. Now with this splitting [5, § 4, Figure 6], the operations remain more local and updates can be parallelized. This approach shares similarities with the recursive computation of the panel described in [4].

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 runtime systems instead of static thread management.

This optimization used in the computation of the slab factorization improved the computation speed by a factor of 2.

Tiled recursive algorithm. Recursive algorithms in dense linear algebra is a natural choice for hierarchical memory systems [18]. For large problems, the geometric nature of the recursion causes that the total area of operands for recursive algorithms is less than that of iterative algorithms [12]. The parallelization of the recursive variant of PLUQ decomposition over finite fields is presented in [5] using OpenMP tasks. 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 9.

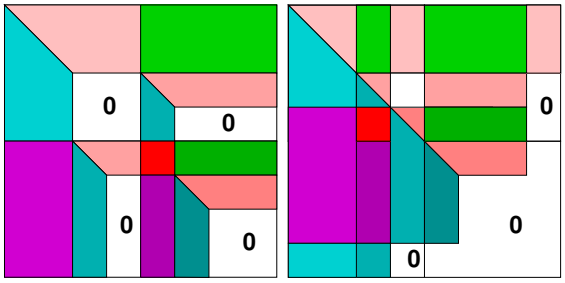

Figure 9: PLUQ quad-recursive scheme

In this work we modify the implementations by adding dependencies between consumed and produced data in the tile iterative and tile recursive implementations. In the PLUQ schemes we have much more dependencies than in the matrix multiplication schemes. Thus, we have much more dependencies to be handled by the runtime system. All experiments in this section are done using the `libkomp` runtime. The optimization of the kernels, mainly `pfgemm`, `pftrsm` and `flaswp`, that are called during the computation, improves greatly the overall performance of parallel PLUQ decomposition and gives a new state of the art over finite fields as shown next.

5.2. Parallel experiments on full rank matrices

Figure 10 shows that the tiled recursive parallel PLUQ implementation, without modular reductions, behaves better than the plasma quark `getrf` and matches the performance of the state of the art MKL `getrf`. This is mainly due to the bi-dimensional cutting which allows for a faster panel elimination, parallel hybrid `pftrsm` kernels, more balanced and adaptive `pfgemm` kernels and some use of Strassen-Winograd's algorithm. The use of the latter speeds up computation when matrx dimension gets larger.

Figures 10 and 11 show that in the `pluq` routine, the number of inner threads (*it*) used for the execution of kernel routines impacts the overall performance. In one hand, with larger number of inner threads (it=1024), the recursive parallel `pluq` routine using dataflow synchronizations has better behavior than with

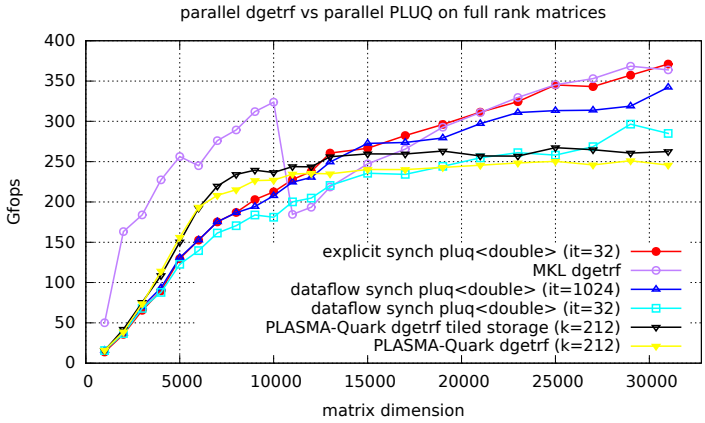

Figure 10: Effective Gfops of parallel LU factorization on full rank matrices without modular operations.

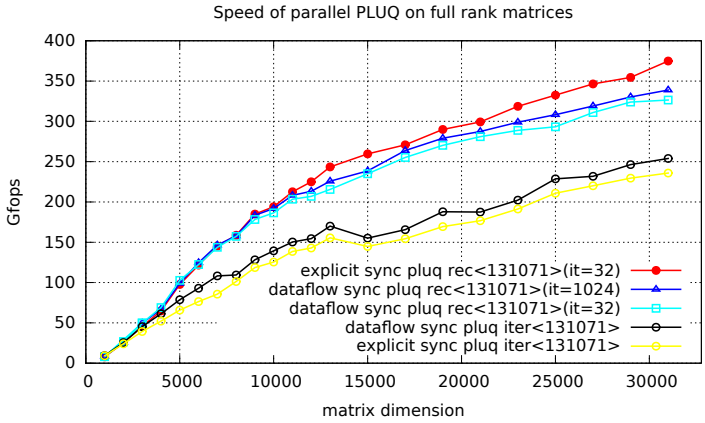

Figure 11: Parallel tiled recursive and iterative PLUQ over $\mathbb{Z}/131071\mathbb{Z}$ on full rank matrices on 32 cores

small number of inner threads (it=32). This is mainly due to the `libkomp` runtime system that handles more efficiently fine grain parallelization. But on the other hand, finer grain penalizes implementation with dataflow synchronizations over finite field because it takes less advantage from the use of fast variants in the matrix product kernels.

Figure 11 shows execution speed using modular reductions. It demonstrates that, parallel recursive `pluq` implementation has asymptotically same behavior with and without modular reductions. This is explained by the fact that the number of modular reductions is independent of the number of tasks in the case of tiled recursive `pluq` as proved in table 1. Whereas, the tiled iterative `pluq`

routine uses smaller granularity and thus is impacted by the number of modular reduction and takes clearly less advantage from fast variants in the `fgemm` kernels. Nevertheless, its implementation using dataflow synchronizations benefits more from small grain size than when using explicit task synchronizations.

5.3. Parallel experiments on rank deficient matrices

We show here performance obtained in the case of rank deficient matrices. Figure 12 shows execution speed of parallel PLUQ versions on matrices with rank equal to half their dimensions. Linearly independent rows and columns of the generated matrix are uniformly distributed on the dimension.

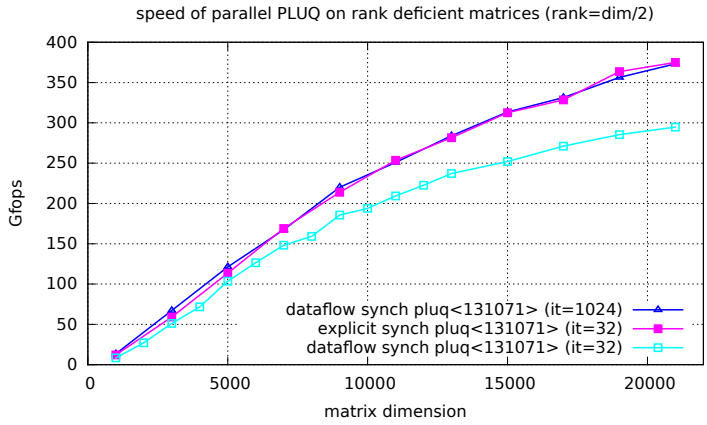

Figure 12: Perforamnce of tiled recursive PLUQ on 32 cores. Matrices rank is equal to half their dimensions. The Speed here ($2 \times n^3/3 \times time$ does not correspond to the real number of operations when rank is less than the dimension)

The implementation with OpenMP of the tiled recursive PLUQ maintained a high efficiency in the case of rank deficient matrices.

Using recursive variants linked against the `libkomp` library with dataflow synchronizations, mapping data on different NUMA nodes that helps reducing 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 tiled recursive PLUQ factorization. Comparing to results in [5] we have 18% gain for matrix multiplication and 23% gain in the case of PLUQ decomposition. The best performance is obtained with the parallel recursive PLUQ variant using the *2D recursive adaptive* variant for matrix multiplication algorithm and the hybrid parallel `pftrsm` variant.

6. Conclusion

We studied in this work several implementations of the PLUQ sub-kernels and showed that the best overall performance is obtained with tiled recursive

and hybrid kernels. Our results depends on different aspects. First, the choice of the algorithm that performs the best in parallel depends on the used sub-kernels. A best behaving parallelized kernel can turn to be less efficient when called concurrently with other kernels. Second, the cost of modular reductions has less impact in the recursive PLUQ scheme. These two features combined with the use of fast variants gave performance nearing state of the art numerical libraries.

Perspective. Our future work focuses on two main optimizations. First, the parallelization of fast variants can speed up performance of matrix multiplication computation. The focus will be on the scheduling heuristics that will reduce as much as possible task dependencies. Second, the distant data accesses has an impact on the overall performance. Thus, adapting the communication avoiding techniques of [11] is highly relevant. Over finite fields, an idea is to precompute the position of all pivots in parallel, then gather them in one block, before beginning the elimination. This reduces distant accesses in the pivot research phase of the Gaussian elimination algorithm.

[1] B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In J. P. May, editor, *ISSAC'2009, Proceedings of the 2009 ACM International Symposium on Symbolic and Algebraic Computation, Seoul, Korea*, pages 135–143. ACM Press, New York, July 2009. URL: http://hal.archives-ouvertes.fr/hal-00163141.

[2] F. Broquedis, T. Gautier, and V. Danjean. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In *IWOMP*, pages 102–115, Rome, Italy, jun 2012.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009. doi:http://dx.doi.org/10.1016/j.parco.2008.10.002.

[4] J. J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014. URL: http://hal.inria.fr/hal-00809765.

[5] J.-G. Dumas, T. Gautier, C. Pernet, and Z. Sultan. Parallel computation of echelon forms. In *Euro-Par 2014, Proceedings of the 20th international conference on parallel processing, Porto, Portugal*, volume 8632 of *Lecture Notes in Computer Science*, pages 499–510, Aug. 2014. URL: http://arxiv.org/abs/1402.3501.

[6] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over prime fields. *ACM TOMS*, 35(3):1–42, Nov. 2008. URL: http://arxiv.org/abs/cs/0601133.

[7] J.-G. Dumas, C. Pernet, and Z. Sultan. Simultaneous computation of the row and column rank profiles. In M. Kauers, editor, *Proc. ISSAC'13, Grenoble, France*, pages 181–188. ACM Press, New York, June 2013.

[8] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

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

[10] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *In Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, jun 2013.

[11] L. Grigori, J. W. Demmel, and H. Xiang. CALU: a communication optimal lu factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.

[12] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–756, 1997.

[13] C.-P. Jeannerod, C. Pernet, and A. Storjohann. Rank-profile revealing Gaussian elimination and the CUP matrix decomposition. *J.Symb.Comp.*, 56:46–68, 2013.

[14] J. Jelinek and *et al.* The GNU OpenMP implementation, 2014. URL: https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[15] K. Klimkowski and R. A. van de Geijn. Anatomy of a parallel out-of-core dense linear solver. In *ICPP*, volume 3, pages 29–33. CRC Press, aug 1995.

[16] A. Rémy, M. Baboulin, M. Sosonkina, and B. Rozoy. Locality optimization on a NUMA architecture for hybrid LU factorization. In M. Bader, A. Bode, H. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 153–162. IOS Press, 2013. URL: https://hal.inria.fr/hal-00957673, doi:10.3233/978-1-61499-381-0-153.

[17] W. Stein. *Modular forms, a computational approach*. Graduate studies in mathematics. AMS, 2007. URL: http://wstein.org/books/modform/modform.

[18] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.