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] *Univ. Grenoble Alpes, LJK, CNRS, Inria*
[b] *Inria, LIG, Univ. Grenoble Alpes, CNRS*
[c] *Univ. Grenoble Alpes, LIP, CNRS, Inria, UCB Lyon 1, ÉNS de Lyon*

Abstract

We present block algorithms and their implementation for the parallelization of sub-cubic Gaussian elimination on shared memory architectures. Contrarily to the classical cubic algorithms in parallel numerical linear algebra, we focus here on recursive algorithms and coarse grain parallelization. Indeed, sub-cubic matrix arithmetic can only be achieved through recursive algorithms making coarse grain block algorithms perform more efficiently than fine grain ones. This work is motivated by the design and implementation of dense linear algebra over a finite field, where fast matrix multiplication is used extensively and where costly modular reductions also advocate for coarse grain block decomposition. 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 tile recursive versions of the PLUQ decomposition, 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 more than 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, Rank deficiency

[☆]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 26 23 39 67; fax : +33 4 72 72 80 80

1. Introduction

Dense Gaussian elimination over a 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 studied for 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, where some intermediate computations also involve dense linear algebra:

- sparse direct methods, may switch to dense Gaussian elimination when the fill-in becomes too large [30, §10.3];
- block iterative methods (like the block-Wiedemann or block-Lanczos algorithms) also require dense linear algebra to handle the block projections.

Recently, efficient sequential exact linear algebra routines have been developed [12]. 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 [15, 3], computational number theory [27], or integer linear programming [18] 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.

Hence a natural ingredient in the design of efficient dense linear algebra routines is the use of block algorithms that results in gathering arithmetic operations in matrix-matrix multiplications [7]. Those can take full advantage of vectorized SIMD instructions and have a high computation per memory access rate, allowing to almost fully overlap the data accesses by computations and hence deliver close to peak performance efficiency. A key feature of exact dense linear algebra, is that fast matrix multiplication algorithms, like Strassen [28] and Strassen-Winograd algorithms [16, 12] can be used with no concern of numerical instability. The complexity improvement of these algorithms also gives a significant speed-up in practice [12]. In order to benefit from these sub-cubic time matrix multiplication algorithms, all other linear algebra computations, including Gaussian elimination need to reduce to it by block recursive algorithms. Hence, among the many variants of block algorithms, we only focus on the recursive ones.

In a previous work [11], we presented our first investigations on the parallelization of exact dense Gaussian elimination for multicore computers. The focus there was on exploring the variants of parallel exact Gaussian elimination algorithms (block iterative or recursive, using slabs or tiles, etc) and showed how and why the tile recursive variant outperforms the others.

We now focus in this manuscript on the building blocks on which these elimination algorithms rely. Our approach is to also apply recursive algorithms

with a task based parallelization, for these building blocks. We explore six variants for the parallelization of the matrix multiplication (five recursive variants compared to the block iterative algorithm of [11]) and introduce a new hybrid parallel algorithm for the triangular system solve with matrix right hand side. We then recall the slab and tile recursive Gaussian elimination algorithms and present their performance using these new building blocks.

The scope of this study extends more generally to the problem of parallelizing any set linear algebra routines based on sub-cubic time matrix arithmetic, such as Strassen's $O(n^{2.81})$ algorithm. In particular, numerical linear algebra based on Strassen's algorithm (if numerical stability issues have been considered acceptable) should clearly benefit from most of its results. Related work on the parallelization of the sub-cubic numerical linear algebra include [1, 24, 6, 25, 2].

Our focus is on parallel implementations using various pivoting strategies that will reveal the echelon form, or the rank profile of the matrix [21, 13]. The latter is a key invariant used in many applications such as Gröbner basis computations [15] and computational number theory [27].

As the 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 [23], or both dimensions, producing tiles [5].

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.

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` [22], the GNU implementation of the `OpenMP` Application Programming Interface and `libkomp` an implementation of the `OpenMP` standard based on the `XKaapi` library [17]. 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 `XKaapi` library handles

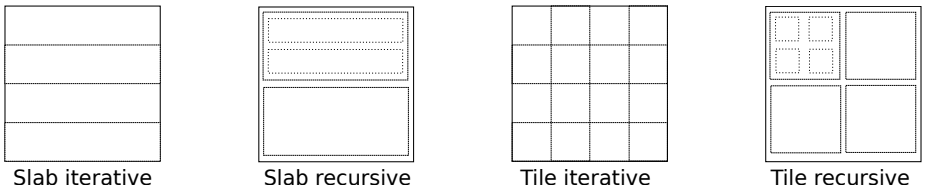

Figure 1: Main types of block splitting

better parallelization with fine-granularity as we show in section 2 by comparing the `libkomp` and `libgomp` runtime systems. However, over a finite field, 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.

Methodology of experiments

All experiments have been conducted on 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` v0.2.11, `Intel-MKL` sp1.1.106, `LAPACK` v3.4.2 and `Plasma-Quark` v2.5.0. We used the `XKaapi`-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, also used in [12, 25, 2] defined as

$$\texttt{Gfops} = \frac{\#\text{ of field ops using classic matrix product}}{\text{time}}.$$

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 with 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 [11]. 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

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

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 [26, 5]. Over a finite field, while some aspects remain similar, the following particularities need to be considered:

Impact of modular reductions. Computations over a finite field are done, first, by embedding the finite field elements in integers stored as floating point numbers. Secondly, modular reduction operations are applied to convert back elements over the finite field. To reduce the number of modular reductions, several multiplications can be accumulated before the reduction while keeping the result exact. To maximizes the computational bandwidth, operations are grouped in floating point matrix multiplications as much as possible, so as to benefit from an optimized BLAS. This approach [10, 12] is only valid as long as the integer computation does not exceed the capacity of the mantissa. For instance, in a matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$, with inner dimension n, the result and any intermediate computation is bounded by $n(p-1)^2$, provided that the algorithm did not perform any other operation than additions and depth 1 multiplications. Hence the computation with an m-bit mantissa is guaranteed to not overflow as long as $n(p-1)^2 < 2^m$.

Furthermore, in a block LU factorization algorithm, the output of a block operation needs to be reduced modulo p. Hence the choice of a small block size increases the overall number of modular reductions, and therefore the computing time. This is one argument in favor of a coarse granularity in our algorithms. Table 1 taken from [11] shows the impact of the block size for iterative and recursive algorithms on the number of modular reductions. This table demonstrates

$k \mid 1$	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	$(1 + \frac{1}{4}\log_2 n)n^2 - \frac{1}{2}n\log_2 n - n$

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.

that the number of modular reductions is smaller in the case of tile recursive LU factorization, which is one motivation for the use of the tile recursive variant over a finite field.

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 [8], in the case of the PLUQ decomposition). It is an important parameter for optimizations: 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 memory bus traffic. In numerical linear algebra, where cubic time algorithm are used, the arithmetic cost is independent of the cutting in blocks. Hence the granularity has very little impact on the efficiency of a block algorithm run sequentially. On the contrary, we saw in Table 1 that over a finite field, a finer granularity can lead to a larger number of costly modular reductions. The use of sub-cubic variants for the sequential matrix multiplications is another reason why coarser a granularity lead to a higher sequential efficiency. On the other hand, the granularity needs to be fine enough so as to generate enough independent tasks to be executed in parallel. Therefore, with a fixed number of resources, we will rather set the number of tasks to be created (usually to the number of available cores, or slightly more), instead of setting a fixed small grain size as usually done in numerical linear algebra. Hence, an increase in the dimensions, will result in a coarser granularity, making each sequential task perform more efficiently.

Asymptotically fast 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 [16, §12] can be systematically used on top of the BLAS [12]. Table 2 shows the impact on the performance of this sub-cubic variant compared to the classical matrix multiplication. In

n	1024	2048	4096	8192	16384
`OpenBLAS sgemm`	**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**
`OpenBLAS dgemm`	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

this table we compare the sequential speed of both variants implemented as the

`fgemm` routine of the `fflas-ffpack` library linked against `OpenBLAS`. In table 2, computations are done over a small finite field (modulo 37), a large finite field (modulo 131071) and over integers without modular reductions, directly calling `OpenBLAS sgemm` or `dgemm` routines, to show the impact of modular reductions. Single precision floats and the `sgemm` routine are used for elements Modulo 37, whereas modulo 131071, double precision and the `dgemm` routine are used.

Table 2 first shows that the overhead of performing the modular reductions in the $O(n^3)$ implementations is noticeable, although limited. Then, when enabling Strassen-Winograd $O(n^{2.81})$ algorithm, a speed-up factor of up to 1.5 can be attained in both single and double precision arithmetic.

Strassen-Winograd algorithm can also naturally be used in parallel. We will restrict ourselves to using a classical block algorithm to generate parallel tasks, each of which will use a sequential Strassen-Winograd algorithm. All our attempts to parallelize Strassen-Winorgrad algorithm directly never reached performances as good as the above strategy.

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 [4] 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 feature of tasks with data-flow dependency of the `OpenMP`-4.0 standard. 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) diagram that precomputes dependencies of all tasks before execution. This feature helps reduce the idle time of resources by removing unnecessary synchronizations. We will see in the next sections the im-

pact 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. 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. In the following section, and more generally in the `fflas-ffpack` library, we use our custom data-mapping for coarse grain data: data are initialized with two parallel for loops. Each iteration is incremented with a fixed chunk size. This is equivalent to using the `numactl -interleave` command, but with a coarser grain, better suited to the data access pattern of our parallel algorithms. 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). When all data are allocated on the same node, the performances are degraded by the latency to transfer part of it to distant nodes. This effect is naturally minimized when all nodes store an equal part of the data. As a comparison, the data placement provided by the `numactl -interleave` command generates a drop of up to 11% in 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 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. For the choice of the classical parallel block algorithm, we explore a variety of well known 2D and 3D cutting strategies, with their iterative or recursive variants. The routines perform the operation $C \leftarrow A \times B$, where A, B and C are dense matrices with dimensions respectively (m, k), (k, n) and (m, n).

3.1. Algorithmic variants

The 2D partitioning. The strategy splits the row dimension of A and the column dimension of B and each parallel task computes a submatrix of C. These tasks are therefore all independent. 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.

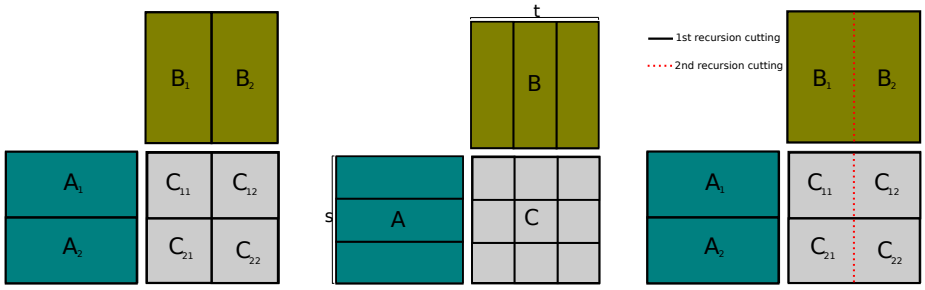

Figure 2: 2D partitioning: recursive (left), iterative (middle) and recursive adaptive (right) cutting.

The 3D partitioning. The strategy parallelizes the computation over dimensions m, n and k and several parallel tasks contribute to the computation of a single submatrix of C. Again we present three variants:

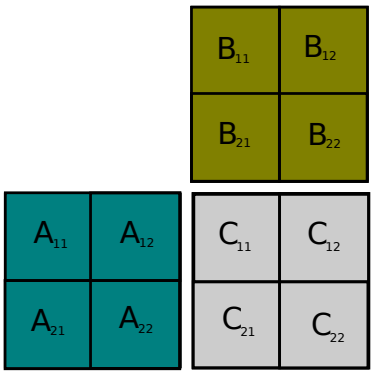

Figure 3: 3D partitioning : cutting A and B according to dimensions m, n and k.

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.

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)

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 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.

Implementation 2 *3D-Inplace recursive* using `OpenMP`-4.0 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)

In this `OpenMP` Implementation 1, each task calls recursively the *3D-Inplace recursive* routine. Using `OpenMP` 4.0 directives helps specifying dependencies between tasks, indicating when 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. Since the split along the inner dimension introduces some overhead, we introduce a weighted penalty system to determine when to split. For a penalty factor of p, the inner dimension only splits when $\max(m, n) < pk$.

In all these recursive schemes, the recursion is stopped when the number of threads allocated 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* variant 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 allows for a better scheduling of the tasks.

3.2. Experiments on square matrices

3.2.1. Comparing all variants over $\mathbb{Z}_{131071}$

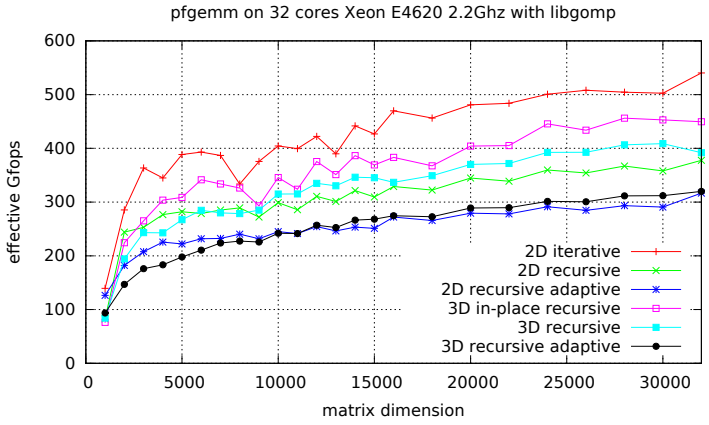

Figure 4: Speed of the matrix multiplication variants using `libgomp`

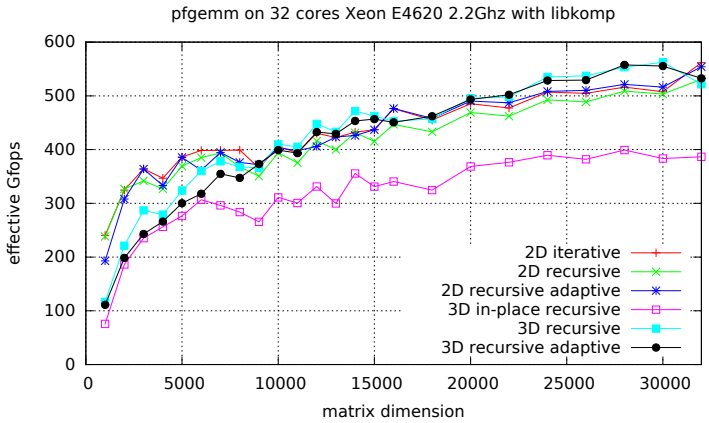

Figure 5: Speed of the matrix multiplication variants using `libkomp`

Figures 4 and 5 show the execution speed of all variants over the field $\mathbb{Z}_{131071}$, using `OpenMP` 4.0 task model, linked with the two runtimes `libgomp` and `libkomp` respectively. The number of sequential tasks is limited to the number of threads available, a constant, which implies that the asymptotic complexity of the parallel algorithm is that of Strassen-Winograd's algorithm: $O(n^{2.81})$.

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 `libgomp`. 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 numerical matrix multiplications: the `dgemm` implementation of `Plasma-Quark`, and `Intel-MKL` and the implementation of `pfgemm` of `fflas-ffpack` using `OpenMP`-4.0 dataflow model. This implementation is run with or without Strassen-Winograd matrix product. `Intel-MKL dgemm` performs consistently faster than all other cubic time variants, which perform rather similarly. However, the speed-up of Strassen-Winograd algorithm makes `pfgemm` faster on larger matrices.

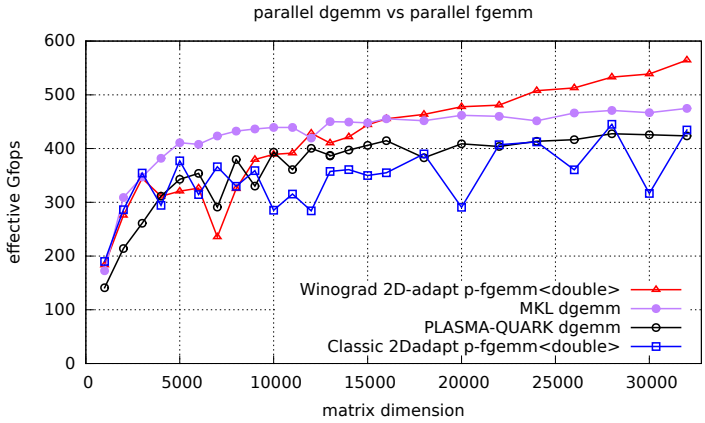

Figure 6: Speed of numerical matrix multiplication routines

3.3. Experiments on rectangular matrices

As shown in the previous 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. 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 speed gets quickly similar in all three variants.

This fact, combined with the results of Figure 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 slightly slower computations.

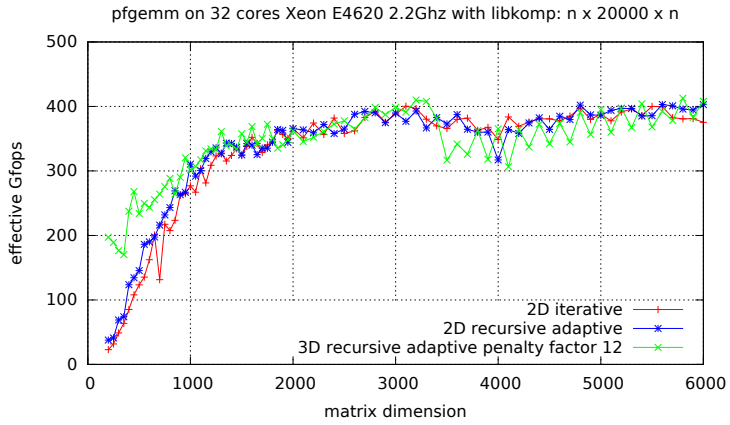

Figure 7: Computation speed on rectangular matrices with large inner dimension k.

4. Parallel triangular solving with matrix unknown

In this section we study various cutting strategies for the computation of parallel `pftrsm` routine. We identify three different types of parallelizations: 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 $X \leftarrow L^{-1}B$.

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

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

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

Algorithm 2 Recursive TRSM

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

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 may 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.

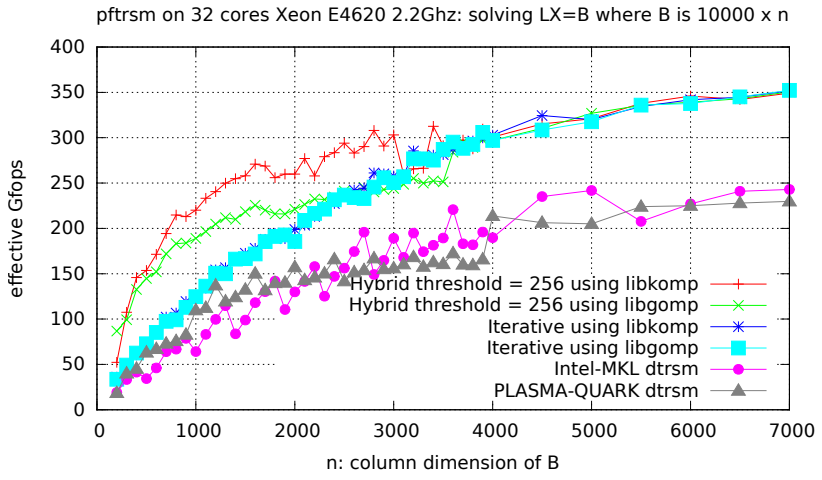

Figure 8: Comparing the Iterative and the Hybrid variants for parallel `ftrsm` using `libkomp` and `libgomp`. Only the outer dimension varies: B and X are $10000 \times n$.

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. Lastly, the performances of the numerical `dtrsm` routines of `Intel-MKL` and `Plasma-Quark` are shown, and appear to be consistently slower.

5. Parallel Gaussian elimination

In this section, we present the parallelization of two different algorithms for the computation of the PLUQ decomposition: a tile iterative and a tile recursive algorithm. Although this computation has already been widely studied for numerical computation [8], applications over a finite field have additional specific requirements and constraints that need to be taken into consideration. In particular, rank deficiency is a well defined notion there, and many applications rely on the computation of the echelon form, or the rank profile of the matrix [21, 13] only revealed by certain pivoting strategies in the PLUQ factorization algorithm.

In [11], parallel iterative and recursive implementations of exact PLUQ decomposition revealing the rank profiles of the matrix are presented. It is there shown that the parallel recursive implementation behaves 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 blocks presented in the previous sections and some new tasking strategies using data-flow dependencies. We will 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 [21] 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 [12]) 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 [13, 14]. As a by product, both row and column rank profiles are also computed simultaneously.

It has been shown in [11] that the slab iterative algorithm performing a PLUQ decomposition is slow due to large sequential tasks (see [11, § 4, Figure 5]). At each iteration a PLUQ decomposition is called sequentially on large

slab blocks of size $k \times n$, where k is the block size and n is the column dimension of the input matrix. These sequential tasks are costly and therefore impose a finer granularity. We will therefore limit our study to the tile iterative and tile recursive variants.

Tile iterative algorithm. We use the tile iterative algorithm presented in [11]. It is constructed from the slab iterative algorithm, where the panel computation is split it into column tiles. With this splitting [11, § 4, Figure 6], the operations remain more local and updates can be parallelized. This optimization used in the computation of the slab factorization improved the computation speed by a factor of 2. This approach shares similarities with the recursive computation of the panel described in [9].

Moreover, the workload of each block operation may strongly vary, depending on the rank of the corresponding slab. Such heterogeneous tasks loads lead us to opt for work-stealing based runtime systems instead of static thread management. This is also a place where data-flow dependencies are expected to behave better than explicit task synchronizations.

Tile recursive algorithm. Recursive algorithms in dense linear algebra are a natural choice for hierarchical memory systems [29]. 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 [20]. The parallelization of the recursive variant of PLUQ decomposition over a finite field is presented in [11] 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. As an illustration, Algorithm 3 shows

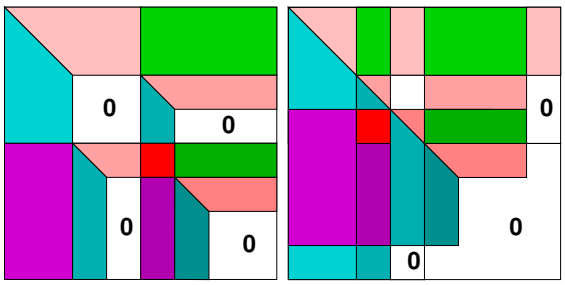

Figure 9: PLUQ quad-recursive scheme

the first half of this algorithm implemented with `OpenMP` tasks with dataflow dependencies. Compared to [11], we introduce in the present implementations the dataflow dependencies between consumed and produced data in the tile iterative and tile recursive implementations, together with the use of optimized building blocks for `pfgemm` and `pftrsm`.

Implementation 3 `PLUQ` (A) tile recursive algorithm

if $\min(m,n) < T$ **then**

Base Case done by an iterative PLUQ decomposition

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`)`

`gemm`(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`)`

`gemm`(C_2, E, V_1) ▷ $G \leftarrow C_2 - E V_1$

`#pragma omp task shared`(A_4, E, D) `depend(in:`E, D`) depend(inout:`A_4`)`

`gemm`(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) ▷ $\begin{bmatrix} D_1 & D_2 \end{bmatrix} \leftarrow D Q_2^T$

`#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 [13, Alg 1])

`#pragma omp taskwait`

5.2. Parallel experiments on full rank matrices

In figures 10 and 11, we compare the parallel behavior of the `PLUQ` decomposition using `explicit synchronizations` and `dataflow synchronizations`. 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.

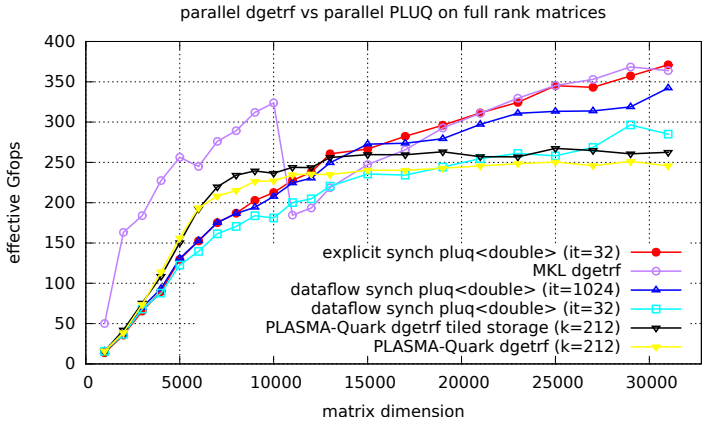

Figure 10: Effective Gfops of numerical parallel LU factorization on full rank matrices.

Figure 10 shows that our tile recursive parallel PLUQ implementation, without modular reduction, 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 algorithm. The use of the latter speeds up computation when the matrix dimension gets larger.

Figure 11 compares execution speed over a finite field. It first shows how the tile recursive variants performs faster than the tile iterative variants, mostly for their lesser 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.

In figures 10 and 11, the number of inner threads (it) allocated to the computation impacts the overall performance. On one hand, a large number of inner threads (it=1024) makes the recursive parallel `PLUQ` routine with dataflow

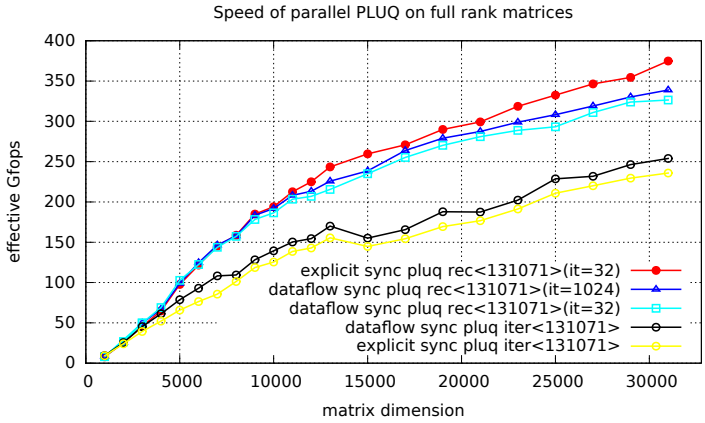

Figure 11: Parallel tile recursive and iterative PLUQ over $\mathbb{Z}/131071\mathbb{Z}$ on full rank matrices on 32 cores

synchronizations perform faster. This illustrates that finer granularity provides more tasks and therefore reduces CPU idle time. The fact that the `libkomp` runtime handles numerous tasks, helps making this happen. On the other hand, finer grain acts also as a penalty because reduces the speed-up of Strassen-Winograd algorithm.

5.3. Parallel experiments on rank deficient matrices

Figure 12 shows the execution speed of the parallel PLUQ variants on matrices with rank equal to half their dimension. Overall the computation speed

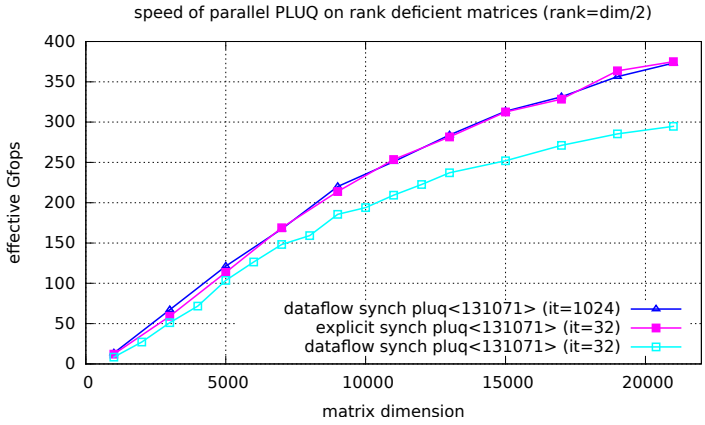

Figure 12: Performance of tile recursive PLUQ on 32 cores. Matrices rank is equal to half their dimensions. Note that the speed here ($2/3 \times n^3/time$) does not correspond to the real number of operations when rank is less than the dimension.

remains of the same order of magnitude. The overhead of additional permutations introduced by the rank deficiency is compensated by the reduced amount of arithmetic operations to be performed. This time again, the variant with explicit synchronizations performs best, but the data-flow synchronization variant matches its speed when the granularity is increased to 1024 inner threads.

6. Conclusion

We studied in this work several implementations of the subroutines used by a parallel recursive PLUQ decomposition algorithm. We identified that the best recursive variant for matrix multiplication is the 2D adaptive variant, combined with a sequential Strassen-Winograd algorithm. While the impact of modular reductions seem to be limited, 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.

These two building blocks, combined in our recent tile recursive algorithm deliver a high computing efficiency. Comparing to our previous results in [11] experiments show a 18% gain for matrix multiplication and 23% gain for 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. 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, which in the other hand, implies a finer grain, and therefore a lesser improvement of the sub-cubic matrix multiplication algorithms.

Perspective. Our future work focuses on two main directions. First, improving the parallelization of the recursive steps of Strassen-Winograd algorithm directly. The focus will be on the scheduling heuristics that will reduce as much as possible task dependencies, while keeping the memory footprint contained. Second, the distant data accesses has an impact on the overall performance. Adapting the communication avoiding techniques of [19] in the framework of our recursive algorithm is highly relevant. Over a finite field, tournament pivoting seem to reduce to computing the union of the non-zero pivots found in concurrent eliminations. It is still unclear whether the rank profile information can still be revealed with such an algorithm.

Acknowledgment

We are grateful to the anonymous referees whose reviews helped to greatly improve the quality of this document.

References

[1] D. H. Bailey, K. Lee, and H. Simon. Using strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1991. doi:10.1007/BF00129836.

[2] 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, pages 42–53, New York, NY, USA, 2015. ACM. doi:10.1145/2688500.2688513.

[3] L. Bettale, J.-C. Faugre, and L. Perret. Cryptanalysis of hfe, multi-hfe and variants for odd and even characteristic. *Designs, Codes and Cryptography*, 69(1):1 – 52, 2013. doi:10.1007/s10623-012-9617-2.

[4] 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.

[5] 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.

[6] 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. *ACM Trans. Math. Softw.*, 38(1):2:1–2:30, Dec. 2011. doi:10.1145/2049662.2049664.

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A proposal for a set of level 3 basic linear algebra subprograms. *SIGNUM Newsl.*, 22(3):2–14, July 1987. doi:10.1145/36318.36319.

[8] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, 1998.

[9] 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.

[10] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *Proc. ISSAC'02*. ACM Press, July 2002. doi:10.1145/780506.780515.

[11] J.-G. Dumas, T. Gautier, C. Pernet, and Z. Sultan. Parallel computation of echelon forms. In F. Silva, I. Dutra, and V. Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing, 2014. doi:10.1007/978-3-319-09873-9_42.

[12] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Trans. on Mathematical Software (TOMS)*, 35(3):1–42, 2008. doi:10.1145/1391989.1391992.

[13] 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. doi:10.1145/2465506.2465517.

[14] J.-G. Dumas, C. Pernet, and Z. Sultan. Computing the Rank Profile Matrix. In *Proc. ISSAC'15, Bath, U.K.*, pages 149–156. ACM Press, New York, July 2015. doi:10.1145/2755996.2756682.

[15] 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.

[16] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3 edition, 2013.

[17] 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.

[18] A. M. Gleixner, D. E. Steffy, and K. Wolter. Improving the accuracy of linear programming solvers with iterative refinement. In *Proc. of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, pages 187–194, New York, NY, USA, 2012. ACM. doi:10.1145/2442829.2442858.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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*, pages 582–588, Apr 1993. doi:10.1109/IPPS.1993.262814.

[25] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz. Communication-avoiding parallel strassen: Implementation and performance. In *Proc.*, SC '12, pages 101:1–101:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=2388996.2389133.

[26] 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, *Proc. Parallel Computing, ParCo 2013, 10-13 September 2013, Garching, Germany*, pages 153–162. IOS Press, 2013. URL: https://hal.inria.fr/hal-00957673, doi:10.3233/978-1-61499-381-0-153.

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

[28] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969. doi:10.1007/BF02165411.

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

[30] Z. Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, 1991.