REPORT ON OpenDreamKit DELIVERABLE D5.14

Implementations of exact linear algebra algorithms on distributed memory et heterogenous architectures: clusters and accelerators. Solving large linear systems over the rationals is the target application.

ALEXIS BREUST, JEAN-GUILLAUME DUMAS, CLÉMENT PERNET, HONGGUANG ZHU

Due on	31/08/2019 (M48)
Delivered on	XX/YY/201Z
Lead	Université Grenoble Alpes (UGA)
Progress on and finalization of this deliverable has been tracked publicly at: https://github.com/OpenDreamKit/OpenDreamKit/issues/112	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #112 ON 2019-08-27

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** Université Joseph Fourier
- **Due:** 2019-08-31 (month 48)
- **Nature:** Demonstrator
- **Task:** T5.3 (#101)
- **Proposal:** p. 51
- **Upcoming report** (sources)

CONTEXT

Computational linear algebra is a key tool delivering high computing throughput to applications requiring large scale computations. In numerical computing, dealing with floating point arithmetic and approximations, a long history of efforts has lead to the design of a full stack of technology for numerical HPC: from the design of stable and fast algorithms, to their implementation in standardized libraries such as LAPACK and BLAS, and their parallelization on shared memory servers or supercomputers with distributed memory.

On the other hand, computational mathematics relies on linear algebra with exact arithmetic, i.e. multiprecision integers and rationals, finite fields, etc. This leads to significant differences in the algorithmic and implementations approaches. Over the last 20 years, a continuous stream of research has improved the exact linear algebra algorithmic; simultaneously, software projects such as LinBox and fflas-ffpack were created to deliver a similar set of kernel linear algebra routines as LAPACK but for exact arithmetic.

The parallelization of these kernels has only been partially addressed in the past, and was mostly focused on shared memory architectures.

GOAL OF THE DELIVERABLE

This deliverable aims at enhancing these libraries so that they can exploit various large scale parallel architectures, including large multi-cores, clusters, and accelerators. The target application is the solver of linear systems of the field of multi-precision rational numbers. For this application, several algorithmic approaches will be studied and experimented, namely

a Chinese Remainder based solver and a p-adic lifting solver. The former exposes a much higher level of parallelism in its tasks, while the latter requires asymptotically much fewer operations.

CONTENTS

1. ALGORITHMIC ASPECT

Solving exactly over the field of rational numbers, requires a careful attention to the size of the numbers being manipulated. Indeed, rational numbers are represented by a pair of multiprecision integers, a numerator and denominator. Applying the standard direct methods designed for solving approximately systems with floating point rational numbers would lead to a blowup in the bit-size of the coefficients, and result in overwhelming computation overheads, with time complexities often not even polynomial.

1.1. State of the art

In a series of algorithmic innovations, summarized in Table 1, the time complexity for solving exactly a linear system over the rationals has been reduced to $O^{\sim}(n^3)$ or $O^{\sim}(n^\omega)$ which is asymptotically as fast as the complexity for solving approximately using floating point numbers.

First, the use of rational reconstruction makes it possible to recover the rational solution from the solution of the same system projected into a large enough finite field. This approach avoids the swell in the bit-size of the intermediate computations and compute with the arithmetic precision of the output for a total cost of $O(n^5)$. Then, using multi-modular representation and the Chinese Remainder Theorem, reduces this cost to $O^{\sim}(n^4)$. This cost, corresponding to the arithmetic complexity of Gaussian elimination multiplied by the bit-size of the output, is sub-optimal. By better balancing the computation load between numerous arithmetic operation on small bit-size and few operations on large bit-size, one can hope to further reduce this complexity. This is achieved by p-adic lifting techniques Dixon [1982], reaching a $O^{\sim}(n^3)$ time complexity. Improving the complexity for this problem is still a very active topic: Storjohann [2005] reduced it to $O^{\sim}(n^\omega)$ using randomization, and the same complexity can now be reached deterministically since this year Birmpilis et al. [2019].

While p-adic lifting techniques achieve the best complexity for sequential computations, they are unfortunately very iterative in their structure, and therefore harder to parallelize at a large scale. On the contrary, Chinese remainder based methods offer an embarrassingly parallel structure, but cost a factor of n more. We have therefore chosen to explore both approaches which will be described in the following subsections.

Method	Bit complexity
Gauss over $\mathbb{Q}$	$2^{O(n)}$
Gauss $\mod$ bound(sol)	$O(n^5)$
CRT $\times$ Gauss $\mod p$	$O\tilde{}(n^4), O\tilde{}(n^\omega + 1)$
p-adic lifting	$O\tilde{}(n^3), O\tilde{}(n^\omega)$

TABLE 1. Brief overview of algorithmic innovations for solving linear systems over the rationals

1.2. The Chinese remainder approach

We recall here the principle of the Chinese Remainder based rational solver: the linear system over the integers is reduced modulo a series of prime numbers. Each of these systems can then be solved independently, therefore exposing a large degree of parallelism. The rational solution can then be recovered in two phases by a Chinese remainder reconstruction followed by a vector rational reconstruction.

Algorithm 1 Chinese Remainder based rational solver

Require: $\mathbf{A} \in \mathbb{Z}^{n\times n}, \mathbf{b} \in \mathbb{Z}^n$
Ensure: $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$
1: $N, D \leftarrow$ SolutionBounds$(\mathbf{A}, b)$
2: Pick ℓ primes $p_1, \ldots, p_\ell$ such that $\prod_{i=1}^{\ell} p_i \geq 2ND$
3: **for** $i \leftarrow 1 \ldots \ell$ **do**
4: $\quad \mathbf{A}^{(i)} \leftarrow \mathbf{A} \mod p_i$
5: $\quad \mathbf{b}^{(i)} \leftarrow \mathbf{b} \mod p_i$
6: $\quad \mathbf{x}^{(i)} \leftarrow (\mathbf{A}^{(i)})^{-1}\mathbf{b}^{(i)} \mod p_i$
7: **end for**
8: $\mathbf{y} \leftarrow$ `ChineseRemainder`$(\mathbf{y}^{(1)}, p_1, \ldots, \mathbf{y}^{(\ell)}, p_\ell)$
9: $\mathbf{x}, d \leftarrow$ `VectorRationalReconstruction`$(\mathbf{y}, \prod_{i=1}^{\ell} p_i)$
10: **return** $\mathbf{x}/d$

In sequential versions of this algorithm, the Chinese remainder and rational reconstruction can be performed on the fly after each iteration, in order to detect stabilization and allow to terminate the loop earlier than what the possibly pessimistic bound on the solution requires. In parallel, this would generate dependency between the parallel tasks and we thus chose to disable this feature.

1.3. The p-adic lifting approach

The p-adic lifting approach, developed in Dixon [1982] is based on a Newton iteration to compute iteratively the digits in the p-adic expansion of the solution vector. This algorithm manages to reduce the cost of Chinese Remaindering by balancing two types of operations: expensive linear algebra (matrix inverse or LU decomposition) is performed only once over in small precision (a finite field) while the high precision part of the computation is contained in a sequence of less expensive linear algebra operations (matrix-vector products).

However, the strongly iterative structure of this algorithm makes it both harder to parallelize and less cache efficient. In Chen and Storjohann [2005], Storjohann and Chen devised a way to improve the cache efficiency thanks to a hybrid combination of p-adic lifting and Chinese remainder techniques. This led to a multi-modular (also called RNS, Residue Number System) p-adic lifting. We further improve this variant in several ways to deliver a parallel RNS p-adic lifting solver. The novel Algorithm 2 reflects these improvements with the two following salient features:

(1) The costly updates in the main loop of Chen and Storjohann [2005] were performed via matrix-vector multiplications with over the arbitrary precision integers with a second RNS basis. We instead gather these updates into a single matrix-matrix multiplication, having better cache efficiency and more suitable for parallelization. This can be seen in line 16 of Algorithm 2.

(2) We combine this algorithm with the parallel RNS improvement of Doliskani et al. [2018] and restrict the second RNS basis to primes individually larger than the ones of the higher-level RNS system, in order to have a free reduction of the input into the second RNS/integer basis.

Algorithm 2 Parallel multi-modular p-adic rational solver

Require: $\mathbf{A} \in \mathbb{Z}^{n\times n}, \mathbf{b} \in \mathbb{Z}^n$, ℓ a parameter
Ensure: $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$

1: $N, D \leftarrow \text{SolutionBounds}(\mathbf{A}, b)$
2: Pick ℓ primes $p_1, \ldots, p_\ell$ and let k such that $\prod_{i=1}^{\ell} p_i^k \geq 2ND$
3: **for** $i \leftarrow 1 \ldots \ell$ in parallel **do**
4: $\quad B_i \leftarrow A^{-1} \mod p_i$
5: $\quad \mathbf{r}^{(i)} \leftarrow \mathbf{b} \mod p_i$
6: $\quad \mathbf{y}^{(i)} \leftarrow 0$
7: **end for**
8: Pick primes $q_1, \ldots, q_\tau$, all $> \max_i\{p_i\}$, such that $\prod_{s=1}^{\tau} q_s \geq n||A||_\infty$
9: **for** $j \leftarrow 0 \ldots k-1$ **do**
10: $\quad$ **for** $i \leftarrow 1 \ldots \ell$ in parallel **do**
11: $\quad\quad (\phi^{(i)}, \rho^{(i)}) \leftarrow \texttt{divmod}(\mathbf{r}^{(i)}, p_i)$ ▷ $\mathbf{r}^{(i)} = p_i\phi^{(i)} + \rho^{(i)}$
12: $\quad\quad \mathbf{c}^{(i)} \leftarrow \mathbf{B_i}\rho^{(i)} \mod p_i$
13: $\quad\quad \mathbf{y}^{(i)} \leftarrow \mathbf{y}^{(i)} + \mathbf{c}^{(i)} p_i^j$
14: $\quad$ **end for**
▷ Now compute simultaneously all the updates $\frac{\mathbf{r}^{(i)} - \mathbf{A}\mathbf{c}^{(i)}}{p_i}$ in parallel:
15: $\quad$ Gather $\mathbf{R} \leftarrow \left[\begin{array}{ccc} \rho^{(1)} & \ldots & \rho^{(\ell)} \end{array}\right]$ and $\mathbf{C} \leftarrow \left[\begin{array}{ccc} \mathbf{c}^{(1)} & \ldots & \mathbf{c}^{(\ell)} \end{array}\right]$
16: $\quad \mathbf{V} \leftarrow \mathbf{R} - \mathbf{A} \cdot \mathbf{C}$ ▷ Parallel RNS with $q_1, \ldots, q_\tau$
17: $\quad$ **for** $i \leftarrow 1 \ldots \ell$ in parallel **do**
18: $\quad\quad \mathbf{r}^{(i)} \leftarrow \phi^{(i)} + \frac{\mathbf{V}_{*,\mathbf{i}}}{p_i}$ ▷ Multiply by $(p_i^{-1} \mod q_s)$ in RNS, then reconvert to $\mathbb{Z}^n$
19: $\quad$ **end for**
20: **end for**
21: $\mathbf{y} \leftarrow \texttt{ParallelChineseRemainder}(\mathbf{y}^{(1)}, p_1^k, \ldots, \mathbf{y}^{(\ell)}, p_\ell^k)$
22: $\mathbf{x}, d \leftarrow \texttt{VectorRationalReconstruction}(\mathbf{y}, \prod_{i=1}^{\ell} p_i^k)$
23: **return** $\mathbf{x}/d$

2. High performance parallelization

The implementations referred to for this deliverable are made available in the LinBox library. Experiments were performed on the Luke and Dahu clusters of the GRICAD-CIMENT[1] HPC federation of Grenoble site. The 2 multi-core servers funded by OpenDreamKit project have been integrated within the luke cluster and share with the community (with highest priority access to OpenDreamKit members). In exchange, we were able to experiment with any server on the data-center. This allowed us to access to a high-end GPU server, and more importantly, to a cluster with many multiple homogeneous nodes, the cost of which requires the implementation of such federation of user projects.

[1] https://gricad.univ-grenoble-alpes.fr/

2.1. **MPI based Chinese remainder algorithm**

We first report on the implementation of the parallelization of Algorithm 1 with various matrix dimensions and entry bit-size. On a cluster where each node is equipped with 2×32 Intel Gold-6130 @2.1GHz cores, we obtain the timings reported in Figure 1. This is for instance a speed up of about 160 on 8 nodes.

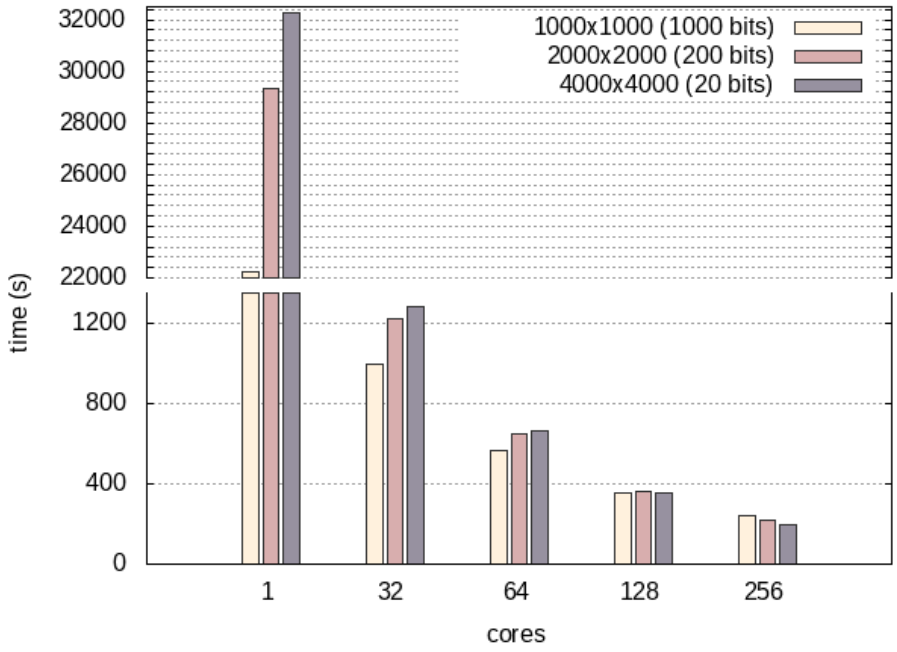

FIGURE 1. MPI-Chinese Remainder based rational solver.

The parallelization here is performed only on the main loop, that is lines 4 and 5 (RNS reduction) and line 6 (Solve), whereas the Chinese Remainder reconstruction (line 8 – CRT) and the rational reconstruction (line 9 – RR) are performed on a single core. Note that the Chinese remaindering reconstruction phase, with such large dimension and bit-size could become a bottleneck with a classic implementation. However the asymptotically fast algorithm in LINBOX for this task, made this operation negligible compared to the core of the iteration. The profiling of the timings, given in Table 2, confirms that the Wall time of these two latter parts is not dominating. As a consequence, it was parallelization of these operations was not of high priority and we chose to delay to future work.

We also report in Table 3 on some prototype hybrid MPI/OpenMP implementation where on each node we switch to use shared memory parallelism with OpenMP instead of having MPI handling also the in-node parallelism. On this particular implementation and cluster OpenMP is, for some unknown reason, much slower than MPI and therefore our Hybrid algorithm is slower than the pure MPI implementation. Nonetheless this hybrid implementation allows to not duplicate the data within each node and therefore enables the solution of much larger matrices, for instance a solving a random 10000×10000 system with 100 bits entries and obtaining an exact solution vector of 2.5GB (that is more than two million bits per coefficient of the output).

2.2. **A multi-core p-adic lifting**

As an alternative to the overwhelming workload of the Chinese Remainder approach, we now focus on Algorithm 2 and its implementation in a multi-core parallel computing setting. All the

Cores	N	Bit-size	ℓ	RNS CPU (s)	Solve CPU (s)	CRT Wall (s)	RR Wall (s)	Total Wall (s)
64	1000	1000	91364	11537.1	17080.8	67.4	69.6	619.6
64	2000	200	37362	6740.0	31138.3	50.1	24.9	702.7
64	4000	20	9450	2702.9	39037.1	19.6	6.9	711.9
128	1000	1000	91364	11536.6	17136.8	67.6	71.1	398.4
128	2000	200	37362	6811.4	31193.5	50.4	25.4	401.7
128	4000	20	9450	2733.1	39307.9	21.1	7.2	379.3
256	1000	1000	91364	11565.7	17138.2	70.2	73.5	290.8
256	2000	200	37362	6814.9	31245.4	51.0	25.5	253.4
256	4000	20	9450	2738.3	39405.1	19.1	6.9	209.9

TABLE 2. Respective timings of the steps of the MPI Chinese Remainder based rational solver on 8 nodes of 32 Intel Xeon Gold 6130 code.

Nodes	2	2	2	2	2	2		
Threads	1	16	1	16	1	16		
N	1000	1000	2000	2000	4000	4000		
Bit-size	1000	1000	200	200	20	20		
Wall-time (s)	425.5	880.2	434.4	812.9	354.0	866.2		
Nodes	4	4	4	4	4	4	6	6
Threads	1	16	1	16	1	16	1	16
N	1000	1000	2000	2000	4000	4000	10000	10000
Bit-size	1000	1000	200	200	20	20	40	40
Wall-time (s)	302.4	519.5	285.2	422.0	240.6	422.7	-	12026.3
Nodes	8	8	8	8	8	8	8	8
Threads	1	16	1	16	1	16	1	16
N	1000	1000	2000	2000	4000	4000	10000	10000
Bit-size	1000	1000	200	200	20	20	100	100
Wall-time (s)	248.4	417.0	242.0	300.8	185.4	325.6	-	21443.8

TABLE 3. Combined MPI/OpenMP prototyping (- for memory thrashing)

experiments in this Section were obtained on a single machine equipped with 4×18 E7-8860v4 cores @2.2GHz.

We implemented Algorithm 2 in the LINBOX library as a full-featured routine with the standard of library code and not of a prototype benchmarking code. It required therefore a significant effort to first cleanup, fix, and redesign the existing p-adic lifting code and its API in the library. Then the new implementation could be smoothly inserted in this API and the library user can now smoothly switch from one implementation to the other and enable parallelization.

The dominant costs of Algorithm 2 are within lines 4, 12 and 16, which we will denote by the INV (matrix inversion), MVmod (matrix-vector iteration) and MMZ (matrix-matrix multiplication over $\mathbb{Z}$). Their respective costs are given by $O(\ell\cdot n^\omega)$, $O(k\ell\cdot n^2)$ and $O(k\cdot n^2\ell^{\omega-1})$, where ℓ is the parameter and $k\ell \approx n\cdot\frac{\texttt{bitsize}}{\log_2(p)}$ is a constant for any ℓ.

Consequently, increasing ℓ leads to an increase of the overall number of operations, but it will hopefully trade some slow operations of MMZ for some very fast operations of INV. We also

see that the dominant cost should remain unchanged within MVmod, but increasing ℓ will also increase the potential for parallelism.

This analysis is backed up by experiments as we show in the following. First, the technique of the double RNS basis, combined with the gathering of updates is shown, in sequential, in Figures 2 and 3. We see that by extending the number of primes ℓ, we increase the overall work but this can pay at different thresholds for different matrix dimensions and bit-sizes. In sequential we can see some 30% gains on a single core, even for small matrices.

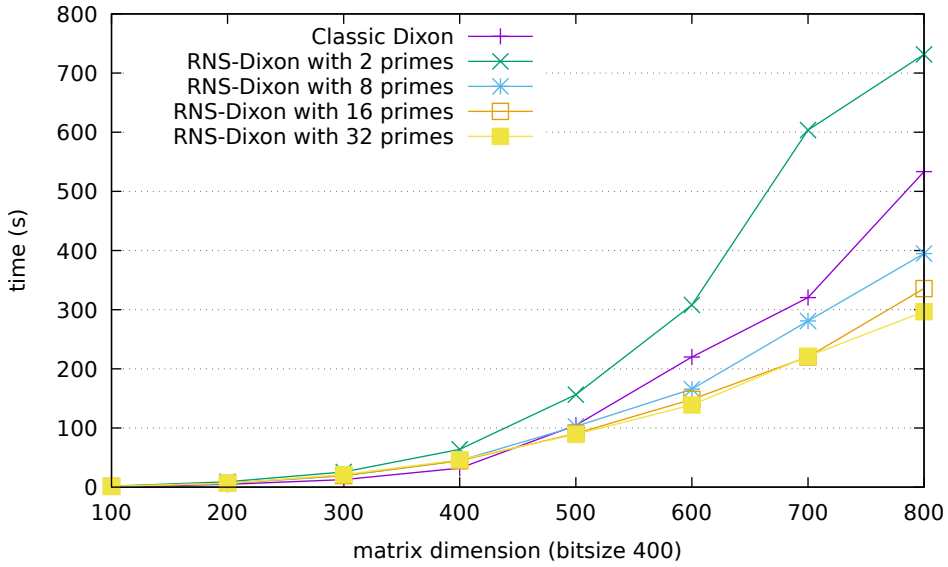

FIGURE 2. Sequential RNS Dixon with fixed bitsize.

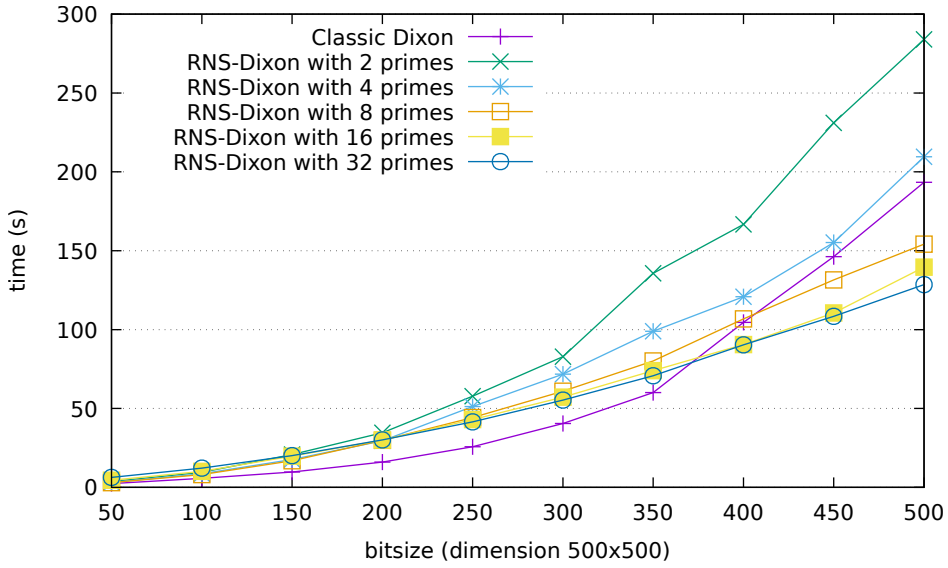

FIGURE 3. Sequential RNS Dixon with fixed matrix dimension.

Now in parallel, we obtain good speed-up, as shown in Figures 4 and 5. Figure 4 illustrates the fact that a rather good scaling can be obtained as long as the number of primes ℓ remain larger than the number of threads.

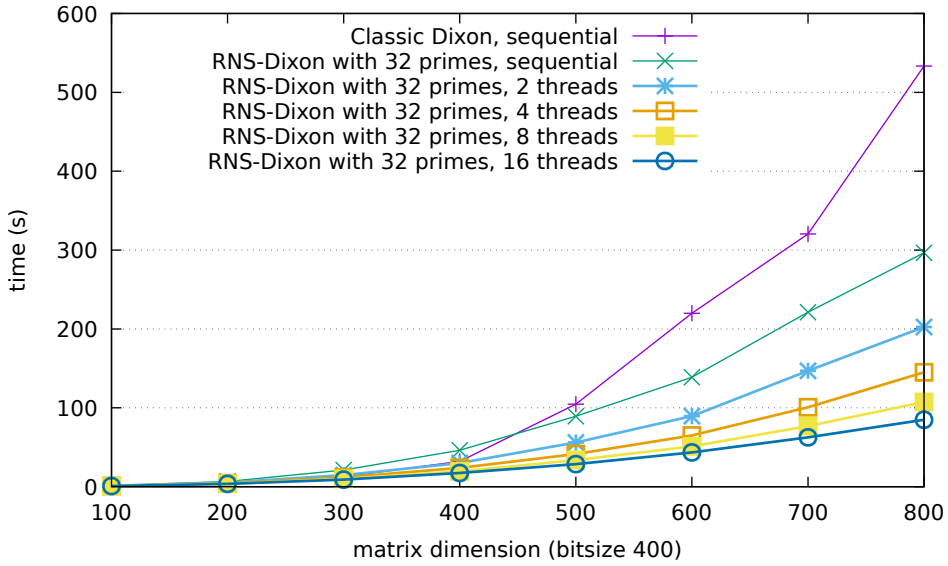

FIGURE 4. Parallel RNS Dixon.

Figure 5 illustrates how the number primes ℓ balances the workload between inversions and lifting. On a small instance (top part), the overall work may increases for large ℓ. This problem disappears for larger instances (bottom part), as the efficiency of the INV operation on large matrices, make it negligible (almost no more "red" in the bottom histogram), and the parallelization pays of (9.6 speed-up for our new Algorithm with 16 cores in the bottom histogram).

Further work is required to tune up at a finer grain the various levels of parallelism used in the large precision RNS matrix-matrix multiplication or to automatically deduce the right choice of parameter ℓ from the characteristics of the matrix and the right hand side.

2.3. **GPU enabled libraries**

Graphic Processing Unit (GPU) have become a widespread component in modern high performance computing platforms. They offer a very high computing throughput for a contained power consumption but only for applications well suited for their constraints:

- the GPU has its own on-board memory and thus data transfer from and to the main memory are expensive and may quickly become a bottleneck
- the type of computation must be very regular (same code kernel applied to multiple sets of data) with as few branching and control instructions as possible.

Matrix multiplication is among these applications which can easily benefit from the power of GPUs. Other computations such as Gaussian elimination, require branching, data permutations, divisions and other operations much less suited for GPUs and therefore require communications back and forth between the main memory and the GPU memory.

Nvidia provides a set of BLAS routines, for dense linear algebra of floating point numbers, supporting their GPUs in the library CUBLAS.

The `fflas-ffpack` library aims at providing a similar set of routines over a finite field by relying exploiting a numerical BLAS under the hood. In this first attempt at supporting GPU's for finite field linear algebra, we therefore focused on interfacing CUBLAS in the `fflas-ffpack` library so that matrix multiplication over a finite field could benefit from CUBLAS' routines.

The user of the library only need to specify at configure time a path where a CUDA library is installed:

```
$> ./configure --with-cuda=/applis/dahu/cuda/cuda-10.0/lib64
```

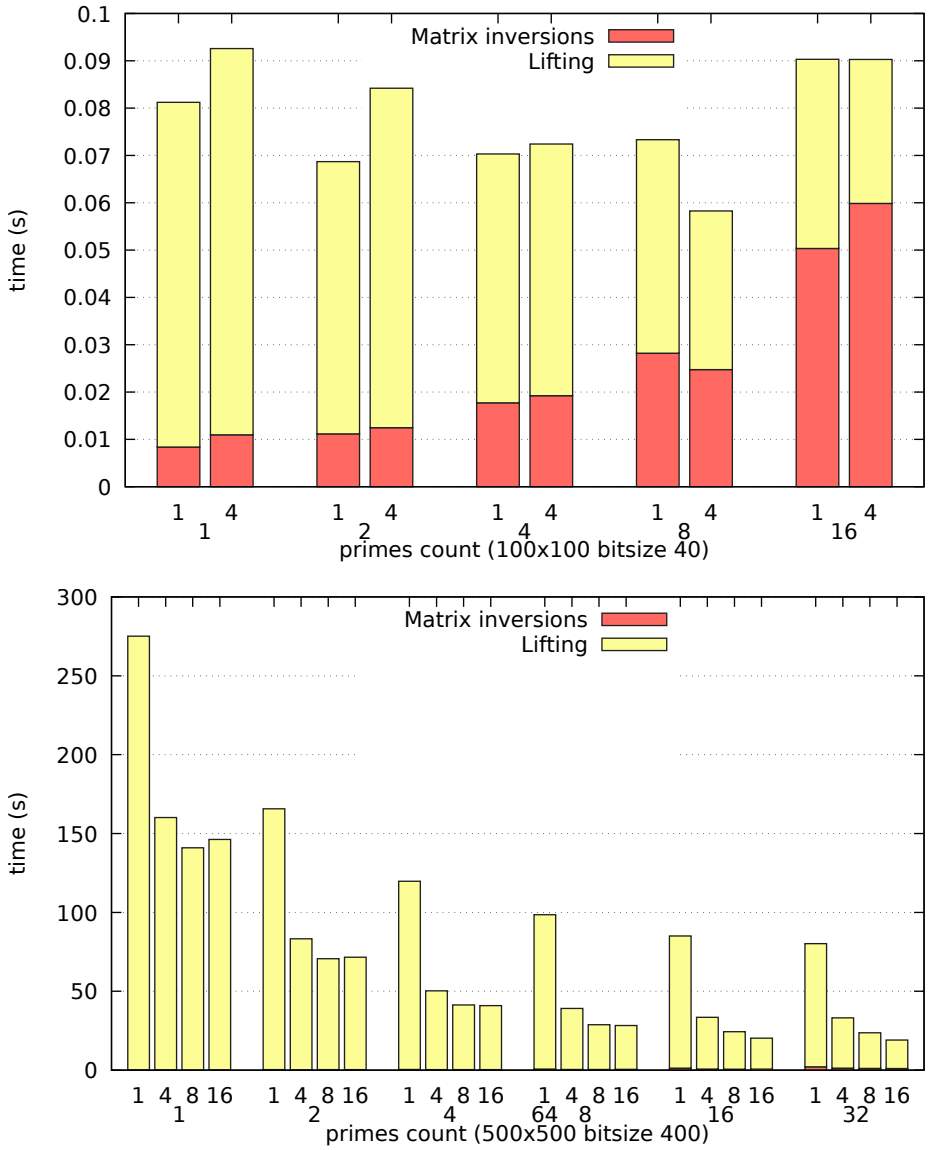

FIGURE 5. Balance between matrix inversion (INV) and lifting phases (MV-mod+MMZ). In each cluster, leftmost column correspond to a sequential run, then to 4, 8 and 16 cores.

He can then transparently run his code and benefit from the speedup of the BLAS library. Figure 6 shows the computation speed achieved on an NVIDIA Tesla V100 GPU, compared to the existing multi-core parallel implementation and the sequential implementation.

The speed-up of one GPU with respect to the sequential implementation is up to a factor 39.5 and up to 2.46 with respect to a parallel execution on 32 cores.

Note that the efficiency of the GPU runs is slowly increasing and therefore perform slower than the multi-core implementation on smaller dimensions. This is due to the communication time spent in transferring the data to and from the GPU. This overhead is quadratic in the matrix dimension and thus becomes negligible compared to as the dimension increases.

This is the reason why other routines relying on many matrix multiplication, such as Gaussian elimination, matrix inverse, etc, cannot blindly benefit from the efficiency of this matrix multiplication routine. Further work will consist in implementing dedicated finite field Gaussian elimination routines on the GPU, and/or exploring the various techniques to schedule the memory transfer so that they overlap with computation.

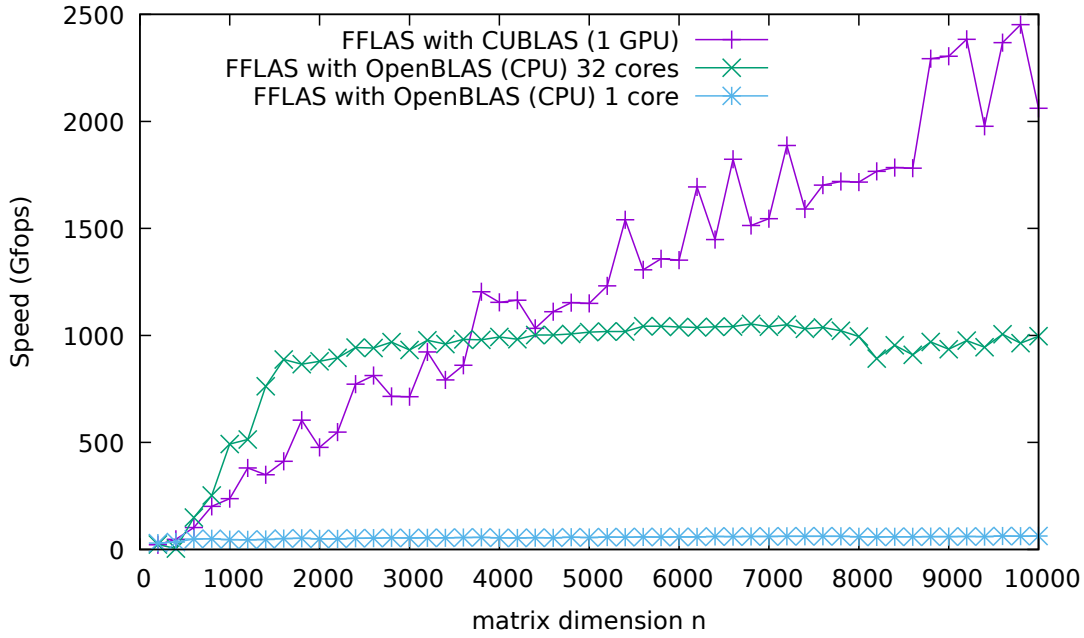

FIGURE 6. Computation speed of FFLAS-FFPACK matrix product over $\mathbb{Z}/131071\mathbb{Z}$ on a server with 1 NVidia Tesla V100 GPU, and 32 Intel Xeon Gold 6130 cores.

REFERENCES

Stavros Birmpilis, George Labahn, and Arne Storjohann. Deterministic Reduction of Integer Nonsingular Linear System Solving to Matrix Multiplication. In *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation*, ISSAC '19, pages 58–65, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6084-5. doi:10.1145/3326229.3326263.

Zhuliang Chen and Arne Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, volume 2005, pages 92–99, January 2005. doi:10.1145/1073884.1073899.

John D. Dixon. Exact solution of linear equations usingP-adic expansions. *Numerische Mathematik*, 40(1):137–141, February 1982. ISSN 0945-3245. doi:10.1007/BF01459082.

Javad Doliskani, Pascal Giorgi, Romain Lebreton, and Éric Schost. Simultaneous conversions with the Residue Number System using linear algebra. *ACM Transactions on Mathematical Software*, 44(3):#27, February 2018. doi:10.1145/3145573. URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-01415472.

Arne Storjohann. The shifted number system for fast linear algebra on integer matrices. *Journal of Complexity*, 21(4):609–650, August 2005. ISSN 0885-064X. doi:10.1016/j.jco.2005.04.002.