A parallel algorithm for Gaussian elimination over finite fields

Stephen Linton, Gabriele Nebe, Alice Niemeyer, Richard Parker, Jon Thackray

June 7, 2018

School of Computer Science, University of St. Andrews, St. Andrews, Fife KY169SX, Scotland *E-mail address*: steve.linton@st-andrews.ac.uk

Lehrstuhl D für Mathematik, RWTH Aachen University, 52056 Aachen, Germany *E-mail address*: nebe@math.rwth-aachen.de

Lehrstuhl B für Mathematik, RWTH Aachen University, 52056 Aachen, Germany *E-mail address*: alice.niemeyer@mathb.rwth-aachen.de

70 York St. Cambridge CB1 2PY, UK
E-mail address: richpark54@hotmail.co.uk

Abstract

In this paper we describe a parallel Gaussian elimination algorithm for matrices with entries in a finite field. Unlike previous approaches, our algorithm subdivides a very large input matrix into smaller submatrices by subdividing both rows and columns into roughly square blocks sized so that computing with individual blocks on individual processors provides adequate concurrency. The algorithm also returns the transformation matrix, which encodes the row operations used. We go to some lengths to avoid storing any unnecessary data as we keep track of the row operations, such as block columns of the transformation matrix known to be zero.

The algorithm is accompanied by a concurrency analysis which shows that the improvement in concurrency is of the same order of magnitude as the number of blocks. An implementation of the algorithm has been tested on matrices as large as 1000000×1000000 over small finite fields.

Keywords: Gaussian elimination, parallel algorithm, finite fields
MSC 2010:15A06, 68W10, 68W05

1 Introduction

Already employed for solving equations by hand in China over 2000 years ago [9], the Gaussian elimination method has become an invaluable tool in many areas of science.

When computing machines became available, this algorithm was one of the first to be implemented on a computer. In general the algorithm takes as input a matrix with entries in a field (or a division ring) and transforms this matrix to a matrix in row echelon form. It can be employed for several different purposes, and computer implementations can be tailored to suit the intended application. Different variants of the Gaussian elimination algorithm can be envisaged, for example computing the rank of a matrix, computing a row echelon form or a reduced row echelon form of a matrix, or computing one of these echelon forms together with the transformation matrix. Often one of these versions of the Gaussian elimination algorithm lies at the heart of other algorithms for solving problems in a broad range of areas and their overall performance is often dictated by the performance of the underlying Gaussian elimination algorithm. Thus an implementation of a Gaussian elimination algorithm is required to display exceptional performance.

Since their invention, computers have become faster and more powerful every year. Yet, for over a decade this increase in computing power is no longer primarily due to faster CPUs but rather to the number of different processors an individual computer has, paired with the increasingly sophisticated memory hierarchy. It is therefore paramount that modern algorithms are tailored to modern computers. In particular this means that they need to be able to perform computations in parallel and store the data for the current computations readily in cache.

With the advance of parallel computers comes the need to design a parallel algorithm to perform Gaussian elimination on a matrix. Such a parallel algorithm would immediately result in immense speedups of higher level algorithms calling the Gaussian elimination without having to introduce parallelism to these algorithms themselves. When designing a parallel Gaussian elimination algorithm it is important to keep the applications of the algorithms in mind. Several versions of a parallel Gaussian elimination algorithm have been described when working over the field of real or complex numbers, see [6] for a survey and PLASMA [11] for implementations. In this paper we describe a parallel version of the Gaussian elimination algorithm which, given a matrix with entries in a finite field, computes a reduced row echelon form together with the transformation matrix. We note that when working with dense matrices over finite fields we are not concerned with sparsity, the selection of suitable pivot elements nor numerical accuracy. In particular, we can always choose the first non-zero element of a given row as our pivot element, ensuring that all entries to the left of a pivot are known to be zero. Moreover, we need not be concerned with producing elements in a field which require more and more memory to store them. Avoiding producing very large field elements would again complicate pivot selection. Thus our main concern is to design a parallel algorithm which makes optimal use of modern parallel computers.

We assume an underlying shared memory computational model in which we have access to k different processors, each of which can run a job independently from any other. The processors communicate with each other through the shared memory system. Our design must take account of the limited total memory bandwidth of the system. The aim of our parallel algorithm is to achieve adequate concurrency by dividing the necessary computational work into smaller jobs and scheduling these to run simultaneously on the

k processors. This in turn calls for a very careful organization of the jobs so that different jobs do not interfere with each other. We will address these issues in Section 2.1.

It is well known, see for example [5, Theorems 28.7, 28.8], that the asymptotic complexities of matrix inversion and matrix multiplication are equal. An algorithm that shows this reduces inversion of a $2n \times 2n$ matrix to 6 multiplications of $n \times n$ matrices together with two inversions, also of $n \times n$ matrices. Applying this approach recursively, almost all of the run-time of inversion is spent in multiplications of various sizes. It is not difficult to see that this extends to our somewhat more general computation of a reduced row echelon form, with transformation matrix.

We envisage that we are given a very large matrix over a finite field for which we need to compute a reduced row echelon form together with a transformation matrix.

Several approaches to achieving this already exist for finite fields. One approach takes advantage of the reduction to multiplication mentioned above, and delegates the problem of parallelising the computation primarily to the much easier problem of parallelising the larger multiplications. Another approach represents finite field elements as floating point real numbers of various sizes in such a way that (with care) the exact result over the finite field can still be recovered. The problem can then be delegated to any of a number of highly efficient parallel floating point linear algebra systems [7]. Another approach by Albrecht *et al.* works over small finite fields of characteristic 2 (see [2] and [3]). A fourth approach in unpublished work by Lübeck repeatedly divides the matrix horizontally, echelonising each block of rows in parallel and then sorting the rows of the matrix, so that rows with similar length initial sequences of zeros come together.

Large modern computers typically have a large number of cores but may well have an order of magnitude less real memory bandwidth per core than a typical laptop or desktop computer. On such a large modern computer, at the lowest level, a core can only be fully occupied if essentially all its data is in the smallest, fastest level of cache memory (L1). At the next level out, this work can only be started if essentially all its data is in the next level of cache (L2). A similar statement is true for L3 cache. To use a modern computer effectively for matrix operations, it is therefore necessary to repeatedly subdivide matrices in **both** directions producing matrices that are roughly square at a scale commensurate with the size of the cache at each level.

It is not too hard to design an algorithm for matrix multiplication with these properties. An approach along these lines to Gaussian elimination for these roughly square submatrices is described in Section 2.3.2.

For the whole matrix, we also need to subdivide to achieve concurrency.

Unlike previous approaches, we subdivide our very large input matrix into smaller submatrices, called *blocks*, by subdividing both rows and columns into roughly square blocks sized so that computing with individual blocks on individual processors provides adequate concurrency. We will show that we gain a concurrency improvement in the same order of magnitude as the number of blocks, see Proposition 5.3 and Theorem 5.4.

As well as computing the reduced row echelon form of the input matrix, we compute the *transformation matrix*, which encodes the row operations used. We go to some lengths to avoid storing any unnecessary data as we keep track of the row operations, such

as block columns of the transformation matrix known to be zero. Our experiments show that the memory usage during the execution of the algorithm remains fairly stable, and is similar to storing the input matrix. The runtime is broadly comparable to multiplication of matrices to the same size as the input matrix. This gives evidence that we have succeeded in keeping the cost of computing the transformation matrix as small as possible and is in accordance with the theoretical analysis of Bürgisser et al. in particular [4, Theorem 16.12].

The parallel Gauss algorithm is designed with three distinct environments in view, although we have only implemented the first so far.

The first (and original) target is to use a single machine with multiple cores with the matrix in shared memory. Here the objective is to subdivide the overall task into many subtasks that can run concurrently.

The second target is to use a single machine where the matrix does not fit in memory, but does fit on disk. Here the objective is to subdivide the matrix so that each piece fits into memory.

The third target is where several computers (probably each with multiple cores) are to work on a single problem simultaneously. Here the objective is again to subdivide the overall task into many subtasks that can run concurrently on different computers with access to the same central disk.

2 Preliminaries

2.1 Computational Model

Our general approach to parallel computing is to decompose the work to be done into relatively small units with explicit dependencies. Units all of whose dependencies are met are called "runnable" and a fixed pool of worker threads carry out the runnable units, thereby discharging the dependencies of other units and making them in turn runnable. A module, called the *scheduler*, is charged with keeping track of the dependencies and scheduling the execution of the individual units. Data describing the units of work and their input and output data reside in a shared data store, but we take considerable care to ensure that the speed of this store is not critical to overall performance. It can thus be the large amount of shared DRAM on a multicore server, or disk storage local to a single server (for the case where we only have one server, but data too large for its RAM) or shared disk provided sufficiently strong consistency can be guaranteed.

In our implementations, we have used two variations of this model. In one, the *task-model*, the units of work are *tasks* and are relatively coarse-grained. A task represents, roughly speaking, all the work that needs to be done in a particular part of the matrix at a particular stage of the algorithm. Dependencies are between tasks, so one task cannot execute until certain others have completed and it will find the data it needs where those previous tasks have stored it. To simplify understanding, we collect different data into *data packages*, the input and output of the tasks. For example a typical output of

the task ClearDown is the data package $\mathbf{A} = (A, M, K, \rho', E, \lambda)$ with six components which we refer to as $\mathbf{A}$.A, $\mathbf{A}$.M, etc.

In the other model, the *job-model*, the units of work are the *jobs* and are significantly more fine-grained and represent a single elementary computation such as a submatrix multiplication. More importantly, the dependencies are between the jobs and the data they produce and consume. Each job requires zero or more input data objects and produces one or more output objects. A job is runnable when all of its input data has been produced. This finer grain approach allows more concurrency. Further gain in efficiency can be achieved by giving the scheduler guidance as to which jobs or tasks are urgent and which are less so. This aspect is mainly ignored in this paper. The implementation of Meataxe64 [10] uses the job-model and identifies the urgent jobs. An implementation in HPC-GAP by Jendrik Brachter and Sergio Siccha is based on the task-model.

The parallel Gaussian elimination algorithm is described in Section 3 as a program called the Chief charged with defining the tasks and the data packages they work on. The Chief is described as a sequential program but the reader should be warned that the tasks are executed in an unpredictable order which may bear little resemblance to the order in which the tasks are submitted by the Chief. The result of running the Chief is a plan consisting of a list of tasks, respectively jobs, together with their inputs and outputs whose collective execution performs the Gaussian elimination.

Below we specify The Chief in terms of tasks, specified in turn by jobs, for which the reader will find a more or less specific description in Section 4.

2.2 Gaussian elimination

This subsection describes the Gaussian elimination process in a way we hope is familiar to the reader, but in our notation. Given a matrix H with entries in a field $\mathbb{F}$ the output of the Gauss algorithm consists of matrices M, K and R and permutation matrices P_ρ and P_γ such that

$$\begin{pmatrix} M & 0 \\ K & 1 \end{pmatrix} P_\rho H P_\gamma = \begin{pmatrix} -1 & R \\ 0 & 0 \end{pmatrix}. \tag{1}$$

The matrices P_ρ and P_γ perform row, respectively column, permutations on the input matrix H such that the top left-hand part of the resulting matrix $P_\rho H P_\gamma$ is invertible (with inverse $-M$) with the same rank as H. It should be noticed that the permutation matrices P_ρ and P_γ in Equation (1) are not uniquely defined. All that matters is that they put the pivotal rows and columns into the top left-hand corner of the matrix. We therefore choose to only specify the sets of row and column numbers containing pivotal elements. As we are chopping our matrix into blocks, we use the word *selected* to specify a row or column in which a pivot has been already been found. Hence we have to apply a permutation to the rows during the course of the algorithm to ensure that our pivots remain located in columns with increasing indices. We formalize how we store these permutation matrices in the Definition 2.1.

Definition 2.1. *When we enumerate elements of a set, we always implicitly assume that these are in order. To a subset* $\rho = \{\rho_1, \ldots, \rho_{|\rho|}\} \subseteq \{1, \ldots, \alpha\}$ *associate a* $0/1$ *matrix*

$$\rho \in \mathbb{F}^{|\rho| \times \alpha} \text{ with } \rho_{i,j} = \begin{cases} 1 & j = \rho_i \\ 0 & \text{else.} \end{cases}$$

We call ρ *the* row-select *matrix and* $\overline{\rho}$ *the* row-nonselect *matrix associated to the set* ρ *and its complement* $\overline{\rho} = \{1, \ldots, \alpha\} \setminus \rho$.

Remark 2.2. *Note that the matrix*

$$P_\rho = \begin{pmatrix} \rho \\ \overline{\rho} \end{pmatrix} \in \mathbb{F}^{\alpha \times \alpha}$$

is a permutation matrix associated to the permutation p_ρ *with* $p_\rho(i) = \rho_i$ *for* $i \leq |\rho|$ *and* $p_\rho(i) = \overline{\rho}_{i-|\rho|}$ *for* $i > |\rho|$. *Note that* $p_\rho = 1$ *if and only if* $\rho = \{1, \ldots, i\}$ *for some* $0 \leq i \leq \alpha$ *but for the other subsets* ρ *we may recover* ρ *from* p_ρ *as the image* $\{p_\rho(1), \ldots, p_\rho(i)\}$ *if* $p_\rho(i+1) < p_\rho(i)$. *In this sense we keep switching between permutations, subsets, and bitstrings in* $\{0,1\}^\alpha$ *which represent the characteristic function of the subset. In particular, this means that a subset of cardinality* $2^\alpha - \alpha$ *of* special permutations *of all the* $\alpha!$ *permutations of* $\{1, \ldots, \alpha\}$ *is sufficient for all our purposes.*

We also note that for a matrix $H \in \mathbb{F}^{\alpha \times \beta}$ and $\rho \subseteq \{1, \ldots, \alpha\}$ and $\gamma \subseteq \{1, \ldots, \beta\}$ the matrix $\rho \times H \in \mathbb{F}^{|\rho| \times \beta}$ consists of those rows of H whose indices lie in ρ (retaining the ordering) and the matrix $H \times \gamma^{tr} \in \mathbb{F}^{\alpha \times |\gamma|}$ consists of those columns of H whose indices lie in γ. Therefore we also call γ^{tr} the *column-select* matrix and $\overline{\gamma}^{tr}$ the *column-nonselect* matrix associated to γ.

Remark 2.3. *Let* $H \in \mathbb{F}^{\alpha \times \beta}$ *be of rank* r. *Then the echelonisation algorithm will produce sets* $\rho \subseteq \{1, \ldots, \alpha\}$ *and* $\gamma \subseteq \{1, \ldots, \beta\}$ *of cardinality* $|\gamma| = |\rho| = r$ *and matrices* $M \in \mathbb{F}^{r \times r}$, $K \in \mathbb{F}^{(\alpha - r) \times r}$, $R \in \mathbb{F}^{r \times (\beta - r)}$ *such that*

$$\begin{pmatrix} M & 0 \\ K & 1 \end{pmatrix} \begin{pmatrix} \rho \\ \overline{\rho} \end{pmatrix} H \begin{pmatrix} \gamma^{tr} & \overline{\gamma}^{tr} \end{pmatrix} = \begin{pmatrix} -1 & R \\ 0 & 0 \end{pmatrix}$$

We will refer to these matrices as

$$(M, K, R, \rho, \gamma) := \mathrm{ECH}(H).$$

Strictly speaking, the job ECH computes the negative row reduced echelon form of the input matrix H. However we will simply call this the *echelon form of* H.

We assume we have an implementation of ECH suitable for use on a single core. The goal of this paper is to show how to scale it up.

Example

We now present an example to highlight some of the structure of the algorithm. Consider the matrix $C \in \mathbb{F}_3^{6\times 6}$ given by

$$\left(\begin{array}{ccc|ccc} 0 & 2 & 2 & 0 & 1 & 0 \\ 0 & 2 & 2 & 1 & 2 & 2 \\ 1 & 0 & 1 & 0 & 2 & 1 \\ \hline 2 & 0 & 1 & 0 & 2 & 2 \\ 0 & 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 2 & 2 & 0 & 0 \end{array}\right)$$

and divided into four blocks as shown. Our first step is to echelonise the top left block. This yields

$$\left(\begin{array}{cc|c} 0 & 2 & 0 \\ 1 & 0 & 0 \\ \hline 2 & 0 & 1 \end{array}\right) P_{\{1,3\}} \left(\begin{array}{ccc} 0 & 2 & 2 \\ 0 & 2 & 2 \\ 1 & 0 & 1 \end{array}\right) P_{\{1,2\}} = \left(\begin{array}{cc|c} 2 & 0 & 2 \\ 0 & 2 & 2 \\ \hline 0 & 0 & 0 \end{array}\right)$$

matching Equation 1. Here

$$P_{\{1,3\}} = \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 1 & 0 \end{array}\right) \text{ and } P_{\{1,2\}} = \left(\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right),$$

where the bars separate selected from non-selected rows or columns. So the outputs from this step are the multiplier $M = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}$, $K = \begin{pmatrix} 2 & 0 \end{pmatrix}$, and $R = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$ as well as the row-select set ρ, selecting the first and third row and the column-select set γ, selecting the first two columns.

After this, two further steps are available, our parallel implementation will do both concurrently. We mimic the row transformations, applied to the top left block, on the top right block. We also use the echelonised top left block to clean out some columns in the block beneath it. We explain them in that order.

Mathematically, we need to left multiply the top right block by $\left(\begin{array}{cc|c} 0 & 2 & 0 \\ 1 & 0 & 0 \\ \hline 2 & 0 & 1 \end{array}\right) P_{\{1,3\}}$. We take advantage of the known structure of this matrix to speed up this computation as follows. We divide the top right block into the selected rows, as described by ρ: $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$ and the rest $\begin{pmatrix} 1 & 2 & 2 \end{pmatrix}$. We add the product of K and the selected rows to the non-selected rows, giving $\begin{pmatrix} 1 & 1 & 2 \end{pmatrix}$, and then multiply the selected rows by M giving $\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 0 \end{pmatrix}$.

The other step requires us to add multiples of the pivot rows from the echelonised top left block $\left(\begin{array}{cc|c} 2 & 0 & 2 \\ 0 & 2 & 2 \end{array}\right)$ to the bottom left block, so as to clear the pivotal columns. We

again take advantage of the known structure of this matrix to speed up this computation as follows. We divide the bottom left block into the selected columns, as described by γ: $\begin{pmatrix} 2 & 0 \\ 0 & 1 \\ 1 & 2 \end{pmatrix}$ and the non-selected columns $\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$. Now we add the product of the selected columns and R to these non-selected columns and obtain $\begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$.

We must now mimic the row transformations used to clear the pivotal columns of the bottom left block by adding multiples of rows from the top right hand block to the bottom right hand block, i.e. we add the product of $\begin{pmatrix} 2 & 0 \\ 0 & 1 \\ 1 & 2 \end{pmatrix}$ and $\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 0 \end{pmatrix}$ to the bottom right hand block, which becomes $\begin{pmatrix} 0 & 1 & 0 \\ 2 & 2 & 1 \\ 2 & 0 & 2 \end{pmatrix}$.

After all these steps, the overall matrix is

$$\left(\begin{array}{ccc|ccc} 2 & 0 & 2 & 0 & 1 & 2 \\ 0 & 2 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 2 \\ \hline 0 & 0 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 2 & 2 & 0 & 2 \end{array}\right).$$

At this stage we have dealt with all the consequences of the pivots found in the top left block. What remains to be done is to echelonise the top-right and bottom-left submatrices in the picture above and deal with the consequences of any pivots found. Finally part of the bottom right hand block will need to be echelonised. This example, chopped 2×2 does not demonstrate pivotal row merging as described in Section 2.3.1

2.3 Some guiding points

2.3.1 Subdividing a huge matrix

Our parallel version of the Gaussian elimination algorithm takes as input a huge matrix and subdivides it, both horizontally and vertically, into blocks. We do this partly to obtain concurrency and partly to reduce the row length for cache reasons. Once the top-left block has been echelonised, the same row operations can be applied simultaneously to all the blocks along the top row. Putting the rest of the left-most block column into echelon form requires addressing the blocks sequentially from the top down. However, in the common case where the co-rank of the top-left block is small, it is not a great deal of computation. Once the top block row and left-most block column are in echelon form, we can update each of the blocks in the other rows and columns concurrently.

It should be remarked that nothing else can happen until this first block is echelonised, suggesting that we should make this block smaller than the rest. A similar comment applies to the last (bottom-right) block.

Proceeding down the left-most block column sequentially enables us to merge into a single row of blocks all those rows whose pivots lie there. This merging is done to reduce the amount of data access. Usually, the work of doing this merging is not great, so that soon after the first block echelonisation is complete, a large block multiply-and-add can be done to every block of the matrix. Without the merging, this multiply-and-add would be done piecemeal, requiring multiple passes through the block.

2.3.2 Echelonisation of a single block

The performance of the echelonisation of a single block (as defined in Remark 2.3) can have a considerable impact on the concurrency, as many later jobs may depend on each echelonisation.

The sequential algorithm used to echelonise individual blocks is recursive, combining elements of the recursive inversion algorithm already mentioned, with the greater generality of the technique described in this paper. A block is divided into two, either horizontally or vertically, and the top (resp. left) part is echelonised. Using a simplified version of ClearDown (resp. UpdateRow), see Sections 4.2.1 and 4.2.2, the remainder of the matrix and the transformation matrix are updated, producing a second block which must also be echelonised. The results of the two echelonisations can be combined to compute the data package consisting of M, K, R, P_ρ and P_γ. Using this technique recursively for all matrices bigger than a threshold size (about 300 dimensions) and a simple direct Gaussian elimination algorithm below this size, echelonisation of a block takes essentially the same time as a block multiply.

2.3.3 Blocks change size and shape

In our description of the algorithm, especially in Equations (3) and (4), we imagine that rows are permuted so that those containing pivots (in blocks to the left) are moved to the top, and the rest are moved to the bottom. In the program, however, these two sets of rows are held in different matrices, but it seemed better to try to include all the information in one matrix, attempting to clarify the relationships between the parts.

As a consequence of moving rows about, the sizes and shapes of the blocks change during the course of the algorithm. We use a superscript to indicate the "stage" of a particular block, so that, for example, $\mathbf{C}_{ik}^{j}$ is the matrix block at the (i,k)-position in its j'th stage. In some ways the original input matrix $\mathcal{C}$ (see Equation (2) below) is gradually converted into the matrix $\mathcal{R}$. An intermediate matrix $\mathbf{B}$ collects the pivotal rows from $\mathcal{C}$ which are subsequently deleted from $\mathcal{C}$.

2.3.4 Riffles

Our subset-stored permutations are used both to pull matrices apart and to merge them together. We call the pulling apart an 'extract' where one matrix is separated into two matrices with the selected rows (or columns) going to one, and the remaining rows to the other. We call the merging a 'riffle' where one matrix is assembled from two inputs with the subset-stored permutation directing from which input each row (or column) comes.

2.3.5 Transformation matrix abbreviation

To compute the transformation matrix, we could apply the same row operations we performed on the input matrix to the identity matrix. In practice this would be wasteful, both of memory and computational effort, since initially the identity matrix contains no information at all, and the early row operations would mainly be adding zeros to other zeros. Considerable effort has been expended in this paper to avoid storing any parts of matrices whose values are known *a priori*, thereby saving both memory, and work manipulating them. The graphs shown in Section 6 suggest that this has been successful. The details of this optimisation are the subject of Section 4.2.3, which may be skipped on first reading.

Note that although the output matrix $\mathcal{M}$ in Equation 2 below is square, the number of blocks in each row may differ from the number of blocks in each column, since the block rows are indexed by the block column in which the pivot was found and *vice versa*.

3 A parallel Gauss algorithm

3.1 The structure of the algorithm

We now describe a parallel version of the Gaussian elimination algorithm which takes as input a huge matrix and subdivides it, both horizontally and vertically, into blocks.

To distinguish between the huge matrix and its blocks we use different fonts.

Let $\mathbb{F}$ be a finite field and $\mathcal{C} \in \mathbb{F}^{m\times n}$ a huge matrix of rank r. We describe a parallel version of the well-known Gauss algorithm, which computes matrices $\mathcal{R}$, a transformation matrix $\mathcal{T} = \begin{pmatrix} \mathcal{M} & 0_{r\times(m-r)} \\ \mathcal{K} & 1_{(m-r)\times(m-r)} \end{pmatrix}$ and subsets $\varrho \subseteq \{1,\dots m\}$ and $\Upsilon \subseteq \{1,\dots,n\}$, both of cardinality r, such that

$$\begin{pmatrix} \mathcal{M} & 0_{r\times(m-r)} \\ \mathcal{K} & 1_{(m-r)\times(m-r)} \end{pmatrix} \begin{pmatrix} \varrho \\ \overline{\varrho} \end{pmatrix} \mathcal{C} \begin{pmatrix} \Upsilon^{tr} & \overline{\Upsilon}^{tr} \end{pmatrix} = \begin{pmatrix} -1_{r\times r} & \mathcal{R} \\ 0 & 0 \end{pmatrix} \tag{2}$$

is in (negative row reduced) echelon form.

Comparing to Remark 2.3, we see that our task is to chop our huge input matrix into smaller blocks and then, using ECH and other jobs on the blocks, to effect the same operation on the huge matrix as ECH does on a single block. We therefore choose

positive integers a_i, b_j such that

$$\sum_{i=1}^{a} a_i = m, \ \sum_{j=1}^{b} b_j = n$$

and our algorithm copies the block-submatrices of the input matrix $\mathcal{C}$ into data packages $\mathbf{C}_{ij} \in \mathbb{F}^{a_i \times b_j}$, called *blocks*, and performs tasks (as described in Section 4) on these smaller blocks.

We call the a matrices $\mathbf{C}_{1j}, \ldots, \mathbf{C}_{aj}$ the j-th *block column* and the b matrices $\mathbf{C}_{i1}, \ldots, \mathbf{C}_{ib}$ the i-th *block row* of $\mathcal{C}$. Echelonising the overall matrix $\mathcal{C}$ is achieved by performing an echelonisation algorithm on individual blocks and using the resulting data to modify others.

The result of the Gaussian elimination as well as the intermediate matrices are partitioned into blocks: When the Gauss algorithm has completed, the matrix $\mathcal{R}$, which occurs in Equation (2), consists of blocks and has the form

$$\mathcal{R} = \begin{pmatrix} R_1 & R'_{12} & \ldots & \ldots & R'_{1b} \\ 0 & R_2 & R'_{23} & \ldots & R'_{2b} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & R_{b-1} & R'_{b-1,b} \\ 0 & \ldots & \ldots & 0 & R_b \end{pmatrix} \tag{3}$$

with $R_j \in \mathbb{F}^{r_j \times (b_j - r_j)}$ and $R'_{jk} \in \mathbb{F}^{r_j \times (b_k - r_k)}$. Here $r = \sum_{j=1}^{b} r_j$ is the rank of $\mathcal{C}$ and for $k = 1, \ldots, b$ the sum $\sum_{j=1}^{k} r_j$ is the rank of the submatrix of $\mathcal{C}$ consisting of the first k block columns.

The time-consuming parts of the Gaussian elimination algorithm consist of Step 1 and Step 3, whereas the intermediate Step 2 is not. After the first step the matrix $\mathcal{C}$ has been transformed into an upper triangular matrix and prior to permuting columns the matrix $(-1_{r\times r}|\mathcal{R}) \in \mathbb{F}^{r \times n}$ has the shape

$$\tilde{\mathcal{R}} = \left(\begin{array}{cccccc} -1 \mid R_1 & X_{12} \mid R_{12} & \ldots & \ldots & X_{1b} \mid R_{1b} \\ 0 \mid 0 & -1 \mid R_2 & X_{23} \mid R_{23} & \ldots & X_{2b} \mid R_{2b} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 \mid 0 & \ldots & 0 \mid 0 & -1 \mid R_{b-1} & X_{b-1,b} \mid R_{b-1,b} \\ 0 \mid 0 & \ldots & \ldots & 0 \mid 0 & -1 \mid R_b \end{array}\right), \tag{4}$$

where $R_{jk} \in \mathbb{F}^{r_j \times (b_k - r_k)}$ and $X_{jk} \in \mathbb{F}^{r_j \times r_k}$. To simplify notation in the algorithms below we define the data packages $\mathbf{B}_{jk} = (X_{jk}|R_{jk}) \in \mathbb{F}^{r_j \times b_k}$ for $1 \leq j < k \leq b$ and $\mathbf{D}_j = (\mathbf{D}_j.\mathrm{R}, \mathbf{D}_j.\gamma)$, where $\mathbf{D}_j.\mathrm{R} = R_j$ and $\mathbf{D}_j.\gamma \subseteq \{1, \ldots, b_j\}$ is the set of indices of pivotal columns in the block column j.

3.1.1 Computing the transformation matrix

During the course of the algorithm we also compute the transformation matrix $\mathcal{T} \in \mathrm{GL}_m(\mathbb{F})$ as given in Equation (2). Of course this could be achieved by simply performing

the same row operations on an identity matrix that were performed on $\mathcal{C}$. This involves considerable work on blocks known to be either zero or identity matrices. For example, if $\mathcal{C}$ is invertible, the work required to compute its inverse is needlessly increased by 50%. To avoid such extra work, we only store the relevant parts of the blocks of the transformation matrix.

During the computation, the data packages $\mathbf{M}_{ji}$ $(j = 1, \ldots, b; i = 1, \ldots, a)$ and $\mathbf{K}_{ih}$ $(i, h = 1, \ldots, a)$ record the matching status of the transformation matrix. If $\tilde{C}_{ik}$ denotes the i,k-block of the original input matrix $\mathcal{C}$, then initially $\mathbf{C}_{ik} = \tilde{C}_{ik}$ and $\mathbf{B}_{jk}$ has no rows. Likewise, initially $\mathbf{K}_{ih}$ is the identity matrix if $i = h$ and the zero matrix otherwise and the matrix $\mathbf{M}_{ji}$ has no rows and no columns. When storing the data packages $\mathbf{K}$ and $\mathbf{M}$, we omit the columns and rows in the blocks that are zero or still unchanged from the identity matrix.

To obtain the row select matrix we maintain further data packages $\mathbf{E}_{ij} = (\mathbf{E}_{ij}.\rho, \mathbf{E}_{ij}.\delta)$ $(1 \leq i \leq a, 1 \leq j \leq b)$ with $\mathbf{E}_{ij}.\rho \subseteq \{1, \ldots, a_i\}$ is the set of indices of pivotal rows in block row i with pivot in some block column $1, \ldots, j$ and $\mathbf{E}_{ij}.\delta \in \{0,1\}^{|\mathbf{E}_{ij}.\rho|}$ records which indices already occurred up to block column $j-1$.

During the algorithm, having handled block row i in Step 1, we have

$$\sum_{h=1}^{a} \mathbf{M}_{jh} \times \mathbf{E}_{hi}.\rho \times \tilde{C}_{hk} = \mathbf{B}_{jk}$$

and

$$\sum_{h=1}^{a} \mathbf{K}_{\ell h} \times \overline{\mathbf{E}_{hi}.\rho} \times \tilde{C}_{hk} = \mathbf{C}_{\ell k}.$$

The final column select matrix is the block diagonal matrix

$$\Upsilon = \mathrm{diag}(\mathbf{D}_1^a.\gamma, \ldots, \mathbf{D}_b^a.\gamma).$$

and the row select matrix is

$$\varrho = \mathrm{diag}(\mathbf{E}_{1b}.\rho, \ldots, \mathbf{E}_{ab}.\rho).$$

3.2 Step 1

Step 1 loops over the block rows. For the j-th column of the i-th block row, the algorithm calls Task ClearDown with the two data packages $\mathbf{C}_{ij}$ and $\mathbf{D}_j^{i-1}$ as input. Task ClearDown amalgamates the pivots in $\mathbf{D}_j^{i-1}.\gamma$ with the pivots in the matrix $\mathbf{C}_{ij}$ to produce the enlarged set $\mathbf{D}_j^i.\gamma$ as well as a new matrix $\mathbf{D}_j^i.\mathrm{R}$ of the (negative) echelon form $(-1 \mid \mathbf{D}_j^i.\mathrm{R})$ followed by 0 rows (up to column permutations which are remembered in $\mathbf{D}_j^i.\gamma$). Moreover, the task ClearDown records in its output data package $\mathbf{A}_{ij}$ the row operations performed. With the help of these data packages, the first step then propagates the same elementary row operations to the remaining blocks in block row i as well as to block row i of the transition matrix using Task UpdateRow.

Hence Step 1 assembles in the block row $\begin{pmatrix} 0 & \dots & 0 & -1 \end{pmatrix} \mathbf{D}_j.\mathrm{R} \quad \mathbf{B}_{j,j+1} \quad \dots \quad \mathbf{B}_{jb}\big)$ the rows of the original input matrix whose pivotal entries lie in block column j for $j \leq b$. These rows are then deleted from the data package $\mathbf{C}$. Thus having treated the j-th block column the matrix $\mathbf{C}$ contains no rows whose pivots lie in block columns $1, \dots, j$ and the number of rows of $\mathbf{C}$ is $m - \sum_{k=1}^{j} r_k$.

In particular, during the course of the entire algorithm the block rows $\mathbf{C}_{i,-}$ contain fewer and fewer rows, whereas the number of rows of the block row $\mathbf{B}_{j,-}$ increases accordingly. After completing the block column j the matrices $\mathbf{B}_{jk}$ remain stable.

Similarly for the transformation matrix, the matrix $\mathbf{M}$ gains rows whereas $\mathbf{K}$ loses rows. However, things here are slightly more complicated due to the fact that we do not store the full transformation matrix. As we only store columns of $\mathbf{K}_{ih}$ that are not known a priori to be zero or columns of an identity matrix, we have to ensure that all the needed columns are present when calling UpdateRowTrafo. The columns that are not yet present are precisely the columns that correspond to the positions of the pivot rows and are stored in $\mathbf{E}_{hj}.\delta$. If $i = h$ this means we need to insert into the correct positions columns of an identity matrix and if $i \neq h$ then columns of a zero matrix. To achieve this, and also to efficiently deal with the cases where the matrices $\mathbf{K}_{ih}$ or $\mathbf{M}_{jh}$ are to be initialized we adapted the task UpdateRow to obtain UpdateRowTrafo.

3.3 Step 2

This intermediate step becomes necessary as we do only store the relevant parts of the transformation matrices $\mathbf{M}_{ji}$. Before the upwards cleaning in Step 3 we need to riffle in zero columns in $\mathbf{M}_{ji}$ so that the number of columns in $\mathbf{M}_{ji}$ is equal to the number of columns in $\mathbf{M}_{bi}$ for all j.

3.4 Step 3

Then back cleaning only performs upwards row operations on the matrix from Equation (4) to eliminate the X_{jk}. The matrices R_{jk} from Equation (4) are stored in the data packages $\mathbf{R}_{jk}$ in the Chief. Having cleaned block columns $b, \dots, k-1$ the algorithm adds the X_{jk} multiple of block row k to block row j for all $j \leq k-1$ to clear block column k. The same row operations are performed on the relevant part $\mathbf{M}$ of the transformation matrix.

Algorithm 1: The Chief

Input : $\mathcal{C} =: (\mathbf{C}^1_{ik})_{i=1,\ldots,a,k=1,\ldots,b}$, where $\mathbf{C}^1_{ik} \in \mathbb{F}^{a_i \times b_k}$

Output: $\mathcal{R} = (\mathbf{R}^{k-j}_{jk})_{j \le k=1,\ldots,b}$, $\mathcal{M} = (\mathbf{M}^{2a}_{jh})_{j=1,\ldots,b,h=1,\ldots,a}$, $\mathcal{K} = (\mathbf{K}^a_{ih})_{i,h=1,\ldots,a}$, a row select matrix $\varrho \subseteq \{1,\ldots,m\}$, the concatenation of the $\mathbf{E}_{ib}.\rho \subseteq \{1,\ldots,a_i\}$ $(i = 1,\ldots,a)$, and a column select matrix $\Upsilon \subseteq \{1,\ldots,n\}$, the concatenation of the $\mathbf{D}^a_j.\gamma \subseteq \{1,\ldots,b_j\}$ $(j = 1,\ldots,b)$, such that

$$\begin{pmatrix} \mathcal{M} & 0 \\ \mathcal{K} & 1 \end{pmatrix} \begin{pmatrix} \varrho \\ \overline{\varrho} \end{pmatrix} \mathcal{C} \begin{pmatrix} \Upsilon & \overline{\Upsilon} \end{pmatrix} = \begin{pmatrix} -1 & \mathcal{R} \\ 0 & 0 \end{pmatrix}$$

Step 1:

for *i from 1 to a* **do**
  for *j from 1 to b* **do**
    $(\mathbf{D}^i_j; \mathbf{A}_{ij}) :=$ ClearDown$(\mathbf{C}^j_{ij}, \mathbf{D}^{i-1}_j, i)$;
    $\mathbf{E}_{ij} :=$Extend$(\mathbf{A}_{ij}, \mathbf{E}_{i,j-1}, j)$;
    for *k from j + 1 to b* **do**
      $(\mathbf{C}^{j+1}_{ik}, \mathbf{B}^i_{jk}) :=$ UpdateRow$(\mathbf{A}_{ij}, \mathbf{C}^j_{ik}, \mathbf{B}^{i-1}_{jk}, i)$;
    for *h from 1 to i* **do**
      $(\mathbf{K}^{j+1}_{ih}, \mathbf{M}^i_{jh}) :=$UpdateRowTrafo$(\mathbf{A}_{ij}, \mathbf{K}^j_{ih}, \mathbf{M}^{i-1}_{jh}, \mathbf{E}_{hj}, i, h, j)$;

Step 2:

for *j from 1 to b* **do**
  for *h from 1 to a* **do**
    $\mathbf{M}^{a+1}_{jh} :=$RowLengthen$(\mathbf{M}^a_{jh}, \mathbf{E}_{hj}, \mathbf{E}_{hb})$;

Step 3:

for *k from* 1 *to b* **do**
  $\mathbf{R}^0_{kk} :=$Copy$(\mathbf{D}^a_k)$;
for *k from b downto* 1 **do**
  for *j from 1 to k* − 1 **do**
    $(\mathbf{X}_{jk}, \mathbf{R}^0_{jk}) :=$ PreClearUp$(\mathbf{B}^a_{jk}, \mathbf{D}^a_k)$;
    for *ℓ from k to b* **do**
      $\mathbf{R}^{\ell-k+1}_{j\ell} :=$ ClearUp$(\mathbf{R}^{\ell-k}_{j\ell}, \mathbf{X}_{jk}, \mathbf{R}^{\ell-k}_{k\ell})$;
    for *h from 1 to a* **do**
      $\mathbf{M}^{a+h}_{jh} :=$ ClearUp$(\mathbf{M}^{a+h-1}_{jh}, \mathbf{X}_{jk}, \mathbf{M}^{a+h-1}_{kh})$;

4 Jobs and Tasks

4.1 The jobs

In this section we describe the jobs. These are fundamental steps that are later used to define the tasks. Many of the jobs take as input one or more matrices. While the input and output matrices of the jobs within the global context of the parallel Gauss algorithm are blocks computed from a huge input matrix, the jobs described in this section work locally only on these matrices. In current implementations, each job can be performed by a single threaded computation, entirely in RAM and in a reasonable amount of time.

CPY This task simply copies the input matrix to the output.

MUL This job performs a matrix **mul**tiplication. It takes as input two matrices $A \in \mathbb{F}^{\alpha\times\beta}$ and $B \in \mathbb{F}^{\beta\times\delta}$ and returns as output the matrix $A \times B \in \mathbb{F}^{\alpha\times\delta}$.

MAD This job performs a matrix **m**ultiplication followed by a matrix **ad**dition. It takes as input matrices $A \in \mathbb{F}^{\alpha\times\delta}$ and $B \in \mathbb{F}^{\alpha\times\beta}$ and $C \in \mathbb{F}^{\beta\times\delta}$ and returns the matrix $A + B \times C \in \mathbb{F}^{\alpha\times\delta}$.

ECH This job performs an **ech**elonisation as described in Remark 2.3. We will refer to the job as

$$(M, K, R, \rho, \gamma) := \mathrm{ECH}(H).$$

CEX This job performs two **c**olumn **ex**tracts. It takes as input a matrix $H \in \mathbb{F}^{\alpha\times\beta}$ and a subset $\gamma \subseteq \{1, \ldots, \beta\}$ and returns the matrices $H \times \gamma^{tr}$ and $H \times \overline{\gamma}^{tr}$, consisting of all those columns of H whose indices lie in γ, respectively do not lie in γ, as described in Definition 2.1.

REX This job performs two **r**ow **ex**tracts. It takes as input a matrix $H \in \mathbb{F}^{\alpha\times\beta}$ and a subset $\rho \subseteq \{1, \ldots, \alpha\}$ and returns the matrices $\rho \times H$ and $\overline{\rho} \times H$, consisting of all those rows of H whose indices lie in ρ, respectively do not lie in ρ, as described in Definition 2.1.

UNH This job performs a **un**ion plus **h**istory. It takes as input a subset $\rho_1 \subseteq \{1, \ldots, \alpha\}$ and, for $\alpha_0 = \alpha - |\rho_1|$, a subset $\rho_2 \subseteq \{1, \ldots, \alpha_0\}$ and returns a subset $\rho \subseteq \{1, \ldots, \alpha\}$ defined as follows. Write $\{1, \ldots, \alpha\} \setminus \rho_1 = \{x_1, \ldots, x_{\alpha_0}\}$. Define

$$\rho = \rho_1 \cup \{x_i \mid i \in \rho_2\} =: \{y_1, \ldots, y_r\}$$

as an ordered set. Then $u \in \{0, 1\}^r$ with $u_\ell = 0$ if $y_\ell \in \rho_1$ and $u_\ell = 1$ otherwise. We refer to this job as

$$(\rho, u) := \mathrm{UNH}(\rho_1, \rho_2).$$

UN0 This job does the same as UNH except that the first input set of UNH is omitted and assumed empty.

MKR It takes two sets $\rho_1 \subseteq \rho_2$ and produces a bitstring $\lambda \subseteq \{0,1\}^{|\rho_2|}$ with 1s corresponding to the elements in ρ_1 and 0s corresponding to the elements in $\rho_2 \backslash \rho_1$.

RRF This job performs a row riffle. The input consists of a bit string $u \in \{0,1\}^r$ and two matrices $B \in \mathbb{F}^{\alpha \times \beta}$ and $C \in \mathbb{F}^{\gamma \times \beta}$ with $\alpha + \gamma = r$, where the number of 0s in u is α and the number of 1s in u is γ. The job returns the new matrix $A \in \mathbb{F}^{r \times b}$ whose rows are the rows of B and C combined according to u. In some sense this is the inverse of row extract.

CRZ Similarly to row riffles we also need column riffles, but we only need to riffle in zero columns.

ADI It takes as input a matrix $K \in \mathbb{F}^{\alpha \times \beta}$ and a bitstring $\delta \in \{0,1\}^{\beta}$ and puts $K_{i,j_i} := 1$ if j_i is the position of the ith 0 in δ. Note that combining CRZ with ADI allows us to riffle in columns of the identity matrix.

4.2 The tasks

We now describe the tasks on which our Gaussian elimination algorithm depends. As mentioned above, a task receives data packages as input, which in turn may consist of several components, and returns data packages as output.

Definition 4.1. *A* data package *is a record of one or several components. A data package is called* ready *(for a given scheduler) if the task that produces it as output has finished, regardless whether this task has computed all its components. If there is no task having this data package as an output, then we also consider it ready.*

Example: Task UPDATEROW.
If called with the parameter $i = 1$ the task UPDATEROW can start, even though the component $\mathbf{A}$.A of the data package $\mathbf{A}$ has not been computed, after the task CLEARDOWN for $i = 1$ has completed. Note that for $i = 1$, no job in the task UPDATEROW takes the component $\mathbf{A}$.A as an input. Also the data package $\mathbf{B}^0_{jk}$ is an input to UPDATEROW but not computed by any task in THE CHIEF. So therefore it is also considered ready.

Task 1: EXTEND

Input : $\mathbf{A} = (A, M, K, \rho', E, \lambda)$, $\mathbf{E} = (\rho, \delta)$ with $\rho \subset \{1, \ldots, \alpha\}$, δ a riffle, j
Output: $\mathbf{E}$.

$j = 1$ (UN0): $(\mathbf{E}.\rho, \mathbf{E}.\delta) := \mathrm{UN0}(\mathbf{A}.\rho')$;
$j \neq 1$ (UNH): $(\mathbf{E}.\rho, \mathbf{E}.\delta) := \mathrm{UNH}(\mathbf{E}.\rho, \mathbf{A}.\rho')$;

Task 2: RowLengthen

Input : $\mathbf{M} \in \mathbb{F}^{\alpha \times g_1}$, $\mathbf{E}_1.\rho \subseteq \mathbf{E}_2.\rho \subseteq \{1, \ldots, \alpha\}$ of sizes g_1, g_2 with $g_1 \leq g_2$.
Output: $\mathbf{M} \in \mathbb{F}^{\alpha \times g_2}$.

(MKR): $\lambda := \mathrm{MKR}(\mathbf{E}_1.\rho, \mathbf{E}_2.\rho)$;
(CRZ): $\mathbf{M} := \mathrm{CRZ}(\mathbf{M}, \lambda)$;

Task 3: ClearUp

Input : $\mathbf{R} \in \mathbb{F}^{\alpha \times \beta}$, $\mathbf{X} \in \mathbb{F}^{\alpha \times \gamma}$, $\mathbf{M} \in \mathbb{F}^{\gamma \times \beta}$.
Output: $\mathbf{R} \in \mathbb{F}^{\alpha \times \beta}$.

(MAD): $\mathbf{R} := \mathbf{R} + \mathbf{X} \times \mathbf{M}$;

Task 4: PreClearUp

Input : $\mathbf{B} \in \mathbb{F}^{\alpha \times \beta}$, $\mathbf{D}$, with $\mathbf{D}.\gamma \subseteq \{1, \ldots, \beta\}$ of cardinality g.
Output: $\mathbf{X} \in \mathbb{F}^{\alpha \times g}$, $\mathbf{R} \in \mathbb{F}^{\alpha \times (\beta - g)}$.

(CEX): $\mathbf{X} := \mathbf{B} \times \mathbf{D}.\gamma^{tr}$; $\mathbf{R} := \mathbf{B} \times \overline{\mathbf{D}.\gamma}^{tr}$;

Task 5: Copy

Input : $\mathbf{D}$, with $\mathbf{D}.\mathrm{R} \in \mathbb{F}^{\alpha \times \beta}$.
Output: $\mathbf{R} \in \mathbb{F}^{\alpha \times \beta}$.

(CPY): $\mathbf{R} := \mathbf{D}.\mathrm{R}$;

4.2.1 Task ClearDown

Task ClearDown works on block columns. Suppose that $j \in \{1, \ldots, b\}$ and ClearDown works on block column j which contains b_j columns. Task ClearDown assumes that block column j truncated after row $i-1$ is in row echelon form and the aim of task ClearDown is to replace the block column j truncated after row i by its row echelon form.

Task ClearDown takes two data packages $\mathbf{C}$ and $\mathbf{D}$ as input. The first data package $\mathbf{C}$ is the block $\mathbf{C}_{ij}$ which is the block in the i-th block row of block column j. The second data set $\mathbf{D}$ contains two data elements. The data element $\mathbf{D}.\mathrm{R}$ is a matrix such that block column j truncated after block row $i-1$ is in row echelon form $(-1 \mid \mathbf{D}.\mathrm{R})$ followed by 0 rows. The data element $\mathbf{D}.\gamma \subseteq \{1, \ldots, b_j\}$ contains indices of the pivots assembled in block column j truncated after block row $i-1$.

The task produces two data packages $\mathbf{A}$ and $\mathbf{D}$ as outputs. The data elements stored in the data package $\mathbf{A}$ are required to propagate row operations performed during the call to Task UpdateRow to other blocks in block row i. The data elements stored in

the data package $\mathbf{D}$ are required for a subsequent call to ClearDown for the block $\mathbf{C}_{i+1,j}$ in block column j.

We begin by partitioning the input block $\mathbf{C}$ according to $\mathbf{D}.\gamma$ into pivotal and non pivotal columns $\mathbf{C} = (\mathbf{A}.\mathrm{A} \mid A')$. Using the rows of the matrix $(-1 \mid \mathbf{D}.\mathrm{R})$ we can reduce $\mathbf{C}$ to $(0 \mid H')$ where $H' = A' + \mathbf{A}.\mathrm{A} \times \mathbf{D}.\mathrm{R}$. The next step is to call job ECH to echelonise H' and obtain

$$(\mathbf{A}.\mathrm{M}, \mathbf{A}.\mathrm{K}, R, \mathbf{A}.\rho', \gamma') := \mathrm{ECH}(H'),$$

where $\mathbf{A}.\rho'$ is the set of pivotal rows of H' and γ' the set of pivotal columns.

As block column j truncated after block row $i-1$ is in row echelon form $(-1_{r\times r} \mid \mathbf{D}.\mathrm{R})$ followed by 0 rows, we now wish to determine a new remnant matrix $\hat{R}$ (which will become the new $\mathbf{D}.\mathrm{R}$) such that block column j truncated after block row i is in row echelon form $(-1_{(r+r')\times(r+r')} \mid \hat{R})$ followed by 0 rows. To achieve this, the we have to add the r' pivots of H' to $-1_{r\times r}$ and reduce $\mathbf{D}.\mathrm{R}$ according to $(-1_{r'\times r'} \mid R)$. This amounts to first separating the columns of $\mathbf{D}.\mathrm{R}$ into those containing pivot entries of H' and those that do not, i.e. writing $\mathbf{D}.\mathrm{R} = (\mathbf{A}.\mathrm{E} \mid R')$ with the help of the row select and row non-select matrices γ' and $\overline{\gamma'}$. We then use the rows of the matrix $(-1_{r'\times r'} \mid R)$ to reduce $\mathbf{D}.\mathrm{R}$ to $(0 \mid R' + \mathbf{A}.\mathrm{E} \times \mathbf{D}.\mathrm{R})$. The new set $\mathbf{D}.\gamma$ of all pivotal columns of block column j truncated after block row i is now obtained by combining the old set $\mathbf{D}.\gamma$ and γ'. We record in $\mathbf{A}.\lambda$ the information which of these indices came from the r' pivots of H'. Finally, the new remnant $\hat{R}$ is obtained by interleaving the rows of $R' + \mathbf{A}.\mathrm{E} \times R$ with the rows of R according to $\mathbf{A}.\lambda$ and storing the resulting matrix as the new $\mathbf{D}.\mathrm{R}$.

The following pseudo code details Task ClearDown:

Task 6: ClearDown

Input : $\mathbf{C} \in \mathbb{F}^{\alpha\times\beta}$, $\mathbf{D}.\gamma \subseteq \{1,\ldots,\beta\}$ of cardinality r, $\mathbf{D}.\mathrm{R} \in \mathbb{F}^{r\times(\beta-r)}$, i;
Output: $\mathbf{D}.\mathrm{R} \in \mathbb{F}^{(r+r')\times(\beta-r-r')}$, $\mathbf{D}.\gamma \subseteq \{1,\ldots,\beta\}$ of cardinality $r+r'$ and
$\mathbf{A} = (A, M, K, \rho', E, \lambda)$ where $A \in \mathbb{F}^{\alpha\times r}$, $M \in \mathbb{F}^{r'\times r'}$, $E \in \mathbb{F}^{r\times r'}$,
$K \in \mathbb{F}^{(\alpha-r')\times r'}$, $\rho' \subseteq \{1,\ldots,\alpha-r\}$ of cardinality r', $\lambda \in \{0,1\}^{r+r'}$.

if $i = 1$ **then**
(ECH): $(\mathbf{A}.\mathrm{M}, \mathbf{A}.\mathrm{K}, \mathbf{D}.\mathrm{R}, \mathbf{A}.\rho', \mathbf{D}.\gamma) := \mathrm{ECH}(\mathbf{C})$;
else
(CEX): $\mathbf{A}.\mathrm{A} := \mathbf{C} \times \mathbf{D}.\gamma^{tr}$; $A' := \mathbf{C} \times \overline{\mathbf{D}.\gamma}^{tr}$;
(MAD): $H := A' + \mathbf{A}.\mathrm{A} \times \mathbf{D}.\mathrm{R}$;
(ECH): $(\mathbf{A}.\mathrm{M}, \mathbf{A}.\mathrm{K}, R, \mathbf{A}.\rho', \gamma') := \mathrm{ECH}(H)$;
(CEX): $\mathbf{A}.\mathrm{E} := \mathbf{D}.\mathrm{R} \times (\gamma')^{tr}$, $R' := \mathbf{D}.\mathrm{R} \times (\overline{\gamma'})^{tr}$;
(MAD): $R' := R' + \mathbf{A}.\mathrm{E} \times R$;
(UNH): $(\mathbf{D}.\gamma, \mathbf{A}.\lambda) := \mathrm{UNH}(\mathbf{D}.\gamma, \gamma')$;
(RRF): $\mathbf{D}.\mathrm{R} := \mathrm{RRF}(\mathbf{A}.\lambda, R', R)$;
end

4.2.2 Task UpdateRow

Given $i \in \{1, \ldots, a\}$, the Task UpdateRow works on block $\mathbf{C} = \mathbf{C}_{ik}$ in block row i and block column k. It takes as input data packages $\mathbf{A}$, $\mathbf{C}$ and $\mathbf{B}$, where the data package $\mathbf{A}$ encodes the necessary information computed by ClearDown when transforming an earlier block in the same block row i into echelon form.

The same row operations that were performed on this earlier block now need to be performed on $\mathbf{C}$. This subroutine also assembles in the matrix $\mathbf{B}$ the rows in block column k whose pivotal entry lies in block column j for $j + 1 \leq k \leq b$. The new data package $\mathbf{C}$ returned by Tasks UpdateRow then is equal to the transformed input matrix $\mathbf{C}$ with these rows deleted.

The following pseudo code details Task UpdateRow:

Task 7: UpdateRow

Input : $\mathbf{A} = (A, M, K, \rho', E, \lambda)$, $\mathbf{C} \in \mathbb{F}^{\alpha \times \beta}$, $\mathbf{B} \in \mathbb{F}^{r \times \beta}$, i.
Output: $\mathbf{C} \in \mathbb{F}^{(\alpha - r') \times \beta}$, $\mathbf{B} \in \mathbb{F}^{(r + r') \times \beta}$.

(1)	$i \neq 1$	(MAD):	$Z := \mathbf{C} + \mathbf{A}.\mathrm{A} \times \mathbf{B}$;
	$i = 1$	(CPY):	$Z := \mathbf{C}$;
(2)	always	(REX):	$V := \mathbf{A}.\rho' \times Z$; and $W := \overline{\mathbf{A}.\rho'} \times Z$;
(3)	always	(MUL):	$X := \mathbf{A}.\mathrm{M} \times V$;
(4)	$i \neq 1$	(MAD):	$S := \mathbf{B} + \mathbf{A}.\mathrm{E} \times X$;
(5)	$i \neq 1$	(RRF):	$\mathbf{B} := \mathrm{RRF}(\mathbf{A}.\lambda, S, X)$;
	$i = 1$	(CPY):	$\mathbf{B} := X$;
(6)	always	(MAD):	$\mathbf{C} := W + \mathbf{A}.\mathrm{K} \times V$;

Remark 4.2. *In the case $i = 1$ in Task* UpdateRow *we work with the first block row. Therefore we do not need to perform the upwards cleaning on the data package* $\mathbf{C}$ *and the data package* $\mathbf{B}$ *is initialized accordingly. Note that for $i = 1$ the task* ClearDown *did not compute the components* $\mathbf{A}.\mathrm{A}$, $\mathbf{A}.\mathrm{E}$ *and* $\mathbf{A}.\lambda$ *and also the input data package* $\mathbf{B}$ *of* UpdateRow *is not present.*

4.2.3 Task UpdateRowTrafo

If one is not too concerned about performance, then it would be possible to generate an identity matrix $\mathbf{K}$ and apply the UpdateRow task replacing $\mathbf{C}$ by $\mathbf{K}$ and $\mathbf{B}$ by $\mathbf{M}$ to mimic the relevant row operations performed to obtain the transformation matrix $\mathbf{M}$ and the cleaner matrix $\mathbf{K}$. This would result in a lot of needless work performed on zero or identity matrices. The main difference is that we never store any columns known to belong to an identity or a zero matrix. Instead we insert these columns just before they are needed. Moreover, we never add a matrix known to be zero or multiply by a matrix known to be the identity.

As a result, we require a separate procedure, UpdateRowTrafo, to mimic the row operations on the transformation matrix. UpdateRowTrafo still performs the

same steps as UpdateRow, identified by the same numbers, however it requires some additional steps, indicated by the symbol + in the first column and which insert some unstored columns into the matrix $\mathbf{K}$. The various instances of a given step are due to the fact that we can often avoid unnecessary work. In particular, UpdateRowTrafo takes as an additional input the integers i, j, h, with $h \leq i$. The integer i indicates the current block row and h the current block column, on which to mimic the row operations performed during UpdateRow on block row i. If $j > 1$ then we already computed some input $\mathbf{K}$ into which we need to riffle in zero (if $i \neq h$) or the relevant columns of the identity matrix (if $i = h$). It turns out that it is easier to always riffle in zero (the first line marked with +) and mimic the special case $i = h$ by adding the correct 0/1 matrix to V later in the other line marked with +. If $j = 1$ then we should initialise $\mathbf{K}$ with zero (if $i \neq h$) or the relevant columns of the identity matrix (if $i = h$). As we only need the input $\mathbf{K}$ to define V and W in line (2), we mimic this by remembering that $W = 0$ in this case and V is either 0 (if $i \neq h$) or the identity matrix if $i = h$. So for $j = 1$ and $h = i$ we omit the multiplication by the identity in lines (3) and (6).

If $i = 1$ again Remark 4.2 applies accordingly to line (4). Note that due to the fact that $h \leq i$, this only happens if $h = i = 1$.

In the following pseudo code describing Task UpdateRowTrafo we indicate in the last column which unstored matrices are implicitly known to be 0 or the identity 1.

Task 8: UpdateRowTrafo

Input : $\mathbf{A} = (A, M, K, \rho', E, \lambda)$, $\mathbf{K} \in \mathbb{F}^{\alpha\times\beta}$, $\mathbf{M} \in \mathbb{F}^{r\times\beta'}$, $\mathbf{E} = (\rho, \delta)$, i, h, j.
Output: $\mathbf{K} \in \mathbb{F}^{(\alpha-r')\times(\beta+|\delta|)}$, $\mathbf{M} \in \mathbb{F}^{(r+r')\times\beta'}$.

	case	job	command	remark
+	$j \neq 1$	(CRZ):	$\mathbf{K} := \mathrm{CRZ}(\mathbf{K}, \mathbf{E}.\delta)$;	
	$j = 1$		–	$\mathbf{K}$ *is* 0
(1)	$h \neq i$, $j \neq 1$	(MAD):	$Z := \mathbf{K} + \mathbf{A}.\mathrm{A} \times \mathbf{M}$;	
	$h \neq i$, $j = 1$	(MUL):	$Z := \mathbf{A}.\mathrm{A} \times \mathbf{M}$;	$\mathbf{K}$ *is* 0
	$h = i$, $j \neq 1$	(CPY):	$Z := \mathbf{K}$;	$\mathbf{M}$ *is* 0
	$h = i$, $j = 1$		–	Z *is* 0
(2)	$\neg(j = 1 \wedge h = i)$	(REX):	$V := \mathbf{A}.\rho' \times Z$;	
			$W := \overline{\mathbf{A}.\rho'} \times Z$;	
	$j = 1 \wedge h = i$		–	V, W *are* 0
+	$j \neq 1 \wedge h = i$	(ADI):	$V := \mathrm{ADI}(V, \mathbf{E}.\delta)$;	
	$j = 1 \wedge h = i$		–	V *is* 1
(3)	$\neg(j = 1 \wedge h = i)$	(MUL):	$X := \mathbf{A}.\mathrm{M} \times V$;	
	$j = 1 \wedge h = i$	(CPY):	$X := \mathbf{A}.\mathrm{M}$	V *is* 1
(4)	$h \neq i$	(MAD):	$S := \mathbf{M} + \mathbf{A}.\mathrm{E} \times X$;	
	$h = i \neq 1$	(MUL):	$S := \mathbf{A}.\mathrm{E} \times X$;	M *is* 0
	$h = i = 1$		–	S *no rows*
(5)	$\neg(h = i = 1)$	(RRF):	$\mathbf{M} := \mathrm{RRF}(\mathbf{A}.\lambda, S, X)$;	
	$h = i = 1$	(CPY):	$\mathbf{M} := X$;	S *no rows*
(6)	$\neg(j = 1 \wedge h = i)$	(MAD):	$\mathbf{K} := W + \mathbf{A}.\mathrm{K} \times V$;	
	$j = 1 \wedge h = i$	(CPY):	$\mathbf{K} := \mathbf{A}.\mathrm{K}$;	V *is* 1, W *is* 0

5 Concurrency analysis

To measure the degree of concurrency of our algorithm we assign costs to each of the tasks. We perform a relative analysis, comparing the cost of a parallel Gauss algorithm with the cost of a sequential algorithm. Therefore, to simplify our analysis, we assume that the cost of a matrix multiplication of $(\alpha \times \beta) \cdot (\beta \times \gamma)$ possibly followed by addition is $\alpha\beta\gamma$ and the cost of echelonising an $\alpha \times \beta$ matrix of rank r is $\alpha\beta r$. It seems plausible that when assuming that these costs are homogeneous functions of some degree, bounded below by ω (see [4, Chapter 16]), then in the results of Proposition 5.3 and Theorem 5.4 the degree of concurrency can be replaced by some constant times $a^{\omega-1}$. We also assume that all blocks are square matrices of size $\alpha \times \alpha$, where α is not too small and

$$\alpha = \frac{n}{a} = \frac{m}{b}.$$

Then the tasks Extend, RowLengthen, PreClearUp, and Copy do not perform any time consuming operations (compared to ClearDown and UpdateRow), so we assign cost 0 to these tasks.

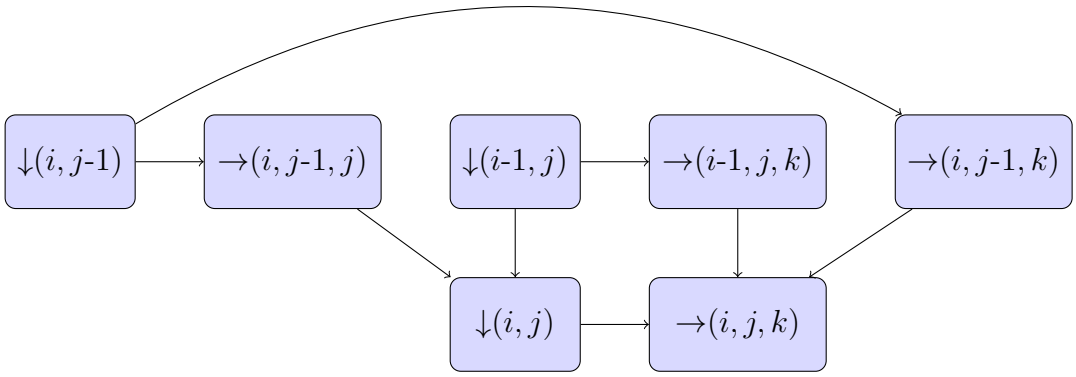


Figure 1: Extract of task dependency graph

Lemma 5.1. *The cost of* ClearDown *is bounded above by* α^3 *and the cost of* UpdateRow *is bounded above by* $1.25\alpha^3$.

Proof. We start analysing the task ClearDown: For $i = 1$ only one job ECH is performed contributing cost α^3. Otherwise the first call of MAD multiplies a matrix $\mathbf{A}.A$ of size $\alpha \times r$ with a matrix $\mathbf{D}.R$ of size $r \times (\alpha - r)$, contributing cost $\alpha r(\alpha - r)$. The echelonisation is done on an $\alpha \times (\alpha - r)$ matrix of rank r' and the second MAD multiplies an $r \times r'$ matrix by an $r' \times (\alpha - r - r')$ matrix. So in total the cost of ClearDown is

$$\alpha r(\alpha - r) + \alpha(\alpha - r)r' + rr'(\alpha - r - r') = \alpha^2(r + r') - rr'(\alpha + r + r') \leq \alpha^3$$

as $r + r' \leq \alpha$.
For the task UpdateRow we similarly obtain the cost $\alpha r\alpha$ for the MAD in row (1), $r'r'\alpha$ for the MUL in row (3), $rr'\alpha$ for the MAD in row (4), and $(\alpha - r')r'\alpha$ for the MAD in row (6). Summing up we obtain

$$\alpha^2(r + r') + \alpha rr' \leq 1.25\alpha^3$$

again since $r + r' \leq \alpha$. □

Ignoring all tasks of cost 0 Step 1 only involves the tasks ClearDown and UpdateRow. The graph of task dependencies decomposes naturally into layers according to the value of $i + j$. We abbreviate the call ClearDown$(\mathbf{C}_{ij}^{j}, \mathbf{D}_{j}^{i-1}, i)$ with data packages depending on i, j by $\downarrow(i, j)$ and similarly UpdateRow$(\mathbf{A}_{ij}, \mathbf{C}_{ik}^{j}, \mathbf{B}_{jk}^{i-1}, i)$ by $\rightarrow(i, j, k)$.Then Figure 1 displays the local task dependencies in Step 1 for layers $i+j-1$ and $i + j$, where $k = j + 1, \ldots, b$.

Recall that a critical path in a task dependency graph is the longest directed path between any pair of start node and finish node. Its length is weighted by the cost of the nodes along the path.

Proposition 5.2. *1. The weighted length of a critical path in the task dependency graph of Step 1 is* $2.25\alpha^3(a+b-1)$.

2. The weighted length of a critical path in Step 3 is $\max((b-1)\alpha^3, a\alpha^3)$.

Proof. 1.) The task dependency graph splits naturally into $a+b-1$ layers according to the value of $i+j \in \{2,\ldots,a+b\}$. Within each layer, the critical paths involve ClearDown and UpdateRow exactly once. So in total the length of a critical path is $(a+b-1)(\alpha^3+1.25\alpha^3)$.
2.) Step 3 only involves the task ClearUp of non-zero cost α^3. The data package $\mathbf{R}_{jh}$ is only changed by the data packages below in the same column so $h-j$ times. The maximum of $h-j$ is achieved at $\mathbf{R}_{1b}$ contributing the cost $\alpha^3(b-1)$. For the transformation matrix, which can be done independently, each of the $\mathbf{M}_{jh}$ is touched a times contributing $a\alpha^3$. □

To determine the average degree of concurrency we divide the cost of the sequential Gaussalgorithm (with transformation matrix) applied to the $m\times n$-matrix ClearDown by the weighted length of a critical path. For simplicity we assume that $m=n$, $a=b$ and that all blocks are of the same size $\alpha\times\beta$ with $\alpha=\beta$.

Proposition 5.3. *Under the assumptions above and using Lemma 5.1 the average degree of concurrency of* the Chief *is* $\frac{1}{5.5}a^2$.

Proof. By our assumptions $n=m$ and the cost of the sequential Gaussalgorithm (with transformation matrix) applied to the huge matrix ClearDown is $n^3=a^3\alpha^3$. By Proposition 5.2 the weighted length of a critical path in the complete algorithm the Chief is $(2.25(2a-1)+a)\alpha^3=(5.5a-2.25)\alpha^3\sim 5.5a\alpha^3$. □

In practical examples the gain of performance is much better. This is partly due to the fact that we split our matrix into blocks that fit into the computer's memory; an echelonisation of the huge matrix, however, would require to permanently read and write data to the hard disk. The other reason is that in random examples the length of a critical path is much shorter. To make this precise we assume, in addition to the assumptions above of starting with a square matrix partitioned into square blocks of equal size α, that our input matrix is *well-conditioned*, by which we mean that the a top-left square submatrices of the input matrix of size $j\alpha$ $(j=1,\ldots,a)$ have full rank. Then, properly implemented, the cost of ClearDown is α^3, if it is called for $i=j$ and 0 otherwise. Also in Update Row $r'=0$ and so the cost of Update Row is α^3 (this can be shown without using the assumptions in Lemma 5.1). In particular in the dependency graph above, the weighted length of a critical path in any odd layer is α^3 and in an even layer, this length is $2\alpha^3$ (resp. α^3 for the last layer). In total this shows the following

Theorem 5.4. *For a well-conditioned square matrix, the length of a critical path in the dependency graph is* $\alpha^3(3a-2)$ *and hence the average degree of concurrency of* the Chief *is* $\frac{1}{3}a^2$.

Remark 5.5. *The concurrency analysis above is not sufficient to ensure the effective use of all processors throughout the entire run that we see in the experimental results below.*

Although the steps are estimated at their worst case in practice many tasks early in the task dependency graph execute a lot faster. Provided there are sufficiently many blocks, enough work is available for execution considerably earlier than the task dependency graph might suggest. Assigning a priority $i + j$ to a task pertaining to the i-th row and j-th column directs an appropriate scheduling.

6 Experimental results

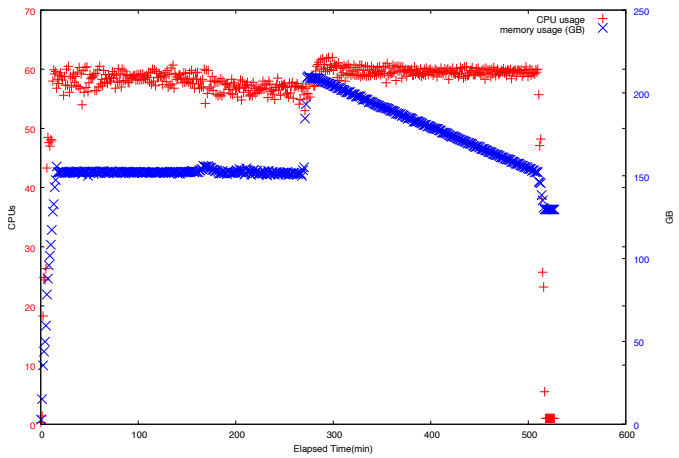
We give timings for the two implementations mentioned in Section 2.1.

The Meataxe

In order to demonstrate the power of this algorithm the following tests were done on a machine with 64-piledriver cores with 512 GB of memory running at 2.7 gigaherz.

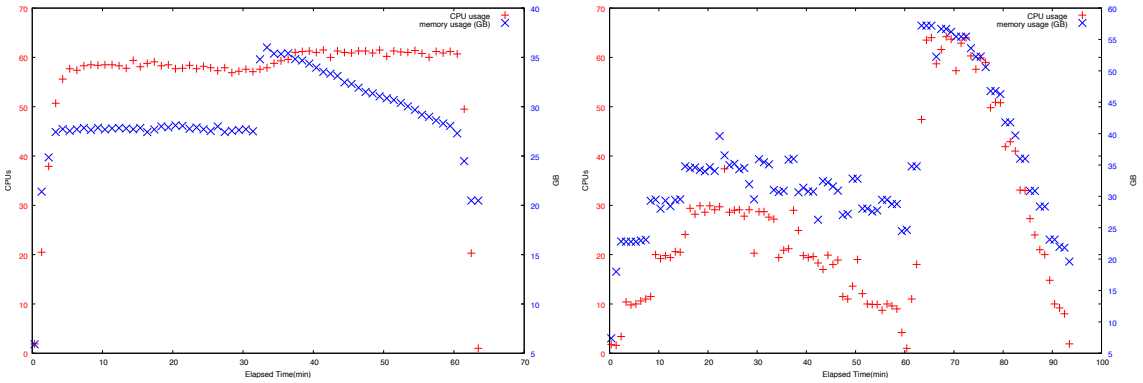
We chose as our first example a random $1,000,000 \times 1,000,000$-matrix with entries in the field of order 2. To put this matrix into reduced echelon form with transformation matrix we chose to chop the matrix into blocks of size $50,000 \times 50,000$. This run took 520min.

The following graph shows the progress of the calculation. The red shows that over 60 cores were used for the vast majority of the time. The blue shows that during Steps 1 and 2 the memory footprint was fairly constant but at the transition to Step 3 about 30% more memory was needed, due mainly to the expansion of the matrix $\mathcal{M}$.



For a second example, on the same machine, we echelonised with transformation matrix a random $600,000 \times 600,000$-matrix with entries in the field of order 3 in 460 min, using a block size of $30,000$. We do not give a graph for this run, as it is almost indistinguishable from the one given above.

The above examples were done with a carefully chosen block size. The following two examples highlight the effect of too large a block size. We echelonised the same random $300,000 \times 300,000$-matrix with entries in the field of order 3 using block sizes of $30,000$ and $15,000$, respectively. In the first graph we see that $15,000$ is again a good choice of block size. Almost all cores are used for most of the time. In the second graph we see that $30,000$ is too large a block size, so that the 64 available cores are seldomly utilised at once, leading to a run time which is about 50% longer.



Note that in general terms the necessary block size agrees with the concurrency analysis.

Our final three examples are intended to demonstrate that our methods are not restricted to tiny fields, nor to prime fields. A random $200,000 \times 200,000$-matrix with entries in the field with 193 took 445 mins, a random $200,000 \times 200,000$-matrix with entries in the field with $1331 = 11^3$ took 615 mins, and a random $100,000 \times 100,000$-matrix with entries in the field $50653 = 37^3$ elements took 200 mins.

7 Acknowledgements

We thank Martin Albrecht for discussions in the early stages of the algorithm design. The first full implementation of this algorithm was developed by Jendrik Brachter in GAP [8] which was essential to getting the details of the algorithm right. A parallel version of this implementation, using HPC-GAP, was produced by Jendrik Brachter and Sergio Siccha.

The second and third author acknowledge the support of the SFB-TR 195.

References

[1] Martin R. Albrecht The M4RIE library for dense linear algebra over small fields with even characteristic. ISSAC 2012-Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation, 28–34, ACM, New York, 2012.

[2] Martin Albrecht and Gregory Bard and William Hart. Algorithm 898: Efficient multiplication of dense matrices over GF (2) *ACM Transactions on Mathematical Software (TOMS)* **37** (1), Article 9, 2010.

[3] Martin Albrecht *et al.*,Linear Algebra over $\mathbb{F}_2$ (and $\mathbb{F}_{2^e}$) `https://malb.bitbucket.io/m4ri-e-website-2008-2015/further.html`

[4] Peter Bürgisser and Michael Clausen and M. Amin Shokrollahi, *Algebraic complexity theory*. With the collaboration of Thomas Lickteig. Grundlehren der Mathematischen Wissenschaften **315**. Springer-Verlag, Berlin, 1997.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*. (Second ed.), The MIT Press, Cambridge, Massachusetts, London McGraw-Hill Book Company, Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis Montreal Toronto, (2001).

[6] Simplice Donfack and Jack Dongarra and Mathieu Faverge and Mark Gates and Jakub Kurzak and Piotr Luszczek and Ichitaro Yamazaki, A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computat.: Pract. Exper.* (2014). `DOI: 10.1002/cpe.3306`

[7] Jean-Guillaume Dumas, Pascal Giorgi and Clément Pernet, Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages. *ACM Trans. on Mathematical Software (TOMS)* **35**(3), ACM Press, NY, USA, (2008), 1–42. `DOI: 10.1145/1391989.1391992`

[8] The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.9.1; 2018. (https://www.gap-system.org)

[9] Joseph F. Grcar, Mathematicians of Gaussian elimination. *Notices Amer. Math. Soc.* **58** (6), (2011), 782–792.

[10] Richard Parker, *Meataxe64 Blog*, `https://meataxe64.wordpress.com`

[11] PLASMA software package for solving problems in dense linear algebra. `http://icl.cs.utk.edu/plasma/`