Alcom-FT Technical Report Series
ALCOMFT-TR-03-6

Communication-Efficient Parallel Gaussian Elimination

Alexander Tiskin *

Department of Computer Science
University of Warwick, Coventry CV4 7AL, UK
tiskin@dcs.warwick.ac.uk

Abstract. The model of bulk-synchronous parallel (BSP) computation is an emerging paradigm of general-purpose parallel computing. In this paper, we consider the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We define a common framework that unifies both problems, and present a new communication-efficient BSP algorithm for their solution. Apart from being a useful addition to the growing collection of efficient BSP algorithms, our result can be viewed as a refinement of the classical "parallelism-communication tradeoff".

1 Introduction

The model of *bulk-synchronous parallel (BSP) computation* (see [19, 10, 12]) provides a simple and practical framework for general-purpose parallel computing. Its main goal is to support the creation of architecture-independent and scalable parallel software. Key features of BSP are its treatment of the communication medium as an abstract fully connected network, and strict separation of all interaction between processors into point-to-point asynchronous data communication and barrier synchronisation. This separation allows an explicit and independent cost analysis of local computation, communication and synchronisation.

In [18], we began the study of the BSP complexity of Gaussian elimination. In this paper, we make a further step in that direction, by considering the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We define a common framework that unifies both problems, and present a new communication-efficient BSP algorithm for their solution.

2 The BSP Model

A *BSP computer*, introduced in [19], consists of p *processors* connected by a communication network. Each processor has a fast *local memory.*

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A BSP computation consists of S supersteps, with costs $w_s + h_s \cdot g + l$, $1 \leq s \leq S$, where g, l are parameters of the computer. The total cost of such computation is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=1}^{S} w_s$ is the local computation cost, $H = \sum_{s=1}^{S} h_s$ is the communication cost, and S is the synchronisation cost. The values of W, H and S typically depend on the number of processors p and on the problem size.

Papers [10, 12] present the McColl–Valiant BSP algorithm for standard (non-Strassen) matrix multiplication, based on an idea from [1]. The BSP cost of this algorithm for $n \times n$ matrices is

$$W = O(n^3/p) \qquad H = O(n^2/p^{2/3}) \qquad S = O(1)$$

Paper [11] extends this result to fast (Strassen-type) matrix multiplication. The local computation, communication and synchronisation costs of the extended algorithm are $W = O(n^\omega/p), H = O(n^2/p^{2/\omega}), S = O(1)$,where ω is the exponent of fast matrix multiplication (currently 2.376 by [5]).

Another important BSP algorithm, presented in [10], concerns the computation of *the cube dag*, which is a three-dimensional grid of nodes with sequential dependence between the nodes in each dimension. The BSP cost of this algorithm for cube dag of size n is

$$W = O(n^3/p) \qquad H = O(n^2/p^{1/2}) \qquad S = O(p^{1/2})$$

A communication-efficient BSP algorithm for Gaussian elimination without pivoting was presented in [18]; we review this algorithm in Section 3. In Section 4 we introduce a new communication-efficient BSP algorithm for Gaussian elimination with pairwise pivoting.

For the sake of simplicity, throughout the paper we ignore small irregularities that arise from imperfect matching of integer parameters. For example, when we say that an array of size n is divided equally across p processors, the value n may not be an exact multiple of p, and therefore the shares may differ in size by ± 1. We use square bracket notation for matrices, referring to an element of an $n \times n$ matrix A as $A[i, j]$, $1 \leq i, j \leq n$.

3 Gaussian Elimination without Pivoting

This section and the next describe a BSP approach to Gaussian elimination, the method primarily used for direct solution of linear systems of equations. More generally, Gaussian elimination and its variations are applied to a broad spectrum of numerical, symbolic and combinatorial problems.

In this section we consider the simplest form of Gaussian elimination, which does not involve the search for pivots. This basic form of elimination is not guaranteed to produce the correct result, or to terminate at all, when performed on arbitrary matrices. However, it works well for matrices over some particular domains, such as closed semirings, or for matrices of some particular types, such as symmetric positive definite matrices over real numbers.

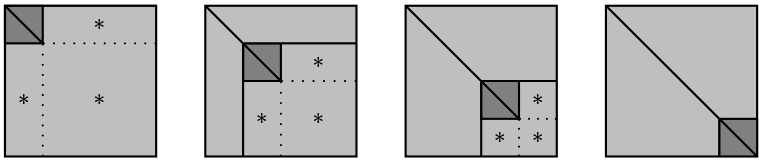


Fig. 1. Iterative block Gaussian elimination

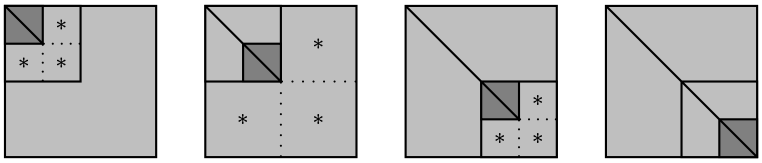


Fig. 2. Recursive block Gaussian elimination

Gaussian elimination can be represented in many ways. In this section we consider it in the form of LU decomposition. Let A be an $n \times n$ real diagonally dominant or symmetric positive definite matrix. The LU decomposition of A is $A = L \cdot U$, where L is an $n \times n$ unit lower triangular, and R is an $n \times n$ upper triangular matrix. This decomposition can be computed in sequential time $O(n^3)$ by plain Gaussian elimination, or in time $O(n^\omega)$ by block Gaussian elimination, using fast matrix multiplication.

The parallel complexity of Gaussian elimination has been extensively studied in various models of parallel computation. In [10] it is shown how to reduce the problem to the computation of a cube dag. The BSP cost of the resulting computation is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$ (see Section 2). The cube dag method is straightforward for LU decomposition, and can be easily adapted to other forms of Gaussian elimination, such as QR decomposition by Givens rotations.

Alternatively, we can apply a more general form of Gaussian elimination, replacing the elimination of single elements by the elimination of square blocks. The resulting method is known as *block Gaussian elimination.* Elimination within a block can be done by the standard pivoting-free algorithm, or take another form, e.g. employing column or full pivoting.

Figure 1 shows the process of block Gaussian elimination. Active blocks are denoted by small squares with darker shading. After block elimination, the resulting transformations must be applied to update the remaining matrix. The updated parts of the matrix are denoted in Figure 1 by asterisks.

In block Gaussian elimination, the block size must be chosen small enough, so that each iteration can be performed in $O(1)$ supersteps. When the size of the diagonal blocks is $n/p^{1/2}$, the BSP cost of the resulting algorithm is asymptotically equal to the cost of the cube dag method (see Section 2).

A lower communication cost for LU decomposition can be achieved by applying the block algorithm recursively (see e.g. [8, 7]). This standard method was suggested as a means of reducing the communication cost in [1] (for the transitive closure problem), and subsequently applied e.g. in [9]. The BSP cost of block Gauss–Jordan elimination was analysed in [18]; we summarise the results here for completeness.

Given a nonsingular matrix A, the algorithm produces the LU decomposition $A = L \cdot U$, together with the inverse matrices L^{-1} and U^{-1}. Matrices A, L, U are partitioned into regular square blocks of size $n/2$,

$$A = L \cdot U : \qquad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \cdot \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \end{pmatrix} \tag{1}$$

where the dot $\cdot$ indicates a zero block. The algorithm proceeds as follows:

1. Apply the algorithm recursively to obtain the LU decomposition of the block $A_{11} = L_{11} \cdot U_{11}$, along with the inverse blocks L_{11}^{-1} and U_{11}^{-1}.

2. Apply these inverse blocks to obtain L_{21}, U_{12}:

$$L_{21} \leftarrow A_{21} \cdot U_{11}^{-1} \qquad U_{12} \leftarrow L_{11}^{-1} \cdot A_{12} \tag{2}$$

3. Apply the algorithm recursively to obtain the LU decomposition $A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$, along with the inverse blocks L_{22}^{-1}, U_{22}^{-1}.

4. Obtain the inverse matrices L^{-1}, U^{-1} by

$$L^{-1} = \begin{pmatrix} L_{11}^{-1} & \cdot \\ -L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1} & L_{22}^{-1} \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} U_{11}^{-1} & -U_{11}^{-1} \cdot U_{12} \cdot U_{22}^{-1} \\ \cdot & U_{22}^{-1} \end{pmatrix} \tag{3}$$

Figure 2 shows the process of recursive block Gaussian elimination. As before, active blocks are denoted by small squares with darker shading, and the updating by asterisks.

We now describe the allocation of block processing and block multiplication tasks in (2)–(3) to the BSP processors. In each level of recursion, every block multiplication in (2)–(3) is performed in parallel by all p processors. Each block LU decomposition is assigned to all p processors, if the block size is large enough. When blocks become sufficiently small, block LU decomposition is computed sequentially by an arbitrarily chosen processor.

The depth at which the algorithm switches from p-processor to single-processor computation can be varied, allowing us to trade off the costs of communication and synchronisation in a certain range. In order to account for this tradeoff, we introduce a real parameter α, controlling the depth of recursion. The algorithm is as follows.

Algorithm 1 (Gaussian elimination without pivoting).
Parameters: integer $n \geq p$; real number α, $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.
Input: $n \times n$ real matrix A; we assume that A is diagonally dominant or symmetric positive definite.
Output: decomposition $A = L \cdot U$, where L is an $n \times n$ unit lower triangular matrix, and U is an $n \times n$ upper triangular matrix.

Description. The computation is defined by recursion on the size of the matrix. Denote the size of the block at the current level of recursion by m, keeping n for the original matrix size. Let $n_0 = n/p^\alpha$. Value $m = n_0$ is the threshold at which the algorithm switches to sequential computation. The recursion proceeds as described earlier in this section.
Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(m^3/p)$	$O(n_0^3)$
$H(m) =$	$2 \cdot H(m/2) + O(m^2/p^{2/3})$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha) \qquad \blacksquare$$

For $\alpha = \alpha_{\min} = 1/2$, the cost of Algorithm 1 is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. This is asymptotically equal to the BSP cost of the cube dag method from [10]. For $\alpha = \alpha_{\max} = 2/3$, the cost of Algorithm 1 is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(p^{2/3})$. In this case, the communication cost is as low as in matrix multiplication. This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. For large n, the communication cost of Algorithm 1 dominates the synchronisation cost, and therefore the communication improvement should outweigh the loss of synchronisation efficiency. This justifies the use of Algorithm 1 with $\alpha = \alpha_{\max} = 2/3$. Smaller values of α, or the cube dag algorithm, should be considered when the problem is moderately sized.

Fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The resulting algorithm has BSP cost $W = O(n^\omega/p), H = O(n^2/p^\alpha), S = O(p^\alpha)$,where $\alpha_{\min} = 1/(\omega - 1) \leq \alpha \leq 2/\omega = \alpha_{\max}$. As ω approaches the value of 2, the range of parameter α becomes tighter. If an $O(n^2)$ matrix multiplication algorithm is eventually discovered, the tradeoff between H and S will disappear.

4 Pairwise Pivoting and Givens Decomposition

In the previous section, we described an efficient parallelisation of the basic, pivoting-free variant of Gaussian elimination. This method works well in situations where pivoting is not necessary. However, in most cases some form of pivoting is essential. This is the case e.g. for numerical matrices without special structure, or for matrices over a finite field. In this section, we extend our Gaussian elimination algorithm with a simple, but useful, version of pivoting applicable in such cases.

Consider, for instance, the case of a matrix A over a finite field. We assume that field operations, including inversion of a nonzero element, can be performed in constant time. Plain Gaussian elimination without pivoting may fail on matrix A, since it requires the inversion of diagonal elements, some of which may be zero initially, or can become zero during elimination. Block Gaussian elimination without pivoting may fail for a similar reason, if some diagonal blocks that need to be inverted are singular or become singular in the course of the elimination.

A standard answer to such concerns in the case of numerical matrices is using column or full pivoting to achieve numerical stability. However, since the computation in our case is performed over a finite field, any elimination method would be suitable as long as it only inverts elements known to be nonzero. We can use this additional freedom of choosing the elimination pattern, in order to save on the BSP communication and synchronisation costs.

As the initial example, consider a 2×1 matrix $A = \left(\begin{smallmatrix} a_1 \\ a_2 \end{smallmatrix}\right)$. To eliminate its bottom element a_2, we pre-multiply matrix A by a 2×2 transformation matrix:

$$\begin{pmatrix} 1 & \cdot \\ -a_2/a_1 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ \cdot \end{pmatrix} \quad \text{if } a_1 \neq 0$$

$$\begin{pmatrix} \cdot & 1 \\ 1 & \cdot \end{pmatrix} \begin{pmatrix} 0 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_2 \\ \cdot \end{pmatrix} \quad \text{if } a_1 = 0$$

In the more general case of a $2 \times n$ matrix A, the above transformation will create a zero in the bottom-left corner, either by modifying the bottom row (if the top-left element is nonzero), or by swapping the rows (if the top-left element is zero). For a general $m \times n$ matrix, several such transformations on blocks of size $2 \times n$ can be carried out simultaneously. This technique is known as *pairwise pivoting* (see e.g. [17, 8]).

A similar pattern can be applied to perform elimination on numerical matrices. In this case, we have to be more careful in choosing the transformation, for reasons of numerical stability. A standard, numerically stable transformation known as *Givens rotation* is defined as

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \cdot \end{pmatrix}$$

where $c = a_1/(a_1^2 + a_2^2)^{1/2}$, $s = a_2/(a_1^2 + a_2^2)^{1/2}$, $b_1 = (a_1^2 + a_2^2)^{1/2}$. Again, for a general $m \times n$ matrix, several such elementary transformations can be performed in parallel on blocks of size $2 \times n$. Givens reduction for parallel solution of linear systems was proposed in [15] (see also [6]). Paper [4] describes a BSP algorithm for Givens reduction; however, its communication, and especially synchronisation efficiency is relatively low.

Since the above two examples share the same elimination pattern, we will refer to both by the term "pairwise pivoting". For our purposes it is convenient to consider, without loss of generality, the elimination process on a $2n \times n$ matrix $\left(\begin{smallmatrix} A_1 \\ A_2 \end{smallmatrix}\right)$, where A_1 is an arbitrary $n \times n$ matrix, and A_2 an upper triangular $n \times n$ matrix. Matrices A_1, A_2 may not have full rank. The problem consists in finding a full-rank $2n \times 2n$ transformation matrix D, and an $n \times n$ upper triangular matrix B_1, such that

$$D \cdot \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = \begin{pmatrix} B_1 \\ \end{pmatrix} \tag{4}$$

This problem is closely related to the problem of transforming a matrix to row echelon form (see [3]).

Similarly to the pivoting-free case, the problem can be solved in the BSP model by the cube dag method, using the standard elimination scheme (see e.g. [15, 13, 14]). Alternatively, we can apply a more general form of pairwise pivoting, replacing the elimination of 2×1 blocks by the elimination of $2k \times k$ blocks. We call the resulting method *block-pairwise elimination*. Elimination within a block can either be done by the standard algorithm with pairwise pivoting, or take another form, e.g. employing column or full pivoting.

Figure 3 shows the process of block-pairwise elimination on a $2n \times n$ matrix. Active blocks are denoted by small rectangles, with the eliminated part marked by darker shading. Blocks that are active simultaneously form *the wavefront*, which is, when viewed at high level, a straight sloping line $2j - i = \text{const}$. After block elimination, the resulting transformations must be applied to update the remaining matrix. The updated parts of the matrix are denoted in Figure 3 by asterisks.

Block-pairwise elimination can be efficiently implemented in the BSP model as follows. We set the block size to $2n/p^{1/2} \times n/p^{1/2}$. The computation proceeds in $2p^{1/2} - 1$ stages. In each stage, at most $p^{1/2}$ blocks are eliminated in one superstep. The updating process can be partitioned into at most p independent multiplications of square matrices of size $2n/p^{1/2}$. Therefore, the BSP cost of a stage is $W = O(n^3/p^{3/2})$, $H = O(n^2/p)$, $S = O(1)$. The overall BSP cost of the block-pairwise elimination algorithm is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. It is asymptotically equal to the cost of the cube dag algorithm (see Section 2) and of iterative pivoting-free block Gaussian elimination (see Section 3).

Similarly to the recursive pivoting-free algorithm described in Section 3, block-pairwise elimination can also be applied recursively (this was first proposed in [16]; see also [3, section 16.5]). The difference from the recursive pivoting-free algorithm is that now we cannot, in general, compute block inverses. After partitioning matrix $\left(\begin{smallmatrix} A_1 \\ A_2 \end{smallmatrix}\right)$ into regular square blocks of size $n/2$, the algorithm proceeds as follows:

1. Apply the algorithm recursively to the $n \times n/2$ matrix $\left(\begin{smallmatrix} A_{21} \\ A_{31} \end{smallmatrix}\right)$. Apply the resulting transformation matrix to update $\left(\begin{smallmatrix} A_{22} \\ A_{32} \end{smallmatrix}\right)$:

$$E \cdot \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ & A_{42} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A'_{21} & A'_{22} \\ & A'_{32} \\ & A_{42} \end{pmatrix} \tag{5}$$

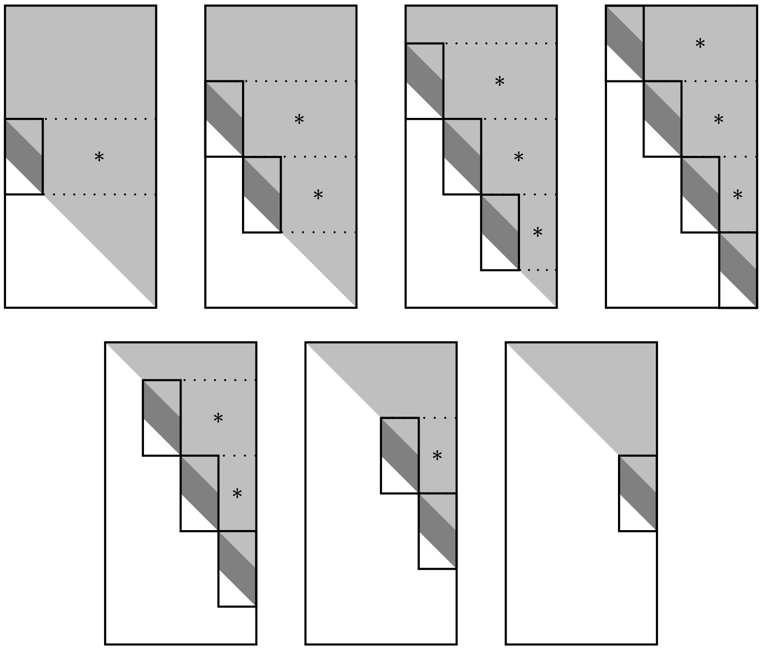


Fig. 3. Iterative block-pairwise elimination

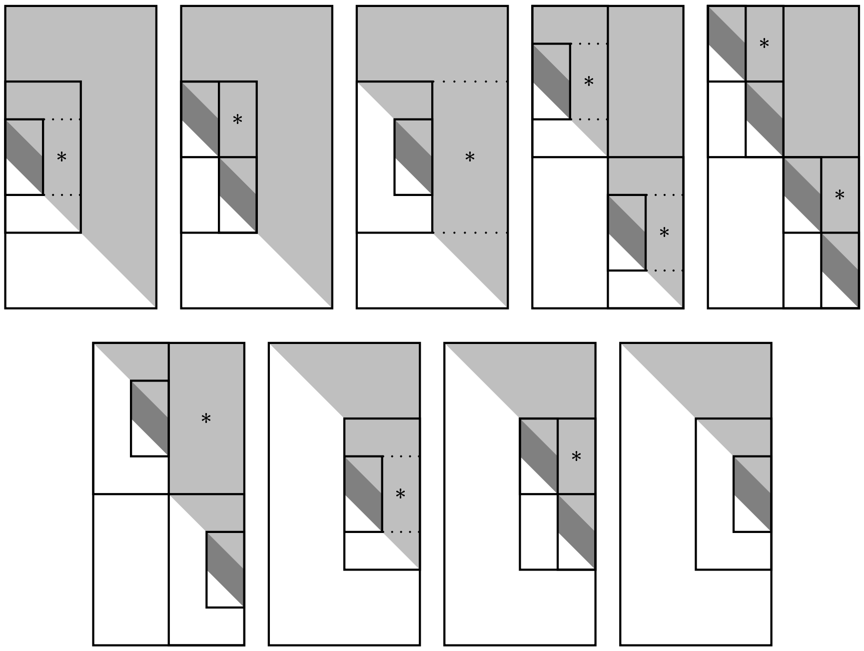
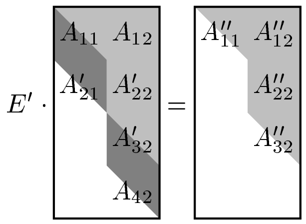


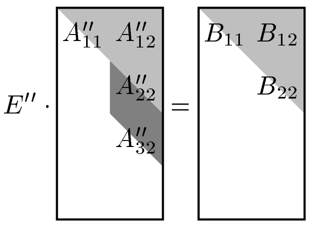
Fig. 4. Recursive block-pairwise elimination

2. Apply the algorithm recursively to matrices $\left(\begin{smallmatrix} A_{11} \\ A'_{21} \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} A'_{32} \\ A_{42} \end{smallmatrix}\right)$. Apply the first resulting transformation matrix to update $\left(\begin{smallmatrix} A_{12} \\ A'_{22} \end{smallmatrix}\right)$:



(6)

3. Apply the algorithm recursively to matrix $\left(\begin{smallmatrix} A''_{22} \\ A''_{32} \end{smallmatrix}\right)$:



(7)

4. Obtain the transformation matrix D in (4) as the product of the three transformation matrices from (5)–(7): $D = E'' \cdot E' \cdot E$.

Figure 4 shows the process of recursive block-pairwise elimination, using the same conventions as in Figure 3. Note that, in contrast with Figure 3, there is no straight-line wavefront.

Overall, the above method resembles recursive pivoting-free Gaussian elimination (Section 3, Algorithm 1); however, it requires four, rather than two, recursive calls on half-sized matrices. Consequently, the communication cost is higher than in the pivoting-free case. Moreover, the recursion has to be significantly deeper to achieve work-optimality, which leads to an increase in the synchronisation cost. We leave it as an exercise for the reader to check that the best synchronisation cost achievable under the work-optimality condition $W = O(n^3/p)$ is $S = O(p^{\log 3}) \approx O(p^{1.585})$, and to work out the corresponding communication cost H.

The inefficiency of the BSP algorithm based on the above approach motivates us to look for an alternative method, that would play for pairwise elimination the role that Algorithm 1 plays for pivoting-free elimination. We present such a method in the rest of this section.

It is clear that, although the recursive blocking approach based directly on [16] fails to provide an efficient BSP algorithm, any alternative method would need to use some form of blocking. One of the first blocked algorithms for Gaussian elimination with pivoting was proposed in [2]. Unfortunately, the quite complicated pivoting scheme of [2] does not lead to any improvement in BSP cost over the iterative block-pairwise elimination. Other existing blocking methods operate on whole columns (or, symmetrically, whole rows) of the matrix (see e.g.

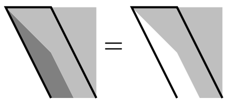
[8]). Such methods are suitable for row or column pivoting, but this comes at the price of a significantly higher communication cost.

In [18], we proposed a BSP algorithm that improves on the iterative block-pairwise elimination; however, its cost is still a polylogarithmic factor higher than the asymptotic cost of the pivoting-free algorithm (Algorithm 1). In the current work, we close this gap, and present an algorithm that achieves the cost asymptotically equal to that of the pivoting-free algorithm. In particular, our algorithm is also work-optimal, and also exhibits a tradeoff between communication and synchronisation.

From our discussion of the standard pairwise, the iterative block-pairwise, and the recursive block-pairwise elimination schemes, it is apparent that an efficient algorithm should maintain a relatively straight wavefront. To achieve that, our main idea is as follows: instead of operating on rectangular blocks, we proceed recursively on strips delimited by sloping lines $2j - i = \text{const}$. We call every such line *a pseudo-diagonal*.

For the presentation of our algorithm, it is convenient to consider elimination with pairwise pivoting on a matrix whose elements are zero below a given pseudo-diagonal $2j - i = r$. We call such a matrix *pseudo-triangular*. For simplicity of exposition, we describe our algorithm in terms of semi-infinite strips, unbounded in the positive direction of i, j. Actual computation will only be required in the region delimited by matrix boundaries. Let $A\langle k : l\rangle_b$ denote a semi-infinite strip in matrix A, that consists of elements $A[i, j]$ with $i \geq b$ and $k \leq 2j - i < l$. The value $l - k$ will be called *the width* of the strip.

The input of our algorithm is a strip $A\langle r : r + 2d\rangle_r$ for given r, d. The algorithm eliminates the lower half-strip $A\langle r : r + d\rangle_r$, apart from its upper triangle, which cannot be eliminated:

$D \cdot$  (8)

In the course of its execution, the algorithm also modifies (but does not eliminate) the upper half-strip $A\langle r + d : r + 2d\rangle_r$. The resulting transformation matrix D is banded of bandwidth $4d$. The part of the input matrix above the strip is left intact. For simplicity of exposition, we sacrifice some constant-factor efficiency by discarding the modifications made by the algorithm to the strip, and considering only the matrix D as the output of the algorithm. The matrix in the right-hand side of (8) can then be obtained by taking the product in the left-hand side, which recomputes the discarded modifications, and also modifies the previously untouched parts of matrix A. This updating pattern is applied in every level of recursion. In practice, recomputation is not necessary, but avoiding it requires a more complicated control structure than the one presented here.

Gaussian elimination on a square matrix can be reduced to the above problem by embedding an $n \times n$ matrix into a strip $A\langle 1 : 1 + 4n\rangle_1$, so that the top-left corner of the embedded matrix is at $A[n, n]$, and all elements outside the embedded matrix are set to zero.

The algorithm proceeds on the strip $A\langle r : r + 2d\rangle_r$ as follows:

1. Save the initial state of the strip $A\langle r : r + 2d\rangle_r$.

2. Apply the algorithm recursively to the lower half-strip $A\langle r : r+d\rangle_r$, reducing the quarter-strip $A\langle r : r+d/2\rangle_r$ to upper-triangular form, and modifying the quarter-strip $A\langle r+d/2 : r+d/2\rangle_r$. Just before returning from the recursive call, the algorithm discards the modifications and reverts the half-strip $A\langle r : r+d\rangle_r$ to its state previous to stage 2. The result of the recursive call is the transformation matrix E of bandwidth $4d$.

3. Apply the transformation matrix E to the current strip $A\langle r : r + 2d\rangle_r$, registering the result only in the middle half-strip $A\langle r + d/2 : r + 3d/2\rangle_{r+d/2}$. The correct updating of the whole strip would require access to matrix entries outside the current strip; we have chosen to update only the middle half-strip so that such a situation can be avoided.

4. Apply the algorithm recursively to the middle half-strip $A\langle r + d/2 : r + 3d/2\rangle_{r+d/2}$, reducing the quarter-strip $A\langle r+d/2 : r+d\rangle_{r+d/2}$ to upper-triangular form, and modifying the quarter-strip $A\langle r + d : r + 3d/2\rangle_{r+d/2}$. Just before returning from the recursive call, the algorithm discards the modifications and reverts the half-strip $A\langle r + d/2 : r + 3d/2\rangle_{r+d/2}$ to its state previous to stage 4. The result of the recursive call is the transformation matrix E' of bandwidth $4d$.

5. Revert the current strip $A\langle r : r + 2d\rangle_r$ to the state of stage 1.

6. Obtain the resulting transformation matrix as the product of the above two transformation matrices: $D = E' \cdot E$. Matrix D is of bandwidth $8d$.

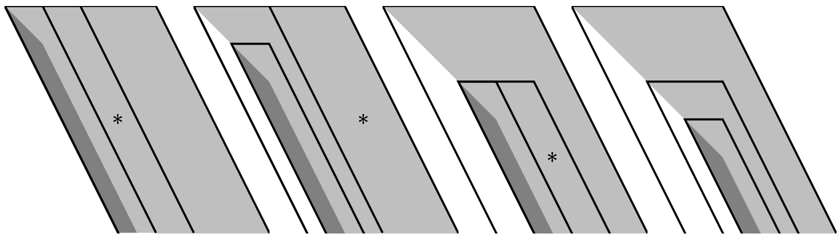


Fig. 5. New algorithm

Figure 5 shows the process of recursive block-pairwise elimination by our algorithm. Due to the recursive nature of the algorithm, the strips in each panel of Figure 5 are overlapped; for example, the first panel shows the overlapping strips $A\langle r : r + 2d\rangle_r$, $A\langle r : r + d\rangle_r$, $A\langle r : r + d/2\rangle_r$, and the last panel the strips $A\langle r : r + 2d\rangle_r$, $A\langle r + d/2 : r + 3d/2\rangle_{r+d/2}$, $A\langle r + 3d/4 : r + 5d/4\rangle_{r+3d/4}$. The eliminated part of the matrix is marked, as usual, by darker shading. The updated parts of the matrix are denoted by asterisks. More precisely, an asterisk indicates the current strip in stage 3 of the algorithm (for convenience of presentation, each asterisk is shifted to the upper half of its strip). The result of the update is registered in the middle half-strip of the current strip. This middle half-strip becomes the current strip in the subsequent recursive call.

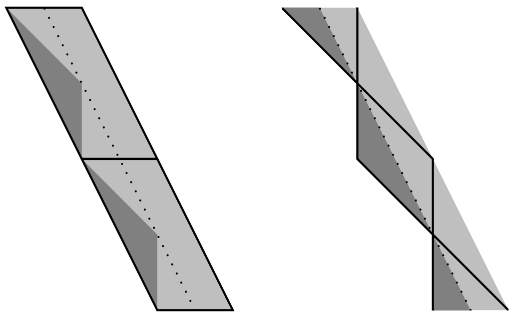


Fig. 6. New algorithm: the recursion base

In contrast with the pivoting-free case, where we could afford to process sufficiently small blocks sequentially, we cannot do so for strips without losing communication efficiency. Therefore, we need to provide a base for our recursion, by specifying a parallel procedure for block-pairwise elimination on a sufficiently narrow strip. Such a procedure is illustrated in Figure 6. For simplicity, the boundaries of the matrix are not shown: the panels represent a "typical" region of the input matrix, far away from the boundaries and not overlapping with the non-eliminated upper triangular part. The processing of the strip at the base of the recursion is performed as follows:

1. Save the initial state of the strip.

2. Partition the strip into parallelogram-shaped blocks, as in Figure 6 (left). In each block, eliminate a triangular "dent" by standard pairwise-pivoting elimination. It is straightforward to check that elimination of the "dents" (as opposed to elimination of the whole lower half-strip) and the computation of the transformation matrix (which in this case is block-diagonal) can be performed using only data local to the parallelogram blocks. After computing the transformation matrix, revert the strip to the state of stage 1.

3. Apply the transformation matrix as in the recursive procedure.

4. Repartition the strip into different parallelogram-shaped blocks, as in Figure 6 (right). Eliminate the lower triangle of each block. Again, it is straightforward to check that the elimination and the computation of the transformation matrix (which is again block-diagonal) can be performed using only data local to the blocks. After computing the transformation matrix, revert the strip to the state of stage 1.

5. Obtain the resulting transformation matrix as the product of the above two transformation matrices.

Note that this procedure is for the recursion base only. It cannot be applied recursively itself, since that would lead to a "ragged" wavefront, something that our recursive procedure has been specifically designed to avoid.

Throughout the algorithm, we multiply banded transformation matrices by sloping matrix strips, as well as by other banded matrices. It remains to verify that such products can be computed efficiently in the BSP model. Consider the multiplication of two such structured matrices, each of which has size $n \times n$ with $O(nm)$ nonzero entries (here m is the bandwidth or the width of the strip). Using the idea of the McColl–Valiant algorithm (Section 2), we can represent the array of elementary products by an $n \times n \times n$ cube. Out of these n^3 elementary products, only $O(nm^2)$ are nontrivial. Partition the cube into $(n/m)^2 \cdot p$ regular cubic blocks of size $n^{1/3}m^{2/3}/p^{1/3}$. The structure of the matrices is such that only $O(p)$ blocks contain nontrivial elementary products. Therefore, the multiplication can be performed on a BSP computer with computation cost $W = O\bigl(nm^2/p\bigr)$, communication cost $H = O\bigl(n^{2/3}m^{4/3}/p^{2/3}\bigr)$, and synchronisation cost $O(1)$.

We now describe the allocation of strip processing and matrix updating tasks to the BSP processors. In each level of recursion, the structured matrix multiplication tasks are performed in parallel by all p processors. Each strip processing task is assigned to all p processors, if the strip is large enough. When the strip becomes sufficiently small, we follow the recursion base procedure, allocating every block of the strip to a different processor.

Similarly to the pivoting-free case (Algorithm 1), the depth at which the algorithm switches to the recursion base procedure can be varied, resulting in a tradeoff between the communication and synchronisation costs. As before, we introduce a real parameter α, controlling the depth of recursion. The algorithm is as follows.

Algorithm 2 (Gaussian elimination with pairwise pivoting).
Parameters: integer $n \geq p$; real number α, $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.
Input: $n \times n$ matrix A.
Output: decomposition $D \cdot A = B$, where D is a full-rank $n \times n$ transformation matrix, and B is an $n \times n$ upper triangular matrix. This generic form of decomposition captures e.g. LU decomposition of matrices over a finite field, or QR decomposition of numerical matrices.
Description. Matrix A is embedded in a suitable pseudo-triangular matrix with a leading strip of width $4n$. The computation is then defined by recursion on the width of the strip. Denote the strip width at the current level of recursion by m, keeping $4n$ for the original width. Let $n_0 = n/p^{\alpha}$. Value $m = n_0$ is the threshold at which the algorithm switches to the recursion base procedure. The recursion proceeds as described earlier in this section.
Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(n \cdot m^2/p)$	$O(n_0^3)$
$H(m) =$	$2 \cdot H(m/2) + O(n^{2/3} \cdot m^{4/3}/p^{2/3})$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \qquad H = O(n^2/p^{\alpha}) \qquad S = O(p^{\alpha}) \qquad \blacksquare$$

Considerations similar to the ones discussed in Section 3 apply to the choice of a particular value of α.

Similarly to the pivoting-free case, fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The resulting algorithm has BSP cost asymptotically equal to the cost of the fast pivoting-free algorithm presented at the end of the previous section.

5 Conclusions

Parallel algorithm complexity is an area of active research, both from the theoretical and the practical perspective. Dozens of parallel cost models have been proposed and used to analyse thousands of algorithms from various fields. One of the common phenomena, observed both in theory and in practice, is the "parallelism-communication tradeoff": the finer the granularity of a parallel algorithm, the more communication it requires.

In this paper, we have considered the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We have defined a common framework that unifies both problems, and presented a new communication-efficient BSP algorithm for their solution. In fact, the communication cost of our algorithm is asymptotically optimal: the lower bound is provided by the communication cost of matrix multiplication. However, the improvement in communication efficiency (relative to previously known algorithms) is offset by a proportional reduction in synchronisation efficiency. Additionally, our method allows one to trade off the costs of communication and synchronisation in a certain range.

Apart from being a useful addition to the growing collection of efficient BSP algorithms, this result (as well as a similar earlier result on pivoting-free Gaussian elimination) can be viewed as a refinement of the classical "parallelism-communication tradeoff". One of the main strengths of the BSP model is that it allows an independent treatment of pure communication and synchronisation. Our analysis shows that, within a certain range of parameters, moving from coarser-grain to finer-grain computation may actually reduce the amount of pure communication. The tradeoff still exists, but only between parallelism and synchronisation. We have argued that for large problem sizes, the finer-grain, communication-efficient end of this tradeoff may be preferable to the coarser-grain, synchronisation-efficient one.

It remains an open question whether the optimal communication and synchronisation costs for Gaussian elimination (even without pivoting) can be achieved simultaneously.

References

1. A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.
2. J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 21(125):231–236, January 1974.

3. P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Number 315 in Grundlehren der mathematischen Wissenschaften. Springer, 1997.
4. R. Calinescu and D. J. Evans. Bulk-synchronous parallel algorithms for QR and QZ matrix factorisation. *Parallel Algorithms and Applications*, 11:97–112, 1997.
5. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.
6. M. Cosnard and E. M. Daoudi. Optimal algorithms for parallel Givens factorization on a coarse-grained PRAM. *Journal of the ACM*, 41(2):399–421, March 1994.
7. J. W. Demmel, N. J. Higham, and R. S. Schreiber. Block LU factorization. *Numerical Linear Algebra with Applications*, 2(2), 1995.
8. K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.
9. D. Irony and S. Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters*, 12:79–94, 2002.
10. W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1995.
11. W. F. McColl. A BSP realisation of Strassen's algorithm. In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 43–46. IOS Press, 1996.
12. W. F. McColl. Universal computing. In L. Bougé et al., editors, *Proceedings of Euro-Par (Part I)*, volume 1123 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 1996.
13. J. J. Modi. *Parallel Algorithms and Matrix Computation*. Oxford Applied Mathematics and Computing Science Series. Clarendon Press, 1988.
14. J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, 1988.
15. A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the ACM*, 25(1):81–91, January 1978.
16. A. Schönhage. Unitäre Transformationen großer Matrizen. *Numerische Mathematik*, 20:409–417, 1973.
17. D. C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, C–34(3):274–278, March 1985.
18. A. Tiskin. Bulk-synchronous parallel Gaussian elimination. *Journal of Mathematical Sciences*, 108(6):977–991, 2002.
19. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.