Записки научных
семинаров ПОМИ
Том 258, 1999 г.

A. Tiskin

BULK-SYNCHRONOUS PARALLEL GAUSSIAN ELIMINATION

Abstract. The model of bulk-synchronous parallel (BSP) computation is an emerging paradigm of general-purpose parallel computing. We study the BSP complexity of Gaussian elimination and related problems. First we analyse Gaussian elimination without pivoting, which can be applied to LU decomposition of symmetric positive definite or diagonally dominant real matrices. Then we analyse Gaussian elimination with Schönhage's recursive local pivoting, suitable for LU decomposition of matrices over a finite field, or QR decomposition of real matrices by Givens rotations. Both versions of Gaussian elimination can be performed in an optimal amount of local computation, but optimal communication and synchronisation costs may not be achievable simultaneously. The presented algorithms allow one to trade off communication and synchronisation costs in a certain range, achieving optimal or near-optimal cost values at the extremes.

1. Introduction

The model of bulk-synchronous parallel (BSP) computation (see [18, 11, 13]) 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. The key features of BSP are the treatment of the communication medium as an abstract fully connected network, and explicit and independent cost analysis of local computation, communication and synchronisation.

In this paper we study the BSP complexity of Gaussian elimination and related problems. Section 3 contains the description of an optimal BSP algorithm for matrix multiplication, due to McColl and Valiant. Section 4 describes another important BSP algorithm: computation of the cube dag. Section 5 presents a BSP algorithm for Gaussian elimination without pivoting. This method can be applied, for example, to LU decomposition of symmetric positive definite or diagonally dominant real matrices. Section 6 describes an extension of this algorithm to Gaussian elimination with local pivoting. This method, which is a parallelisation of the recursive pivoting procedure proposed by Shönhage, can be used for LU decomposition of matrices over a finite field, or QR decomposition of real matrices by Givens rotations. Both methods are based on multiplication of large matrix blocks by the McColl–Valiant algorithm. The analysis of the algorithms shows that they are optimal in local computation cost, but do not allow one to achieve optimal communication and synchronisation costs simultaneously. Instead, we can trade off the costs of communication and synchronisation in a certain range, achieving optimal or near-optimal cost values at the extremes of this range.

2. The BSP model

A *BSP computer*, introduced in [17–19], consists of p *processors* connected by a communication network (see Figure 1). Each processor has a fast *local memory*. The processors may follow different threads of computation. A BSP computation is a sequence of *supersteps* (see Figure 2). A superstep consists of an *input phase*, a *local computation phase* and an *output phase*. In the input phase a processor receives data that were sent to it in the previous superstep; in the output phase it can send data to other processors, to be received in the next superstep. The processors are synchronised between supersteps. The computation within a superstep is asynchronous.

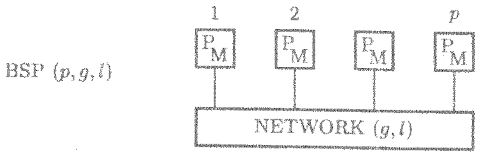

Fig. 1. A BSP computer.

The *cost unit* is the cost of performing a basic arithmetic operation or a local memory access. If for a particular superstep w is the maximum number of local operations performed by each processor, h' (respectively h'') is the maximum number of data units received (respectively sent) by each processor, and $h = h' + h''$, then the cost of the superstep is defined as $w + h \cdot g + l$. Here g and l are parameters of the computer. The value g is the *communication throughput ratio* (also called "bandwidth inefficiency" or "gap"), the value l is the *communication latency* (also called "synchronisation periodicity"). We write BSP (p, g, l) to denote a BSP computer with the given values of p, g and l. The values of w and

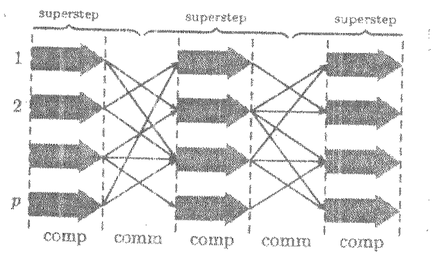

Fig. 2. A BSP computation.

h typically depend on the number of processors p and on the problem size. If a computation consists of S supersteps with costs $w_s + h_s \cdot g + l$, $1 \leqslant s \leqslant S$, then its total cost 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. (We will omit the factors g and l when dealing with communication and synchronisation separately from local computation.)

In order to utilise the computer resources efficiently, a typical BSP program regards the values p, g and l as configuration parameters. Algorithm design should aim to minimise local computation, communication and synchronisation costs for any realistic values of these parameters. For most problems, a balanced distribution of data and computation work will lead to algorithms that achieve optimal cost values simultaneously. However, for some other problems a need to trade off the costs will arise.

An example of a communication-synchronisation tradeoff is the problem of broadcasting a single value from a processor: it can be performed with $H = S = O(\log p)$ by a balanced binary tree, or with $H = O(p)$ and $S = O(1)$ by sending the value directly to every processor (this was observed in [18]). On the other hand, a technique known as *two-phase broadcast* allows one to achieve perfect balance for the problem of broadcasting $n \geqslant p$ values from one processor. By dividing the values into p blocks of size n/p, scattering the blocks so that each one gets to a distinct processor, and then performing total exchange of the blocks, the problem can be solved with $H = O(n)$ and $S = O(1)$ – this is obviously optimal.

The algorithms presented in this paper, as well as many other BSP algorithms, are defined for input sizes that are sufficiently large with respect to the number of processors. This requirement is loosely referred to as *slackness*.

For the sake of simplicity, we ignore small irregularities that arise from imperfect matching of integer parameters. For example, when we write "divide an array of size n 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 $A[i,j]$, $1 \leqslant i,j \leqslant n$. When the matrix is partitioned into regular blocks of size $m \times m$, we refer to each individual block as $A[[s,t]]$, $1 \leqslant s,t \leqslant n/m$.

3. Matrix multiplication

In this section we describe a BSP algorithm for one of the most common problems in scientific computation: dense matrix multiplication. We deal with the general problem of computing the matrix product $A \cdot B = C$, where A, B, C are $n \times n$ matrices over a semiring. We assume that the matrices are distributed across the processors evenly, but otherwise arbitrarily.

We aim to parallelise the standard $\Theta(n^3)$ method, asymptotically optimal for sequential matrix multiplication over a general semiring (see [9]). The method consists in straightforward computation of the family of bilinear forms

$$C[i,k] = \sum_{j=1}^{n} A[i,j] \cdot B[j,k] \qquad 1 \leqslant i,k \leqslant n \tag{1}$$

Following (1), we need to set

$$C[i,k] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,n \tag{2}$$

and then compute

$$V[i,j,k] \leftarrow A[i,j] \cdot B[j,k] \qquad C[i,k] \leftarrow C[i,k] + V[i,j,k] \tag{3}$$

for all i,j,k, $1 \leqslant i,j,k \leqslant n$. Computation (3) for different triples i,j,k is independent (although it requires concurrent reading from $A[i,j]$ and $B[j,k]$, and concurrent writing to $C[i,k]$), and therefore can be performed in parallel.

The McColl–Valiant algorithm implementing this method is described in [11, 13], and based on an idea from [1]. The algorithm works by a straightforward partitioning of the problem graph. Array V is represented as a cube of volume n^3 in integer three-dimensional space (see Figure 3). Arrays A, B, C are represented as projections of the cube V onto

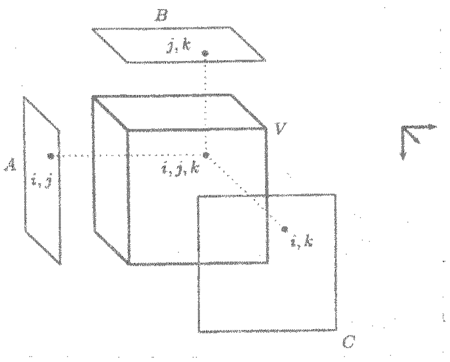

Fig. 3. Matrix multiplication graph .

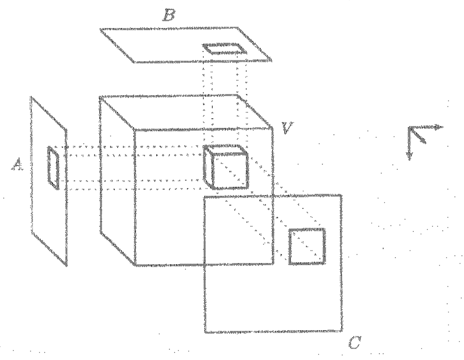

Fig. 4. Matrix multiplication in BSP.

the coordinate planes. Computation of the node $V[i,j,k]$ requires the input of its A and B projections $A[i,j]$ and $B[j,k]$, and the output of its C projection $C[i,k]$. In order to provide a communication-efficient BSP algorithm, the array V must be divided into p regular cubic blocks of size $n/p^{1/3}$ (see Figure 4). Such partitioning induces a partition of the matrices A, B and C into $p^{2/3}$ regular square blocks of size $n/p^{1/3}$,

$$A = \begin{pmatrix} A[[1,1]] & \cdots & A[[1,p^{1/3}]] \\ \vdots & \ddots & \vdots \\ A[[p^{1/3},1]] & \cdots & A[[p^{1/3},p^{1/3}]] \end{pmatrix} \tag{4}$$

and similarly for B and C (see Figure 4). Computation (2), (3) can be expressed in terms of blocks as

$$C[[i,k]] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,p^{1/3} \tag{5}$$

and then

$$V[[i,j,k]] \leftarrow A[[i,j]] \cdot B[[j,k]] \qquad C[[i,k]] \leftarrow C[[i,k]] + V[[i,j,k]] \tag{6}$$

for all i, j, k, $1 \leqslant i, j, k \leqslant p^{1/3}$. Each processor computes a block product $V[[i,j,k]] = A[[i,j]] \cdot B[[j,k]]$ sequentially by (2), (3). The algorithm is as follows.

Algorithm 1. *Matrix multiplication.*

Input: $n \times n$ matrices A and B over a semiring.
Output: $n \times n$ matrix $C = A \cdot B$.
Description. We assume $n \geq p^{1/3}$. After the initialisation step (5), the computation proceeds in one superstep. Each processor performs the computation (6) for a particular triple i, j, k. In the input phase, the processor receives the blocks $A[[i,j]]$ and $B[[j,k]]$. Then it computes the product $V[[i,j,k]] = A[[i,j]] \cdot B[[j,k]]$ by (2), (3). The block $V[[i,j,k]]$ is then written to $C[[i,k]]$ by sending it to the relevant processor. Concurrent writing is resolved by addition of the written blocks. The resulting matrix C is the product of A and B.
Cost analysis. The local computation, communication and synchronisation costs are

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

It can be shown that Algorithm 1 is an optimal BSP algorithm for matrix multiplication over a general semiring. The n^3 operations performed by the algorithm and their representation as a cube are essentially unique (see [9]). The lower bounds $W = \Omega(n^3/p)$ and $S = \Omega(1)$ are trivial. To prove the lower bound on H, we use the following fact, suggested for this proof by [16]. Let A be a finite set of points in $\mathbf{Z}^3$, and let A_1, A_2, A_3 be the orthogonal projections of A onto the coordinate planes. Then the *discrete Loomis–Whitney inequality* (see [10, 8, 2]) states that

$$|A| \leq |A_1|^{1/2} \cdot |A_2|^{1/2} \cdot |A_3|^{1/2} \tag{7}$$

Since n^3 nodes are computed by p processors, there is a processor q that computes at least n^3/p nodes. We apply the discrete Loomis–Whitney inequality, and then the arithmetic-geometric mean inequality, to the set of nodes computed by processor q. Since the size of this set is at least n^3/p, one of the three projections of the set must contain at least $n^2/p^{2/3}$ nodes, therefore $H = \Omega(n^2/p^{2/3})$.

Paper [12] generalises Algorithm 1 to a BSP implementation of any fast bilinear algorithm, such as Strassen's. The method works for matrices over a commutative ring with unit. The BSP cost of this algorithm is $W = O(n^\omega/p)$, $H = O(n^2/p^{2/\omega})$, $S = O(1)$, where ω is the exponent of the bilinear algorithm (the best current value is 2.376 by [4, 5]). For any particular bilinear algorithm these cost values are optimal.

4. Cube dag computation

The cube dag defines the dependence pattern that characterises a large number of scientific algorithms. Here we describe the BSP cube dag algorithm from [11]. For simplicity, we consider the computation of a three-dimensional cube dag; the algorithm for other dimensions is similar.

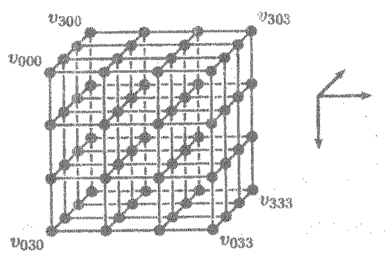

Fig. 5. Cube dag $\mathrm{cube}_3(4)$.

The three-dimensional cube dag $cube_3(n)$ with inputs $x^{(1)}_{jk}$, $x^{(2)}_{ik}$, $x^{(3)}_{ij}$, and outputs $y^{(1)}_{jk}$, $y^{(2)}_{ik}$, $y^{(3)}_{ij}$, $0 \leqslant i, j, k < n$, contains n^3 nodes v_{ijk}, such that

$$\begin{aligned}
&v_{0jk},\ v_{i0k},\ v_{ik0} \text{ take respectively } x^{(1)}_{jk},\ x^{(2)}_{ik},\ x^{(3)}_{ij}\\
&v_{ijk} \text{ contributes to each of the nodes}\\
&v_{i+1,j,k},\ v_{i,j+1,k},\ v_{i,j,k+1} \text{ whenever such node exists}\\
&v_{n-1,j,k},\ v_{i,n-1,k},\ v_{i,j,n-1} \text{ produce respectively } y^{(1)}_{jk},\ y^{(2)}_{ik},\ y^{(3)}_{ij}
\end{aligned} \tag{8}$$

Figure 5 shows the cube dag $\mathrm{cube}_3(4)$.

The BSP algorithm for computing the dag $\mathrm{cube}_3(n)$ is given in [11]. In this algorithm, the array $V = (v_{ijk})$ is partitioned into $p^{3/2}$ regular cubic blocks of volume $(n/p^{1/2})^3$. We denote these blocks by V_{ijk}, $0 \leqslant i, j, k < p^{1/2}$. Each block defines a dag isomorphic to $\mathrm{cube}_3(n/p^{1/2})$. The algorithm computes a block $V_{i+1,j+1,k+1}$ as soon as the data from its predecessors $V_{i,j+1,k+1}$, $V_{i+1,j,k+1}$ and $V_{i+1,j+1,k}$ become available. The independent blocks computed simultaneously form a "layer", or "wavefront" of the dag $cube_3(n)$.

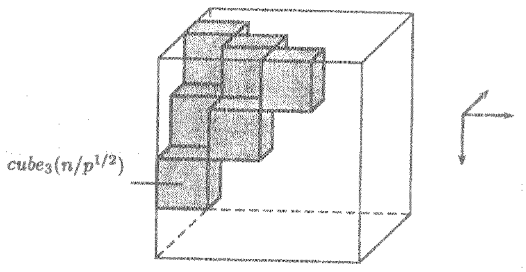

Fig. 6. BSP computation of $\mathrm{cube}_3(n)$.

Figure 6 shows a stage in the BSP computation of $\mathrm{cube}_3(n)$. The current wavefront is shaded. The total number of wavefronts is $3p^{1/2} - 2$, therefore the computation can be completed in $O(p^{1/2})$ supersteps.

Algorithm 2. *Computation of the cube dag $cube_3(n)$*

Input: arrays $\mathbf{x}^{(1)} = (x^{(1)}_{jk})$, $\mathbf{x}^{(2)} = (x^{(2)}_{ik})$, $\mathbf{x}^{(3)} = (x^{(3)}_{ij})$, $0 \leqslant i, j, k < n$.

Output: arrays $\mathbf{y}^{(1)} = (y^{(1)}_{jk})$, $\mathbf{y}^{(2)} = (y^{(2)}_{ik})$, $\mathbf{y}^{(3)} = (y^{(3)}_{ij})$, $0 \leqslant i, j, k < n$, defined by (8).

Description. We assume $n \geqslant p^{1/2}$. The computation proceeds in $3p^{1/2} - 2$ stages, each comprising a constant number of supersteps. In stage s, $0 \leqslant s < 3p^{1/2} - 3$, the blocks V_{ijk} with $i + j + k = s$ are computed. The maximum number of blocks computed in any one stage is $3p/4$.

Cost analysis. The local computation cost is $W = O(n^3/p)$. The computation of a block requires the communication of $O(n^2/p)$ values on the surface of the block. Therefore, the communication cost of each stage is $h = O(n^2/p)$. The total communication cost is $H = h \cdot p^{1/2} = O(n^2/p^{1/2})$. The synchronisation cost is $S = O(p^{1/2})$. •

The definition of the three-dimensional cube dag can be naturally generalised to any dimension. The two-dimensional dag $\mathrm{cube}_2(n)$ is called *diamond dag*. It can be computed by an algorithm similar to Algorithm 2 at the BSP cost $W = O(n^2/p)$, $H = O(n)$, $S = O(p)$. In general, the BSP cost of computing a cube dag $\mathrm{cube}_d(n)$ is $W = O(n^d/p)$, $H = O\left(n^{d-1}/p^{\frac{d-2}{d-1}}\right)$, $S = O\left(p^{\frac{1}{d-1}}\right)$.

5. Gaussian elimination without pivoting

This section describes a BSP approach to Gaussian elimination, a method primarily used for direct solution of linear systems of equations. In general, Gaussian elimination and its variations apply 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 of pivots. This basic form of elimination is not guaranteed to produce correct result, or terminate at all, when applied to 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. We will consider pivoting in Section 6.

Gaussian elimination can be described in many alternative ways. In this section we consider it as LU decomposition of a real matrix.

Let A be an $n \times n$ real diagonally dominant or symmetric positive definite matrix. The LU decomposition of A is $A = LU$, 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 many models of parallel computation. A BSP algorithm proposed in [11] works by reducing the problem to the computation of a cube dag $cube_3(n)$. The BSP cost of this computation is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$ (see Section 4). 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.

A lower communication cost for LU decomposition can be achieved by an alternative algorithm, based on recursive block Gauss–Jordan elimination (see [7, 3, 6]). This well-known method was suggested as a means of reducing the communication cost in [1] (for the transitive closure problem). Given a nonsingular matrix A, the algorithm produces the LU decomposition $A = LU$, together with the inverse matrices L^{-1} and U^{-1}. The algorithm works by dividing the matrices A, L, U into square blocks of size $n/2$,

$$A = LU: \qquad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} \tag{9}$$

where the dot $\cdot$ indicates a zero block. First we find 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}, by applying the algorithm recursively. Then we apply block Gauss–Jordan elimination to find the blocks L_{21} and U_{12}:

$$L_{21} \leftarrow A_{21} \cdot U_{11}^{-1} \qquad U_{12} \leftarrow L_{11}^{-1} \cdot A_{12} \tag{10}$$

A second recursive application of the algorithm yields the LU decomposition $A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$, and the inverse blocks L_{22}^{-1} and U_{22}^{-1}. We complete the computation by taking

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

Let us now describe the allocation of the block LU decomposition and block multiplication tasks in (10)–(11) to the BSP processors. We assume that the matrices are distributed across the processors evenly, but otherwise arbitrarily. Initially, all p processors are available to compute the LU decomposition $A = LU$. There is no substantial parallelism between block decomposition and block multiplication tasks in (10)–(11); we can only exploit the parallelism within block multiplication. Therefore, the recursion tree has to be computed in depth-first order. On each level of recursion every block multiplication in (10)–(11) is performed in parallel by all processors available at this level. Each block LU decomposition is also performed in parallel by all processors available at this level, 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. This variation allows 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, which controls the depth of parallel recursion. The algorithm is as follows.

Algorithm 3. *Gaussian elimination without pivoting.*

Parameter: real number α, $\alpha_{\min} = 1/2 \leqslant \alpha \leqslant 2/3 = \alpha_{\max}$.

Input: $n \times n$ real matrix A; we assume that A is diagonally dominant or symmetric positive definite.

Output: decomposition $A = LU$, where L is an $n \times n$ unit lower triangular matrix, and U is an $n \times n$ upper triangular matrix.

Description. We assume $n \geqslant p^{1/2}$. The computation is defined by recursion on the size of the matrix. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^{\alpha}$. The value n_0 is the threshold at which the algorithm switches from parallel to sequential computation.

On each step of recursion the matrix is divided into regular square blocks of size $m/2$ as shown in (9). Then, computation (10)–(11) is performed by the following schedule.

Small blocks. If $1 \leqslant m \leqslant n_0$, choose an arbitrary processor from all currently available, and compute (10)–(11) on this processor.

Large blocks. If $n_0 < m \leqslant n$, compute L_{11}, U_{11}, L_{11}^{-1}, U_{11}^{-1} by recursion. Then compute L_{21}, U_{12} and $A_{22} - L_{21} \cdot U_{12}$ by Algorithm 1. After that, compute L_{22}, U_{22}, L_{22}^{-1}, U_{22}^{-1} by recursion. Finally, compute $-L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1}$ and $-U_{11}^{-1} \cdot U_{12} \cdot U_{22}^{-1}$ by Algorithm 1. Each of these computations is performed with all available processors. The result is the decomposition (9).

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

$$W(m) = \begin{cases} 2 \cdot W(m/2) + O(m^3/p) & \text{if } n_0 < m \leqslant n \\ O(n_0^3) & \text{if } m = n_0 \end{cases}$$

$$H(m) = \begin{cases} 2 \cdot H(m/2) + O(m^2/p^{2/3}) & \text{if } n_0 < m \leqslant n \\ O(n_0^2) & \text{if } m = n_0 \end{cases}$$

$$S(m) = \begin{cases} 2 \cdot S(m/2) + O(1) & \text{if } n_0 < m \leqslant n \\ O(1) & \text{if } m = n_0 \end{cases}$$

which gives

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

For $\alpha = \alpha_{\min} = 1/2$ the cost of Algorithm 3 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 [11]. For $\alpha = \alpha_{\max} = 2/3$, the cost of Algorithm 3 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 (Algorithm 1). This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. For large n, the communication cost of Algorithm 3 dominates the synchronisation cost, and therefore the communication improvement should outweigh the loss of synchronisation efficiency. This justifies the use of Algorithm 3 with $\alpha = 2/3$. Smaller values of α, or the cube dag algorithm, should be considered when the problem is moderately sized.

6. Local pivoting and Givens rotations

In this section we extend the results of the previous section, obtaining an efficient BSP algorithm for certain forms of Gaussian elimination with pivoting.

Let A be an $n \times n$ matrix over a finite field. We assume that the inverse of a nonzero field element can be computed in time $O(1)$. Plain Gaussian elimination without pivoting may fail to find the LU decomposition of A, because some diagonal elements may be zero initially, or become zero during elimination. Block Gaussian elimination without pivoting may fail for a similar reason, if some diagonal blocks are singular or become singular during elimination. A particular feature of computation over a finite field is that any nonzero element, or any nonsingular block, can serve as a pivot. As we show below, this allows us to use a restricted version of pivoting, where pivot search is done locally in every block. We call this technique *local pivoting*, since the pivot search is done locally in a block. A similar approach applies to computation of the QR decomposition of a real matrix by Givens rotations.

To describe local pivoting, we consider Gaussian elimination on rectangular matrices of a special form. Let $\left(\begin{smallmatrix} A \\ V \end{smallmatrix}\right)$ be a $2n \times n$ matrix over a finite field. Here A is an arbitrary $n \times n$ matrix, and V an upper triangular $n \times n$ matrix. Matrices A, V may not have full rank. The problem consists in finding a full-rank $2n \times 2n$ transformation matrix $\left(\begin{smallmatrix} D & E \\ F & G \end{smallmatrix}\right)$, and an $n \times n$ upper triangular matrix U, such that

$$\begin{pmatrix} D & E \\ F & G \end{pmatrix} \begin{pmatrix} A \\ V \end{pmatrix} = \begin{pmatrix} U \\ \cdot \end{pmatrix}. \qquad (12)$$

This problem is closely related to the problem of transforming $\begin{pmatrix} A \\ V \end{pmatrix}$ to echelon form.

As in the previous section, the problem can be solved by the cube dag method, using the standard elimination scheme (see e.g. [14, 15]). Alternatively, we can compute the decomposition (12) by a recursive procedure. This recursive procedure differs from the one described in Section 5 in that we cannot compute block inverses. The matrices $\begin{pmatrix} D & E \\ F & G \end{pmatrix}$, $\begin{pmatrix} A \\ V \end{pmatrix}$, $\begin{pmatrix} U \\ \cdot \end{pmatrix}$ are partitioned into regular square blocks of size $n/2$,

$$\begin{pmatrix} D_{11} & D_{12} & E_{11} & E_{12} \\ D_{21} & D_{22} & E_{21} & E_{22} \\ F_{11} & F_{12} & G_{11} & G_{12} \\ F_{21} & F_{22} & G_{21} & G_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ V_{11} & V_{12} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \tag{13}$$

First the algorithm is applied recursively to $n \times n/2$ matrix $\begin{pmatrix} A_{21} \\ V_{11} \end{pmatrix}$. Taking the decomposition

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{21} \\ V_{11} \end{pmatrix} = \begin{pmatrix} Y_{11} \\ \cdot \end{pmatrix}$$

we obtain

$$\begin{pmatrix} I & \cdot & \cdot & \cdot \\ \cdot & P_{11} & P_{12} & \cdot \\ \cdot & P_{21} & P_{22} & \cdot \\ \cdot & \cdot & \cdot & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ V_{11} & V_{12} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ Y_{11} & Y_{12} \\ \cdot & Y_{22} \\ \cdot & V_{22} \end{pmatrix} \tag{14}$$

where

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{22} \\ V_{12} \end{pmatrix} = \begin{pmatrix} Y_{12} \\ Y_{22} \end{pmatrix}$$

In the next stage we apply the algorithm recursively to matrices $\begin{pmatrix} A_{11} \\ Y_{11} \end{pmatrix}$ and $\begin{pmatrix} Y_{22} \\ V_{22} \end{pmatrix}$, obtaining

$$\begin{pmatrix} Q_{11} & Q_{12} & \cdot & \cdot \\ Q_{21} & Q_{22} & \cdot & \cdot \\ \cdot & \cdot & Q_{33} & Q_{34} \\ \cdot & \cdot & Q_{43} & Q_{44} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ Y_{11} & Y_{12} \\ \cdot & Y_{22} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Z_{12} \\ \cdot & Z_{22} \\ \cdot & \cdot \end{pmatrix} \tag{15}$$

Finally, we apply the algorithm recursively to the matrix $\left(\begin{smallmatrix} Z_{12} \\ Z_{22} \end{smallmatrix}\right)$, obtaining

$$\begin{pmatrix} I & \cdot & \cdot & \cdot \\ \cdot & R_{11} & R_{12} & \cdot \\ \cdot & R_{21} & R_{22} & \cdot \\ \cdot & \cdot & \cdot & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Z_{12} \\ \cdot & Z_{22} \\ \cdot & \cdot \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \tag{16}$$

Matrix $\left(\begin{smallmatrix} D & E \\ F & G \end{smallmatrix}\right)$ can now be computed as the product of the three transformation matrices from (14)–(16).

As opposed to Gaussian elimination without pivoting, the recursive procedure (14)–(16) is not efficient when implemented in BSP, due to a large number of recursive calls. However, we can improve the efficiency by changing the order in which low-level block operations are executed. We represent the block elimination order defined by (14)–(16) schematically as

$$\begin{array}{c|c} 2 & 3 \\ 1 & 2 \end{array}$$

Here 1 denotes the elimination of V_{11}, 2 denotes the elimination of Y_{11} and V_{22}, and 3 denotes the elimination of Z_{22}. Vertical bars | represent block multiplication. In this schematic representation, a block elimination can be performed after the block immediately below has been eliminated, and block multiplication immediately to the left has been performed. A block multiplication can be performed after all blocks immediately to the left have been eliminated.

For 4×4 matrices, the elimination scheme of the recursive algorithm is

$$\begin{array}{c|c|c|c} 5 & 6 & 8 & 9 \\ 4 & 5 & 7 & 8 \\ 2 & 3 & 5 & 6 \\ 1 & 2 & 4 & 5 \end{array}$$

Here the length of the vertical bars corresponds to the size of matrices being multiplied. Note that the elimination 4 in the bottom row has to be computed after the matrix multiplication immediately to its left, and therefore after elimination 3. However, elimination 4 in the left column can be computed immediately after elimination 2, in parallel with elimination 3. In general, we can eliminate any block as soon as the result of multiplication immediately on its left is available. The optimised elimination scheme is

4	5	7	8
3	4	6	7
2	3	5	6
1	2	4	5

Similarly, for 8×8 matrices the original, recursive elimination scheme is

14	15	17	18	23	24	26	27
13	14	16	17	22	23	25	26
11	12	14	15	20	21	23	24
10	11	13	14	19	20	22	23
5	6	8	9	14	15	17	18
4	5	7	8	13	14	16	17
2	3	5	6	11	12	14	15
1	2	4	5	10	11	13	14

and the optimised scheme is

8	9	11	12	16	17	19	20
7	8	10	11	15	16	18	19
6	7	9	10	14	15	17	18
5	6	8	9	13	14	16	17
4	5	7	8	12	13	15	16
3	4	6	7	11	12	14	15
2	3	5	6	10	11	13	14
1	2	4	5	9	10	12	13

This method can be generalised to arbitrary matrix sizes. Decomposition of a $2n \times n$ matrix can be computed by an $r \times r$ optimised elimination scheme, where each entry corresponds to a block of size $n_0 = n/r$. We call such blocks *elementary*. Elimination of each elementary block is performed sequentially by a single BSP processor.

We now describe the allocation of the elementary block decomposition and matrix multiplication tasks to the BSP processors. The computation alternates between decomposition of elementary blocks and parallel multiplication of the resulting matrices. Each of these stages is implemented by a superstep. In each decomposition stage at most one entry from every column of the elimination scheme is computed. In each matrix multiplication stage at most one matrix multiplication from every column is performed. For any k, we allocate p/k processors to multiplication of matrices of size n/k.

The resulting algorithm is similar to Algorithm 3, in that a real parameter α controls the depth at which block decomposition is performed sequentially. We use an optimised elimination scheme of size $r = p^\alpha$. As before, variation of α results in a tradeoff between the communication and synchronisation costs.

Algorithm 4. *Gaussian elimination with local pivoting.*

Parameter: real number α,

$$\alpha_{\min} = \frac{1}{2} + \frac{\log\log p}{2\log p} \leqslant \alpha \leqslant \frac{2}{3} + \frac{\log\log p}{\log p} = \alpha_{\max}.$$

Input: $n \times n$ matrix A over a finite field.

Output: decomposition $DA = U$, where D is a full-rank $n \times n$ matrix, and U is an $n \times n$ upper triangular matrix.

Description. We assume $n \geqslant p^{1/2}$. We apply the optimised block elimination procedure to the matrix $\left(\begin{smallmatrix} A \\ 0 \end{smallmatrix}\right)$. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^\alpha$. The value n_0 is the size of elementary blocks.

We decompose the matrix using an optimised elimination scheme of size $r = p^\alpha$, as described above. Decomposition of elementary blocks is computed sequentially. Matrix multiplication is performed by Algorithm 1.

Cost analysis. The value for $S = S(n)$ can be found from the following recurrence relation:

$$S(m) = \begin{cases} 2 \cdot S(m/2) + O(1) & \text{if } n_0 < m \leqslant n \\ O(1) & \text{if } m = n_0 \end{cases}$$

which gives $S = O(p^\alpha \cdot \log p)$.

Since $r = p^\alpha \leq p$, all elementary block decompositions occurring in the same superstep can be performed in parallel. The total cost of elementary block decompositions is

$$\begin{aligned} W_0 &= S_0 \cdot O(n_0^3) = O(n^3 \cdot \log p/p^{2\alpha}) \\ H_0 &= S_0 \cdot O(n_0^2) = O(n^2 \cdot \log p/p^{\alpha}) \\ S_0 &= O(p^\alpha \cdot \log p) \end{aligned}$$

The total computation and communication cost of matrix multiplications is dominated by the cost of the largest matrix multiplication.

Indeed, in each matrix multiplication stage we compute at most one matrix product of size n, at most two matrix products of size $n/2$, etc. In general, for any k we compute at most k matrix products of size k. Each such product is computed on $p/(2k)$ processors. Let n/K be the size of the largest matrix product occuring in a particular matrix multiplication superstep. The computation cost of this superstep is at most

$$O\left(\frac{(n/K)^3}{p/K} \sqcup \frac{(n/(2K))^3}{p/(2K)} \sqcup \frac{(n/(4K))^3}{p/(4K)} \sqcup \ldots\right) = O\left(\frac{n^3}{K^2 \cdot p}\right)$$

and the synchronisation cost is at most

$$O\left(\frac{(n/K)^2}{(p/K)^{2/3}} \sqcup \frac{(n/2K)^2}{(p/2K)^{2/3}} \sqcup \frac{(n/4K)^2}{(p/4K)^{2/3}} \sqcup \ldots\right) = O\left(\frac{n^2}{K^{4/3} \cdot p^{2/3}}\right)$$

where $\sqcup$ denotes the maximum operator. There are at most two supersteps with $K = 1$, at most four supersteps with $K = 2$, etc. In general, there are at most $2K$ supersteps for any particular K. Therefore, the total cost of matrix multiplications is

$$W_1 = O\left(\frac{n^3}{p} + 2 \cdot \frac{n^3}{2^2 \cdot p} + 4 \cdot \frac{n^3}{4^2 \cdot p} + \ldots\right) = O(n^3/p)$$

$$H_1 = O\left(\frac{n^2}{p^{2/3}} + 2 \cdot \frac{n^2}{2^{4/3} \cdot p^{2/3}} + 4 \cdot \frac{n^2}{4^{4/3} \cdot p^{2/3}} + \ldots\right) = O(n^2/p^{2/3})$$

$$S_1 = O(p^\alpha \cdot \log p)$$

Since $W_0 = O(W_1)$, $H_1 = O(H_0)$, the total computation cost is

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

For $\alpha = \alpha_{\min}$ the cost of Algorithm 4 is $W = O(n^3/p)$, $H = O(n^2 \cdot (\log p)^{1/2}/p^{1/2})$, $S = O(p^{1/2} \cdot (\log p)^{3/2})$. This is slightly higher than the BSP cost of the cube dag method. For $\alpha = \alpha_{\max}$, the cost of Algorithm 4 is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(p^{2/3} \cdot (\log p)^2)$. In this case the communication cost is as low as in matrix multiplication (Algorithm 1) and Gaussian elimination without pivoting (Algorithm 3). This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. Considerations similar to the ones discussed in Section 5 apply to the choice of a particular value of α.

Since for $\alpha = \alpha_{\min}$ the cube dag method is slightly better than Algorithm 4, one might expect that a lower BSP cost may be achieved by a hybrid algorithm, performing elimination by an optimised scheme at the upper level, and decomposing elementary blocks by the cube dag algorithm at the lower level. This would be straightforward if Algorithm 4 were a pure recursive algorithm, similar to Algorithm 3. However, since Algorithm 4 computes the elimination scheme in an optimised non-recursive order, the problem of finding an efficient hybrid algorithm remains open.

Algorithm 4 can be used to compute the QR decomposition of a real matrix. For a 2×1 matrix $\binom{a}{v}$, the decomposition (12) is given by the Givens rotation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ v \end{pmatrix} = \begin{pmatrix} u \\ \cdot \end{pmatrix}$$

where $c = a/(a^2+v^2)^{1/2}$, $s = v/(a^2+v^2)^{1/2}$, $u = (a^2+v^2)^{1/2}$. Therefore, equations (14)–(16) and Algorithm 4 can be directly applied to computation of the QR decomposition of a real matrix by Givens rotations.

7. Conclusions

We have presented two BSP algorithms for Gaussian elimination: one without pivoting, another with a local pivoting scheme proposed by Schönhage. Both algorithms are more efficient in communication than the previously known BSP algorithms based on the cube dag approach. In fact, the communication cost of these algorithms is optimal under certain natural assumptions. The improvement in communication efficiency is offset by a reduction in synchronisation efficiency. The algorithms allow us to trade off the costs of communication and synchronisation in a certain range, achieving optimal or near-optimal cost values at the extremes of this range. We have argued that for large problem sizes, a communication-efficient algorithm may be preferable to a synchronisation-efficient one. It remains an open question whether the optimal communication and synchronisation costs for Gaussian elimination can be achieved simultaneously.

References

1. A. Aggarwal, A. K. Chandra, and M. Snir, *Communication complexity of PRAMs. Theoretical Computer Science*, **71** (1990), 3–28.
2. Yu. D. Burago and V. A. Zalgaller, *Geometric Inequalities.* Number 285 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1988.
3. J. Choi et al., *The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines.* Technical Report ORNL/TM-12470, 1994.
4. D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation*, **9(3)** (1990), 251–280.
5. D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions.* In E. Kaltofen, editor, *Computational Algebraic Complexity*, Academic Press (1990), 23–52.

6. J. W. Demmel, N. J. Higham, and R. S. Schreiber, *Block LU factorization. Numerical Linear Algebra with Applications,* **2(2)** *(1995).*
7. K. A. Gallivan, R. J. Plemmons, and A. H. Semeh, *Parallel algorithms for dense linear algebra computations. SIAM Review,* **32(1)** (1990), 54–135.
8. H. Hadwiger, *Vorlesungen über Inhalt, Oberfläche und Isoperimetrie.* Number 93 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1957.
9. J. E. Hopcroft and L. R. Kerr, *On minimizing the number of multiplications necessary for matrix multiplication. SIAM Journal of Applied Mathematics,* **20(1)** (1971), 30–36.
10. L. H. Loomis and H. Whitney, *An inequality related to the isoperimetric inequality. Bulletin of the AMS,* **55** (1949), 961–962.
11. W. F. McColl, *Scalable computing.* In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments,* **1000** *of Lecture Notes in Computer Science,* Springer-Verlag (1995), 46–61.
12. W. F. McColl, *A BSP realisation of Strassen's algorithm..* In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distribution Computing.* IOS Press, 1966.
13. W. F. McColl, *Universal computing.* In L. Bouǵe, P. Fraigniaud, A. Mignotte, and Y.Robert, editors, *Proceedings of Euro-Par'96-I,* **1123** of Lecture Notes in Computer Science, Springer-Verlag (1996), 25–36.
14. J. J. Modi, *Parallel Algorithms and Matrix Computation.* Oxford applied mathematics and computing science series. Clarendon Press, 1988.
15. J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems.* Frontires of Computer Science. Plenum Press, 1988.
16. M. S. Paterson, *Private communication.* 1993.
17. L. G. Valiant, *Bulk-synchronous computers. In M. Reeve, editor. Parallel Processing and Artificial Intelligence,* John Wiley & Sons, chapter 2 (1989), 15–22.
18. L. G. Valiant, *A bridging model for parallel computation. Communications of the ACM,* **33(8)** (1990), 103–111.
19. L. G. Valiant, *General purpose parallel architectures. In J. van Leeuwen, editor,. Handbook of Theoretical Computer Science,* chapter 18 (1990), 943–971.

Oxford University
Computing Laboratory
Wolfson Building,
Parks Rd., Oxford OX1 3QD, UK

Поступило 12 мая 1999 г.