TEMPUS: Activity 2

Parallel Algorithms, Chapter 2
Numerical and Matrix Algorithms

I. Vassányi, G. Dózsa, KFKI-MSZKI

20th July 1996

Contents

1 Introduction **3**
1.1 Notations and Terminology 4
1.2 Overview on the Chapter 4

2 Algorithms for General Dense Matrices **5**
2.1 Transposition 6
2.1.1 Block-Checkerboard on a Mesh 6
2.1.2 Block-Checkerboard on a Cube 7
2.2 Matrix-vector Multiplication 9
2.2.1 Row-wise Block Striping 9
2.2.2 Block-Checkerboard 9
2.3 Matrix Multiplication 10
2.3.1 Block-Checkerboard 10
2.3.2 Cannon's Algorithm on a Mesh 11
2.3.3 The DNS Algorithm on a 3-D Mesh or Cube 12
2.4 Matrix Inversion 13
2.5 Solving a Linear System and the LU Decomposition 14
2.5.1 Gaussian Elimination 14
2.5.2 Elimination with Pivoting 17
2.5.3 Back-Substitution 18
2.6 Inversion-based Solution of Linear Systems 18
2.6.1 Recursive LU Decomposition 19
2.6.2 Inversion-based Back-substituion 19

3 Algorithms for Special Matrices **21**
3.1 The Finite Element Method 21
3.2 Multiplication of a block-tridiagonal matrix by a vector 23
3.3 Jacobi Iteration for Solving Linear Systems 24
3.4 Matrix-vector Product for Toeplitz Matrices 26

4 Further Reading **27**

1 Introduction

This chapter attempts to give an overview on the most important parallel algorithms for the wide class of problems which are most naturally expressed as operations on matrices. Our goal is to reformulate the serial algorithms in order to reveal all the possible parallelism inherent in the problem, and then assess the resulting parallel algorithms on the basis of performance and optimality, with respect to various architectures and problem partitioning schemes, wherever they are relevant. We will focus on the practical engineering applicability of the algorithms presented, so excessive mathematical details and elaborate analyses will be omitted except for a few sample cases. The interested reader may wish to give a more precise mathematical formulation, for example, for the estimated performance metrics for some of the algorithms.

The parallel algorithms presented here vary greatly in the complexity of their serial components, that is, in their granularity. In some cases, the operations are rather simple (such as the transposition of a sub-matrix), and considering the communication costs, it may seem unwise to split the task among many processors. Also, we may seemingly overcomplicate a parallel algorithm just to get a small complexity bound improvement. However, these techniques are not only of theoretical importance only, since they do produce a significant speedup above a certain problem (matrix) size. Such very large matrices are often encountered in digital signal processing, modelling of physical systems and in industrial control. For instance, the control of some chemical reactors requires the real-time solution (with a time frame of some minutes) of linear systems of size 2000..3000.

When trying to find the best parallel algorithm for a certain problem on a certain target architecture, then the actual communication properties of the architecture (such as I/O bottlenecks, disk organization and optimal transfer block size variations) may outweigh the usual performance arguments, making general algorithm assessment and comparison very hard for such basic problems as matrix multiplication. In this respect, the algorithms of this chapter are to be taken just as a collection of useful ideas for parallelisation, which need to be revised and modified in a certain real-world environment.

In the following, we will often make such assumptions as, for example, $N = k\sqrt{P}; k = 1, 2, \ldots$, which is seldom the case in reality. However, these assumptions definitely simplify the presentation and analysis of the algo-

rithms, which are in turn usually simple to modify for a given environment.

In this chapter, we will assume that the reader is familiar with the mesh, linear array, and hypercube architectures and the CREW/EREW PRAM models as they are used in the previous chapters, along with their communication properties. The reader is also expected to understand the purpose and use of performance metrics and computational complexity bounds.

1.1 Notations and Terminology

If not stated otherwise, a matrix A will be assumed to be an $N \times N$ array of real numbers. The element in the i-th row and j-th column will be denoted by $a_{i,j}$, $a_{1,1}$ being the upper left element. That is, $i, j = 1, 2, \ldots, N$. Similarly, $PE_{i,j}$ means the Processing Element, or processor, in the i-th row and j-th column of the PE array in a processor mesh architecture. A column vector $\mathbf{b}$ is a one-dimensional array of elements $b_i, i = 1, \ldots, N$. The i-th column of a matrix A is $\mathbf{a_{*,i}}$, the i-th row is $\mathbf{a_{i,*}^T}$. We will use the symbol $I(k)$ for the $k \times k$ unity matrix, which has 1's in its main diagonal (i.e. in elements $i_{k,l}$, where $k = l$), and zeros in all other positions. In figures showing schematics of matrices, areas left blank represent zero elements, while shaded or *****-ed parts are nonzero elements.

Regarding computational complexity bounds, our basic assumption is that the four basic operations of $+$, $-$, $/$, and $*$ all take unit time on real numbers. This may not be true on some architectures, however assigning different execution times to all of them would overcomplicate the analysis. Again, the interested reader is encouraged to derive more elaborate complexity formulae. We will denote the runtime of the sequential algorithm by T_S and its parallel version by T_P. We will state that a parallel algorithm is *cost-optimal* or *optimal* if T_S is in the order of T_P multiplied by the number of the processors P, i.e. if $\Theta(T_S) = \Theta(P \cdot T_P)$, where $C = P \cdot T_P$ is the *cost* of the parallel algorithm.

1.2 Overview on the Chapter

This chapter divides matrix-related problems in the classes of general dense matrix algorithms and special matrix algorithms, the latter exploiting some special feature of the matrices, such as sparsity. In each of these sections, a

subsection is devoted for a certain problem, in some cases contatining sub-subsections for various architectures.

Problems are first stated and formulated, then a serial solution is given along with the run-time of the solution. Then the parallel algorithm and its analysis are presented.

2 Algorithms for General Dense Matrices

This section gives parallel solutions for transposition, multiplication and inversion of matrices, and it presents two distinct approaches for the solution of linear systems.

In order to share computational load, matrices must be partitioned (distributed) among processors. The most obvious matrix partitioning strategies are as follows:

- **Striping** We assign complete rows or columns to each PE (row-wise or column-wise striping), such that a PE holds N/P of them. If we assign a continuous section of rows or columns to the same PE, then we do *block partitioning*, however, there are other schemes which will produce better load distribution among PEs in some cases. Such is *cyclic partitioning*, where we assign rows or columns such that each PE should have one from each N/P-wide stripe of the matrix. This is illustrated in Figure 1.

  We may have a maximum of N PEs using striped partitioning.

- **Checkerboard** In this scheme we break rows and columns as well, and we assign
$$\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$$
elements of the matrix to each PE. Again, the elements can be assigned in a block or cyclic manner. This partitioning scheme is more natural, as it follows the original structure of the matrix. It allows the use of a maximum of N^2 PEs, however in practical cases $P \ll N$, so a block or cyclic assignment is used.

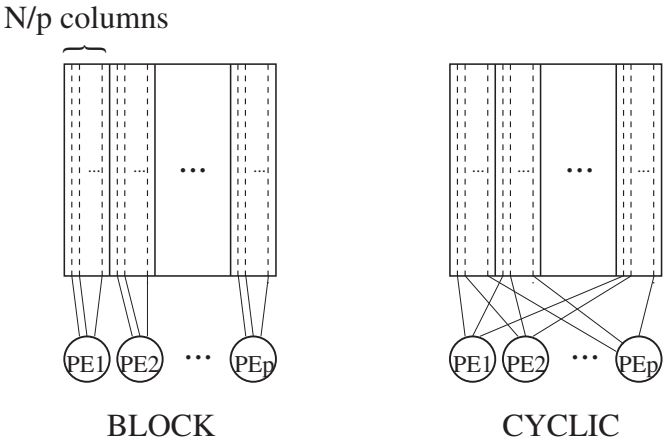

Figure 1: *Block and cyclic column-wise striping*

2.1 Transposition

The transposition problem is to mirror a matrix with respect to its main diagonal, i.e. $a_{i,j} \leftrightarrow a_{j,i}$. Clearly, the runtime of a trivial serial algorithm is $T_S = \Theta(N^2)$.

2.1.1 Block-Checkerboard on a Mesh

For this algorithm we assume that $N = k\sqrt{P}; k = 1, 2, \ldots$, and each PE holds a $N/\sqrt{P} \times N/\sqrt{P}$ submatrix of the original matrix.

Parallel Algorithm: First, we exchange nondiagonal blocks using nearest neighbour communication (Figure 2), then each PE transposes in parallel its locally stored block using the sequential algorithm.

Analysis: One block has N^2/P elements, and the lower left and upper right blocks must travel $2\sqrt{P}$ far, which gives

$$\Theta\left(\frac{N^2}{\sqrt{P}}\right)$$

time units for the block transfer, and then we need

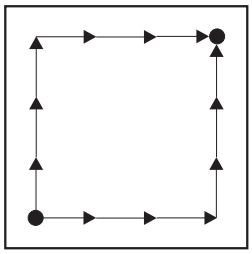

Figure 2: *Mesh transpose with neighbour communication*

$$\Theta\left(\frac{N^2}{P}\right)$$

time units for the local transpose. This means that the block transfer dominates the execution time. Multiplying it with P, we have

$$C = \Theta\left(N^2 \cdot \sqrt{P}\right),$$

which is more than the complexity of the sequential algorithm, so the parallel algorithm is not optimal.

2.1.2 Block-Checkerboard on a Cube

For this algorithm we assume that $P = 2^{2x}$ and $N = k\sqrt{P}; x, k = 1, 2, \ldots$, and we partition the matrix elements among the processors in a recursive way as illustrated by Figure 3 for $P = 16, x = 2$. In each step of this assignment, we divide each current submatrix into four sub-submatrices, and assign them to the four sub-subcubes of the current subcubes. We stop when the size of the current subcubes is 1.

Parallel Algorithm: We recursively swap the contents of the subcubes that hold non-diagonal submatrices. Following the assignment scheme of Figure 3, each such step takes one time unit if $N = \sqrt{P}$, so we need $2 \log P = x$ time steps altogether. Then each PE transposes its local submatrix using a serial algorithm.

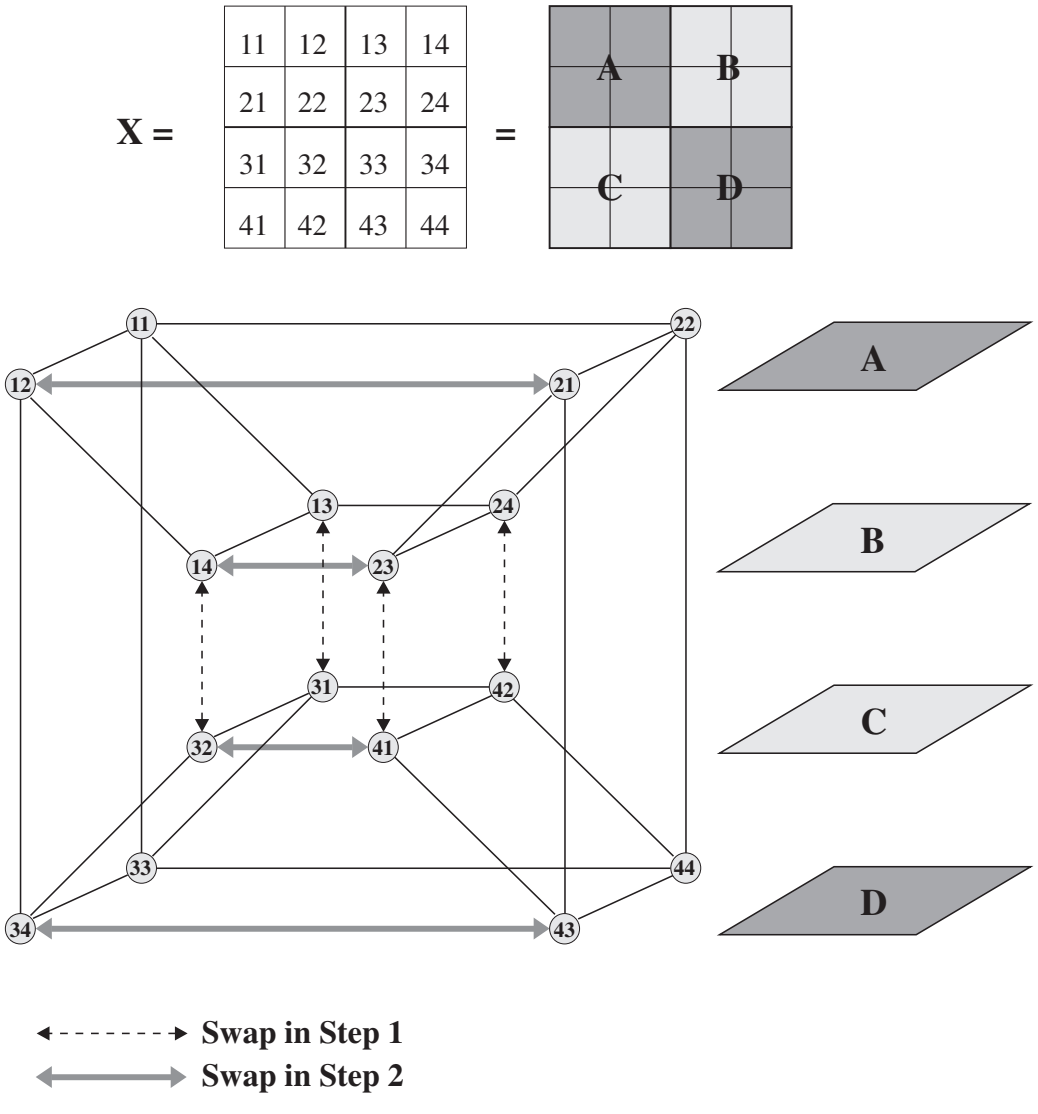

Figure 3: *Transpose on a 16-cube cube in two steps*

Analysis: The analysis is similar to that of the mesh architecture except that, due to the global connectivity of the hypercube architecture, we need only

$$\Theta\left(\frac{N^2}{P} \cdot \log P\right)$$

for the transfer of submatrix blocks, which improves the cost to

$$C = \Theta\left(N^2 \cdot \log P\right).$$

2.2 Matrix-vector Multiplication

This problem can be formulated as the computation of $A\mathbf{x} = \mathbf{y}$, where $y_i = \sum_{j=1}^{N} a_{i,j}x_j$, so for each element of $\mathbf{y}$ we need one row of A. It is clear that the complexity of the sequential algorithm is $T_S = \Theta(N^2)$.

2.2.1 Row-wise Block Striping

We assume that $N = kP, k = 1, 2, \ldots$ and we assign a chunk of neighbouring rows to each PE.

Parallel Algorithm: First all the elements of the vector $\mathbf{x}$ are loaded into all PEs. Then each PE computes in parallel those elements of $\mathbf{y}$ for which it holds the corresponding rows of A.

Analysis: We need $\Theta(N^2/P)$ time units for the local products, and we need $\Theta(N)$ time units for the global broadcast on a mesh or linear array, giving an optimal

$$C = \Theta(N^2).$$

2.2.2 Block-Checkerboard

We follow here the same block partitioning scheme as in 2.1.1 with $P = (N/k)^2$ processors.

Parallel Algorithm: The simple algorithm is shown in Figure 4, for a case of $P = N^2$. It contains two broadcast steps followed by a local product computation step and an accummulate-add step to produce the segments of the vector $\mathbf{y}$.

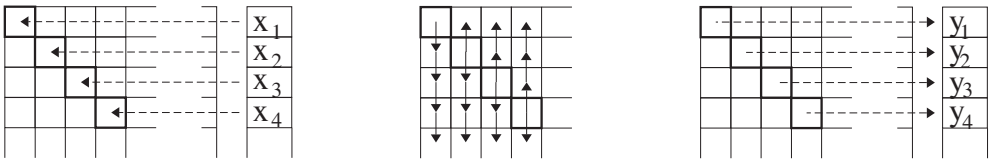

Figure 4: *Matrix-vector multiplication on a mesh*

Analysis: For the detailed analysis of this algorithm, the interested reader is referred to the bibliography. We only note here that although performance can be improved with this partititoning as compared to striping for a given number of PEs, the algorithm is not optimal.

2.3 Matrix Multiplication

This problem can be formulated as the computation of

$$A \cdot B = C, \quad where \quad c_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

, so for each element of $\mathbf{y}$ we need one row of A and one column of B. Again, it is easy to see that the complexity of the simple sequential algorithm is $T_S = \Theta(N^3)$, which will be denoted by $\Theta(M)$ in the following[1].

2.3.1 Block-Checkerboard

We follow here the same block partitioning scheme as in 2.1.1 with $P = (N/k)^2$ processors. This partitioning scheme is based on the concept of per-

[1]There exists a better computational bound for serial matrix multiplication, which is currently at about $M = N^{2.37}$.

forming the product of matrices as the sum of the products of their sub-blocks, as if the sub-blocks were normal matrix elements. For instance, if A and B are divided into four sub-matrices,

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} and\ B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix},$$

then

$$AB = \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix}.$$

Parallel Algorithm: Initially each PE stores a sub-block of both A and B. Then each PE broadcasts its locally stored A-block in its *row* and its locally stored B-block in its *column*. After this, each PE computes a sub-block of C using all the sub-blocks received, which make up whole rows and columns of A and B.

Analysis: We need $\Theta(N^3/P)$ time units for the local products, and as it can be checked, this will dominate the execution time on both the hypercube and mesh architectures, giving an optimal

$$C = \Theta(N^3).$$

2.3.2 Cannon's Algorithm on a Mesh

In the above algorithm, we can save a lot of memory in the PEs if we do not store all the blocks broadcast, but follow a pipeline strategy instead.

Parallel Algorithm: Initially each PE stores a sub-block of both A and B. Each PE multiplies its local A and B block and stores it in a partial result P. Then in $\sqrt{P}$ steps each PE sends its local A_{old} block to its left neighbour and its local B_{old} block to its upper neighbour, with wraparound at the edges of the PE array. Figure 5 shows the corresponding communication pattern for $\sqrt{P} = 4$. Then in each step each PE updates its partial result: $P = P + A_{new}B_{new}$. We have the blocks of the result with the same partitioning as used for the initial assignment.

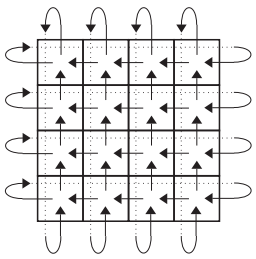

Figure 5: *Matrix multiplication on a wraparound mesh*

Analysis: The performance is the same as for the original algorithm in 2.3.1, but we need only $\Theta(3 \cdot N^2/P)$ words of memory in the PEs, significantly less than with the original algorithm.

Note: On architectures favouring one-to-all broadcast communication, a variant of the algorithm presented called the Fox algorithm can be used. The difference is that we broadcast a block in each step of the computation row-wise and we shift the position of the broadcasting PE after each step in each row. T_P can be improved this way by interleaving computation and pipelined communication.

2.3.3 The DNS Algorithm on a 3-D Mesh or Cube

This algorithm is the fastest one, and it is useful if we have enough processors to make up a 3-D array of reasonable size. We describe the algorithm for $P = N^3$ for simplicity; from this the block-partitioned version can be easily derived.

Parallel Algorithm: The idea is that we partition the 3-D multiplication index space (i, j, k) in a 3-D array of PEs. Initially the matrices A and B are partitioned in the plane $k = 1$ of the PE array in the usual way. Then the algorithm consists of four steps:

- **Distribution of A:** In this step each column of A is copied to a different plane of the array in two sub-steps, as shown in Figure 6. This step will place $a_{i,k}$ to $PE_{i,*,k}$.

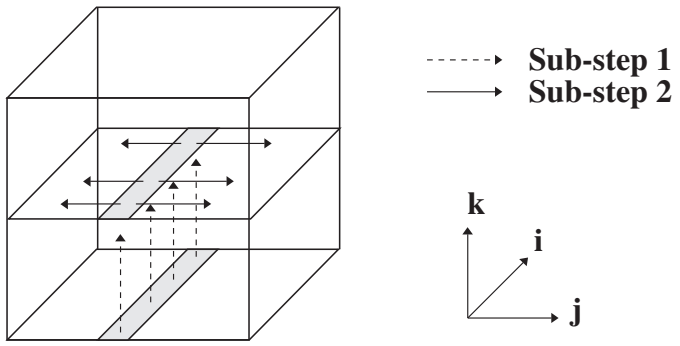

Figure 6: *Step 1 of matrix multiplication on a 3-D mesh*

- **Distribution of B:** In this step the rows of B are distributed in a similar manner. This step will place $b_{k,j}$ to $PE_{*,j,k}$.
- **Product computation:** Each PE computes the partial result $PR_{i,j,k} = a_{i,k} \cdot b_{k,j}$.
- **Addition:** The PEs of each column add their partial results $PR_{i,j,*}$ in parallel to produce the result $c_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$.

Analysis: On a hypercube (in which a mesh can be embedded easily) all the column/row broadcast and addition steps take only $T_P = \Theta(\log N)$ time, however the algorithm is not optimal ($C = \Theta(N^3 \cdot \log N)$).

2.4 Matrix Inversion

The problem is to find A^{-1} for a nonsingular matrix A such that $AA^{-1} = I(N)$. We show a very useful recursive divide-and-conquer strategy which uses matrix multiplication as a building block. Therefore, any of the algorithms in Section 2.3 can be used.

Parallel Algorithm: Suppose $N = 2^x$ and we have a matrix containing four $N/2 \times N/2$ sub-blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

Then it is easy to check that

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} -A_{11}^{-1} & 0 \\ 0 & X^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix},$$

where $X = A_{22} - A_{21}A_{11}^{-1}A_{12}$. We recursively compute A_{11}^{-1} and X^{-1}, which are both $N/2 \times N/2$ matrices.

Analysis: The runtime of the parallel algorithm is

$$T_P(N) = 2T_P(N/2) + \Theta(PM),$$

where PM is the bound of the parallel matrix multiplication algorithm used. With the fast DNS algorithm this evaluates to $T_P(N) = \Theta(\log^2 N)$. The cost of the algorithm depends on the actual implementation and architecture.

2.5 Solving a Linear System and the LU Decomposition

In this subsection, we present a traditional, elimination-based algorithm for the important problem of linear system solution. The problem is stated as $A\mathbf{x} = \mathbf{b}$, where A and $\mathbf{b}$ are known and we want to find the solution vector $\mathbf{x}$. The basic idea of the algorithm is to decompose the coefficient matrix, which is supposed to be nonsingular, into the product of a lower triangular matrix L and an upper triangular matrix U. Then the original equation can be written as

$$LU\mathbf{x} = \mathbf{b},$$

which is a new linear system with a lower triangular coefficient matrix L and a with new vector of unknowns $\mathbf{y} = U\mathbf{x}$. The reason for this approach is that it is very easy to solve linear systems with triangular matrices in a process known as back-substitution. Once we have solved $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$, we solve $U\mathbf{x} = \mathbf{y}$, which is also a triangular system, for $\mathbf{x}$.

2.5.1 Gaussian Elimination

In the algorithm presented below, we compute U and $\mathbf{y}$ in a single process in N steps. Starting from the upper left corner, the algorithm transforms the matrix such that after each step it has one more row starting with a section of all zeros (Figure 7).

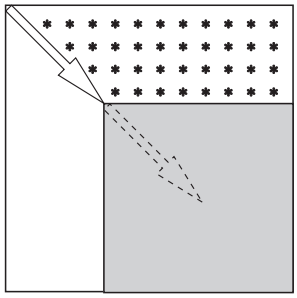

Figure 7: *Gaussian elimination of a dense matrix*

Serial Algorithm: In the k-th step there are all zeros in the first $k-1$ positions in all the rows below the k-th row. We suppose that $a_{k,k} \neq 0$, and divide the k-th row and b_k by $a_{k,k}$ (division step). Then, for all the rows below the k-th row ($row_i, k < i \leq N$) we substract from the row $\mathbf{a_i^T}$ and its matching b_i the appropriate multiple of the k-th row such that all $a_{i,k}$, where $k < i \leq N$ should become zero[2] (elimination step).

We proceed until $k = N$. Then in the last step only the division is necessary.

Connection of Elimination and LU Decomposition. In certain cases, when we have to solve the same linear system with a different $\mathbf{b}$ many times, we may wish to do only the LU decomposition alone, without the simultaneous solution of $L\mathbf{y} = \mathbf{b}$. Then we can compute the solution vector $\mathbf{x}$ for an arbitrary $\mathbf{b}$ in two back-substitution steps. In this case, we do not care about updating $\mathbf{b}$, and we use a slightly simplified version of the above algorithm:

- In the division step, we preserve the original value of the diagonal element and the elements before it in its row, which were produced by previous elimination steps. In other words, we divide by the diagonal element only the row segment *right* to the diagonal element.
- We perform the elimination step only in the columns *right* to the column of the diagonal element.

[2]This operation is very easy to understand if we remember that rows of the matrix represent equations which can be multiplied on both sides and added.

Then we can easily separate the resulting matrix A' into the L and U matrices we wanted to find, by simply inserting a diagonal of 1's in the U matrix (Figure 8). Since the computational steps for LU-decomposition are very

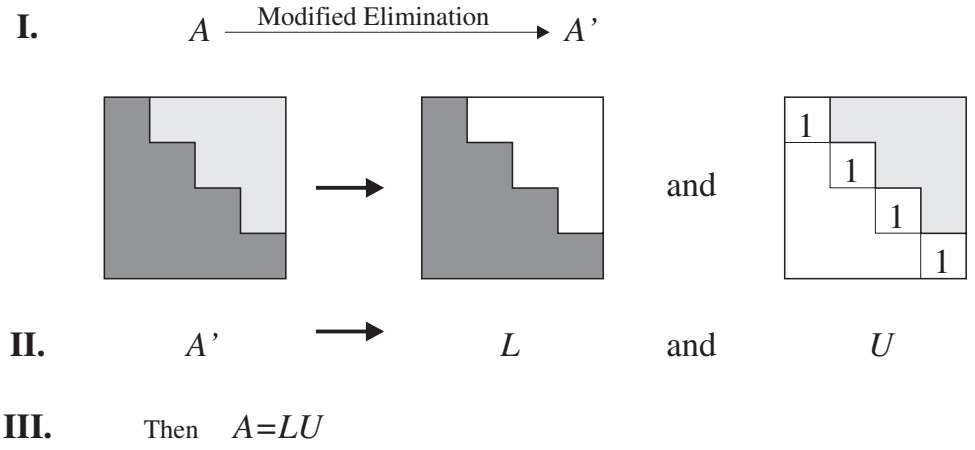

Figure 8: *How elimination solves the LU decomposition problem*

similar to those of Gaussian elimination, the same algorithm can be used for both as far as parallel implementation is concerned.

Since we need approximately $\Theta(N^2)$ elimination and division steps for each of the N diagonal element, the runtime of the serial algorithm presented is $T_S = \Theta(N^3)$.

Parallel Algorithm with Block-Striped Partitioning: We assign a row, or a chunk of rows in the case of block striped partitioning, to each PE. The algorithm is similar to the serial one. When the PE holding the active row finished the division step, it broadcasts the whole row to the PEs below it.

Analysis for a Hypercube: In each of the N steps we need to transfer $\Theta(N)$ elements, each taking $\Theta(\log N)$ time units on a cube. This gives $T_P = \Theta(N^2 \log N)$ and a non-optimal $C = \Theta(N^3 \log N)$ for $P = N$. The algoritm can be made optimal if the PEs operate in a pipeline i.e. they do not wait for the end of the current iteration. The same holds for the practical case when $P \ll N$.

Since the computation frontier moves downwards, we have more and more idle PEs towards the end of the iteration sequence. A better load distribution and overall runtime can be achieved with cyclic striping, in which case the overhead caused by PE idling is only $\Theta(N^2)$.

Parallel Algorithm with Block-Checkerboard Partitioning: With block-checkerboard partitioning we assign a $N/\sqrt{P} \times N\sqrt{P}$ sub-matrix to each PE, and execute the same algorithm with both row-wise and column-wise pipelining. This results in a computational frontier moving downwards in the direction of the main diagonal (Figure 9).

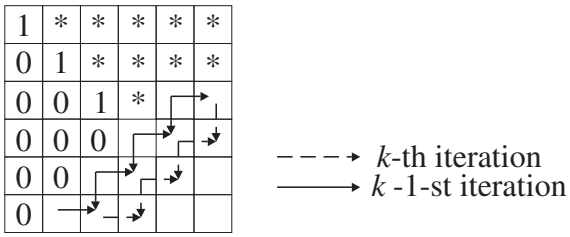

Figure 9: *Diagonal computation frontier in Gaussian elimination*

Analysis: Since each of the N movements of the frontier takes constant time, we have $C = \Theta(N^3)$, which can be improved by cyclic partitioning just like striped partitioning.

2.5.2 Elimination with Pivoting

The Gaussian elimination as described above assumes that $a_{k,k} \neq 0$, and even with this assumption we cannot guarantee the numerical stability of the algorithm if $|a| \approx 0$. A possible solution to this problem is that in each iteration step, we find the greatest absolute value element in the current row in the division step, and swap two columns to make it diagonal. To put it another way, we *pivot the row*[3]. The result of the elimination with pivoting will be an upper triangular matrix, but with permuted columns.

[3]This operation is equivalent to exchanging two unknowns of the vector x and as such it does not affect the values of the solution vector.

Serial Algorithm: For pivoting, the original serial algorithm needs two extra steps: pivot search and column exchange. If the exchange is only logical, we need to keep track of the exchanges by maintaining an index permutation vector.

Parallel Algorithm with Row-wise Cyclic Striped Partitioning: This partitioning scheme provides uniform load distribution and allows the logical (intra-PE) exchange of columns, so no physical data transfer is needed. The parallel algorithm is the same as without pivoting, with the two extra steps.

Columnwise striped partitioning is not feasible since it necessitates inter-PE communication for column exchange, and checkerboard partitioning is not feasible since pivoting seriously limits pipelineability.

2.5.3 Back-Substitution

The problem is to solve $U\mathbf{x} = \mathbf{y}$, where $u_{k,k} = 1$ and U is an upper triangular matrix.

Serial Algorithm: We start with the last equation $x_N = y_N$ and compute the other unknowns with the recursive formula

$$x_k = y_k - u_{k,k+1}x_{k+1} - \ldots - u_{k,N}x_N.$$

It is easy to see that the runtime of the serial algorithm is $T_S = \Theta(N^2)$.

Parallel Algorithm with Row-wise Block Striped Partitoning: We use a pipelined parallel form of the serial algorithm, in which each PE produces a chunk of the unknowns $x_1, \ldots, x_N$.

Analysis: Since each PE performs N/P multiplications and substractions and there are N steps, we have $T_P = \Theta(N^2/P)$ and $C = \Theta(N^2)$.

2.6 Inversion-based Solution of Linear Systems

In this section we show an alternative method for linear system solution which also first finds the LU decomposition of the matrix and then solves the resulting triangular system, but it uses a higher level approach in which

matrix multiplication and inversion are just building blocks. Therefore, no architectural analyses are presented, since they are dependent on the implemented multiplication and inversion algorithm. In both cases, we assume that N is a natural power of 2.

2.6.1 Recursive LU Decomposition

We aim to find the decomposition of A in the following form:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}.$$

Then, for the required, correct L and U matrices, the following equations hold:

$$A_{11} = L_{11}U_{11}; \quad A_{12} = L_{11}U_{12};$$

$$A_{21} = L_{21}U_{11}; \quad A_{22} = L_{21}U_{12} + L_{22}U_{22}.$$

First we compute A_{11}^{-1} (inversion of an $N/2 \times N/2$ matrix) and then we recursively compute the LU decomposition of A_{11} and $Y = A_{22} - A_{21}A_{11}^{-1}A_{12}$. The result of the former decomposition are the L_{11} and U_{11} matrices and the result of the latter are the L_{22} and U_{22} matrices, respectively.

In this way we have the submatrices L_{11}, L_{22}, U_{11}, and U_{22}. Then we set

$$L_{21} = A_{21}A_{11}^{-1}L_{11} \quad \textit{and} \quad U_{12} = U_{11}A_{11}^{-1}A_{12}.$$

This requires the multiplication of $N/2 \times N/2$ matrices.

Substituting back the expressions of L_{21}, U_{12}, A_{11} and Y into the required equations of the subblocks of A, it is easy to see that they hold, which means that the algorithm is correct.

With the best possible parallel runtime for the matrix multiplication and inversion primitives (see Subsection 2.4), the achievable parallel runtime of the algorithm is

$$T_P(N) = T_P(N/2) + \Theta(\log^2 N) = \Theta(\log^3 N).$$

2.6.2 Inversion-based Back-substituion

We solve the triangular system $L\mathbf{x} = \mathbf{y}$ by an inversion and a vector multiplication: $\mathbf{x} = L^{-1}\mathbf{y}$. Similarly to the general case of dense matrix inversion

(Subsection 2.4), it is easy to check that if

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}$$

, where both A_1 and A_3 are lower triangular matrices, then

$$A^{-1} = \begin{bmatrix} -A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix},$$

so we can recursively compute A_1^{-1} and A_3^{-1}. Then we need a parallel algorithm of Subsection 2.2, that best matches our target architecture, to compute $\mathbf{x}$.

3 Algorithms for Special Matrices

Matrices with special features often arise in various fields of scientific computing. Naturally, algorithms operating on these matrices need to exploit these features in order to be efficient.

It would be impossible to show all the important application fields with their specialized parallel algorithms within this section. Rather, we focus on the finite element method, which involves two types of special matrices; block-tridiagonal and Toeplitz-type matrices.

3.1 The Finite Element Method

This method is a very important tool in scientific computations, particularly in simulation of physical systems, where we want to solve a spatial differential (or other type) equation over an application-specific problem space. The requested solution of the differential equation would describe the variations of the domain variable (e.g. temperature in a heated metal block or concentration in a diluted liquid), in each point of the problem space. In many cases, the proper analytical solution of the differential equation is prohibitively expensive, but good estimates are valuable.

The main idea of the finite element method is to split the homogenous problem space into small regions called elements, usually via imposing a uniform grid, and to assume that the domain variable is the same within the whole area of any single element (element variable). This discretization turns the original spatial differential equation into as many difference equations as many elements we have. Each of these equations describes the dependence of an element variable on a *limited section* of neighbouring element variables, in most cases only on nearest neighbours. The solution of the linear system consisting of these equations provides an approximation of the solution of the original differential equation, and the accuracy of the approximation can be improved by downsizing the elements i.e. imposing a finer grid.

For simplicity, let us suppose that we have a square, two-dimensional problem space. Such an example is a heat sink sheet attached to a heavy-duty microelectronic component, and in this case we are interested in the steady-state temperature distribution of the sheet. With a $G \times G$ grid we have G^2 element variables. These are marked with circles in Figure 10. The simplest way of assembling these equations is the so-called lexicographical ordering,

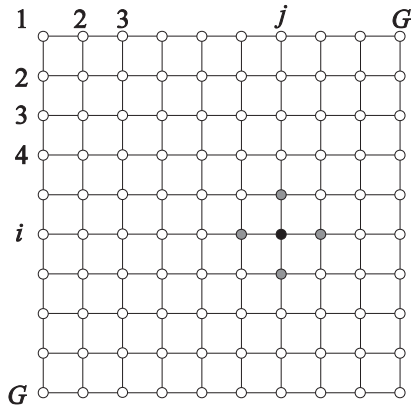

Figure 10: *A $G \times G$ grid over a square 2-D problem space*

by which we assign indices of the solution vector to element variables row-by-row. Figure 11 shows such an assignment if each element depends on its four nearest neighbours (which is the case in the usual approximation of heat distribution). As the figure shows, for an element (i, j) the corresponding index is $k = (i-1)G + j$. We can now formulate the original problem as a

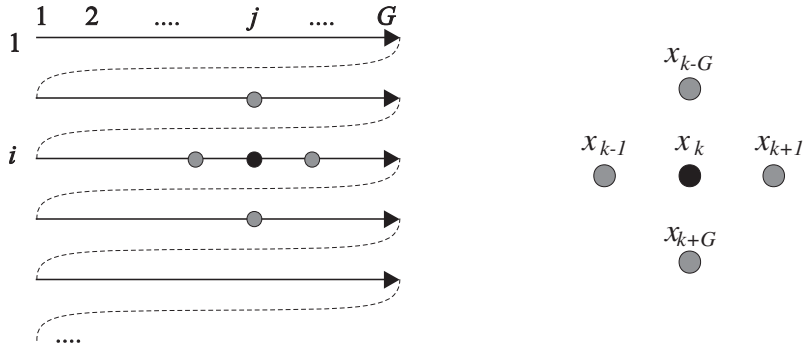

Figure 11: *Lexicographical ordering*

linear system $A\mathbf{x} = \mathbf{y}$, where A is a block-tridiagonal sparse matrix of size $G^2 \times G^2$, which has G pieces of $G \times G$ tridiagonal blocks in its principal diagonal plus an upper and a lower diagonal of nonzero elements. In the k-th row of the matrix the element in the principal diagonal is the coefficient of the k-th variable x_k, and the coefficients in the outer diagonals represent

the grid element above and below the grid element of x_k. The structure of the matrix is shown in Figure 12. There are many ways to solve systems

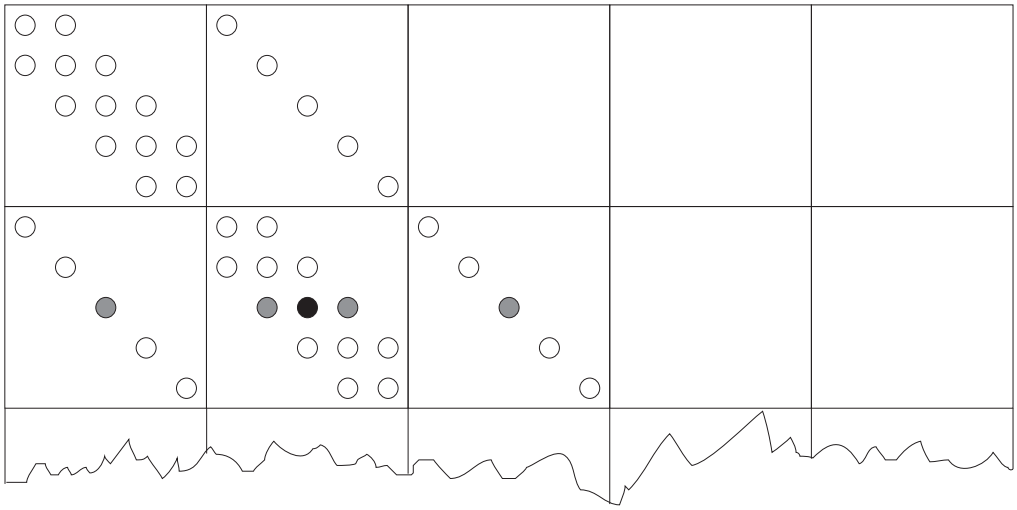

Figure 12: *Upper part of a block-tridiagonal matrix*

with block-tridiagonal matrices. In the subsections to follow we will show an iterative algorithm and its parallel implementation. For this we will need an efficient method for the multiplication of a block-tridiagonal matrix and a dense vector.

3.2 Multiplication of a block-tridiagonal matrix by a vector

The problem can be formulated as $A\mathbf{x} = \mathbf{y}$, where we wish to compute the vector $\mathbf{y}$. We suppose that the $G^2 \times G^2$ matrix is rowwise block-striped on P processors, each holding $l \geq G$ rows[4], where G is the number of elements in the grid (Figure 13). Moreover, each PE holds a matching segment of the vector $\mathbf{x}$.

Parallel Algorithm: In order to compute each element of $\mathbf{y}$, we need five elements of $\mathbf{x}$. For the indices that fall in the middle of the chunk,

[4]Only the nonzero elements of the matrix are stored, preferably in a diagonal format.

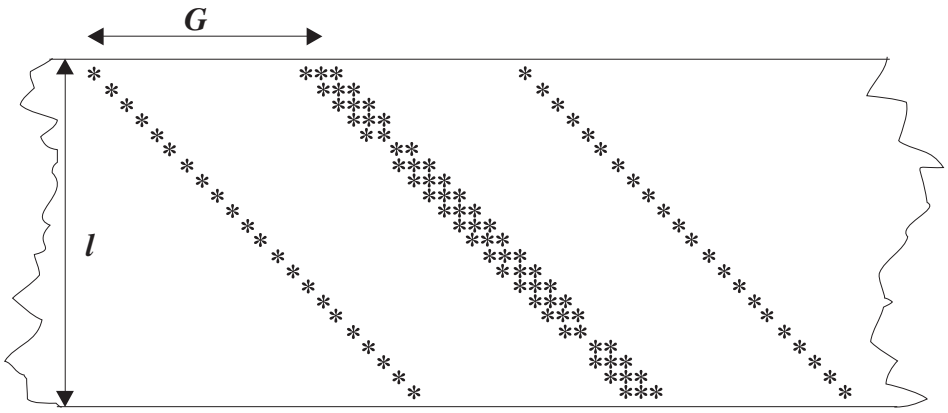

Figure 13: *A chunk of a block-tridiagonal matrix*

there is no need for inter-processor communication, but for the uppermost and lowermost G rows, elements of $\mathbf{x}$ must be received from the upper and lower neighbour of the PE, respectively. This can be accomplished by a simultaneous (not wrapped-around) up/down shift operation in the PE array.

Note: If $l < G$, then the elements of $\mathbf{x}$ required due to the outer diagonals will need to be fetched from farther PEs, which means longer execution time. More precisely, in this case $\lfloor G/l \rfloor$ shift steps are necessary.

3.3 Jacobi Iteration for Solving Linear Systems

Iterative equation solver methods attempt to improve an initial, random or estimated solution through a series of steps or *iterations*. If the difference between subsequent iterations keeps diminishing and eventually yields an approximate solution, then they are said to converge[5]. These methods differ from direct methods such as Gaussian elimination in that the number of iterations required for a certain precision, is data dependent, and the performance of the algorithm can only be assessed for the average case. Generally speaking, iterative methods are faster than direct methods, however we need to perform regular convergence checks.

[5]Even direct methods cannot give exact solutions on digital computers due to the final length variable precision.

The Jacobi iteration is one of the simplest ones, and it is well outperformed, as for the number of iterations, by other methods such as the SOR and the Conjugate Gradient methods, but it is quite simple and it is easy to parallelize since it does not use the partial result of an iteration before the end of the iteration.

Serial Algorithm: We formulate the problem as $A\mathbf{x} = \mathbf{b}$ and start with a randomly chosen initial solution vector $\mathbf{x}[0]$. Then, in the k-th iteration step, we

- Substep 1: Compute the elements of the vector $\mathbf{x}[k]$, indexed by $i = 1, 2, \ldots, N$, according to the formula

$$x_i[k] = \frac{r_i[k-1]}{a_{i,i}} + x_i[k-1]; \quad a_{i,i} \neq 0$$

- Substep 2: Compute the k-th *residual vector* using the formula[6]

$$\mathbf{r}[k] = \mathbf{b} - A\mathbf{x}[k].$$

The convergence check is performed after every few iteration steps via evaluating the norm (inner product) of the residual vector $\mathbf{r}[k]$. It should be decreasing, and if it is below a predefined threshold (precision), then the algorithm is finished.

Parallel Algorithm: We focus on the implementation of a single iteration step with block rowwise striping of both A, $\mathbf{x}$ and $\mathbf{b}$ (see Figure 13. The PEs also store their part of the result vector $\mathbf{x}[k-1]$ and the residual vector $\mathbf{r}[k-1]$ of the previous iteration. Then there is no need for inter-processor communication in Substep 1.

In Substep 2 we need a matrix-vector multiplication operation (see Subsection 3.2 for an implementation) and, in every few iterations, an inner product operation. The latter means simply to compute the square of each element of $\mathbf{r}[k]$ locally, and then add up the subsums in parallel. This takes $\Theta(\log N)$ time on a hypercube.

[6]These formulae are originated through simply rewriting the i-th equation $\sum_{i=1}^{N} a_{i,j} x_j = b_i$; for the details please refer to [1] or [4].

3.4 Matrix-vector Product for Toeplitz Matrices

We have seen in the previous subsections how block-tridiagonal matrices emerged in the finite element method in the simplest case, when we used an approximation of the original differential equation with only the four nearest neighbours. Due to the effect of nearest neighbours, we had an upper and lower diagonal along the principal diagonal, both having the same elements in their full length. The outer diagonals appeared because of the two-dimensional problem space, which also caused the interrupts in the near diagonals. In the more general case, we can consider also the effect of more remote finite elements by using a more complicated, not so localized approximation.

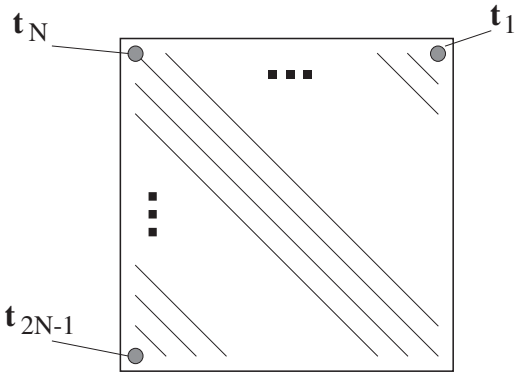

Figure 14: *The structure of a Toeplitz matrix*

Toeplitz matrices, which have diagonals with the same elements in all their length (Figure 14), emerge in the approximate solution of one-dimensional, constant coefficient differential equations. The special structure of these matrices allows their treatment as a vector $\mathbf{t}[2N-1]$ that defines the matrix, as shown in Figure 14.

The product computation problem is formulated as $T\mathbf{a} = \mathbf{d}$, and we wish to compute $\mathbf{d}$. Then it is easy to see that $\mathbf{d}$ is part of $\mathbf{a} \otimes \mathbf{t}$, the convolution of $\mathbf{a}$ and $\mathbf{t}$, which means that

$$d_k = \sum_{j=1}^{k} a_j t_{k-j}.$$

This is easy to understand if we notice that, due to the diagonally striped structure of T, each row in T is a right-shifted version of the previous one. The convolution of the N long vector $\mathbf{a}$ and the $2N-2$ long vector $\mathbf{t}$ is a $3N-3$ long series of numbers, and we select N out of these beginning at the index position $N-1$ to form the result vector $\mathbf{d}$.

The efficient computation of the convolution of two vectors is a core problem in digital signal processing, so the reader is referred to the relevant chapter in this course material. We only note here that the FFT algorithm can be used to perform the convolution of a length a and a length b vector in $\Theta(\log a + b)$ time on a hypercube, which gives a logarithmic running time for our problem as well.

4 Further Reading

This chapter tried to show some interesting fields of parallel methods for linear algebra, with an emphasis on simple, easy-to-follow presentation to make it suitable for all (undergraduate) levels of background knowledge. However, in some cases this approach might have led to the over-simplification of important questions that emerge in a real engineering implementaion. Also, a lot of interesting and important application fields had to be omitted due to the limited space. For those readers who wish to gain more in-depth knowledge of the state-of-art of the topics presented, or wish to extend the scope of the material, we recommend the following summarizing textbooks, which also served as the basic sources for the writing of this chapter.

- [1] Rózsa Pál: *Lineáris algebra és alkalmazásai*, Tankönyvkiadó, Budapest, 1991.
- [2] J. Jájá: *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- [3] S. Y. Kung: *VLSI Array Processors*, Prentice Hall, 1992.
- [4] V. Kumar et al.: *Introduction to parallel computing*, Benjamin/Cummings Publishing Co., 1994.
- [5] S. G. Akl: *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.