8. *Parallel Methods for Matrix Multiplication*

Matrix multiplication is one of the essential problems in matrix calculations. This Section discusses several parallel algorithms for carrying out the operation. Two of them are based on block-striped data decomposition scheme. The other two methods are based on checkerboard block scheme decomposition. They are the well known the Fox algorithm and the Cannon method.

This Section has been written based essentially on the teaching materials given in Quinn (2004).

8.1. Problem Statement

Multiplying an $m \times n$ matrix *A* with *m* rows and *n* columns and an $n \times l$ matrix *B* with *n* rows and *l* columns produces an $m \times l$ matrix *C* with *m* rows and *l* columns. Each element of the matrix *C* is calculated according to the formula:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l. \tag{8.1}$$

As it can be seen in (8.1), each element of the matrix *C* is the result of the inner product of the corresponding row of the matrix *A* and column of the matrix *B*:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0}, a_{i1}, ..., a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, ..., b_{n-1j}\right)^T. \tag{8.2}$$

This algorithm executes $m \cdot n \cdot l$ multiplications and the same number of additions of the initial matrix elements. In case of square matrices, the size of which is $n \times n$, the number of the executed operations is the order $O(n^3)$. There are also sequential matrix multiplication algorithms of smaller computational complexity (for instance, the Strassen algorithm). But studying these algorithms though requires certain efforts and for simplicity we will use the above described sequential algorithm as the basis for parallel method development in this section. We will also assume further that all matrices are square and their sizes are $n \times n$.

8.2. Sequential Algorithm

The sequential matrix multiplication algorithm includes three nested loops:

```
// Algorithms 8.2
// Sequential matrix multiplication algorithm
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Algorithm 8.1. Sequential matrix multiplication algorithm

This algorithm is an iterative procedure and calculates sequentially the rows of the matrix *C*. In fact, a result matrix row is computed per outer loop (loop variable i) iteration (see Figure 8.1)

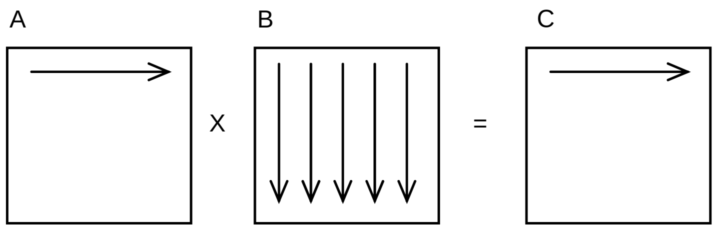

Figure 8.1. During the first iteration of loop variable *i* the first matrix *A* row and all the columns of matrix *B* are used to compute the elements of the first result matrix *C* row

As each result matrix element is a scalar product of the initial matrix *A* row and the initial matrix *B* column, it is necessary to carry out $n^2(2n-1)$ operations to compute all elements of the matrix *C*. As a result the time complexity of matrix multiplication is

$$T_1 = n^2 \cdot (2n-1) \cdot \tau \qquad (8.3)$$

where τ is the execution time for an elementary computational operation such as multiplication or addition.

8.3. Matrix Multiplication in Case of Block-Striped Data Decomposition

Let us consider two parallel matrix multiplication algorithms. Matrices *A* and *B* are partitioned into continuous sequences of rows or columns (*stripes*).

8.3.1. Computation Decomposition

As it is clear from the definition of matrix multiplication, all elements of the matrix *C* may be computed independently. As a result, a possible approach for parallelizing the matrix multiplication is to define the basic computational subtask as the problem of computing an element of the result matrix *C*. To carry out all the necessary computations each subtask must contain a row of the matrix *A* and a column of the matrix *B*. The total number of subtasks in case of this approach appears to be equal to n^2 (according to the number of elements of the matrix *C*).

One may note that the level of parallelism achieved in this approach is somewhat excessive. As a rule, in carrying out practical computations the number of the subtasks formed exceeds the number of the available processors. As a result, the aggregation stage of basic subtasks becomes inevitable. In this respect it is reasonable to aggregate the computations at the stage of selecting the basic subtasks. A possible solution is to combine all the computations related not with one, but with several elements of the result matrix *C* in a single subtask. For further discussion we will define the basic computational subtask as the problem of computing all row elements of the matrix *C*. This approach decreases the total number of subtasks up to value *n*.

A row of the matrix *A* and all the columns of the matrix *B* must be available for carrying out all the necessary computations of the basic subtasks. The simple solution to the problem is duplicating the matrix *B* in all the subtasks, but it is unacceptable because of sizeable memory expenses needed for data storage. As a result, computations should be implemented so that subtasks contain only a part of the data needed for the computations at any given moment. The access to the other part of the data should be provided by means of data communications. Two possible ways to carry out parallel computations of this type are considered in 8.3.2.

8.3.2. Analysis of Information Dependencies

To compute a row of the matrix *C* each subtask must have a row of the matrix *A* and access to all columns of the matrix *B*. Possible ways to organize parallel computations are described below.

1. The first algorithm. The algorithm is an iterative procedure, the number of iterations is equal to the number of subtasks. Each subtask holds a row of the matrix *A* and a column of the matrix *B* at each algorithm iteration. At each iteration the scalar products of rows and columns containing in the subtasks are computed, and the corresponding elements of the result matrix *C* are obtained. After completing of all iteration computations the columns of matrix *B* must be transmitted so that subtasks should have new columns of the matrix *B* and new elements of the matrix *C* could be calculated. This transmission of columns among the subtasks must be executed in such a way that all the columns of matrix *B* should have appeared in each subtask sequentially.

A possible simple scheme to provide the required communications of the columns of matrix *B* among the subtasks is to present the topology of the information dependencies of the subtasks as a ring structure. In this case the subtask i, $0 \le i < n$, will transmit its column of matrix *B* to the subtask $i+1$ at each iteration (in accordance with the ring structure subtask $n-1$ transmits its data to the subtask 0) – see Figure 8.2. After the algorithm termination the required condition will be provided, i.e. all the columns of matrix *B* will appear sequentially in each subtask.

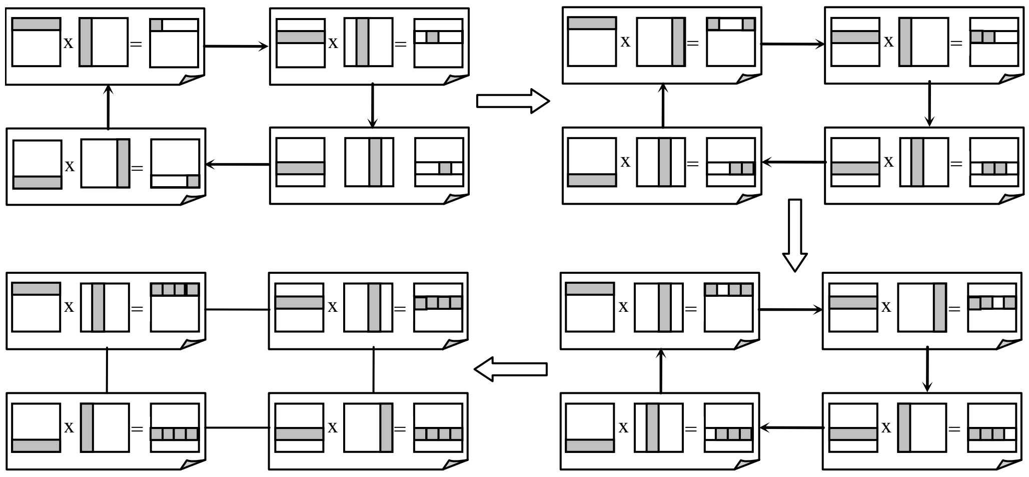

Figure 8.2. General scheme of data communications for the first parallel algorithm of matrix multiplication in case of block-striped decomposition

Figure 8.2 presents the iterations of the matrix multiplication algorithm for the case when matrices have four rows and four columns (n=4). At the beginning of the computations each subtask i, $0 \le i < n$, holds i-th row of the matrix *A* and i-th column of the matrix *B*. As a result the subtask i can compute the element c_{ii} of the result matrix *C*. Further each subtask transmits its column of matrix *B* to the following subtask in accordance with the ring structure. These actions should be repeated until all the iterations of the parallel algorithm are completed.

2. The second algorithm. The difference of the second algorithm from the first one is that the subtasks contain not columns but rows of matrix *B*. As a result, data multiplication of each subtask is the multiplication of the row elements of the matrix *B* by a corresponding row element of the matrix *A*. Therefore, a row of partial results for matrix *C* is obtained in each subtask.

In case of this scheme of data decomposition for matrix multiplication, it is necessary to provide sequential obtaining all rows of the matrix *B* by all in the subtasks, the multiplication of the row elements of the matrix *B* by a corresponding row element of the matrix *A* and summation of the new values and the previously computed ones. The ring structure of information dependencies may be also used to provide the necessary sequence of communications of the rows of the matrix *B* among the subtasks (see Figure 8.3).

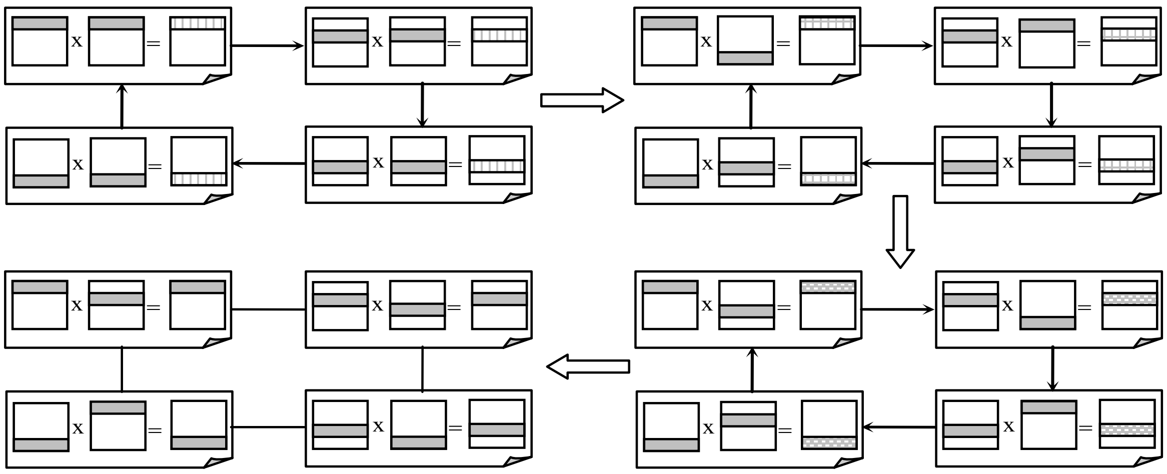

Figure 8.3. General scheme of data communications for the second parallel algorithm of matrix multiplication in case of block-striped decomposition

Figure 8.3 presents the iterations of the matrix multiplication algorithm in the case when matrices have 4 rows and 4 columns (n=4). At the beginning of the computations each subtask i, $0\le i<n$, holds i-th rows of the matrix A and the matrix B. As a result of multiplication the subtask defines i-th row of the partial results for the matrix C. Then each subtask transmits its row of the matrix B to the following subtask according to the ring structure of information dependencies. The described actions are repeated until all the iterations of the parallel algorithm are completed.

8.3.3. Scaling and Distributing Subtasks among the Processors

As a result of proper data decomposition the basic subtasks have the same computational complexity and the same intensity of the data communications. If the size of matrices n appears to be greater than the number of processors p, the basic subtasks may be aggregated by combining in one subtask several neighboring rows and columns of the multiplied matrices. In this case the initial matrix A is partitioned into a number of horizontal stripes, and matrix B is presented as a set of vertical (for the first algorithm) or horizontal (for the second algorithm) stripes. The stripe size should be equal to $k=n/p$ (assuming that n is divisible by p), as it will make possible to provide equal distribution of the computational load among the processors.

Any method of subtask distribution among the processors may be used if it efficiently presents the ring topology of the subtask communications. It might be sufficient, for instance, for the neighboring subtasks in the ring topology to be assigned to the processors linked by direct data communication lines.

8.3.4. Efficiency Analysis

Let us estimate the efficiency of the first matrix multiplication parallel algorithm.

The total time complexity of the sequential algorithm, as it has been stated earlier, is proportional to n^3. In case of the parallel algorithm each processor multiplies the stripes of the matrix A and the matrix B at each iteration (the stripe size is equal to n/p and, as a result, the total number of the multiplication operations performed is equal to n^3/p^2). As the number of the algorithm iterations is the same as the number of processors the complexity of the parallel algorithm, with no account for data communication, may be evaluated by means of the following expression:

$$T_p = (n^3/p^2)\cdot p = n^3/p\,. \tag{8.4}$$

With regard to this estimation, the speedup and efficiency of the given parallel algorithm of matrix multiplication look as follows:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{and} \quad E_p = \frac{n^3}{p\cdot\left(n^3/p\right)} = 1\,. \tag{8.5}$$

Thus, the general efficiency analysis gives ideal characteristics of the parallel computation efficiency. To specify the obtained relations we should estimate more precisely the number of computational operations of the algorithm and take into account the overhead of data communications among the processors.

With regards to the number and the duration of the operations the time for carrying out the computations for parallel algorithm may be estimated as follows:

$$T_p\left(calc\right) = (n^2/p)\cdot\left(2n-1\right)\cdot\tau \tag{8.6}$$

(where, as previously, τ is the execution time of an basic computational operation).

For the purpose of estimating the communication complexity of parallel computations we will assume that all data communication operations among the processors in the course of an algorithm iteration may be executed in parallel. The amount of the data transmitted among the processors is determined by the stripe size and is equal to *n/p* rows or columns of size *n*. The total number of parallel data communication operations is equal to the number of algorithm iterations minus one (at the last iteration data communication is not compulsory). Thus, the time complexity estimation for the data communication operations performed may be evaluated as:

$$T_p(comm) = (p-1)\cdot(\alpha + w\cdot n\cdot(n/p)/\beta), \tag{8.7}$$

where α is the latency, β is the network bandwidth, and w is the size of the matrix element in bytes.

With regard to the relations obtained the total execution time for the parallel algorithm of matrix multiplication can be estimated by the following expression:

$$T_p = (n^2/p)(2n-1)\cdot\tau + (p-1)\cdot(\alpha + w\cdot n\cdot(n/p)/\beta). \tag{8.8}$$

8.3.5. Computational Experiment Results

The computational experiments for estimating the efficiency of the first parallel algorithm of matrix multiplication in case of block-striped data decomposition scheme were carried out under the conditions given in 7.6.5.

The experiments were carried out on the computational cluster on the basis of processors Intel XEON 4 EM64T, 3000 Mhz and Gigabit Ethernet under OS Microsoft Windows Server 2003 Standard x64 Edition.

For estimating the time τ of the basic computational operation execution the problem of matrix multiplication was solved by the use of the sequential algorithm. The calculation execution time obtained in this way was divided into the total number of the operations performed. As a result of the experiments, τ was found to be equal to 6.4 nsec. The experiments performed in order to get the data communication network parameters gave the values of the latency α and the bandwidth β 47 msec and 53.29 Mbyte/sec correspondingly. All computations were performed over the numerical values of the double type, i.e. the value w is equal to 8 bytes.

The results of the computational experiments are shown in Table 8.1. The experiments were performed with the use of 2, 4 and 8 processors.

Table 8.1. The results of the computational experiments for the first parallel algorithm of matrix multiplication based on the block-striped data decomposition

Matrix Size	Serial Algorithm	2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,8752	0,3758	2,3287	0,1535	5,6982	0,0968	9,0371
1000	12,8787	5,4427	2,3662	2,2628	5,6912	0,6998	18,4014
1500	43,4731	20,9503	2,0750	11,0804	3,9234	5,1766	8,3978
2000	103,0561	45,7436	2,2529	21,6001	4,7710	9,4127	10,9485
2500	201,2915	99,5097	2,0228	56,9203	3,5363	18,3303	10,9813
3000	347,8434	171,9232	2,0232	111,9642	3,1067	45,5482	7,6368

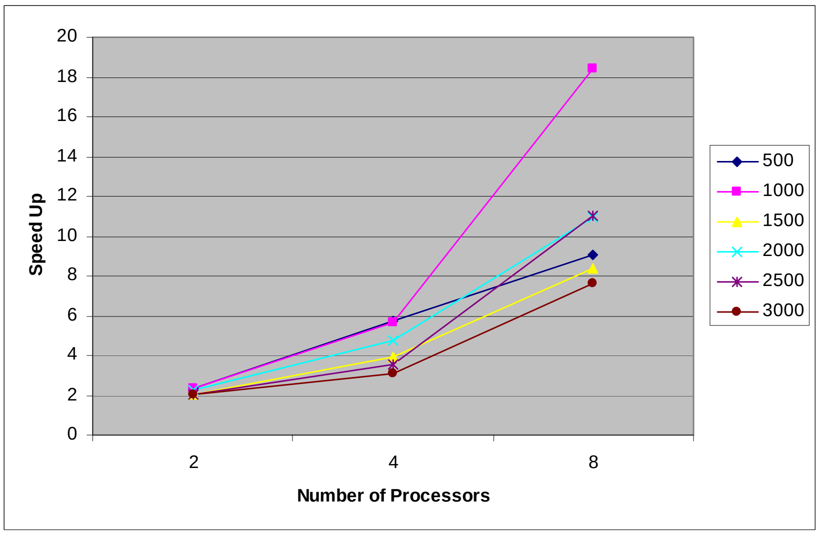

Figure 8.4. Speedup for the first parallel algorithm of matrix multiplication (block-striped matrix decomposition)

The comparison of the experimental execution time T_p^* and the theoretical time T_p from expression (8.8) is given in Table 8.2 and in Figure 8.5.

Table 8.2. The comparison of the experimental and theoretical execution time of the first matrix multiplication parallel algorithm based on the block-striped data decomposition

Matrix Size	2 processors		4 processors		8 processors	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	0,8243	0,3758	0,4313	0,1535	0,2353	0,0968
1000	6,51822	5,4427	3,3349	2,2628	1,7436	0,6998
1500	21,9137	20,9503	11,1270	11,0804	5,7340	5,1766
2000	51,8429	45,7436	26,2236	21,6001	13,4144	9,4127
2500	101,1377	99,5097	51,0408	56,9203	25,9928	18,3303
3000	174,6301	171,9232	87,9946	111,9642	44,6772	45,5482

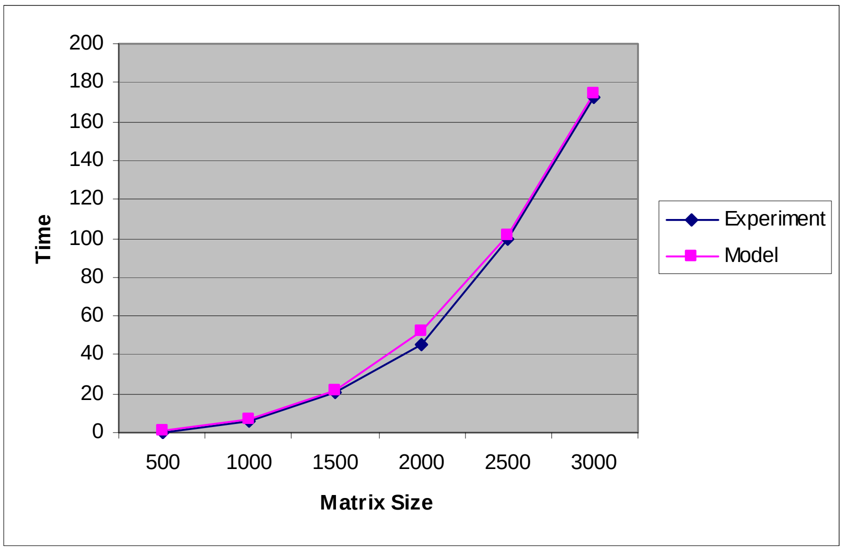

Figure 8.5. Theoretical and experimental execution time with respect to matrix size (block-striped matrix decomposition, 2 processors)

8.4. Fox Algorithm of Matrix Multiplication in Case of Checkerboard Data Decomposition

In designing the parallel methods of matrix multiplication the checkerboard block matrix decomposition is widely used just as the block-striped matrix partitioning. Let us analyze this method of computations in detail.

8.4.1. Computation Decomposition

The checkerboard block scheme of matrix partitioning is described in detail in 7.2. In case of this method of data decomposition the initial matrices *A* and *B* and the result matrix *C* are subdivided into sets of blocks. For simplicity the further explanations we will assume all the matrices are square of $n\times n$ size, the number of vertical blocks and the number of horizontal blocks are the same and are equal to *q* (i.e. the size of all block is equal to $k\times k$, $k=n/q$). In case of this data decomposition method the multiplying matrices *A* and *B* as blocks may be represented as follows:

$$\begin{pmatrix} A_{00}A_{01}\ldots A_{0q-1} \\ \cdots \\ A_{q-10}A_{q-11}\ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01}\ldots B_{0q-1} \\ \cdots \\ B_{q-10}B_{q-11}\ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01}\ldots C_{0q-1} \\ \cdots \\ c_{q-10}C_{q-11}\ldots C_{q-1q-1} \end{pmatrix},$$

where each block C_{ij} of matrix *C* is computed in accordance with the expression:

$$C_{ij}=\sum_{s=0}^{q-1}A_{is}B_{sj}\ .$$

In case of the checkerboard block data decomposition it is reasonable to define the basic computational subtasks on the basis of the computations performed over the matrix blocks. As a result the basic subtask can be defined as the problem of computing of a block of the matrix *C*.

To perform all the necessary computations the basic subtasks should have the corresponding sets of the matrix *A* rows and the matrix *B* columns. The placement all the necessary data in each subtask will inevitably lead to duplicating and to a considerable increase of the size of memory used. As a result, the computations must be executed in such a way that the subtasks should contain only a part of the data necessary for computations at any given moment, and the access to the rest of the data should be provided by means of data communications. One of the possible approaches (*the Fox algorithm*) will be discussed further in this Section. The second method (*the Cannon algorithm*) will be discussed in 8.5.

8.4.2. Analysis of Information Dependencies

To develop a parallel matrix multiplication method based on the checkerboard decomposition scheme it should be reminded that in this case the basic subtasks are responsible for computing the separate blocks of the matrix *C*. It is also required that each subtask should hold only one block of the multiplying matrices at each iteration.

To enumerate the subtasks the indices of the blocks C_{ij} contained in the subtasks can be used for enumeration. Thus, the subtask *(i,j)* computes the block C_{ij}. So the set of subtasks forms a square grid, which corresponds to the structure of the checkerboard block decomposition of the matrix C.

The Fox algorithm can be used to perform matrix multiplication computations under these conditions (see for instance, Fox et al. (1987), Kumar et al. (1994)).

In accordance with the Fox algorithm each basic subtasks *(i,j)* holds four matrix blocks:

- Block C_{ij} of matrix *C*, computed by the subtask;
- Block A_{ij} of matrix *A*, placed in the subtask before the beginning of computations;
- Blocks A'_{ij}, B'_{ij} of matrices *A* and *B*, obtained by the subtask in the course of computations.

Parallel algorithm execution includes:

- **The initialization stage**. Each subtask *(i,j)* obtains blocks A_{ij}, B_{ij}. All elements of blocks C_{ij} in all subtasks are set to zero;
- **The computation stage**. At this stage the following operations are carried out at each iteration $l,\ 0\le l<q$,:
  - For each row $i,\ 0\le i<q$, the block A_{ij} of subtask *(i,j)* is transmitted to all the subtasks of the same processor grid row; index *j*, which defines the position of the subtask in the row, is computed according to the following expression:

    $$j = (\,i+l\,)\ mod\ q,$$

    where *mod* is operation of obtaining the remainder in integer division;:
  - Blocks A'_{ij}, B'_{ij} obtained as a result of subtask communications are multiplied and added to block C_{ij}:

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$

– Blocks B'_{ij} of each subtask *(i,j)* are transmitted to the subtasks, which are upper neighbors in the processor grid columns (the first row blocks are transmitted to the last row of the grid).

To illustrate these rules we show the state of blocks in each subtask in the course of executing iterations of the computation stage (for the grid of 2×2). See Figure 8.6.

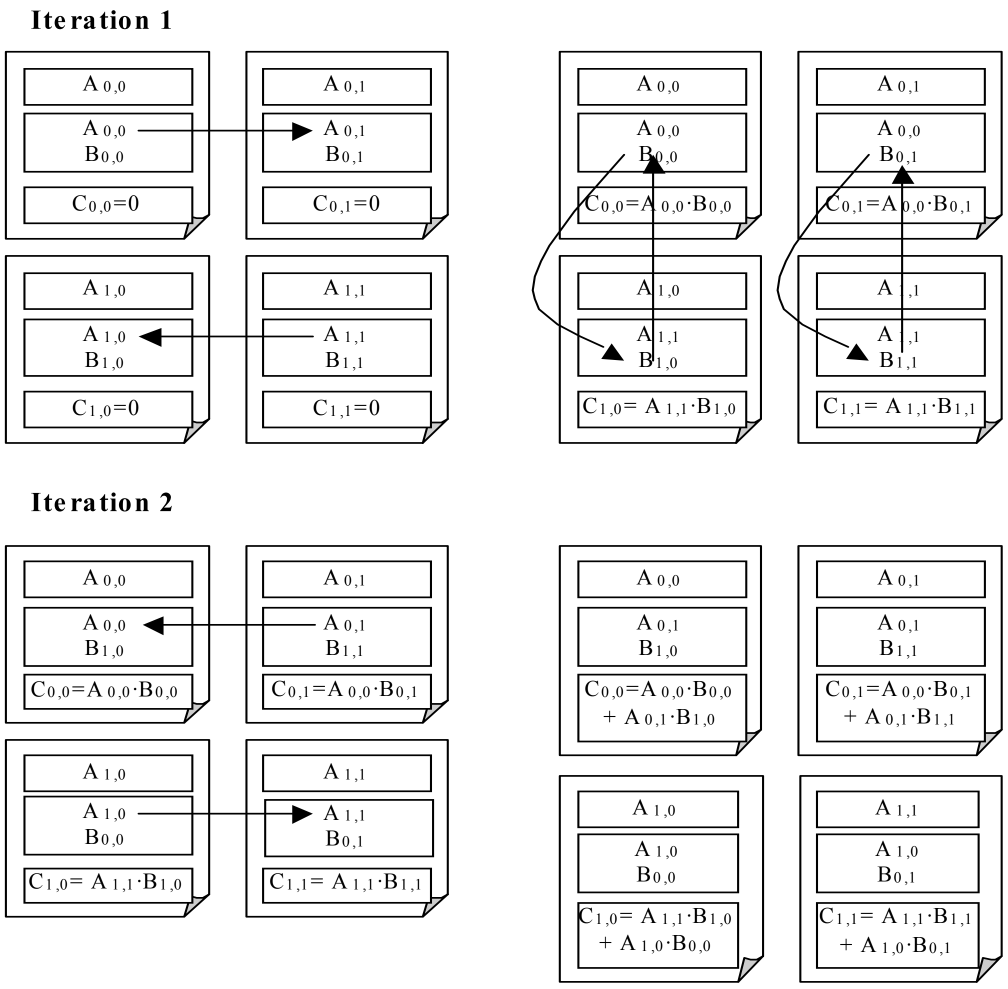

Figure 8.6. Block distribution among subtasks on iterations of the Fox algorithm

8.4.3. Scaling and Distributing Subtasks among Processors

The number of blocks at the checkerboard decomposition scheme cam be regulated by variation of matrix block sizes. These sizes may be chosen so that the total number of the basic subtasks coincides with the number of processors *p*. Thus, for instance, in the simplest case when the number of processors is equal to $p=\delta^2$, the size of the block grid may be chosen equal to δ (i.e. $q=\delta$). This way to define the number of blocks makes the amount of computations in each subtask the same and, thus, uniform balancing of the computational load is achieved. In a more general case, when the number of processors and the sizes of matrices are arbitrary, computational load may not be distributed among processors equally but proper setting the sizes of the matrix blocks can provide uniform load balancing with adequate accuracy.

To execute the Fox algorithm efficiently, when the basic subtasks form a square grid and data communications consist in block transmission along rows and columns of the subtask grid, the network topology should be also a square grid. In this case it is possible to map easily the set of subtasks onto the set of processors by placing the basic subtasks (*i,j*) on processors $P_{i,j}$. The required structure of the data communication network may be provided at the physical level, if the network topology is a grid or a complete graph.

8.4.4. Efficiency Analysis s

Let us evaluate the computational complexity of the Fox algorithm. To formulate the required estimations we will suppose that all the previous assumptions are met, i.e. all matrices are square, their sizes are $n\times n$, the block grid is square and its size is equal to q (i.e. the size of all blocks is $k\times k$, $k=n/q$), processors form a square grid and their number is $p=q^2$.

As it has been previously mentioned, the execution of the Fox algorithm requires q iterations, during which each processor multiplies its current blocks of the matrices A and B, and adds the multiplication results to the current block of the matrix C. With regard to the above mentioned assumptions, the total number of the executed operations will be of the order n^3/p. As a result, the speedup and efficiency of the algorithm look as follows:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{and} \quad E_p = \frac{n^3}{p\cdot\left(n^3/p\right)} = 1\,. \tag{8.9}$$

The general efficiency analysis indicates that parallel computation efficiency is ideal. To make the obtained relations more precise we will estimate more exactly the number of algorithm computational operations and take into account the overhead related with data communications among the processors.

Let us evaluate the number of computational operations. The complexity of scalar multiplication of the block row of the matrix A by the block column of the matrix B may be estimated as $2(n/q)-1$. The number of rows and columns in the blocks is equal to n/q. As a result, the time complexity of block multiplication appears to be equal to $(n^2/p)(2n/q-1)$. The addition of the blocks requires n^2/p operations. With regard to the above given expressions the computational time of the Fox algorithm may be estimated in the following way:

$$T_p\left(calc\right) = q[(n^2/p)\cdot\left(2n/q-1\right)+(n^2/p)]\cdot\tau\,. \tag{8.10}$$

(as previously that τ is the execution time of an basic computational operation).

Let us estimate now the overhead on data communications among the processors. One of the processors of the processor grid row transmits its matrix A block to the rest of the grid row processors at each iteration. As it has been mentioned previously, the execution of this operation may be provided in log_2q steps, if the topology of the network is a hypercube or a complete graph. As a result, the time complexity of data communications in accordance with the Hockney model may be estimated as follows:

$$T_{\mathrm{p}}^1\left(comm\right) = \log_2 q\,(\alpha + w(n^2/p)/\beta) \tag{8.11}$$

where α is the latency, β is the network bandwidth, w is the size of a matrix elements in bytes. In case of the ring topology the expression for time estimation looks as follows:

$$\tilde{T}_{\mathrm{p}}^1\left(comm\right) = (q/2)\,(\alpha + w(n^2/p)/\beta)\,.$$

After multiplying the matrix blocks the processors send their matrix B blocks to the processors, which are upper neighbors in the processor grid columns (the first row processors send their data to the last row processors). These operations may be carried out by the processors in parallel and, thus, the time complexity of these communication operations is the following:

$$T_p^2\left(comm\right) = \alpha + w\cdot(n^2/p)/\beta\,. \tag{8.12}$$

After the summation of all the obtained expressions, it becomes clear that the total execution time for the Fox algorithm may be defined by means of the following relations:

$$\begin{aligned} T_p &= q[(n^2/p)\cdot\left(2n/q-1\right)+(n^2/p)]\cdot\tau + q\log_2 q\,(\alpha + w(n^2/p)/\beta) + \left(q-1\right)\cdot(\alpha + w(n^2/p)/\beta) = \\ &= q[(n^2/p)\cdot\left(2n/q-1\right)+(n^2/p)]\cdot\tau + (q\log_2 q + (q-1))(\alpha + w(n^2/p)/\beta) \end{aligned} \tag{8.13}$$

(it should be reminded that parameter q defines the size of the processor grid and $q=\sqrt{p}$).

8.4.5. Software Implementation

Here we discuss possible software implementation of the Fox algorithm for matrix multiplication in case of the checkerboard block data decomposition. The given program code contains the basic modules of the parallel program. The absence of some auxiliary functions will not hinder the process of understanding of this parallel computation scheme.

1. The main function. The main function implements the computational method scheme by sequential calling out the necessary subprograms.

```
// Program 8.1
// The Fox algorithm of matrix multiplication – checkerboard decomposition
// Program execution conditions:
//   all matrices and their blocks are square,
//   matrix blocks and processors form square grids of the same size

int ProcNum = 0;      // Number of available processes
int ProcRank = 0;     // Rank of current process
int GridSize;         // Size of virtual processor grid
int GridCoords[2];    // Coordinates of current processor in grid
MPI_Comm GridComm;    // Grid communicator
MPI_Comm ColComm;     // Column communicator
MPI_Comm RowComm;     // Row communicator

void main ( int argc, char * argv[] ) {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;          // Size of matricies
  int BlockSize;     // Sizes of matrix blocks on current process
  double *pAblock;   // Initial block of matrix A on current process
  double *pBblock;   // Initial block of matrix B on current process
  double *pCblock;   // Block of result matrix C on current process
  double *pMatrixAblock;
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  GridSize = sqrt((double)ProcNum);
  if (ProcNum != GridSize*GridSize) {
    if (ProcRank == 0) {
      printf ("Number of processes must be a perfect square \n");
    }
  }
  else {
    if (ProcRank == 0)
      printf("Parallel matrix multiplication program\n");

    // Creating the cartesian grid, row and column communcators
    CreateGridCommunicators();

    // Memory allocation and initialization of matrix elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock, Size, BlockSize );

    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
      BlockSize);

    // Execution of Fox method
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
      pCblock, BlockSize);

    ResultCollection(pCMatrix, pCblock, Size, BlockSize);
    TestResult(pAMatrix, pBMatrix, pCMatrix, Size);

    // Process Termination
    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock);
  }

  MPI_Finalize();
}
```

2. The function *CreateGridCommunicators*. This function creates a communicator as a two-dimensional square grid, determines the coordinates of each process in the grid and creates communicators for each row and each column separately.

The grid is created by the function *MPI_Cart_create* (the vector *Periodic* defines the permissibility of data communications among the bordering processes of the grid columns and rows). After the grid has been created, each parallel program process will have its coordinates in the grid. The coordinates may be obtained by means of the function *MPI_Cart_coords*.

Then in addition to the grid topology a set of communicators for each grid column and row separately is created by the function *MPI_Cart_sub*.

```
void CreateGridCommunicators() {
  int DimSize[2];  // Number of processes in each dimension of the grid
  int Periodic[2]; // =1, if the grid dimension should be periodic
  int Subdims[2];  // =1, if the grid dimension should be fixed

  DimSize[0] = GridSize;
  DimSize[1] = GridSize;
  Periodic[0] = 0;
  Periodic[1] = 0;

  // Creation of the Cartesian communicator
  MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

  // Determination of the cartesian coordinates for every process
  MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

  // Creating communicators for rows
  Subdims[0] = 0;  // Dimensionality fixing
  Subdims[1] = 1;  // The presence of the given dimension in the subgrid
  MPI_Cart_sub(GridComm, Subdims, &RowComm);

  // Creating communicators for columns
  Subdims[0] = 1;
  Subdims[1] = 0;
  MPI_Cart_sub(GridComm, Subdims, &ColComm);
}
```

3. The function *ProcessInitialization*. This function sets the matrix sizes and allocates memory for storing the initial matrices and their blocks, initializes all the original problem data. In order to determine the elements of the initial matrices we will use the functions *DummyDataInitialization* and *RandomDataInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
  double* &pTemporaryAblock, int &Size, int &BlockSize ) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);

      if (Size%GridSize != 0) {
        printf ("Size of matricies must be divisible by the grid size! \n");
      }
    }
    while (Size%GridSize != 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  BlockSize = Size/GridSize;

  pAblock = new double [BlockSize*BlockSize];
  pBblock = new double [BlockSize*BlockSize];
  pCblock = new double [BlockSize*BlockSize];
  pTemporaryAblock = new double [BlockSize*BlockSize];
  for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
  }
  if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    //DummyDataInitialization(pAMatrix, pBMatrix, Size);
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
  }
}
```

4. The function *ParallelResultCalculation*. The function *ParallelResultCalculation* executes the parallel Fox algorithm of matrix multiplication. The matrix blocks and their sizes must be given to the function as its arguments.

According to the scheme of parallel computations described in Exercise 3, it is necessary to carry out *GridSize* iterations in order to execute matrix multiplication with the use of Fox algorithm. Each of the iterations consists of the execution of the following operations:

- The broadcast of the matrix *A* block along the processor grid row (to execute the step we should develop the function *ABlockCommunication*),
- The multiplication of matrix blocks (to carry out the multiplication of matrix blocks we may use the function *SerialResultCalculation*, which was implemented in the course of the development of the serial matrix multiplication program),

The cyclic shift of the matrix *B* blocks along the column of the processor grid (the function *BBlockCommunication*).

```
void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,
  double* pBblock, double* pCblock, int BlockSize) {
  for (int iter = 0; iter < GridSize; iter ++) {
    // Sending blocks of matrix A to the process grid rows
    ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);
    // Block multiplication
    BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
    // Cyclic shift of blocks of matrix B in process grid columns
    BblockCommunication(pBblock, BlockSize);
  }
}
```

5. The function *AblockCommunication*. The function broadcasts matrix A blocks to the process grid rows. The leading process *Pivot* that responsible for sending is chosen in each row of the grid. For broadcasting the pivot processes are used their blocks *pMatrixAblock* (let us to remind that these blocks transmitted to the processes at the moment of the initial data distribution). The required communications are executed by means of the function *MPI_Bcast*. It should be noted that the operation is collective, and its localization in separate process grid rows is provided by the communicators *RowComm*, which are created for the set of processes of each row separately.

```
// Broadcasting matrix A blocks to process grid rows
void ABlockCommunication (int iter, double *pAblock, double* pMatrixAblock,
  int BlockSize) {

  // Defining the leading process of the process grid row
  int Pivot = (GridCoords[0] + iter) % GridSize;

  // Copying the transmitted block in a separate memory buffer
  if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
      pAblock[i] = pMatrixAblock[i];
  }

  // Block broadcasting
  MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot, RowComm);
}
```

6. The function *BlockMultiplication*. The function executes block multiplication of the matrices *A* and *B*. The easiest way to perform this multiplication is to use the serial matrix multiplication algorithm, described in 8.2. It should be noted that we provide the simplest variant of the function implementation for better understanding of the program. These calculations may be optimized to decrease the computation time. This optimization may be aimed, for instance, at increasing the efficiency of the processor cache, vectorizing the executed operations etc.

7. The function *BblockCommunication*. The function performs the cyclic shift of blocks of the matrix *B* in the process grid columns. Each process transmits its block to the upper neighboring process *NextProc* in the process column and receives the block transmitted from the process *PrevProc* , which stands below it in the grid column. Data transmission is executed by means of the function *MPI_SendRecv_replace*, which provides all the necessary block transmissions using the same memory buffer *pBblock*. Besides, this function prevents possible deadlocks, which happen when data transmission begins to be performed simultaneously by several processes in the ring network topology.

```
// Cyclic shift of matrix B blocks in the process grid columns
void BblockCommunication (double *pBblock, int BlockSize) {
  MPI_Status Status;
  int NextProc = GridCoords[0] + 1;
  if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
  int PrevProc = GridCoords[0] - 1;
  if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

  MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColComm, &Status);
}
```

8.4.6. Computational Experiment Results

The computational experiments for estimating the parallel algorithm efficiency were carried out under the same conditions as those described in 8.3.5. The results of the experiments with the use of 4 and 9 processors are given in Table 8.3.

Table 8.3 The Results of the computational experiments for estimating the Fox parallel algorithm efficiency

Matrix Size	Serial Algorithm	Parallel Algorithm			
		4 processors		9 processors	
		Time	Speed Up	Time	Speed Up
500	0,8527	0,2190	3,8925	0,1468	5,8079
1000	12,8787	3,0910	4,1664	2,1565	5,9719
1500	43,4731	10,8678	4,0001	7,2502	5,9960
2000	103,0561	24,1421	4,2687	21,4157	4,8121
2500	201,2915	51,4735	3,9105	41,2159	4,8838
3000	347,8434	87,0538	3,9957	58,2022	5,9764

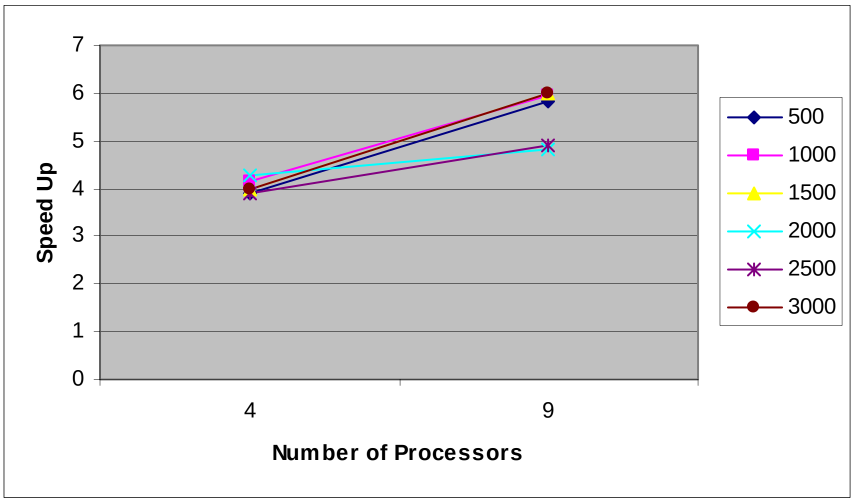

Figure 8.7. Speedup of the Fox Parallel Algorithm with Respect to Number of Processors

The comparison of the experiment execution time T_p^* and the theoretical time T_p, calculated according to expression 8.13, is shown in Table 8.4 and in Figure 8.8.

Table 8.4. The comparison of the experimental and theoretical execution time for the Fox parallel algorithm

Matrix Size	4 processors		9 processors	
	T_p	T_p^*	T_p	T_p^*
500	0,4217	0,2190	0,2200	0,1468
1000	3,2970	3,0910	1,5924	2,1565
1500	11,0419	10,8678	5,1920	7,2502
2000	26,0726	24,1421	12,0927	21,4157
2500	50,8049	51,4735	23,3682	41,2159
3000	87,6548	87,0538	40,0923	58,2022

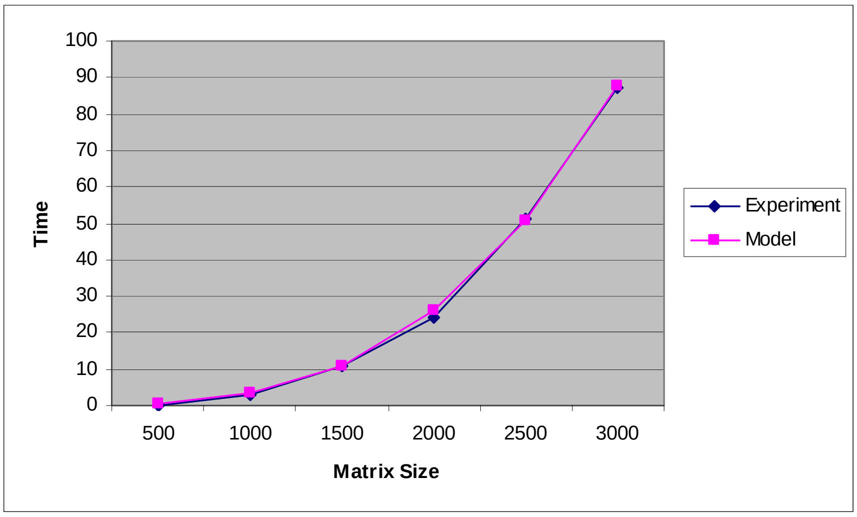

Figure 8.8. Experimental and theoretical execution time of the Fox parallel algorithm with respect to matrix size (checkerboard block matrix decomposition, 4 processors)

8.5. Cannon Algorithm of Matrix Multiplication in Case of Checkerboard Data Decomposition

Let us discuss one more parallel algorithm of matrix multiplication based on checkerboard block matrix partitioning.

8.5.1. Computation Decomposition

As it was in case of the Fox algorithm, the computations for calculating a block of the result matrix *C* will be chosen as the basic computational subtask. As it has been mentioned previously, a subtask should have access to the elements of the horizontal stripe of the matrix *A* and the elements of the vertical stripe of the matrix *B* in order to compute the elements of the block of the matrix *C*.

8.5.2. Analysis of the Information Dependencies

The difference between the Cannon algorithm and the Fox method is the initial distribution scheme of the multiplied matrix blocks among the subtasks. The initial block distribution in the Cannon algorithm is chosen so that the blocks located in the subtasks can be multiplied without additional data transmission. The block distribution may be provided so that transmitting blocks among the subtasks in the course of computations can be performed by means of simpler communication operations.

With regard to these remarks, the initialization stage of the Cannon algorithm includes the execution of the following data communication operations:

- Blocks A_{ij}, B_{ij} are transmitted into each subtask *(i,j)*;
- For each subtask grid row *i* blocks of the matrix *A* are shifted *(i-1)* positions left;
- For each subtask grid column *j* blocks of the matrix *B* are shifted *(j-1)* positions up.

The data communication procedures, carried out in redistribution of matrix blocks, are examples of the cyclic shift operations – see Section 3. To illustrate the method of the initial data distribution we give an example of block placement for the subtask grid 3×3 in Figure 8.9.

Redistribution of matrix A and B blocks

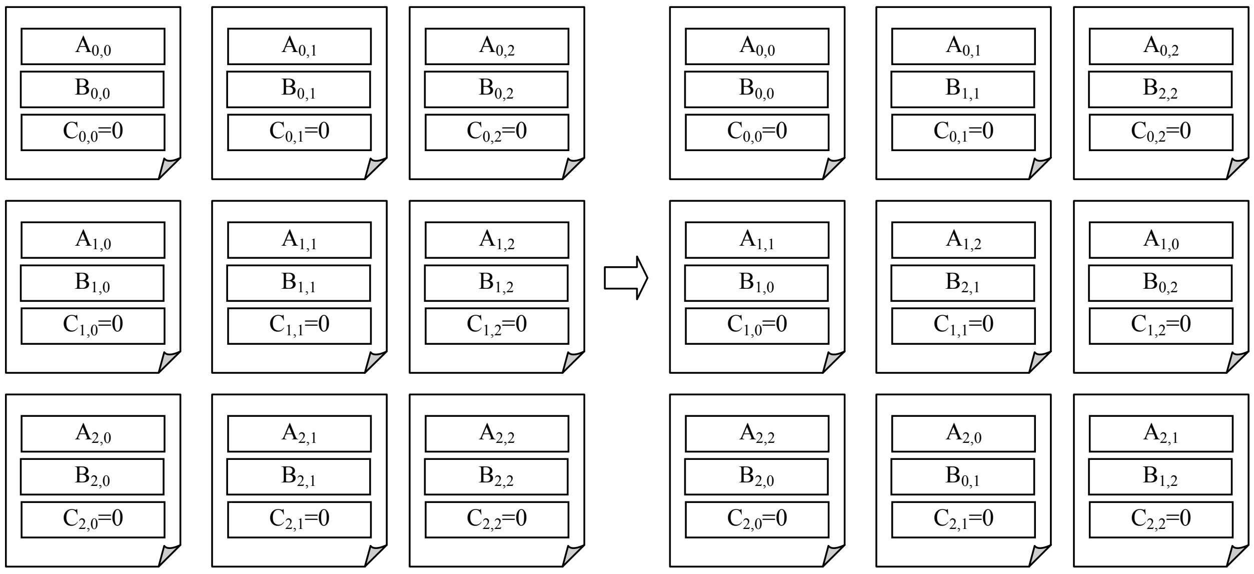

Figure 8.9. Redistributing the blocks of the initial matrices among the processes in the Cannon algorithm

As a result of this initial distribution each subtask will have the blocks, which may be multiplied without additional data communications. Besides, obtaining the further blocks for all the subtasks may be provided by means of simple communication operations. After the operation of block multiplication each block of the matrix *A* should be transmitted to the preceding subtask to the left in the subtask grid rows, and each block of the matrix *B* should be transmitted to the preceding subtask up in the grid columns. It can be shown that the successive iterations of these cyclic shifts and the multiplication of the obtained matrix blocks lead to obtaining the corresponding blocks of the result matrix *C* in the basic subtasks.

8.5.3. Scaling and Distributing Subtasks among Processors

As in case of the Fox method the block size for the Cannon algorithm may be chosen so that the number of the basic subtasks and the number of processors becames the same. As the amount of computations in each subtask is identical, it provides uniform balance of the computational load among the processors.

The scheme, which was used in the Fox algorithm - i.e. representing the available processors as a square grid and placing the basic subtasks (i,j) on the corresponding processors $P_{i,j}$ of the processor grid– can be also applied for the Cannon algorithm. The adequate data communication network structure may be as previously provided at the physical level, if the topology of the computer system is a grid or a complete graph.

8.5.4. Efficiency Analysis

Before carrying out the efficiency analysis we should mention that only difference between the Cannon algorithm and the Fox method is the type of the communication operations performed in the course of computations. As a result, we will analyze only the communication complexity of the Cannon algorithm using the estimations of the execution time of the computational operations given in 8.4.4.

According to the algorithm the block redistribution of the matrix *A* and matrix *B* is performed by means of the cyclic shift of the matrix blocks in the processor grid columns and rows at the stage of initialization. The time complexity of such data communication operation depends essentially on the network topology. For the complete graph topology all the blocks may be transmitted simultaneously (i.e. the duration of operation appears to be equal to the time of transmitting a matrix block between the neighboring processors). For the hypercube topology network the cyclic shift operation may require the execution of $log_2 q$ iterations. For the ring topology the necessary number of iteration appears to be equal to $q-1$. The methods of executing the cyclic shift operation are discussed in detail in Section 3. To estimate the communication complexity of the initialization stage we will use the complete graph topology as it corresponds to cluster computational systems. As a result the execution time of the initial block redistribution may be estimated as follows:

$$T_p^1(comm) = 2 \cdot \left(\alpha + w \cdot (n^2/p)/\beta\right) \tag{8.14}$$

(the expression n^2/p defines the sizes of the transmitted blocks, and the coefficient 2 corresponds to two executed operations of the cyclic shift).

Let us estimate the overhead of data communications among the processors during the computation stage of the Cannon algorithm. At each algorithm iteration after multiplying the matrix blocks the processors transmit the blocks to the preceding processors in the rows (for blocks of the matrix *A*) and the columns (for blocks of the matrix *B*) of the processor grid. These operations may be performed by the processors in parallel. Thus, the duration of the communication operations is as follows:

$$T_p^2(comm) = 2 \cdot \left(\alpha + w \cdot (n^2/p)/\beta\right). \qquad (8.15)$$

As the number of the Cannon algorithm iterations is equal to *q*, the total execution time of parallel computation, with regard to estimation 8.13, may be evaluated by means of the following expression:

$$T_p = q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau+(2q+2)(\alpha+w(n^2/p)/\beta) \qquad (8.16)$$

(it should be reminded that the parameter $q = \sqrt{p}$ defines the size of the processor grid).

8.5.5. Computational Experiments Results

The computational experiments for estimation of the parallel algorithm efficiency were carried out under the same conditions as those described in 8.3.5. The results of the experiments for 4 and 9 processors are given in Table 8.5.

Table 8.5. The results of the computational experiments for estimating the Cannon parallel algorithm efficiency

Matrix Size	Serial Algorithm	Parallel Algorithm			
		4 processors		9 processors	
		Time	Speed Up	Time	Speed Up
1000	12,8787	3,0806	4,1805	1,1889	10,8324
1500	43,4731	11,1716	3,8913	4,6310	9,3872
2000	103,0561	24,0502	4,2850	14,4759	7,1191
2500	201,2915	53,1444	3,7876	23,5398	8,5511
3000	347,8434	88,2979	3,9394	36,3688	9,5643

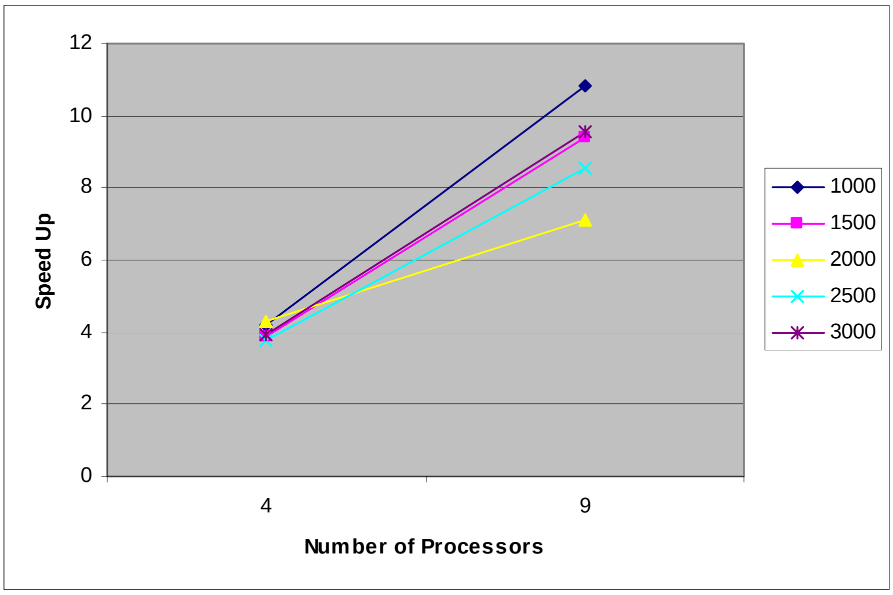

Figure 8.10. Speedup of the Cannon parallel algorithm with respect to Number of Processors

The comparison of the experiment execution time T_p^* and theoretical time T_p, calculated according to expression 8.16 is given in Table 8.6 and Figure 8.13.

Table 8.6. The comparison of the experimental execution time and the theoretical execution time for the Cannon parallel algorithm

Matrix Size	4 processors		9 processors	
	T_p	T_p^*	T_p	T_p^*

1000	3,4485	3,0806	1,5669	1,1889
1500	11,3821	11,1716	5,1348	4,6310
2000	26,6769	24,0502	11,9912	14,4759
2500	51,7488	53,1444	23,2098	23,5398
3000	89,0138	88,2979	39,8643	36,3688

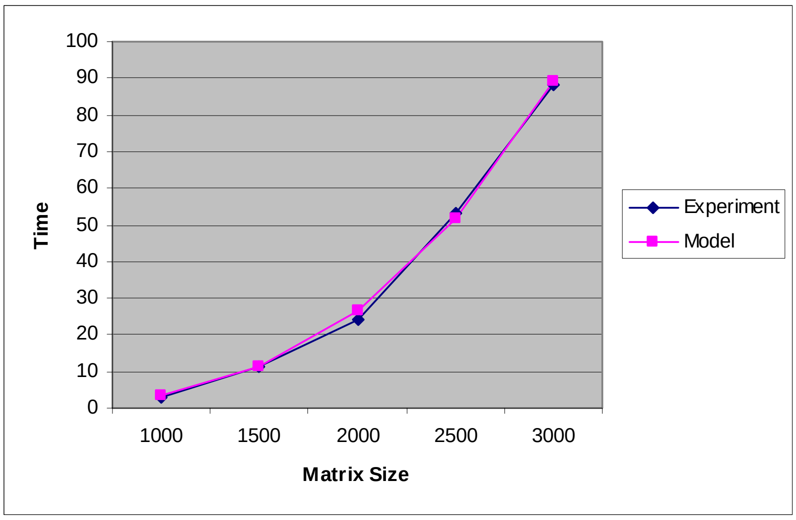

Figure 8.11. Experimental and theoretical execution time of the Cannon parallel algorithm with respect to matrix size (checkerboard block matrix decomposition, 4 processors)

8.6. Summary

The Section discussed three parallel methods of matrix multiplication. The first algorithm is based on the block-striped matrix distribution among the processors. Two variants of this method are described in the Section. The first variant is based on different partitioning of the multiplied matrices when the first matrix (the matrix *A*) is partitioned into horizontal striped, and the second matrix (the matrix *B*) is partitioned into vertical stripes. In case of the second variant both matrices are partitioned into horizontal stripes.

This Section also discussed the well-known the Fox and the Cannon algorithms, based on the checkerboard block matrix decomposition. Using the same schemes of matrix decomposition these algorithms differ from each other in the data communication operations. For the Fox algorithm matrix block broadcasting and matrix block cyclic shift are performed in the course of computations; while in Cannon algorithm only the cyclic shift operation is performed.

The difference in the ways of data partitioning leads to different topologies of communication network, which enhance the efficiency of the parallel algorithm execution. Thus, the algorithms based on the block-striped data decomposition are designed at the hypercube or the complete graph topology. For the algorithms based on the checkerboard block data decomposition the grid topology is the most efficient.

The summary graph in Figure 8.12 presents the speedup values obtained as a result of the computational experiments for all the discussed algorithms. The computations have shown that increasing the number of processors improves the checkerboard block multiplication algorithm efficiency.

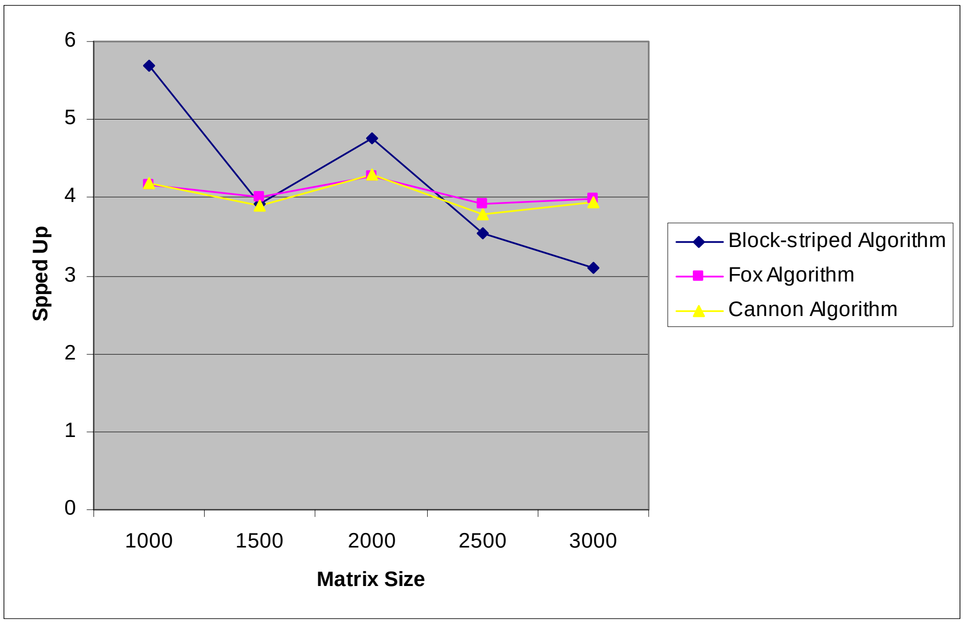

Figure 8.12. Speedup of the matrix parallel multiplication algorithms according with computational experiments (4 processors)

8.7. References

The problem of matrix multiplication is broadly discussed in science. As additional training materials we may recommend the works by Kumar, et al. (1994) and Quinn (2004). The problems of parallel execution of matrix multiplication are also discussed in Dongarra, et al. (1999).

Blackford, et al. (1997) may be useful for considering some aspects of parallel software development. This book describes the software library of numerical methods ScaLAPACK, which is well-known and widely used.

8.8. Discussions

1. What is the statement of the matrix multiplication problem?
2. Give the examples of the problems, which make use of the matrix multiplication operations.
3. Give the examples of various sequential algorithms of matrix multiplication operations. Is the complexity various in case of different algorithms?
4. What methods of data distribution are used in developing parallel algorithms of matrix multiplication?
5. Describe the general schemes of the parallel algorithms considered in the Section.
6. Analyze and compute the efficiency of the block-striped algorithm for horizontal partitioning of the multiplied matrices.
7. What information communications are carried out for the algorithms in case of the block-striped data decomposition?
8. What information interactions are carried out in case of the checkerboard block matrix multiplication algorithms?
9. What communication network topology is efficient for each of the algorithms discussed?
10. Which of the discussed algorithms requires the least memory size and which of them requires the greatest necessary memory size?
11. Which of the discussed algorithms has the best speedup and efficiency?
12. Estimate the possibility to carry out matrix multiplication as a sequence of matrix-vector operations.
13. Give the general description of the software implementation for the Fox algorithm. In what way may be differences of software implementation of other algorithms?
14. What functions of the library MPI appear to be necessary in the software implementation of the algorithms?

8.9. Problems

1. Develop the implementation of two block-striped algorithms of matrix multiplication. Compare the execution time of these algorithms.

2. Develop the Cannon algorithm implementation. Formulate the theoretical estimation of the algorithm execution time. Execute the computational experiments. Compare the experimental results with the theoretical estimations.

3. Develop the implementation of the checkerboard block algorithm of matrix multiplication, which may be carried out for rectangular processor grids.

4. Develop the implementation of matrix multiplication using the previously developed programs of matrix-vector multiplication.

References

Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999). Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, D. Walker, R.C. Whaley, K. (1997). Scalapack Users' Guide (Software, Environments, Tools). Soc for Industrial & Applied Math.

Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.

Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)

Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.