

## **7.Parallel Methods for Matrix-Vector Multiplication**

7.	Parallel Methods for Matrix-Vector Multiplication.....	1
7.1.	Introduction .....	1
7.2.	Parallelization Principles.....	2
7.3.	Problem Statement .....	3
7.4.	Sequential Algorithm.....	3
7.5.	Data Distribution .....	3
7.6.	Matrix-Vector Multiplication in Case of Rowwise Data Decomposition .....	4
7.6.1.	Analysis of Information Dependencies .....	4
7.6.2.	Scaling and Subtask Distribution among Processors .....	4
7.6.3.	Efficiency Analysis .....	4
7.6.4.	Program Implementation.....	5
7.6.5.	Computational Experiment Results .....	5
7.7.	Matrix-Vector Multiplication in Case of Columnwise Data Decomposition .....	9
7.7.1.	Computation Decomposition and Analysis of Information Dependencies .....	9
7.7.2.	Scaling and Subtask Distribution among Processors .....	10
7.7.3.	Efficiency Analysis .....	10
7.7.4.	Computational Experiment Results .....	11
7.8.	Matrix-Vector Multiplication in Case of Checkerboard Data Decomposition .....	12
7.8.1.	Computation Decomposition.....	12
7.8.2.	Analysis of Information Dependencies .....	13
7.8.3.	Scaling and Distributing Subtasks among Processors .....	13
7.8.4.	Efficiency Analysis .....	14
7.8.5.	Computational Experiment Results .....	14
7.9.	Summary .....	15
7.10.	References .....	16
7.11.	Discussions .....	16
7.12.	Exercises .....	17

### **7.1. Introduction**

Matrices and matrix operations are widely used in mathematical modeling of various processes, phenomena and systems. Matrix calculations are the basis of many scientific and engineering calculations. Computational mathematics, physics, economics are only some of the areas of their application.

As the efficiency of carrying out matrix computations is highly important many standard software libraries contain procedures for various matrix operations. The amount of software for matrix processing is constantly increasing. New efficient storage structures for special type matrix (triangle, banded, sparse etc.) are being created. Highly efficient machine-dependent algorithm implementations are being developed. The theoretical research into searching faster matrix calculation method is being carried out.

Being highly time consuming, matrix computations are the classical area of applying parallel computations. On the one hand, the use of highly efficient multiprocessor systems makes possible to substantially increase the complexity of the problem solved. On the other hand, matrix operations, due to their rather simple formulation, give a nice opportunity to demonstrate various techniques and methods of parallel programming.

In this chapter we will discuss the parallel programming methods for matrix-vector multiplication. In the next Section (Section 8) we will discuss a more general case of matrix multiplication. Solving linear equation systems, which are an important type of matrix calculations, are discussed in Section 9. The problem of matrix distributing between the processors is common for all the above mentioned matrix calculations and it is discussed in Subsection 7.2.

Let us assume that the matrices, we are considering, are dense, i.e. the number of zero elements in them is insignificant in comparison to the general number of matrix elements.

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

## 7.2. Parallelization Principles

The repetition of the same computational operations for different matrix elements is typical of different matrix calculation methods. In this case we can say that there exist *data parallelism*. As a result, the problem to parallelize matrix operations can be reduced in most cases to matrix distributing among the processors of the computer system. The choice of matrix distribution method determines the use of the definite parallel computation method. The availability of various data distribution schemes generates a range of parallel algorithms of matrix computations.

The most general and the most widely used matrix distribution methods consist in partitioning data into *stripes* (vertically and horizontally) or rectangular fragments (*blocks*).

**1. Block-striped matrix partitioning.** In case of block-striped partitioning each processor is assigned a certain subset of matrix rows (*rowwise* or *horizontal partitioning*) or matrix columns (*columnwise* or *vertical partitioning*) (Figure 7.1). Rows and columns are in most cases subdivided into stripes on a continuous sequential basis. In case of such approach, in rowwise decomposition (see Figure 7.1), for instance, matrix  $A$  is represented as follows:

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}), i_j = ik + j, 0 \leq j < k, k = m/p, \quad (7.1)$$

where  $a_i = (a_{i1}, a_{i2}, \dots, a_{in})$ ,  $0 \leq i < m$ , is  $i$ -th row of matrix  $A$  (it is assumed, that the number of rows  $m$  is divisible by the number of processors  $p$  without a remainder, i.e.  $m = k \cdot p$ ). Data partitioning on the continuous basis is used in all matrix and matrix-vector multiplication algorithms, which are considered in this and the following sections.

Another possible approach to forming rows is the use of a certain row or column alternation (*cyclic*) scheme. As a rule, the number of processors  $p$  is used as an alternation parameter. In this case the horizontal partitioning of matrix  $A$  looks as follows:

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}), i_j = i + jp, 0 \leq j < k, k = m/p. \quad (7.2)$$

The cyclic scheme of forming rows may appear to be useful for better balancing of computational load (for instance, it may be useful in case of solving a linear equation system with the use of the Gauss method, see Section 9).

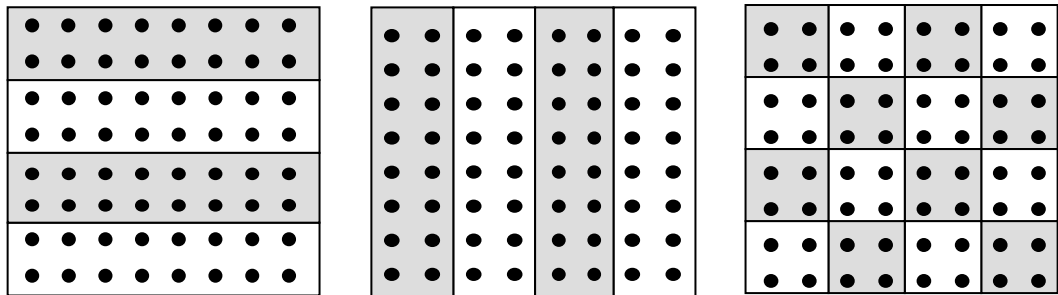
**2. Checkerboard Block Matrix Partitioning.** In this case the matrix is subdivided into rectangular sets of elements. As a rule, it is being done on a continuous basis. Let the number of processors be  $p = s \cdot q$ , the number of matrix rows is divisible by  $s$ , the number of columns is divisible by  $q$ , i.e.  $m = k \cdot s$  and  $n = l \cdot q$ . Then the matrix  $A$  may be represented as follows:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0q-1} \\ & \dots & & \\ A_{s-11} & A_{s-12} & \dots & A_{s-1q-1} \end{pmatrix},$$

where  $A_{ij}$  - is a matrix block, which consists of the elements:

$$A_{ij} = \begin{pmatrix} a_{i_0j_0} & a_{i_0j_1} & \dots & a_{i_0j_{l-1}} \\ & \dots & & \\ a_{i_{k-1}j_0} & a_{i_{k-1}j_1} & \dots & a_{i_{k-1}j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \leq v < k, k = m/s, j_u = jl + u, 0 \leq u < l, l = n/q. \quad (7.3)$$

In case of this approach it is advisable that a computer system have a physical or at least a logical processor grid topology of  $s$  rows and  $q$  columns. Then, for data distribution on a continuous basis the processors neighboring in grid structure will process adjoining matrix blocks. It should be noted however that cyclic alteration of rows and columns can be also used for the checkerboard block scheme.



**Figure 7.1.** Most widely used matrix decomposition schemes

In this chapter three parallel algorithms are considered for square matrix multiplication by a vector. Each approach is based on different types of given data (matrix elements and vector) distribution among the processors. The data distribution type changes the processor interaction scheme. Therefore, each method considered here differs from the others significantly.

### 7.3. Problem Statement

The result of multiplying the matrix  $A$  of order  $m \times n$  by vector  $b$ , which consists of  $n$  elements, is the vector  $c$  of size  $m$ , each  $i$ -th element of which is the result of inner multiplication of  $i$ -th matrix  $A$  row (let us denote this row by  $a_i$ ) by vector  $b$ :

$$c_i = (a_i, b) = \sum_{j=0}^{n-1} a_{ij} b_j, \quad 0 \leq i \leq m-1. \quad (7.4)$$

Thus, obtaining the result vector  $c$  can be provided by the set of the same operations of multiplying the rows of matrix  $A$  by the vector  $b$ . Each operation includes multiplying the matrix row elements by the elements of vector  $b$  ( $n$  operations) and the following summing the obtained products ( $n-1$  operations). The total number of necessary scalar operations is the value

$$T_1 = m \cdot (2n - 1).$$

### 7.4. Sequential Algorithm

The sequential algorithm of multiplying matrix by vector may be represented in the following way:

```
// Algorithm 7.1
// Sequential algorithm of multiplying matrix by vector
for (i = 0; i < m; i++){
    c[i] = 0;
    for (j = 0; j < n; j++){
        c[i] += A[i][j]*b[j]
    }
}
```

**Algorithm 12.1.** Sequential algorithm of matrix-vector multiplication

In the given program code the following notation is used:

- Input data:
  - $A[m][n]$  – matrix of order  $m \times n$ ,
  - $b[n]$  – vector of  $n$  elements,
- Result:
  - $c[m]$  – vector of  $m$  elements.

Matrix-vector multiplication is the sequence of inner product computations. As each computation of inner multiplication of vectors of size  $n$  requires execution of  $n$  multiplications and  $n-1$  additions, its time complexity is the order  $O(n)$ . To execute matrix-vector multiplication it is necessary to execute  $m$  operations of inner multiplication. Thus, the algorithm's time complexity is the order  $O(mn)$ .

### 7.5. Data Distribution

While executing the parallel algorithm of matrix-vector multiplication, it is necessary to distribute not only the matrix  $A$ , but also the vector  $b$  and the result vector  $c$ . The vector elements can be *duplicated*, i.e. all the vector elements can be copied to all the processors of the multiprocessor computer system, or *distributed* among the processors. In case of block partitioning of the vector consisting of  $n$  elements, each processor processes the continuous sequence of  $k$  vector elements (we assume that the vector size  $n$  is divisible by the number of processors  $p$ , i.e.  $n = kp$ ).

Let us make clear, why duplicating vectors  $b$  and  $c$  among the processors is an admissible decision (for simplicity further we will assume that  $m=n$ ). Vectors  $b$  and  $c$  consist of  $n$  elements, i.e. contain as much data as one matrix row or column. If the processor holds a matrix row or column and single elements of the vectors  $b$  and  $c$ , the total size of used memory is the order  $O(n)$ . If the processor holds a matrix row (column) and all the elements of the vectors  $b$  and  $c$ , the total number of used memory is the same order  $O(n)$ . Thus, in cases of vector duplicating and vector distributing the requirements to memory size are equivalent.

## 7.6. Matrix-Vector Multiplication in Case of Rowwise Data Decomposition

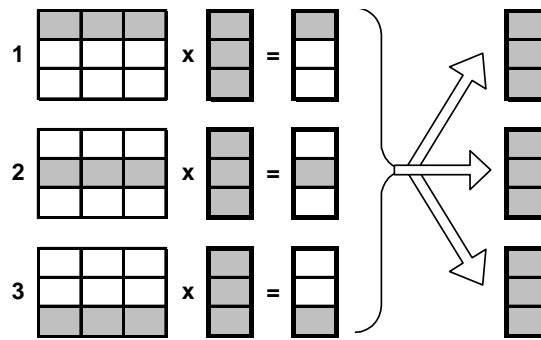
As the first example of parallel matrix computations, let us consider the algorithm of matrix-vector multiplication, which is based on rowwise block-striped matrix decomposition scheme. If this case, the operation of inner multiplication of a row of the matrix  $A$  and the vector  $b$  can be chosen as the basic computational subtask.

### 7.6.1. Analysis of Information Dependencies

To execute the basic subtask of inner multiplication the processor must contain the corresponding row of matrix  $A$  and the copy of vector  $b$ . After computation completion each basic subtask determines one of the elements of the result vector  $c$ .

To combine the computation results and to obtain the total vector  $c$  on each processor of the computer system, it is necessary to execute the all gather operation (see Sections 3-4, 6), in which each processor transmits its computed element of vector  $c$  to all the other processors. This can be executed, for instance, with the use of the function `MPI_Allgather` of MPI library.

The general scheme of informational interactions among subtasks in the course of computation is shown in Figure 7.2.



**Figure 7.2.** Computation scheme for parallel matrix-vector multiplication based on rowwise striped matrix decomposition

### 7.6.2. Scaling and Subtask Distribution among Processors

In the process of matrix-vector multiplication the number of computational operations for computing the inner product is the same for all the basic subtasks. Therefore, in case when the number of processors  $p$  is less than the number of basic subtasks  $m$ , we can combine the basic subtasks in such a way that each processor would execute several of these tasks. In this case each subtask will hold a row stripe of the matrix  $A$ . After completing computations, each aggregated basic subtask determines several elements of the result vector  $c$ .

Subtasks distribution among the processors of the computer system may be performed in an arbitrary way.

### 7.6.3. Efficiency Analysis

To analyze the efficiency of parallel computations, two kinds of estimations will be formed henceforward. To form the first type of them algorithm complexity is measured by the number of computational operations that are necessary for solving the given problem (without taking into account the overhead caused by data communications among the processors); the duration of all elementary computational operations (for instance, addition and multiplication) is considered to be the same. Besides, the obtained constants are not taken into consideration in relations. It provides to obtain the order of algorithm complexity and, as a result, in most cases such estimations are rather simple and they can be used for the initial efficiency analysis of the developed parallel algorithms and methods.

The second type of estimation is aimed at forming as many exact relationships for predicting the execution time of algorithms as possible. Such estimations are usually obtained with the help of refinement of the expressions resulting from the first stage. For that purpose the parameters, which determine the execution time, are introduced in the existing relations; time complexity of communication operations are estimated; all the necessary constants are stated. The accuracy of the obtained expressions is examined with the help of computational experiments. On the basis of their results the time of executed computations is compared to the theoretically predicted estimation of the execution time. As a result, such estimations are, as a rule, more complex, but they make it possible to estimate the efficiency of the developed parallel computation methods more precisely.

Let us consider the time complexity of the algorithm of matrix-vector multiplication. If matrix  $A$  is square ( $m=n$ ), the sequential algorithm of matrix-vector multiplication has the complexity  $T_1=n^2$ . In case of parallel computations each processor performs multiplication of only a part (stripe) of the matrix  $A$  by the vector  $b$ . The size

of these stripes is equal to  $n/p$  rows. In case of computing the inner product of one matrix row by a vector, it is necessary to perform the  $n$  multiplications and  $(n-1)$  additions. Therefore, computational complexity of the parallel algorithm is determined as:

$$T_p = n^2/p. \quad (7.5)$$

Taking into account this estimation, the criteria of speedup and efficiency of the parallel algorithm are:

$$S_p = \frac{n^2}{n^2/p} = p, \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \quad (7.6)$$

The estimations (7.5), (7.6) of the computation execution time are expressed in the number of operations. Besides, they are formed without taking into consideration the execution of data communication operations. Let us use the above mentioned assumptions that the executed multiplications and additions are of equal duration  $\tau$ . Besides, let us assume that the computer system is homogeneous, i.e. all the processors of the system have the same performance. With regard to the introduced assumptions, the computation time of the parallel algorithm is:

$$T_p(\text{calc}) = \lceil n/p \rceil \cdot (2n-1) \cdot \tau$$

( $\lceil \cdot \rceil$  is denoted rounding up to the nearest integer number).

The estimation of the all gather operation efficiency was performed in Subsection 6.3.4. As it has been mentioned before, this operation can be executed in  $\lceil \log_2 p \rceil$  iterations<sup>1)</sup>. At the first iteration the interacting pairs of processors exchange messages of size  $w \lceil n/p \rceil$  ( $w$  is the size of one element of the vector  $c$  in bytes). At the second iteration the size becomes doubled and is equal to  $2w \lceil n/p \rceil$  etc. As a result, the all gather operation execution time when the Hockney model is used can be represented as:

$$T_p(\text{comm}) = \sum_{i=1}^{\lceil \log_2 p \rceil} (\alpha + 2^{i-1} w \lceil n/p \rceil / \beta) = \alpha \lceil \log_2 p \rceil + w \lceil n/p \rceil (2^{\lceil \log_2 p \rceil} - 1) / \beta, \quad (7.7)$$

where  $\alpha$  is the latency of data communication network,  $\beta$  is the network bandwidth. Thus, the total time of parallel algorithm execution is

$$T_p = (n/p) \cdot (2n-1) \cdot \tau + \alpha \cdot \log_2 p + w(n/p)(p-1)/\beta. \quad (7.8)$$

(to simplify the expression, it was assumed that the values  $n/p$  and  $\log_2 p$  are whole numbers).

#### 7.6.4. Program Implementation

Let us take a possible variant of parallel program for a matrix- vector multiplication with the use of the algorithm of rowwise matrix partitioning. The realization of separate modules is not given, if their absence does not influence the process of understanding of general scheme of parallel computations.

**1. The main program function.** The main program function realizes the logic of the algorithm operations and sequentially calls out the necessary subprograms.

```
// Program 7.1
// Multiplication of a matrix by a vector - stripe horizontal partitioning
// (the source and the result vectors are doubled among the processors)
int ProcRank;           // Rank of current process
int ProcNum;            // Number of processes
void main(int argc, char* argv[]) {
    double* pMatrix;    // The first argument - initial matrix
    double* pVector;    // The second argument - initial vector
    double* pResult;    // Result vector for matrix-vector multiplication
    int Size;           // Sizes of initial matrix and vector
    double* pProcRows;
    double* pProcResult;
    int RowNum;
    double Start, Finish, Duration;

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

<sup>1)</sup> Let us assume that the topology of the computer system allows to carry out this efficient method of all gather operation (it is possible, in particular, if the structure of the data communication network is a hypercube or a complete graph).

```

    ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
        Size, RowNum);

    DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

    ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

    ResultReplication(pProcResult, pResult, Size, RowNum);

    ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);

    MPI_Finalize();
}

```

**2. ProcessInitialization.** This function defines the initial data for matrix A and vector b. The values for matrix A and vector b are formed in function *RandomDataInitialization*.

```

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, double* &pProcRows, double* &pProcResult,
    int &Size, int &RowNum) {
    int RestRows; // Number of rows, that haven't been distributed yet
    int i;        // Loop variable

    if (ProcRank == 0) {
        do {
            printf("\nEnter size of the initial objects: ");
            scanf("%d", &Size);
            if (Size < ProcNum) {
                printf("Size of the objects must be greater than
                    number of processes! \n ");
            }
        }
        while (Size < ProcNum);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    RestRows = Size;
    for (i=0; i<ProcRank; i++)
        RestRows = RestRows-RestRows/(ProcNum-i);
    RowNum = RestRows/(ProcNum-ProcRank);

    pVector = new double [Size];
    pResult = new double [Size];
    pProcRows = new double [RowNum*Size];
    pProcResult = new double [RowNum];

    if (ProcRank == 0) {
        pMatrix = new double [Size*Size];
        RandomDataInitialization(pMatrix, pVector, Size);
    }
}

```

**3. DataDistribution.** *DataDistribution* pushes out vector b and distributes the rows of initial matrix A among the processes of the computational system. It should be noted that in case when the number of matrix rows  $n$  is not divisible by the number of processors  $p$ , the amount of data transferred for the processes may appear to be different. In this case it is necessary to use function *MPI\_Scatterv* of MPI library for message passing.

```

// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
    int Size, int RowNum) {
    int *pSendNum; // the number of elements sent to the process
    int *pSendInd; // the index of the first data element sent to the process
    int RestRows=Size; // Number of rows, that haven't been distributed yet

```

```

MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Alloc memory for temporary objects
pSendInd = new int [ProcNum];
pSendNum = new int [ProcNum];

// Define the disposition of the matrix rows for current process
RowNum = (Size/ProcNum);
pSendNum[0] = RowNum*Size;
pSendInd[0] = 0;
for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
}

// Scatter the rows
MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Free the memory
delete [] pSendNum;
delete [] pSendInd;
}

```

It should be noted that such separation of initial data generalization and initial data broadcast among processes might not be justified in real parallel computations with big amounts of data. The approach, which is widely used in such cases, consists in arranging data transfer to the processes immediately after the data of the processors are generated. The decrease of memory resources needed for data storage may be achieved also at the expense of data generation in the last process ( in case of such approach the memory for the transferred data and for the process data may be the same).

**4. ParallelResultCalculation.** *ResultCalculation* performs the multiplication of the matrix rows, which are at a given moment distributed to a given process, by a vector. Thus, the function forms the block of the result vector *c*.

```

// Function for calculating partial matrix-vector multiplication
void ParallelResultCalculation(double* pProcRows, double* pVector, double*
pProcResult, int Size, int RowNum) {
    int i, j; // Loop variables
    for (i=0; i<RowNum; i++) {
        pProcResult[i] = 0;
        for (j=0; j<Size; j++)
            pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
    }
}

```

**5. ResultReplication.** This function unites the blocks of the result vector *c*, which have been obtained on different processors and replicates the result vector to all the computational system processes.

```

// Function for gathering the result vector
void ResultReplication(double* pProcResult, double* pResult, int Size,
    int RowNum) {
    int i; // Loop variable
    int *pReceiveNum; // Number of elements, that current process sends
    int *pReceiveInd; /* Index of the first element from current process
                        in result vector */
    int RestRows=Size; // Number of rows, that haven't been distributed yet

    //Alloc memory for temporary objects
    pReceiveNum = new int [ProcNum];
    pReceiveInd = new int [ProcNum];

    //Define the disposition of the result vector block of current processor
    pReceiveInd[0] = 0;
    pReceiveNum[0] = Size/ProcNum;
    for (i=1; i<ProcNum; i++) {

```

```

RestRows -= pReceiveNum[i-1];
pReceiveNum[i] = RestRows/(ProcNum-i);
pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
}
//Gather the whole result vector on every processor
MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank], MPI_DOUBLE, pResult,
pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

//Free the memory
delete [] pReceiveNum;
delete [] pReceiveInd;
}

```

### 7.6.5. Computational Experiment Results

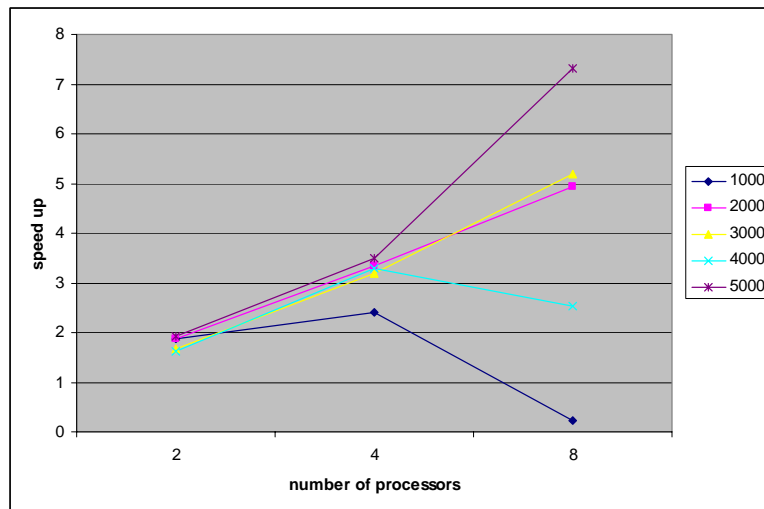
Let us analyze the results of the computational experiments carried out in order to estimate the efficiency of the discussed parallel algorithm of matrix-vector multiplication. Besides, the obtained results will be used for the comparison of the theoretical estimations and experimental values of the computation time. Thus, the accuracy of the obtained analytical relations will be checked. The experiments were carried out on the computational cluster on the basis of the processors Intel XEON 4 EM64T, 3000 Mhz and the network Gigabit Ethernet under OS Microsoft Windows Server 2003 Standard x64 Edition.

Now let us describe the way the parameters of the theoretical dependencies (values  $\tau$ ,  $w$ ,  $\alpha$ ,  $\beta$ ) were evaluated. To estimate the duration  $\tau$  of the basic scalar computational operation, we solved the problem of matrix-vector multiplication using the sequential algorithm. The computation time obtained by this method was divided into the total number of the operations performed. As a result of the experiments the value of  $\tau$  was equal to 1.93 nsec. The experiments carried out in order to determine the data communication network parameters demonstrated the value of latency  $\alpha$  and bandwidth  $\beta$  correspondingly 47 msec and 53.29 Mbyte/sec. All the 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 7.1. The experiments were carried out with the use of 2, 4 and 8 processors. The algorithm execution time is given in seconds.

**Table 7.1.** The results of the computational experiments for the parallel algorithm of matrix-vector multiplication with rowwise block-striped data decomposition

Matrix Size	Sequential Algorithm	Parallel Algorithm					
		2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
1000	0,0041	0,0021	1,8798	0,0017	2,4089	0,0175	0,2333
2000	0,016	0,0084	1,8843	0,0047	3,3388	0,0032	4,9443
3000	0,031	0,0185	1,6700	0,0097	3,1778	0,0059	5,1952
4000	0,062	0,0381	1,6263	0,0188	3,2838	0,0244	2,5329
5000	0,11	0,0574	1,9156	0,0314	3,4993	0,0150	7,3216



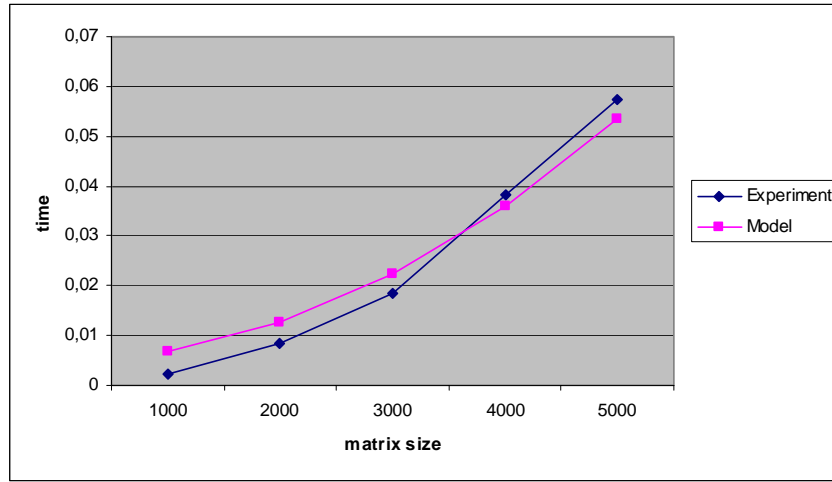
**Figure 7.3.** Speedup for parallel matrix-vector multiplication (rowwise block-striped matrix decomposition)



The comparison of the experiment execution time  $T_p^*$  and the theoretical time  $T_p$  calculated in accordance with the expression (7.8) is shown in Table 7.2. It is also shown graphically in Figures 7.3 and 7.4.

**Table 7.2.** The comparison of the experimental and theoretical execution time for parallel algorithm of matrix-vector multiplication based on rowwise matrix decomposition

Matrix Size	2 processors		4 processors		6 processors	
	$T_p$ (model)	$T_p^*$	$T_p$ (model)	$T_p^*$	$T_p$ (model)	$T_p^*$
1000	0,0069	0,0021	0,0108	0,0017	0,0152	0,0175
2000	0,0132	0,0084	0,0140	0,0047	0,0169	0,0032
3000	0,0235	0,0185	0,0193	0,0097	0,0196	0,0059
4000	0,0379	0,0381	0,0265	0,0188	0,0233	0,0244
5000	0,0565	0,0574	0,0359	0,0314	0,0280	0,0150



**Figure 7.4.** Experimental and theoretical execution time with respect to matrix size (rowwise block-striped matrix decomposition, 2 processors)

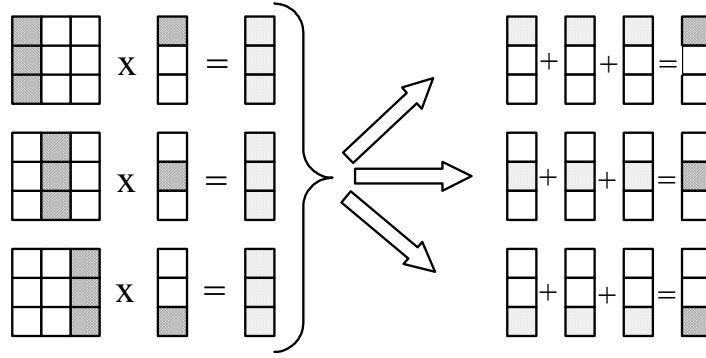
## 7.7. Matrix-Vector Multiplication in Case of Columnwise Data Decomposition

Let us analyze the other approach to parallel matrix-vector multiplication, which is based on partitioning the matrix into continuous sets (vertical stripes) of columns.

### 7.7.1. Computation Decomposition and Analysis of Information Dependencies

In case of columnwise matrix decomposition the operation of multiplying a column of matrix  $A$  by one of the vector  $b$  elements may be chosen as the basis computational subtask. As a result to perform computations each basic subtask  $i$ ,  $0 \leq i < n$ , must contain the  $i$ -th column of matrix  $A$  and the  $i$ -th elements  $b_i$  and  $c_i$  of vectors  $b$  and  $c$ .

At the starting point of the parallel algorithm of matrix-vector multiplication each basic task  $i$  carries out the multiplication of its matrix  $A$  column by element  $b_i$ . As a result, vector  $c'(i)$  (the vector of intermediate results) is obtained in each subtask. The subtasks must further exchange their intermediate data in order to obtain the elements of the result vector  $c$  (element  $j$ ,  $0 \leq j < n$ , of the partial result  $c'(i)$  of the subtask  $i$ ,  $0 \leq i < n$ , must be sent to the subtask  $j$ ). This *all-to-all communication* or *total exchange* is the most general communication procedure and may be executed with the help of the function `MPI_Alltoall` of MPI library. After the completion of data communications each basic subtask  $i$ ,  $0 \leq i < n$ , will contain  $n$  partial values  $c'_i(j)$ ,  $0 \leq j < n$ . Element  $c_i$  of the result vector  $c$  is determined after the addition of the partial values (see Figure 7.5).



**Figure 7.5.** Computation scheme for parallel matrix-vector multiplication based on columnwise striped matrix decomposition

### 7.7.2. Scaling and Subtask Distribution among Processors

The selected basic subtasks are of equal computational intensity and have the same amount of the data transferred. If the number of matrix columns exceeds the number of processors, the basic subtasks may be aggregated by uniting several neighboring columns within one subtask. In this case, the initial matrix  $A$  is partitioned into a number of vertical stripes. If all the stripe sizes are the same the above discussed method of computation aggregating provides equal distribution of the computational load among the processors.

As with the previous algorithm, the subtasks may be arbitrarily distributed among the computer system processors.

### 7.7.3. Efficiency Analysis

As previously, let matrix  $A$  be square, i.e.  $m=n$ . At the first stage of computations each processor multiplies its matrix columns by the vector  $b$  elements. The obtained values are summed for each matrix row separately:

$$c'_s(i) = \sum_{j=j_0}^{j_{i-1}} a_{sj} b_j, \quad 0 \leq s < n, \quad (7.9)$$

( $j_0$  and  $j_{i-1}$  are the initial and the final column indices of the basic subtasks  $i$ ,  $0 \leq i < n$ ). As the sizes of matrix stripes and the block of the vector  $b$  are equal  $n/p$ , the time complexity of such computations may be estimated as  $T' = n^2/p$  operations. After the subtasks have exchanged the data at the second stage of computations each processor sums the obtained values up for its block of the result vector  $c$ . The number of the summed values for each element  $c_i$  of vector  $c$  coincides with the number of processors  $p$ . The size of result vector block is equal to  $n/p$ . Thus, the number of operations carried out for the second stage appears to be equal to  $T'' = n$ . With regards to the relations obtained the speedup and the efficiency of the parallel algorithm may be expressed as follows:

$$S_p = \frac{n^2}{n^2/p} = p, \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \quad (7.10)$$

Now let us consider more accurate relations for estimation of the time of parallel algorithm execution. With regard to the above discussion the execution time of the parallel algorithm computations may be estimated by means of the following expression:

$$T_p(\text{calc}) = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau. \quad (7.11)$$

(here, as previously,  $\tau$  is the execution time for one basic scalar computational operation).

Let us discuss two possible methods to carry out the total exchange (see also Section 3). The first method is provided by the algorithm, according to which each processor sends its data to all the rest of computer system processors sequentially. Let us assume that the processors may simultaneously send and receive messages, and there is a direct communication line between any pair of processors. The time complexity estimation (execution time) of the total exchange algorithm may be written as follows:

$$T_p^1(\text{comm}) = (p-1)(\alpha + w \lceil n/p \rceil / \beta). \quad (7.12)$$

(where  $\alpha$  is the communication network latency,  $\beta$  is the network bandwidth,  $w$  is the data element size in bytes)

The second method of carrying out the total exchange is considered in Section 3 (the computational network topology should be represented as a hypercube). As it has been shown above, the algorithm may be executed in

$\lceil \log_2 p \rceil$  steps. At each step each processor sends and receives a message of  $n/2$  elements. As a result, the execution time of data communications is:

$$T_p^2(comm) = \lceil \log_2 p \rceil (\alpha + w(n/2)/\beta). \quad (7.13)$$

With regards to the obtained relations the total execution time for the parallel algorithm of matrix-vector multiplication in case of columnwise data decomposition is expressed by the following expressions:

- For the first method of data communications:

$$T_p^1 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + (p-1)(\alpha + w \lceil n/p \rceil / \beta). \quad (7.14)$$

- For the second method of data communications:

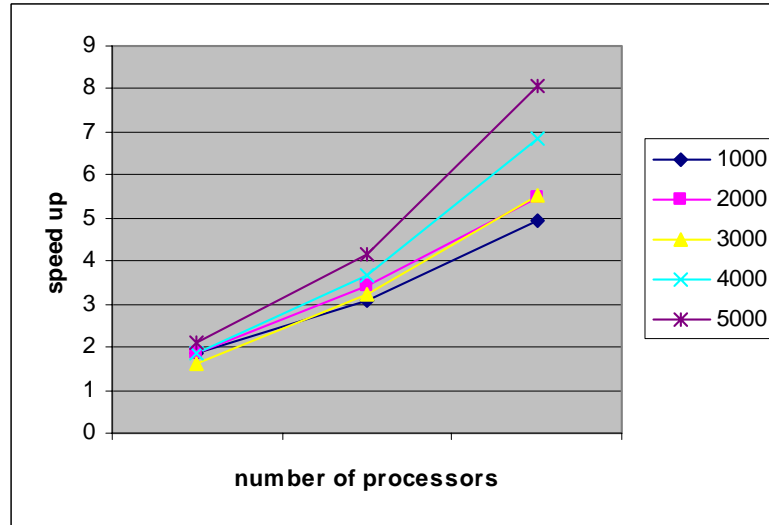
$$T_p^2 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + \lceil \log_2 p \rceil (\alpha + w(n/2)/\beta). \quad (7.15)$$

#### 7.7.4. Computational Experiment Results

The computational experiments for estimating the efficiency of the parallel algorithm of matrix-vector multiplication in case of columnwise matrix decomposition were carried out under the conditions given in Section 7.6.5. The results of the computational experiments are given in Table 7.3.

**Table 7.3.** The results of the computational experiments for parallel matrix-vector multiplication algorithm based on columnwise matrix decomposition

Matrix Size	Sequential Algorithm	2 processors		4 processors		8 processors	
		Time	Speed up	Time	Speed up	Time	Speed up
1000	0,0041	0,0022	1,8352	0,0132	0,3100	0,0008	4,9409
2000	0,016	0,0085	1,8799	0,0046	3,4246	0,0029	5,4682
3000	0,031	0,019	1,6315	0,0095	3,2413	0,0055	5,5456
4000	0,062	0,0331	1,8679	0,0168	3,6714	0,0090	6,8599
5000	0,11	0,0518	2,1228	0,0265	4,1361	0,0136	8,0580



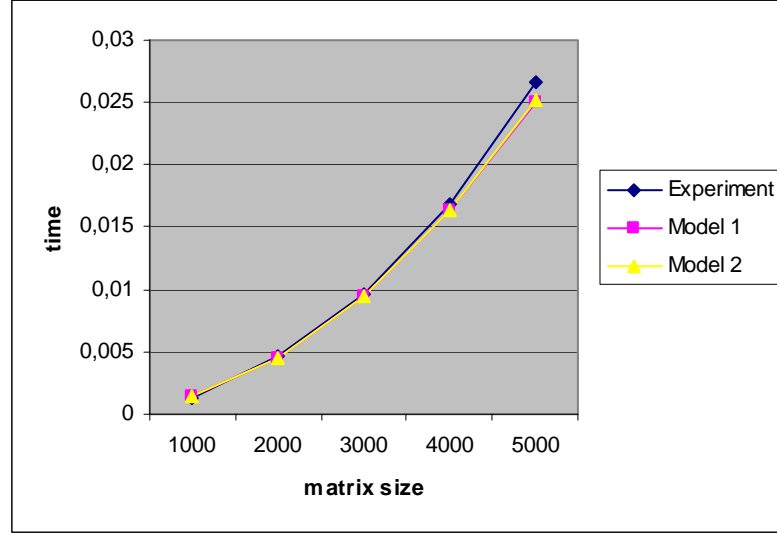
**Figure 7.6.** Speedup for parallel matrix-vector multiplication (columnwise block-striped matrix decomposition)

The comparison of the experiment execution time  $T_p^*$ , and time  $T_p$ , calculated according to relations (7.14), (7.15), are shown in Table 7.4 and in Figures 7.6 and 7.7. The theoretical time  $T_p^1$  is calculated according to (7.14), and the theoretical time  $T_p^2$  is calculated in accordance with (7.15).

**Table 7.4.** The comparison of the experimental and the theoretical time for parallel matrix-vector multiplication algorithm based on columnwise matrix decomposition

Matrix Size	2 processors			4 processors			8 processors		
	$T_p^1$	$T_p^2$	$T_p^*$	$T_p^1$	$T_p^2$	$T_p^*$	$T_p^1$	$T_p^2$	$T_p^*$

1000	0,0021	0,0021	0,0022	0,0014	0,0013	0,0013	0,0015	0,0011	0,0008
2000	0,0080	0,0080	0,0085	0,0044	0,0044	0,0046	0,0031	0,0027	0,0029
3000	0,0177	0,0177	0,019	0,0094	0,0094	0,0095	0,0056	0,0054	0,0055
4000	0,0313	0,0313	0,0331	0,0162	0,0163	0,0168	0,0091	0,0090	0,0090
5000	0,0487	0,0487	0,0518	0,0251	0,0251	0,0265	0,0136	0,0135	0,0136



**Figure 7.7.** Experimental and theoretical execution time with respect to matrix size (columnwise block-striped matrix decomposition, 4 processors)

## 7.8. Matrix-Vector Multiplication in Case of Checkerboard Data Decomposition

Let us consider the parallel matrix-vector multiplication algorithm, which is based on the other method of data decomposition, i.e. in case when the matrix is subdivided into rectangular fragments (blocks).

### 7.8.1. Computation Decomposition

Checkerboard block scheme of matrix decomposition is described in detail in Subsection 7.2. Under this method of data decomposition the matrix  $A$  is presented as a set of rectangular blocks:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots A_{0q-1} \\ & \dots & \\ A_{s-11} & A_{s-12} & \dots A_{s-1q-1} \end{pmatrix},$$

where  $A_{ij}$ ,  $0 \leq i < s$ ,  $0 \leq j < q$ , is the matrix block:

$$A_{ij} = \begin{pmatrix} a_{i_0j_0} & a_{i_0j_1} & \dots a_{i_0j_{l-1}} \\ & \dots & \\ a_{i_{k-1}j_0} & a_{i_{k-1}j_1} & \dots a_{i_{k-1}j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \leq v < k, k = m/s, j_u = jl + u, 0 \leq u < l, l = n/q$$

(as previously, we assume here that  $p = s \cdot q$ , the number of matrix rows is divisible by  $s$ , and the number of columns is divisible by  $q$ , i.e.  $m = k \cdot s$  and  $n = l \cdot q$ ).

In case of checkerboard block decomposition of matrix  $A$ , it is convenient to define the basic computational subtasks on the basis of computations carried over the matrix blocks. For subtask enumeration it is possible to use the indices of the blocks of the matrix  $A$  contained in the subtasks, i.e. the subtask  $(i, j)$  holds the block  $A_{ij}$ . Beside the matrix block each subtask must contain also a block of the vector  $b$ . Certain correspondence rules must be observed for the blocks of the same subtask. Namely, the multiplication of the block  $A_{ij}$  may be performed only if block  $b'(i, j)$  of the vector  $b$  looks as follows:

$$b'(i, j) = (b'_0(i, j), \dots, b'_{l-1}(i, j)), \text{ where } b'_u(i, j) = b_{j_u}, j_u = jl + u, 0 \leq u < l, l = n/q.$$

### 7.8.2. Analysis of Information Dependencies

Let us consider the general scheme of parallel computations for matrix-vector multiplication based on checkerboard block decomposition. After multiplication of matrix blocks by blocks of the vector  $b$  each subtask  $(i,j)$  will contain the partial results vector  $c'(i,j)$ , which is determined according to the following expressions:

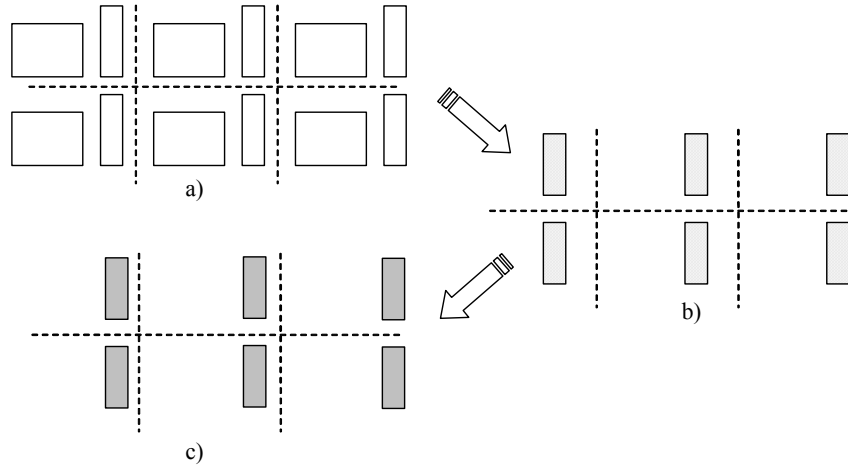
$$c'_v(i,j) = \sum_{u=0}^{l-1} a_{i_v j_u} b_{j_u}, i_v = ik + v, 0 \leq v < k, k = m/s, j_u = jl + u, 0 \leq u \leq l, l = n/q.$$

Element by element addition of vectors of partial results for each horizontal stripe (this procedure is often referred to as reduction – see section 3) of matrix blocks leads to obtaining the result vector  $c$ :

$$c_\eta = \sum_{j=0}^{q-1} c'_v(i,j), 0 \leq \eta < m, i = \eta/s, v = \eta - i \cdot s.$$

To allocate the vector  $c$  we will use the same scheme that was used for the vector  $b$ . Let us organize the computations in such a way that after the computation termination the vector  $c$  would be distributed in each of the vertical stripes of matrix blocks. Hence, each block of the vector  $c$  must be duplicated on each horizontal stripe. Addition of partial results and duplicated of the result vector blocks are the necessary operations. They may be provided with the help of the function `MPI_Allreduce` of MPI library.

The general scheme of the performed computations for matrix-vector multiplication in case of checkerboard block data distribution is shown in Figure 7.8.



**Figure 7.8.** General scheme of parallel matrix-vector multiplication based on checkerboard block decomposition: a) the initial result of data distribution, b) the distribution of partial result vectors, c) the block distribution of result vector  $c$

On the basis of analysis of this parallel computation scheme it is possible to draw the conclusion that the information dependencies of the basic subtasks appears only at the stage of summing the results of multiplication of matrix blocks by blocks of the vector  $c$ . These computations may be performed according to the cascade scheme (see Section 2). As a result, the structure of the available information communications for the subtasks of the same horizontal block stripe corresponds to the topology of the binary tree.

### 7.8.3. Scaling and Distributing Subtasks among Processors

The size of matrix blocks may be chosen in such a way that the total number of the basic subtasks would coincide with the number of processors  $p$ . Thus, for instance, if the size of the block grid is defined as  $p = s \cdot q$ , then

$$k = m/s, l = n/q,$$

where  $k$  and  $l$  are the number of rows and columns in matrix blocks. This method of defining the block sizes makes the amount of computations in each subtask be the same. Hence, complete balance of computational load among processors is achieved.

Choice is also possible in defining the sizes of matrix block structure. A great number of horizontal blocks leads to the increase of the number of iterations in the reduction of block multiplication results. The increase of the vertical size of the block grid enhances the amount of the data transmitted among the processors. The simple frequently used solution consists in the use of the same number of blocks both vertically and horizontally, i.e.

$$s = q = \sqrt{p}.$$

It should be noted, that the checkerboard block scheme of data decomposition can be considered as the generalization of all the approaches discussed in this Section. Actually, if  $q=1$  the block grid is reduced to the set of horizontal stripes, if  $s=1$  the matrix  $A$  is partitioned into the set of vertical stripes.

The possibility to carry out data reduction efficiently must be taken into account in solving the problem of distributing subtasks among the processors. A suitable approach may consist in distributing the subtasks of the same horizontal stripes among processors for which the network topology is a hypercube or a complete graph.

#### 7.8.4. Efficiency Analysis

Let us carry out the efficiency analysis for the parallel matrix-vector multiplication algorithm making the usual assumptions that matrix  $A$  is square, i.e.  $m=n$ . We will also assume that the network topology is rectangular grid  $p=s \times q$  ( $s$  is the number of rows in the processor grid and  $q$  is the number of columns).

The general efficiency analysis leads to ideal parallel algorithm characteristics:

$$S_p = \frac{n^2}{n^2/p} = p, \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \quad (7.15)$$

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.

The total time of multiplication of matrix blocks by the vector  $b$  may be evaluated as follows:

$$T_p(calc) = \lceil n/s \rceil \cdot (2 \cdot \lceil n/q \rceil - 1) \cdot \tau. \quad (7.16)$$

Data reduction may be performed with the use of the cascade scheme. Thus, it includes  $\log_2 q$  iterations of transmitting messages of  $w \lceil n/s \rceil$  size. As a result, the estimation of communication overhead in accordance with the Hockney model may be evaluated by means of the following expression:

$$T_p(comm) = (\alpha + w \lceil n/s \rceil / \beta) \lceil \log_2 q \rceil. \quad (7.17)$$

Thus, the total time of carrying out parallel algorithm of matrix-vector multiplication based on checkerboard block decomposition is the following:

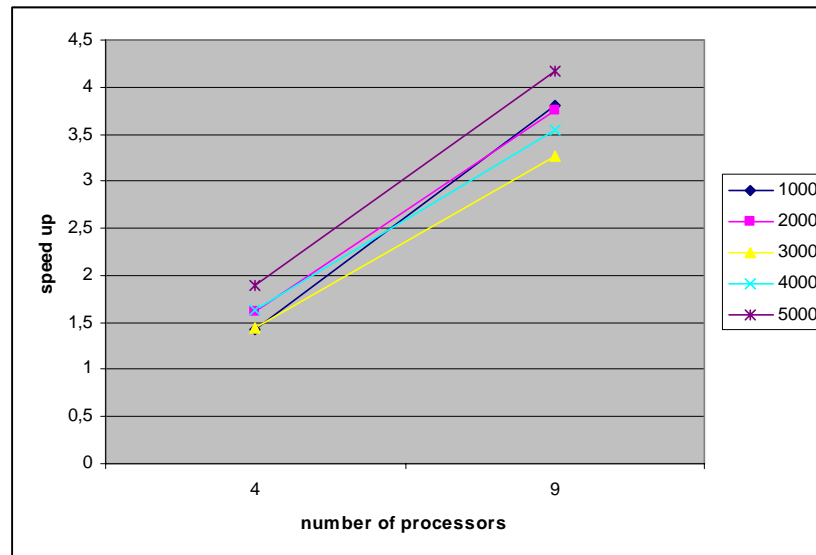
$$T_p = \lceil n/s \rceil \cdot (2 \cdot \lceil n/q \rceil - 1) \cdot \tau + (\alpha + w \lceil n/s \rceil / \beta) \lceil \log_2 q \rceil. \quad (7.18)$$

#### 7.8.5. Computational Experiment Results

The computational experiments to estimate parallel algorithm efficiency we carried out under the same conditions that were observed for the previous computations (see Subsection 7.6.5). The results of the experiments are shown in Table 7.5 and in Figure 7.9. The computations were carried out with the use of 4 and 9 processors.

**Table. 7.5.** The results of the computational experiments for parallel matrix-vector multiplication algorithm based on checkerboard block decomposition

Matrix Size	Sequential Algorithm	Parallel Algorithm			
		4 processors		9 processors	
		Time	Speed Up	Time	Speed Up
1000	0,0041	0,0028	1,4260	0,0011	3,7998
2000	0,016	0,0099	1,6127	0,0042	3,7514
3000	0,031	0,0214	1,4441	0,0095	3,2614
4000	0,062	0,0381	1,6254	0,0175	3,5420
5000	0,11	0,0583	1,8860	0,0263	4,1755

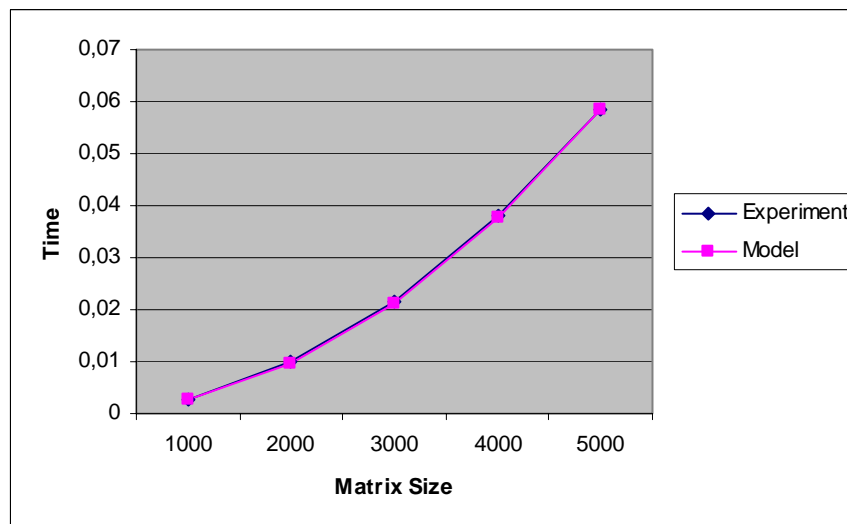


**Figure 7.9.** Speedup for parallel matrix-vector multiplication (checkerboard block matrix decomposition)

The comparison of the experiment execution time  $T_p^*$  the experiment and the theoretical time  $T_p$ , calculated in accordance with the expression (7.18), is given in Table 7.6 and Figure 7.10.

**Table 7.6.** The comparison of the experimental and the theoretical time for executing parallel matrix-vector multiplication in case of checkerboard block decomposition

Matrix Size	4 processors		9 processors	
	$T_p$	$T_p^*$	$T_p$	$T_p^*$
1000	0,0025	0,0028	0,0012	0,0010
2000	0,0095	0,0099	0,0043	0,0042
3000	0,0212	0,0214	0,0095	0,0095
4000	0,0376	0,0381	0,0168	0,0175
5000	0,0586	0,0583	0,0262	0,0263



**Figure 7.10.** Experimental and theoretical execution time with respect to matrix size (checkerboard block matrix decomposition, 4 processors)

## 7.9. Summary

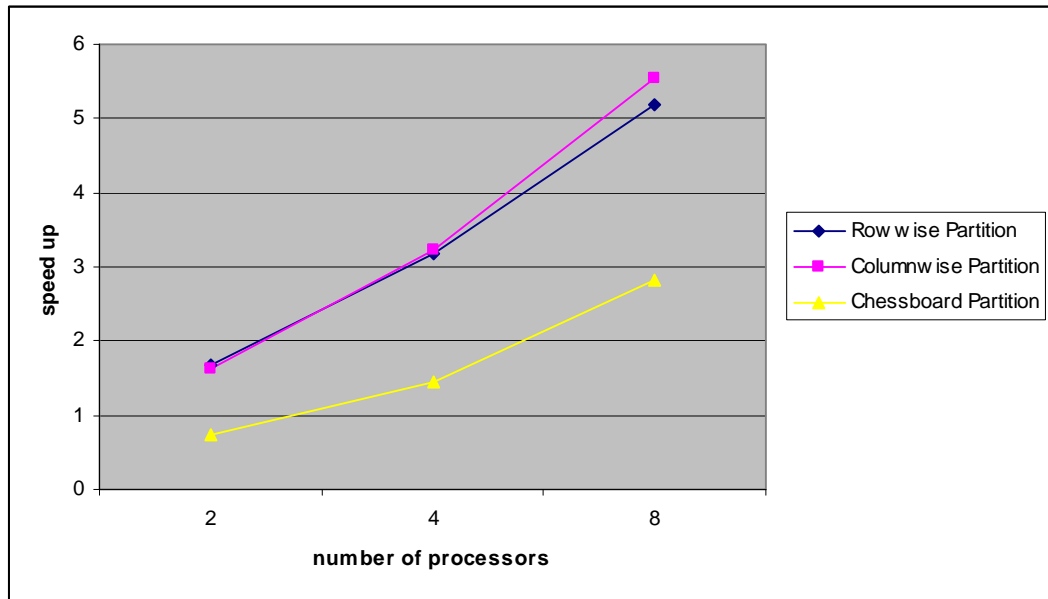
The Section discusses the possible schemes of matrix decomposition among the processors of a multiprocessor computer system in the problems of matrix-vector multiplication. Among the schemes discussed there are the

methods of matrix partitioning into *stripes* (vertically and horizontally) or matrix partitioning into the rectangular fragments (*blocks*).

The matrix decomposition schemes produce three possible approaches of carrying out matrix-vector multiplication in parallel. The first algorithm is based on *the rowwise striped matrix distribution* among the processors. The second one is based on *the columnwise striped matrix partitioning*. And the last one is based on *the checkerboard block distribution*. Each algorithm is presented in accordance with the general scheme described in Section 6. First the basic subtasks are defined. Then the information dependencies of the subtasks are analyzed, and after that scaling and distributing subtasks among the processors are considered. In conclusion each algorithm is provided with the analysis of the parallel computation efficiency. The results of the computational experiments are also given. The Section describes a sample of software implementation for the matrix-vector multiplication in case of the rowwise striped distribution.

The efficiency characteristics obtained in the course of the experiments demonstrate that all the methods of data partitioning lead to equal balance of the computational load. The differences concern only the time complexity of the information communications among the processors. In this respect it is very important to analyze the way in which the choice of the data distribution method influences the type of the necessary data communication operations. It is also interesting to find out the main differences in the communication operations of the algorithms. The selection of the most suitable network communication topology among the processors is also very important for performing the corresponding parallel algorithm efficiently. Thus, for instance, the algorithms based on the block-striped data distribution are designed for the network topology in the form of a hypercube or a complete graph. The availability of the grid topology is necessary for implementation of the algorithm based on the checkerboard block data decomposition.

The speedup values experimentally obtained for the discussed algorithms are shown in the Figure 7.11. The computations have shown that increasing the number of processors improves the checkerboard block multiplication algorithm efficiency.



**Figure 7.11.** Speedup of parallel multiplication algorithms according with computational experiments (size of the matrix  $A$  and the vector  $b$  is equal to 2000)

## 7.10. References

The problem of matrix-vector multiplication is frequently used as an example of parallel programming and, as a result, is widely discussed. The books by Kumar, et al. (1994) and Quinn (2004) may be recommended as additional materials on the problem. Parallel matrix computations are discussed in detail 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.

## 7.11. Discussions

1. What are the main methods of distributing matrix elements among processors?
2. What is the statement of the matrix-vector multiplication problem?
3. What is the computational complexity of the sequential matrix-vector multiplication?



4. Why is it admissible to duplicate the vector-operand to all the processors in developing a parallel algorithm of matrix-vector multiplication?
5. What approaches of the development of parallel algorithms may be suggested?
6. Describe the general schemes of the parallel algorithms discussed in the Section.
7. Evaluate the efficiency characteristics for one of the algorithms discussed in the Section?
8. Which of the algorithms has the best speedup and efficiency?
9. Can the use of the cyclic data distribution scheme influence the execution time of each of the algorithms?
10. What information communications are carried out for the algorithms in case of block-striped data distribution scheme? What is the difference between the data communications in case of rowwise matrix distribution and those required in case of columnwise distribution?
11. What information communications are performed for the checkerboard block matrix-vector multiplication algorithm?
12. What kind of communication network topology is adequate for each algorithm discussed in the Section?
13. Analyze the software implementation of matrix-vector algorithm in case of rowwise data distribution. How may software implementations of the other algorithms differ from each other?
14. What functions of the library MPI appeared to be necessary in the software implementation of the algorithms?

### **7.12. Exercises**

1. Develop the implementation of the parallel algorithm based on the columnwise striped matrix decomposition. Estimate theoretically the algorithm execution time. Carry out the computational experiments. Compare the results of the experiments with the theoretical estimations.
2. Develop the implementation of the parallel algorithm based on the checkerboard block decomposition. Estimate theoretically the algorithm execution time. Carry out the computational experiments. Compare the results of the experiments with the theoretical estimations.

### **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.