The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to MPI

Lecture 6-7. Parallel Methods for Matrix-Vector Multiplication.

Nizhni Novgorod

2014

Lecture_6,7_. Parallel Methods for Matrix-Vector Multiplication

Lectures 6 and 7 describe basic principles of parallel matrix-vector multiplication algorithm construction.

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.

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.

LECTURE 6

6.1. 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 decomposition data into *stripes* (vertically and horizontally) or rectangular fragments (*blocks*).

1. Block-striped matrix decomposition. In case of block-striped decomposition each processor is assigned a certain subset of matrix rows (*rowwise* or *horizontal decomposition*) or matrix columns (*columnwise* or *vertical decomposition*) (Figure 6.1). 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 6.1), for instance, matrix *A* is represented as follows:

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = ik + j,\ 0 \le j < k,\ k = m / p, \quad (6.1)$$

where $a_i = (a_{i1}, a_{i2}, \ldots a_{in}),\ 0 \le i < m,$ 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 decomposition 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 decomposition of matrix A looks as follows:

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = i + jp,\ 0 \le j < k,\ k = m / p. \quad (6.2)$$

2. Checkerboard Block Matrix Decomposition. 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:

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

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

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \ldots a_{i_0 j_{l-1}} \\ & \ldots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix},\ i_v = ik + v,\ 0 \le v < k,\ k = m / s,\ j_u = jl + u,\ 0 \le u \le l,\ l = n / q. \quad (6.3)$$

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.

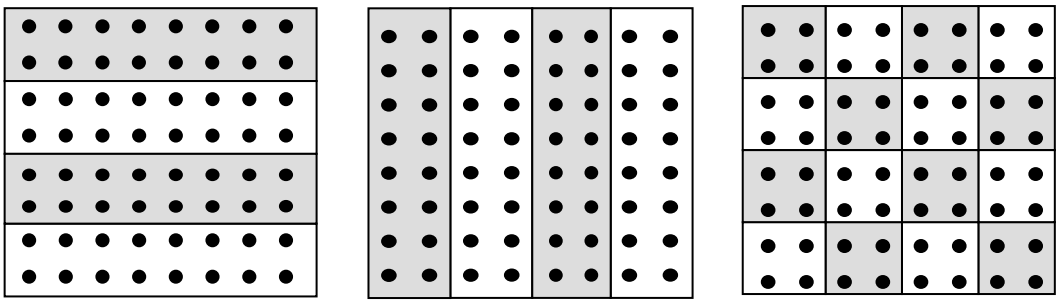

Figure 6.1 Most widely used matrix decomposition schemes

In this lecture 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.

6.2. 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*:

$$c_i = (a_i, b) = \sum_{j=0}^{n-1} a_{ij} b_j \ , \ 0 \le i \le m-1 . \qquad (6.4)$$

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-l* operations). The total number of necessary scalar operations is the value

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

6.3. Sequential Algorithm

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

```
// 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]
  }
}
```

In the given program code the following notation is used:

a. Input data:
   i. A[m][n] – matrix of order $m \times n$,
   ii. b[n] – vector of *n* elements,
b. Result:
   i. 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-l* 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).*

6.4. 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 decomposition 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. $n = k \cdot p$).

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.

6.5. 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.

6.5.1. 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, 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 computationS is shown in Figure 6.2.

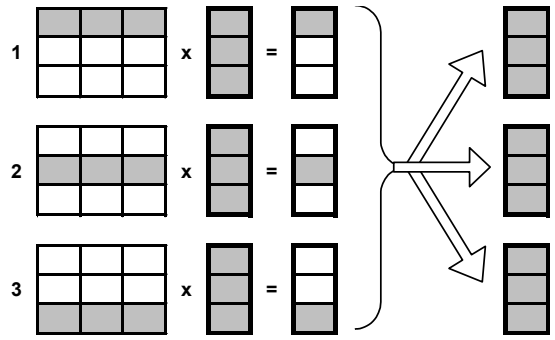

Figure 6.2 Computation scheme for parallel matrix-vector multiplication based on rowwise striped matrix decomposition

6.5.2. 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.

6.5.1. 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 decomposition. 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.

```
// Multiplication of a matrix by a vector – stripe horizontal decomposition
// (the source and the result vectors are doubled amoung 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);

  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. ParallelResultCaculation. *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;
}
```

6.5.2. 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.

The results of the computational experiments are shown in Table 6.1. The experiments were carried out with the use of 2, 4 and 8 processors. The algorithm execution time is given in seconds.

Table 6.1 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

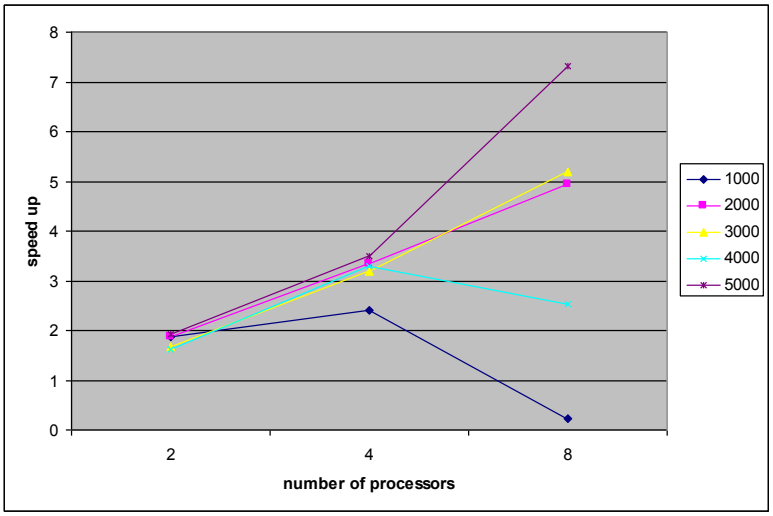

Figure 6.3 Speedup for parallel matrix-vector multiplication (rowwise block-striped matrix decomposition)

6.6. *Review of 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.

LECTURE 7

7.1. Matrix-Vector Multiplication in Case of Columnwise Data Decomposition

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

7.2 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*, $0 \le i < n$, 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*, $0 \le j < n$, of the partial result *c'(i)* of the subtask *i*, $0 \le i < n$, 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*, $0 \le i < n$, will contain *n* partial values $c'_i(j)$, $0 \le j < n$. Element c_i of the result vector *c* is determined after the addition of the partial values (see Figure 7.1).

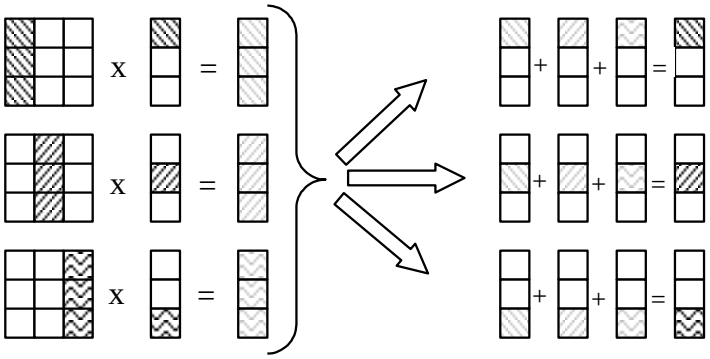

Figure 7.1 Computation scheme for parallel matrix-vector multiplication based on columnwise striped matrix decomposition

7.3 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.

7.4 Computational Experiment Results

The results of the computational experiments are given in Table 7.1.

Table 7.1. 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

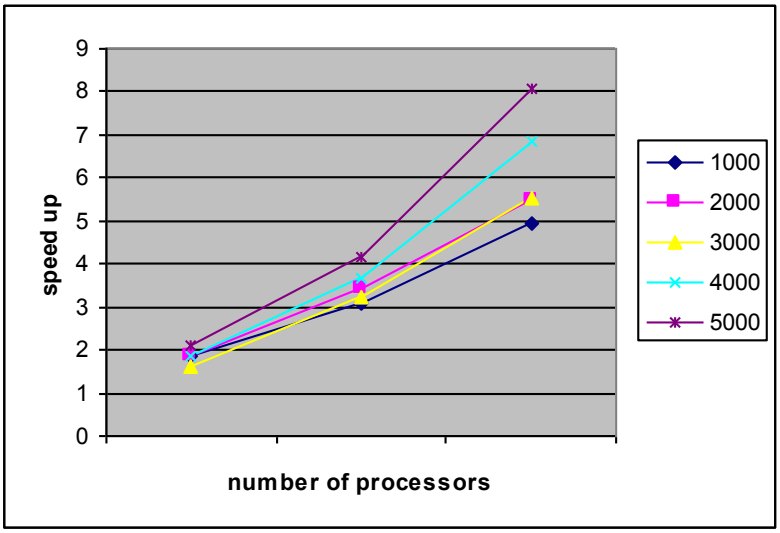

Figure 7.2 Speedup for parallel matrix-vector multiplication (columnwise block-striped matrix decomposition)

Test questions

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. What functions of the library MPI appeared to be necessary in the software implementation of the algorithms?

Practice

1. Develop the implementation of the parallel algorithm based on the columnwise striped matrix decomposition. Carry out computational experiments. Compare actual results to those given in the lecture.

2. Develop the implementation of the parallel algorithm based on checkerboard block matrix decomposition. Carry out computational experiments. Compare actual results to those given in the lecture.

References

1. **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/
2. **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.
3. **Foster**, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.