9. Parallel Methods for Solving Linear Equation Systems

Linear equation systems appear in the course of solving a number of applied problems, which are formulated by differential, integral equations or by systems of non-linear (transcendent) equations. They may appear also in the problems of mathematical programming, statistical data processing, function approximation, or in discretization of boundary differential problems by methods of finite differences or of finite elements, etc.

The coefficient matrices of linear equation systems may be of various structure and have various characteristics. The matrices of the systems solved may be dense and their order may reach several thousands of rows and columns. In solving many problems there can be the systems, which possess symmetric positively definite stripe matrices with the order of tens of thousands and the width of the stripe of the several thousands elements. And finally in consideration of a great number of problems there may appear systems of linear equations with sparse matrices of the order of millions of columns and rows.

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

9.1. Problem Setting

A linear equation with n independent unknowns $x_0, x_1, ..., x_{n-1}$ may be described by means of the expression

$$a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} = b \tag{9.1}$$

where values $a_0, a_1, ..., a_{n-1}$ and b are constant.

A set of n linear equations

$$\begin{array}{llll} a_{0,0}x_0 & +a_{0,1}x_1 & +...+a_{0,n-1}x_{n-1} & = b_0 \\ a_{1,0}x_0 & +a_{1,1}x_1 & +...+a_{1,n-1}x_{n-1} & = b_1 \\ ... & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +...+a_{n-1,n-1}x_{n-1} & = b_{n-1} \end{array} \tag{9.2}$$

is called *a system of linear equations* or *a linear system*. In matrix notation this system may be written as

$$Ax = b \ ,$$

where $A=(a_{i,j})$ is a real matrix of size $n\times n$, and b and x are vectors of n elements.

The problem of solving a linear equation system for the given matrix A and the vector b is considered to be the problem of searching the value of unknown vector x whereby all the system equations hold.

9.2. The Gauss Algorithm

Gauss method is a well known *direct* algorithm of solving systems of linear equations, the coefficient matrices of which are *dense*. If a system of linear equations is nondegenerate, then the Gauss method guarantees solving the problem with the error determined by the computation accuracy. The main concept of the method is a modification of matrix A by means of equivalent transformations (which do not change the solution of system (9.2)) to a triangle form. After that the values of the desired unknown variables may be obtained directly in an explicit form.

The subsection gives the general description of the Gauss method, which is sufficient for its initial understanding and which allows to consider possible methods of parallel computations in solving linear equation systems. A more detailed description and more rigorous consideration of accuracy issues for the obtained solutions may be found, for instance, in Kahaner, Moler and Nash (1988), Bertsekas and Tsitsiklis (1989) etc.

9.2.1. Sequential Algorithm

The Gauss method is based on the possibility to carry out the transformation of linear equations, which do not change the solution of the system under consideration (such transformations are referred to as *equivalent*). They include the following transformations:

- The multiplication of any equation by a nonzero constant,
- The permutation of equations,
- Addition of any system equation to other equation.

The Gauss method includes sequential execution of two stages. At the first stage (*the Gaussian elimination stage*) the initial system of linear equations is reduced to the upper triangle form by means of sequential elimination of unknowns:

$$Ux = c\ ,$$

where the coefficient matrix of the obtained system looks as follows

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}.$$

At *the back substitution* (the second stage of the algorithm) the values of unknown variables are calculated. The value of the variable x_{n-1} may be calculated from the last equation of the transformed system. After that it becomes possible to find the value of the variable x_{n-2} from the second to last equation etc.

9.2.1.1 Gaussian Elimination

The Gaussian elimination stage consists in sequential elimination of the unknowns in the equations of the linear equation system being solved. At iteration $i,\ 0\le i<n-1,$ of the method the variable x_i is eliminated for all the equations with numbers k greater than i (i.e. $i< k\le n-1,$). In order to do so the row i multiplied by the constant (a_{ki}/a_{ii}) is subtracted from these equations so that the resulting coefficient of unknown x_i in the rows appears zero. All the necessary computations may be described by the following relations:

$$\begin{array}{r} a'_{kj} = a_{kj} - (a_{ki}/a_{ii})\cdot a_{ij}, \\ b'_k = b_k - (a_{ki}/a_{ii})\cdot b_i, \end{array} \quad i\le j\le n-1,\ i<k\le n-1,\ 0\le i<n-1$$

(it should be noted that similar computations are performed over the elements of the vector b as well).

Let us demonstrate the Gaussian elimination stage using the following system of linear equations as an example:

$$\begin{array}{llll} x_0 & +3x_1 & +2x_2 & = & 1 \\ 2x_0 & +7x_1 & +5x_2 & = & 18\ . \\ x_0 & +4x_1 & +6x_2 & = & 26 \end{array}$$

At the first iteration the unknown x_0 is eliminated in the second and the third rows. For this the first row multiplied correspondingly by 2 and by 1 is subtracted from these rows. After these transformations the system looks as follows:

$$\begin{array}{llll} x_0 & +3x_1 & +2x_2 & = & 1 \\ & x_1 & +\ x_2 & = & 16\ . \\ & x_1 & +4x_2 & = & 25 \end{array}$$

As a result, we need to perform the last iteration and eliminate the unknown x_1 in the third equation. For this it is necessary to subtract the second row. In the final form the system looks as follows:

$$\begin{array}{rrrcl} x_0 & +3x_1 & +2x_2 & = & 1 \\ & x_1 & +x_2 & = & 16\,. \\ & & 3x_2 & = & 9 \end{array}$$

Figure 9.1 shows the general scheme of the data state at i-th iteration of the Gaussian elimination stage. All the coefficients of the unknowns, which are located lower than the main diagonal and to the left of column i are already zero. At i-th iteration of the Gaussian elimination stage the coefficients of column i located lower than the main diagonal are set to zero. It is done by means of subtracting the row i multiplied by the adequate nonzero constant. After executing *(n-1)* similar iterations the matrix of linear equation coefficients is transformed in the upper triangle form.

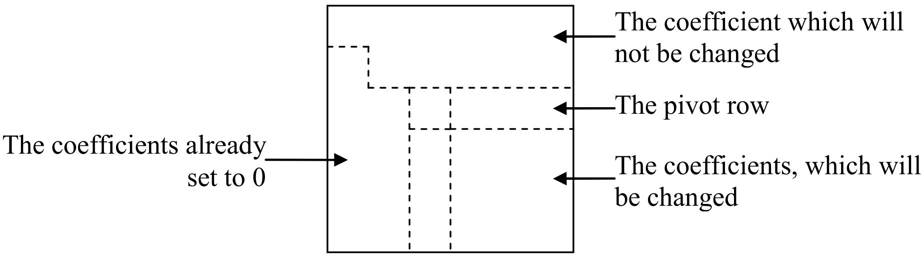
The coefficient which will not be changed
The pivot row
The coefficients already set to 0
The coefficients, which will be changed

Figure 9.1. The iteration of the Gaussian elimination stage

During the execution of the Gaussian elimination stage the row, which is used for eliminating unknowns is called the *pivot* one, and the diagonal element of the pivot row is called the pivot element. As it can be noted it is possible to perform computations only if the leading element is a nonzero value. Moreover, if the pivot element $a_{i,i}$ has a small value, then the division and the multiplication of rows by this element may lead to accumulation of the computational errors and the computational instability of the algorithm.

A possible way to avoid this problem may consist in the following. At each iteration of the Gaussian elimination stage it is necessary to determine the coefficient with the maximum absolute value in the column, which corresponds to the eliminated variable, i.e.

$$y = \max_{i \le k \le n-1} |a_{ki}|\,,$$

and choose the row, which holds this coefficient, as the pivot one (this scheme of choosing the pivot value is called *the partial pivoting scheme*).

The computational complexity of the Gaussian elimination stage with partial pivoting is the order $O(n^3)$.

9.2.1.2 Back Substitution

After the matrix of the coefficients has been reduced to the upper triangle form, it becomes possible to find the values of the unknowns. The value of the variable x_{n-1} may be calculated from the last equation of the transformed system. After that it is possible to determine the variable x_{n-2} from the second to last equation and so on. In the general form the computations performed during the back substitution stage may be presented by means of the following relations:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$
$$x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \; i = n-2, n-3, ..., 0\,.$$

It may be explained as previously using the example of the linear equation systems discussed in subsection 9.2.1.1

$$\begin{array}{rrrcl} x_0 & +3x_1 & +2x_2 & = & 1 \\ & x_1 & +x_2 & = & 16\,. \\ & & 3x_2 & = & 9 \end{array}$$

From the last equation of the system, the value of the variable x_2 is 3. As a result, it becomes possible to solve the second equation and to find the value of the unknown x_1=13, i.e.

$$\begin{array}{rrrcl} x_0 & +3x_1 & +2x_2 & = & 1 \\ & x_1 & & = & 13\,. \\ & & x_2 & = & 3 \end{array}$$

The value of the unknown x_0, which is equal to -44, is determined at the last iteration of the back substitution stage.

With regard to the following parallel execution, it is possible to note that the obtained unknown values may be used for recalculations in all the system equations in parallel (these operations may be performed in the equations simultaneously and independently). Thus, in the example under consideration the system after the determination of the value of the unknown x_2 may be reduced to the following form:

$$\begin{array}{lllcr} x_0 & +3x_1 & & = & -5 \\ & x_1 & & = & 13 \\ & & x_2 & = & 3 \end{array}$$

The computational complexity of the back substitution stage is $O(n^2)$.

9.2.2. Computation Decomposition

In close consideration of the Gauss method it is possible to note that all the computations are reduced to uniform computational operations on the rows of coefficient matrix of the linear equation system. As a result, the parallel implementation of the Gauss method may be based on the data parallelization principle. All the computations related with processing a row of the matrix A and the corresponding element of the vector b may be taken as the basic computational subtask in this case.

9.2.3. Analysis of Information Dependencies

Let us consider the general scheme of parallel computations and the information dependencies among the basic subtasks, which appear in the course of computations.

For the execution of the Gaussian elimination stage it is necessary to perform $(n-1)$ iterations of eliminating the unknown variables in order to transform the matrix of coefficients A to the upper triangle form.

The execution of iteration i, $0\le i<n-1$, of the Gaussian elimination stage includes a number of sequential operations. First of all, at the very beginning of the iteration it is necessary to select the pivot row, which (if the partial pivoting scheme is used) is determined by the search of the row with the greatest absolute value among the elements of the column i, which corresponds to the eliminated variable x_i. As the rows of matrix A are distributed among subtasks, the subtasks with the numbers k, $k>i$, should exchange their coefficients of the eliminated variable x_i for the maximum value search. After all the necessary data has been accumulated in each subtask, it is possible to determine, which of them holds the pivot row, and which value is the pivot element.

To carry out the computations further the pivot subtask has to broadcast its pivot row of the matrix A and the corresponding element of the vector c to all the other subtasks with the numbers k, $k>i$. After the reception of the pivot row the subtasks perform the subtraction of rows, thus, providing the elimination of the corresponding variable x_i.

During the execution of the back substitution stage the subtasks perform the necessary computations for calculating the value of the unknowns. As soon as some subtask i, $0\le i<n-1$, determines the value of its variable x_i, this value must be broadcasting to all the subtasks with the numbers k, $k<i$. After communications the subtasks substitute the variables x_i for the obtained value and modify the elements of the vector b.

9.2.4. Scaling and Distributing Subtasks among Processors

In case when the size of the linear equation system appears to be grater greater than the number of the available processors (i.e. $p<n$), the basic subtasks should be aggregated by uniting several matrix rows in a single subtask. However, the application of the sequential scheme of data partitioning, which was applied in sections 7 and 8, for parallel solving the systems of linear equations may lead to an unequal computational load of processors. In the course of eliminating (the Gaussian elimination stage) or calculating (the back substitution stage) the unknown variables the computations for an increasing part of processors will be terminated and the processor will appear to be idle. A possible solution of the balancing problem may consist in the use of cyclic striped scheme (see Section 7) for distributing data among the aggregated subtasks. In this case matrix A is divided into sets (stripes) of rows of the following form (see Figure 9.2):

$$A=(A_0,A_2,...,A_{p-1})^T,\ A_i=(a_{i_0},a_{i_1},...,a_{i_{k-1}}),\ i_j=i+jp,\ 0\le j<k,\ k=m/p\ .$$

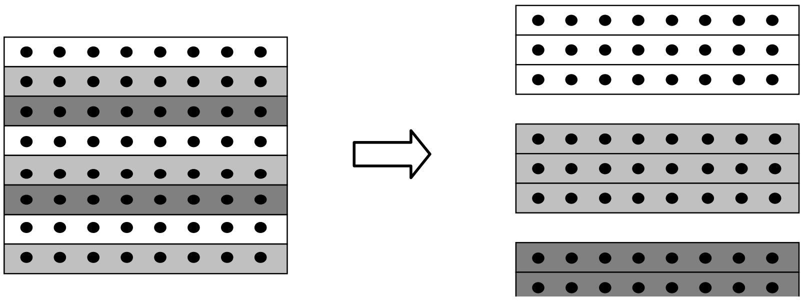

Figure 9.2. The example of using the cyclic striped scheme for partitioning matrix rows among 3 processors

The comparison of the data decomposition scheme and computations to be executed in the Gauss method allows to conclude that the use of the cyclic striped scheme makes possible to provide a better balancing of the computational load among the subtasks.

The distribution of the subtasks among the processors must take into account the data communications performed by the Gauss method. One-to-all broadcast is the main form of the information transmissions of the subtasks. As a result, the data communication network topology must be a hypercube or a complete graph in order to execute efficiently the required information transmissions among the basic subtasks.

9.2.5. Efficiency Analysis

Let us estimate the time complexity of the Gauss method. As previously, let n be the order of the linear equation system being solved, and $p,\ p<n,$ be the number of processors used. Thus, the coefficient matrix A is of $n \times n$ size and, correspondingly, n/p is the size of matrix stripe on each processor.

First of all, it is easy to demonstrate that the total execution time for the sequential variant of the Gauss method is the following:

$$T_1 = 2n^3/3 + n^2 \,. \qquad (9.3)$$

Let us determine the complexity of the parallel variant of the Gauss method. In the Gaussian elimination stage each processor should choose the maximum values in the column with the eliminated unknown within its stripe. This computations should be executed at each iteration in order to choose the pivot row. The initial size of the stripes is equal to n/p. As the unknowns are eliminated the number of rows in the stripes to be processed is gradually reduced. The current size of the stripes may be estimated approximately as $(n\text{-}i)/p$, where $i,\ 0\le i<n\text{-}1,$ is the iteration number of the Gaussian elimination stage being performed. After the obtained maximum values are accumulated on each processor, the pivot row should be determined and transmitted to all processors. Then each processor must perform the subtraction of the pivot row from each row of its remaining part of the matrix A. The number of row elements to be processed is also reduced as the unknowns are eliminated and the current number of the row elements to be computed is estimated by the value $(n\text{-}i)$. Thus, the complexity of the row subtraction can be estimated as $2\cdot(n\text{-}i)$ operations (before being subtracted the pivot row is multiplied by the scaling value a_{ik}/a_{ii}). With regard to the number of iterations performed the total number of operations for the parallel variant of the Gaussian elimination stage is determined by the following expression:

$$T_p^1 = \sum_{i=0}^{n-2}\left[\frac{(n-i)}{p}+\frac{(n-i)}{p}\cdot 2(n-i)\right] = \frac{1}{p}\sum_{i=0}^{n-2}\left[(n-i)\cdot +2(n-i)^2\right].$$

At each iteration of the back substitution stage after broadcasting the computed value of the current unknown each processor have to recalculate the values of its part of the matrix A and the vector b. Hence, the time complexity of the parallel variant of the back substitution stage can be estimated as the following value:

$$T_p^2 = \sum_{i=0}^{n-2} 2\cdot(n-i)/p \,.$$

After summation of the obtained expressions we get the following:

$$T_p = \frac{1}{p}\sum_{i=0}^{n-2}\left[(n-i)\cdot +2(n-i)^2\right] + \frac{2}{p}\sum_{i=0}^{n-2}(n-i) = \frac{1}{p}\sum_{i=0}^{n-2}\left[3(n-i)+2(n-i)^2\right] = \frac{1}{p}\sum_{i=2}^{n}(3i+2i^2) \,.$$

As a result of the analysis, the speedup and efficiency characteristics for the parallel variant of the Gauss method may be evaluated by means of the following formulas:

$$S_p = \frac{(2n^3/3+n^2)}{\frac{1}{p}\sum_{i=2}^{n}(3i+2i^2)}, \qquad E_p = \frac{(2n^3/3+n^2)}{\sum_{i=2}^{n}(3i+2i^2)}. \tag{9.4}$$

The obtained relations are rather complicated for analyzing. Alongside with this it is possible to show that the parallel algorithm complexity is of order *~(2n³/3)/p*. As a result, we can conclude that computational load balancing among the processor is approximately uniform.

Let us estimate the overhead of data communications among the processors. At the Gaussian elimination stage the processors exchange the locally obtained maximum values in the column with the eliminated variable in order to determine the pivot row at each iteration. The execution of this operation simultaneously with the determination of the maximum value among the accumulated values may be provided by the function *MPI_Allreduce* of MPI library. As it can be shown this operation may be executed in $log_2 p$ steps. Now it is possible to estimate the time necessary to execute reduction operation by means of the following expression:

$$T_p^1(comm) = (n-1)\cdot\log_2 p\cdot(\alpha + w/\beta),$$

where, as previously, α is the communication latency, β is the network bandwidth, w is the size of the data element to be transmitted.

Additionally a each iteration of the Gaussian elimination stage the selected pivot row should be transmitted to all processor. The complexity of this data communication operation is equal to the following value:

$$T_p^2(comm) = (n-1)\cdot\log_2 p\cdot(\alpha + wn/\beta).$$

At each iteration of the back substitution stage the computed value of the current unknown has to be distributed among all the processors. The total execution time of these operations may be estimated as follows:

$$T_p^3(comm) = (n-1)\cdot\log_2 p\cdot(\alpha + w/\beta).$$

Finally with regard to all the obtained relations the time complexity of the parallel variant of the Gauss method is, , the following:

$$T_p = \frac{1}{p}\sum_{i=2}^{n}(3i+2i^2)\,\tau + (n-1)\cdot\log_2 p\cdot(3\alpha + w(n+2)/\beta), \tag{9.5}$$

where τ is the execution time of the basic computational operation.

9.2.6. Software Implementation

Let us discuss a possible variant of parallel Gauss method implementation for solving a system of linear equations. It should be noted that program code of several modules is not given as its absence does not influence the understanding of the general scheme of parallel computations.

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

```
// Program 9.1
// The Gauss Algorithm for solving the systems of linear equations
int ProcNum;            // Number of the available processes
int ProcRank;           // Rank of the current process
int *pParallelPivotPos; // Number of rows selected as the pivot ones
int *pProcPivotIter;    // Number of iterations, at which the processor
                        // rows were used as the pivot ones

int* pProcInd; // Number of the first row located on the processes
int* pProcNum; // Number of the linear system rows located on the processes

void main(int argc, char* argv[]) {
  double* pMatrix;        // Matrix of the linear system
  double* pVector;        // Right parts of the linear system
  double* pResult;        // Result vector
  double *pProcRows;      // Rows of the matrix A
  double *pProcVector;    // Block of the vector b
  double *pProcResult;    // Block of the vector x
  int     Size;           // Size of the matrix and vectors
  int     RowNum;         // Number of  the matrix rows
  setvbuf(stdout, 0, _IONBF, 0);

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

  if (ProcRank == 0)
    printf("Parallel Gauss algorithm for solving linear systems\n");

  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult,
    pProcRows, pProcVector, pProcResult, Size, RowNum);

  // The execution of the parallel Gauss algorithm
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);

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

  ResultCollection(pProcResult, pResult);

  TestResult(pMatrix, pVector, pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult);

  MPI_Finalize();
}
```

It is necessary to explain the use of auxiliary arrays. The elements of the array *pParallelPivotPos* determine the numbers of the matrix rows selected as the pivot ones according to the Gaussian elimination iterations. This is exactly the order in which the back substitution iterations should be executed in order to find the values of the unknowns of a linear equation system. The array *pParallelPivotPos* is global and its every change in any process requires broadcasting the changed data to all the program processes.

The elements of *pProcPivotIter* determine the numbers of the Gaussian elimination iterations, at which the process rows were used as the pivot ones (i.e. the row *i* of the process was selected the pivot one at the *iteration pProcPivotIter[i]*). The initial values of the array elements are set to zero. Thus, the zero value of pProcPivotIter*[i]* element is the sign that the row *i* of the process is still to be processed. Besides, it should be noted that iteration numbers stored in *pProcPivotIter* elements additionally signify the numbers of the unknowns, which must be determined by means of the equations, which correspond to the rows. The array *pProcPivotIter* is local for each process.

The function *ProcessInitialization* determines the initial data of the problem being solved (the number of unknowns), allocates memory for storing the data, inputs the coefficient matrix of the linear equation system and the vector of the right part (or it forms the data by means of random number generator). Function *DataDistribution* distributes the data among the processors.

The function *ResultCollection* accumulates separate parts of the vector of the unknowns from all the processes.

The function *ProcessTermination* executes the necessary output of the results of problem solving and releases all the previously allocated memory for storing the data.

The implementation of all the above mentioned functions may be performed on the analogy with the examples, which have been discussed earlier and is given to the reader a training exercise.

2. The function *ParallelResultCalculation*. This function contains the calls of the functions for executing the Gauss algorithm stages.

```
/ Function for execution of the parallel Gauss algorithm
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  // Gaussian elimination
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  // Back substitution
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);
}
```

3. The function *ParallelGaussianElimination*. It executes the parallel variant of the Gaussian elimination stage.

```
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;   // Value of the pivot element of the process
  int    PivotPos;   // Position of the pivot row in the process stripe
  // Structure for the pivot row selection
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;

  // pPivotRow is used for storing the pivot row and the corresponding
  // element of the vector b
  double* pPivotRow = new double [Size+1];

  // The iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {

    // Calculating the local pivot row
    double MaxValue = 0;
    for (int j=0; j<RowNum; j++) {
      if ((pProcPivotIter[j] == -1) &&
          (MaxValue < fabs(pProcRows[j*Size+i]))) {
        MaxValue = fabs(pProcRows[j*Size+i]);
        PivotPos = j;
      }
    }
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;

    // Finding the pivot process (process with the maximum value of MaxValue)
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
                  MPI_COMM_WORLD);

    // Broadcasting the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      pProcPivotIter[PivotPos]= i; //iteration number
      pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
    }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);

    if ( ProcRank == Pivot.ProcRank ){
      // Fill the pivot row
      for (int j=0; j<Size; j++) {
        pPivotRow[j] = pProcRows[PivotPos*Size + j];
      }
      pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank, MPI_COMM_WORLD);

    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size,
      RowNum, i);
  }
}
```

The function *ParallelEliminateColumns* subtracts the pivot row from the process rows, which have not been used as the pivot ones yet (i.e. rows for which the elements of the array *pProcPivotIter* are equal to zero).

4. The function *ParallelBackSubstitution*. This function executes the parallel variant of the back substitution stage.

```
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank;     // Rank of the process with the current pivot row
  int IterPivotPos;     // Position of the pivot row of the process
  double IterResult;   // Calculated value of the current unknown
  double val;

  // Iterations of the back substitution stage
  for (int i=Size-1; i>=0; i--) {

    // Calculating the rank of the process, which holds the pivot row
  FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank, IterPivotPos);

    // Calculating the unknown
    if (ProcRank == IterProcRank) {
      IterResult = pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
        pProcResult[IterPivotPos] = IterResult;
    }
    // Broadcasting the value of the current unknown
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

    // Updating the values of the vector b
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}
```

The back substitution execution consists of *Size* iterations. At each iteration it is necessary to determine the row, which makes possible to calculate the value of the next result vector element. The row number is stored in the array *pParallelPivotIter*. Using the row number The function *FindBackPivotRow* determines the number of process, where the row is stored, and the number of the row in the stripe *pProcRows* of the process.

9.2.7. Computational Experiment Results

The computational experiments for the estimation of the parallel variant of the Gauss method efficiency for solving the systems of linear equations were carried out under the conditions specified in 7.6.5. In brief terms these conditions are the following.

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.

The duration τ of the basic computational operation has been estimated experimentally by solving several systems of linear equations using the sequential algorithm. The computation time obtained that way was then divided by the total number of the executed operations. As a result the value 4.7 nsec was obtained for the parameter τ. The experiments performed for estimating the data communication network parameters demonstrated the values of the latency α and the bandwidth β correspondingly 47 msec and 53.29 Mbyte/sec. All the computations were carried out with the numerical values of double type, i.e.value w is equal to 8 bytes.

The results of the computational experiments are given in Table 9.1. The experiments were carried out with the use of 2, 4 and 8 processors.

Table 9.1. The results of the computational experiments for the parallel Gauss plgorithm

Matrix Size	Serial Algorithm	Parallel Algorithm					
		2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,36	0,3302	1,0901	0,5170	0,6963	0,7504	0,4796
1000	3,313	1,5950	2,0770	1,6152	2,0511	1,8715	1,7701
1500	11,437	4,1788	2,7368	3,8802	2,9474	3,7567	3,0443
2000	26,688	9,3432	2,8563	7,2590	3,6765	7,3713	3,6204
2500	50,125	16,9860	2,9509	11,9957	4,1785	11,6530	4,3014
3000	85,485	28,4948	3,0000	19,1255	4,4696	17,6864	4,8333

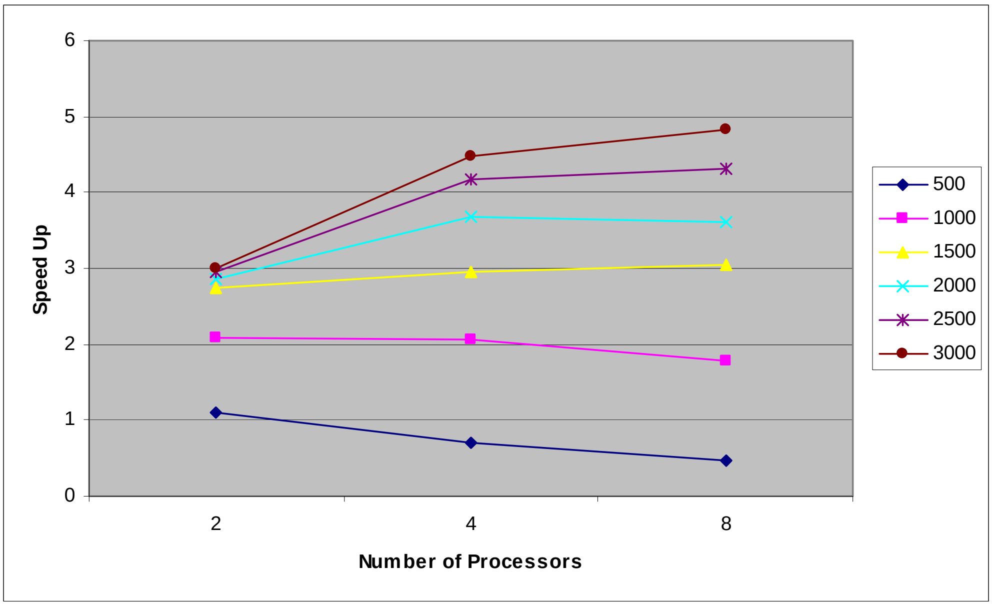

Figure 9.3. Speedup of the parallel Gauss algorithm for solving linear equation systems (rowwise block-striped matrix decomposition)

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (9.5) is given in Table 9.2 and in Figure. 9.4.

Table 9.2. The comparison of the experimental execution time and the theoretical estimation for the parallel Gauss algorithm

Matrix Size	2 processors		4 processors		8 processors	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	0,2393	0,3302	0,2819	0,5170	0,3573	0,7504
1000	1,3373	1,5950	1,1066	1,6152	1,1372	1,8715
1500	4,0750	4,1788	2,8643	3,8802	2,5345	3,7567
2000	9,2336	9,3432	5,9457	7,2590	4,7447	7,3713
2500	17,5941	16,9860	10,7412	11,9957	7,9628	11,6530
3000	29,9377	28,4948	17,6415	19,1255	12,3843	17,6864

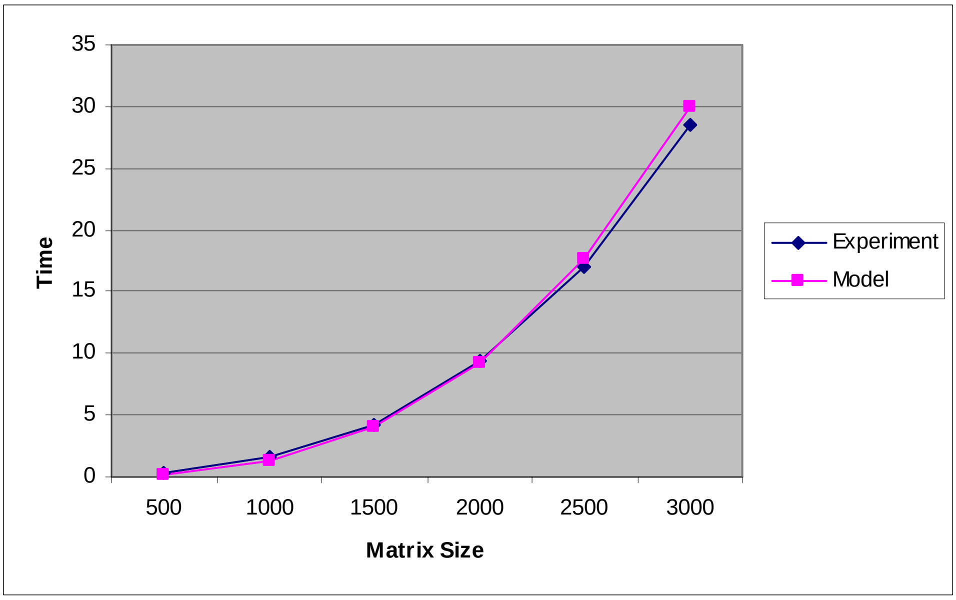

Figure 9.4. Theoretical and experimental execution time with respect to matrix size (rowwise block-striped matrix decomposition, 2 processors)

9.3. The Conjugate Gradient Method

Let us consider another approach to solving linear equation systems. According to this approach the sequence of the approximate solutions x^0, x^1,…, x^k,… is calculated for approximating the exact solution x^* of system $Ax=b$. In addition the computations are performed in such a way that each next approximation gives the estimation of the exact solution with a decreasing error and, as a result, in case of continuous computations the estimation for the exact solution may be obtained with any desired degree of accuracy. Such iterative methods of solving linear equation systems are widely used in the practice of computations. Among advantages of these methods it should be mentioned a smaller amount (in comparison, for instance, to the Gauss method) of the computations necessary for solving the sparse linear equation systems, possibility of faster obtaining the initial approximation of the exact solution, the availability of efficient methods of parallel computations. Additional information and the description of these methods, consideration of the issues of convergence and accuracy of the obtained solutions may be found, for instance, in Bertsekas and Tsitsiklis (1989)

The conjugate gradient method is one of the best known iterative algorithms. This method may be used for solving a symmetric positive definite linear equation system of high dimensionality.

Let's recall that a matrix *A* is symmetric if it coincides with its transpose, i.e. $A=A^T$. A matrix *A* is positive definite, if $x^T Ax > 0$ takes place for each vector *x*.

It is known (see, for instance, Bertsekas and Tsitsiklis (1989)), that after the execution of *n* algorithm iterations (*n* is the order of the linear equation system being solved), the next approximation x^n coincide with the exact solution.

9.3.1. Sequential Algorithm

If matrix *A* is symmetric and positive definite, then the function

$$q(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T b + c \tag{9.6}$$

has a single minimizer at point x^*, which coincides with the solution of the linear equation system (9.2). The conjugate gradient method is one from wide group of iterative algorithms, which allow to find the solution (9.2) by means of minimizing function *q(x)*.

The iteration of conjugate gradient method consists in computing the next approximation to the exact solution in accordance to the following rule:

$$x^k = x^{k-1} + s^k d^k \, , \tag{9.7}$$

i.e., the new value of approximation x^k is computed with regard to the approximation obtained at the pervious step x^{k-1}, a scalar step s^k and a direction vector d^k.

Before the execution of the first iteration the vectors x^0 and d^0 are assumed to be equal to zero and the value –*b* is assigned to vector g^0 . Then at each iteration the computation of the next approximation value x^k includes the execution of the following four steps:

Step 1: Compute the gradient:

$$g^k = A \cdot x^{k-1} - b \, ; \tag{9.8}$$

Step 2: Compute the direction vector:

$$d^k = -g^k + \frac{\left((g^k)^T, g^k\right)}{\left((g^{k-1})^T, g^{k-1}\right)} d^{k-1} \, , \tag{9.9}$$

where (g^T,g) is the inner product of vectors;

Step 3: Compute the step value:

$$s^k = \frac{\left(d^k, g^k\right)}{(d^k)^T \cdot A \cdot d^k} \, ; \tag{9.10}$$

Step 4: Compute the new approximation:

$$x^k := x^{k-1} + s^k d^k \, . \tag{9.11}$$

As it can be noted, these expressions include two operations of matrix-vector multiplication, four operations of inner product and five vector operations. As a result, the total number of operations executed at each iteration is the following:

$$t_1=2n^2+13n.$$

As it has been previously mentioned, to obtain the exact solution for linear equation system with a symmetric positive definite matrix, it is necessary to perform *n* iterations. Thus, to solve the system it is necessary to perform

$$T_1=2n^3+13n^2 \tag{9.12}$$

operations, so the algorithm complexity is the order $O(n^3)$.

Let's demonstrate the conjugate gradient method by solving the following system of linear equations:

$$\begin{array}{rrcl} 3x_0 & -x_1 & = & 3 \\ -x_0 & +3x_1 & = & 7 \end{array}.$$

The matrix of this system is symmetric positive definite and, the conjugate gradient method may be applied. As it was previously mentioned, for finding the exact decision of this system it is required to execute only two iterations of a method.

On the first iteration it was computed that the gradient g^1 = (-3,-7), the direction vector d^1 = (3, 7), the step size s^1=0.439. Accordingly, the next approximation of the exact solution is x^1 = (1.318, 3.076).

On the second iteration it was computed that the gradient g^2 = (-2.121, 0.909), the direction vector d^2 = (2.397,-0.266) and the step size s^2=0.284. The next approximation is x^2 = (2, 3) and it coincides with the exact solution of the system.

Figure 9.3 shows the sequence of the approximations to the exact solution, calculated by the conjugate gradient method (as the initial approximation x^0 the point (0,0) has been chosen)).

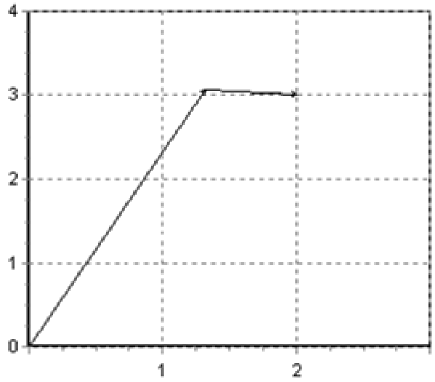

Figure 9.5. Iterations of the conjugate gradient method for solving linear equation system with two unknowns

9.3.2. Parallel Algorithm

In the course of development of the parallel conjugate gradient method for solving a linear equation system, it is first of all necessary to take into account the fact that method iterations are executed sequentially. So the most reasonable approach is to parallelize the computations performed in the course of iteration execution.

The analysis of the relations (9.8)-(9.11) shows that the main computations performed in accordance with the method consist in multiplying the matrix *A* by the vectors *x* and *d*. As a result, all the results of Section 7 may be used in developing parallel computations.

Additional computations in (9.8)-(9.11), which are of lower complexity order, are various vector computational operations (inner product, summation and subtraction, multiplication by a scalar). The implementation of such computations should be, of course, matched with the selected parallel method of executing the operation of matrix-vector multiplication. The general recommendations may be the following: if vectors are of small sizes, it is possible to duplicate the vectors among the processors, if the linear equation system being solved is of great order, then the vector block decomposition is more efficient.

9.3.3. Efficiency Analysis

To analyze the efficiency of parallel computations let us choose the parallel algorithm of matrix-vector multiplication in case of the rowwise block-striped matrix partitioning and the duplication approach of all the processed vectors.

The time complexity for the sequential conjugate gradient method was determined in (9.12).

Let us determine the execution time for the parallel implementation of this method. The computational complexity of parallel matrix-vector multiplication operations in case of the rowwise block-striped matrix decomposition is the following:

$T_p^1(calc) = 2n\lceil n/p \rceil \cdot (2n-1)$ (see Section 7).

As a result, if all the computations on the vectors are duplicated, the total computational complexity for the parallel variant of the method is the following:

$$T_p(calc) = n\left(2\cdot\lceil n/p\rceil\cdot(2n-1)+13n\right).$$

Taking into account the obtained estimations, we may express the speedup and efficiency using the following relations:

$$S_p = \frac{2n^3+13n^2}{n\left(2\lceil n/p\rceil\cdot(2n-1)+13n\right)},\quad E_p = \frac{2n^3+13n^2}{p\cdot n\left(2\lceil n/p\rceil\cdot(2n-1)+13n\right)}.$$

The analysis of these expressions shows that computational load balancing among the processor is approximately uniform.

Let us specify the expressions used in the subsection. We will take into account the duration of the computational operations performed and estimate the time complexity of the data communications among the processors. As it can be noted, the information communications are appeared only in case of matrix-vector multiplication. With regard to the results of Section 7, we may consider the communication complexity of the parallel computations to be equal to the following:

$$T_p(comm) = 2n\left(\alpha\cdot\lceil\log p\rceil + w(n/p)(p-1)/\beta\right),$$

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

In its final form the execution time for the parallel variant of the conjugate gradient method for solving linear equation systems looks as follows:

$$T_p = n\cdot\left[\left(2\lceil n/p\rceil\cdot(2n-1)+13n\right)\cdot\tau + 2\left(\alpha\cdot\lceil\log p\rceil + w(n/p)(p-1)/\beta\right)\right], \tag{9.13}$$

Where τ is the execution time of the basic computational operation.

9.3.4. Computational Experiment Results

The computational experiments for the estimation of the parallel variant of conjugate gradient method efficiency for solving the systems of linear equations were carried out under the conditions specified in 7.6.5.

The results of the experiments are given in Table 9.3. The experiments were carried out on the computational systems, which consist of 2, 4 and 8 processors.

Table 9.3. The results of the computational experiments for the parallel conjugate gradient method for solving the linear equation systems

Matrix Size	Serial Algorithm	Parallel Algorithm					
		2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,5	0,4634	1,0787	0,4706	1,0623	1,3020	0,3840
1000	8,14	3,9207	2,0761	3,6354	2,2390	3,5092	2,3195
1500	31,391	17,9505	1,7487	14,4102	2,1783	20,2001	1,5539
2000	92,36	51,3204	1,7996	40,7451	2,2667	37,9319	2,4348
2500	170,549	125,3005	1,3611	85,0761	2,0046	87,2626	1,9544
3000	363,476	223,3364	1,6274	146,1308	2,4873	134,1359	2,7097

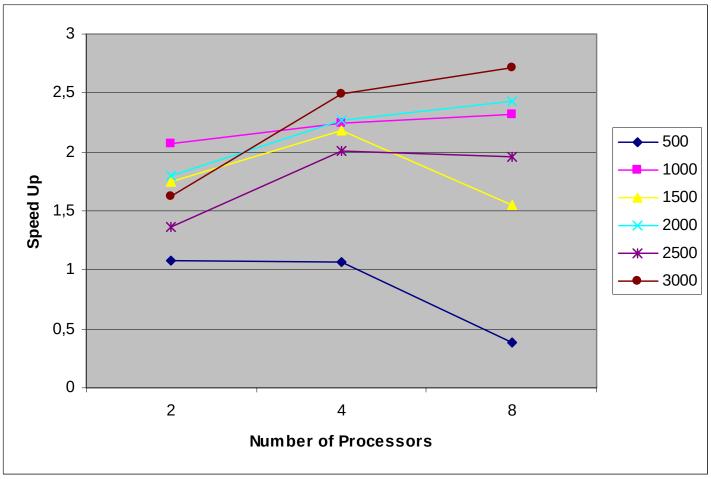

Figure 9.6. Speedup for the parallel conjugate gradient method for solving the linear equation systems (rowwise block-striped matrix decomposition)

The comparison of the execution time T_p^* and the theoretical estimation T_p from (9.13) is given in Table 9.4 and in Figure 9. 6.

Table 9.4. The comparison of the experimental and the theoretical execution time for the parallel conjugate gradient method for solving linear equation systems

Matrix Size	2 processors		4 processors		8 processors	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	1,3042	0,4634	0,6607	0,4706	0,3390	1,3020
1000	10,3713	3,9207	5,2194	3,6354	2,6435	3,5092
1500	34,9333	17,9505	17,5424	14,4102	8,8470	20,2001
2000	82,7220	51,3204	41,4954	40,7451	20,8822	37,9319
2500	161,4695	125,3005	80,9446	85,0761	40,6823	87,2626
3000	278,9077	223,3364	139,7560	146,1308	70,1801	134,1359

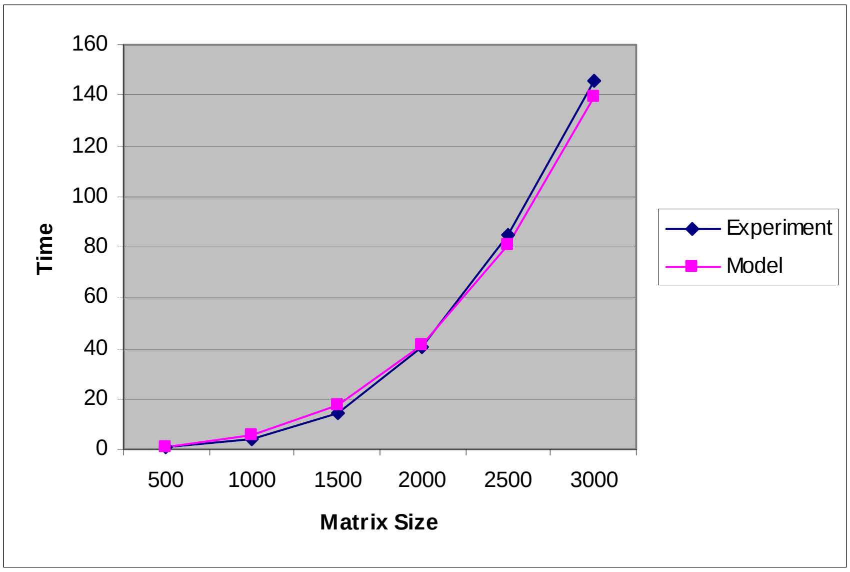

Figure 9.7. Theoretical and experimental execution time with respect to matrix size (rowwise block-striped matrix decomposition, 4 processors)

9.4. Summary

The Section is devoted to the problem of parallel computations for solving linear equation systems. The two widely known algorithms are discussed in the Section – *the Gauss method*, as an example of direct algorithms for solving the problem, and *the iterative conjugate gradient method.*

The parallel variant of the Gauss method (Subsection 9.2) is based on the rowwise block-striped matrix distribution among the processors and the use of the cyclic scheme of row distribution. This scheme makes possible to balance the computational load. To develop the parallel variant of the method a complete design cycle is carried out in the Subsection. The basic computational subtasks are defined, information communications are analyzed, the issues of scaling are discussed, efficiency characteristics are estimated, software implementation is suggested and the results of the computational experiments are given. In general, according to the efficiency analysis described in the Subsection, the use of the parallel variant of the Gauss method does not provide the speedup of computations (see Figure 9.7), because of the big number of communication operations.

An important aspect in the consideration of the parallel variant of the conjugate gradient method (Subsection 9.3) is that the method of parallel computations for this method may be obtained through the previously considered parallel algorithms of the executed computational operations (the operations of matrix-vector multiplication, scalar-vector product, summation and subtraction of vectors). The approach chosen for the discussion consists in parallelizing the most computationally time-consuming operation of matrix-vector multiplication, while all the vector computations are duplicated on each processor in order to decrease the number of data communications.

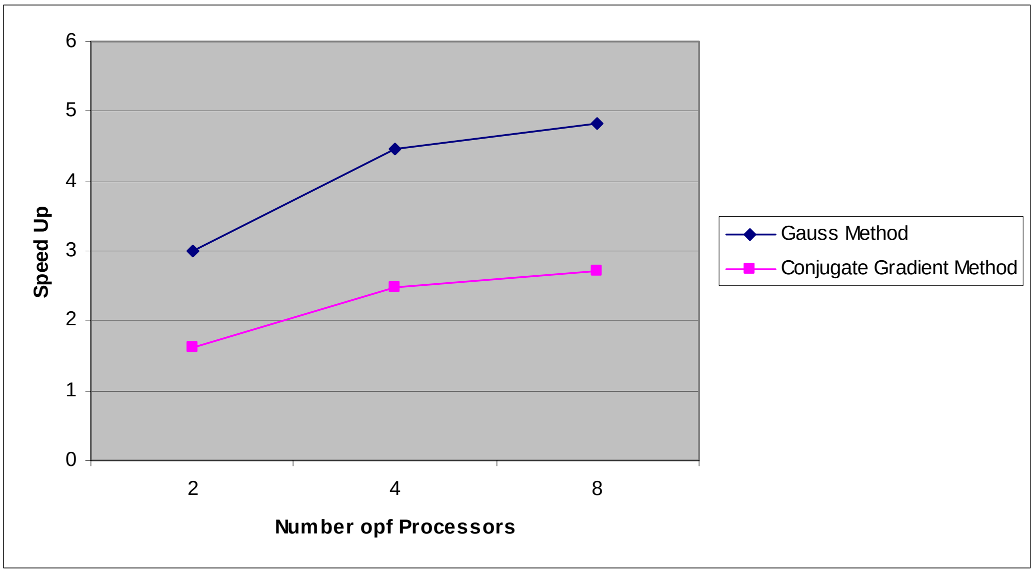

Figure 9.8. Speedup of the parallel algorithms for solving a linear equation system (4 processors, matrix size 3000×3000)

9.5. References

The problem of numerical solving linear equation systems is broadly discussed. For training purposes we may recommend the works by Kumar, et al. (1994) and Quinn (2004). The issues of parallel computations for solving this type of problems are also discussed in Bertsekas and Tsitsiklis (1989), 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.

9.6. Discussions

1. What is a system of linear equations? What types of systems do you know? What methods may be used for solving systems of various types?

2. What is the problem statement for solving linear equation systems?

3. What is the essence of the parallel Gauss method implementation?

4. What information communications are there among the basic subtasks in the parallel variant of Gauss method?

5. What are the efficiency characteristics for the parallel variant of Gauss method?

6. What is the scheme of program implementation of the parallel variant of Gauss method?

7. What is the concept of the parallel implementation of the conjugate gradient method?

8. Which of the algorithms has the greater communication complexity?

9.7. Exercises

1. Analyze the efficiency of the parallel computations for the Gaussian elimination and the back substitution stages of the Gauss method separately. At which stage is there the greater decrease of parallel efficiency?

2. Develop the parallel variant of the Gauss method based on the columnwise block-striped matrix decomposition. Estimate theoretically the execution time of the algorithm. Carry out the computational experiments. Compare the experimental results with the theoretical estimations obtained previously.

3. Develop the implementation of the parallel conjugate gradient method. Estimate theoretically the execution time of the algorithm. Carry out the computational experiments. Compare the experimental results with the theoretical estimations obtained previously.

4. Develop the parallel variant of the Jacobi and Seidel methods of solving linear equation systems (see, for instance, Kumar, et al. (1994), Quinn (2004)). Estimate theoretically the execution time of the algorithm. Carry out the computational experiments. Compare the experimental results with the theoretical estimations obtained previously.

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.