12. Parallel methods for partial differential equations

12. Parallel methods for partial differential equations 1
12.1. Sequential Method 1
12.2. Parallel Computations for Shared Memory Systems 3
12.2.1. Parallel Programming with OpenMP 3
12.2.2. Computation Synchronization 4
12.2.3. Computation Indeterminacy 7
12.2.4. Computation Deadlock 8
12.2.5. Indeterminacy Elimination 9
12.2.6. Wavefront Computation Scheme 10
12.2.7. Computation Load Balancing 14
12.3. Parallel Computation for Distributed Memory Systems 15
12.3.1. Data Distribution 15
12.3.2. Information Communications among Processors 16
12.3.3. Collective Communications 18
12.3.4. Wavefront Computations 18
12.3.5. Checkerboard Data Distribution 19
12.3.6. Communication Complexity Analysis 21
12.4. Summary 23
12.5. References 23
12.6. Discussions 23
12.7. Exercises 24

Partial differential equations are widely used in various scientific and technical fields. Unfortunately, analytical solution of these equations may only be possible in some special cases. Therefore approximate numerical methods are usually used for solving partial differential equations. The amount of computations to perform here is usually significant. Using high-performance systems is traditional for this sphere of computational mathematics. Numerical solution of differential equations in partial derivatives is a subject of intensive research (see, for instance, Fox, et al. (1988)).

Let us consider the numerical solution of *the Dirichlet problem for the Poisson equation*. It is defined as a problem of finding function $u = u(x, y)$ that satisfies in the domain *D* the following equation:

$$\begin{cases} \frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = f(x,y), & (x,y) \in D, \\ u(x,y) = g(x,y), & (x,y) \in D^0, \end{cases}$$

and takes the value $g(x, y)$ at the boundary D^0 of the domain D when continuous functions f and g are given. Such model can be used for describing steady liquid flow, stationary thermal fields, heat transportation processes with the internal heat sources, elastic plate deformation etc. This problem is often used as a training problem for demonstrating various aspects of parallel computations (see, for instance, Fox, et al. (1988), Pfister (1998)).

For simplicity hereinafter the unit square

$D = \{(x, y) \in D : 0 \le x, y \le 1\}$

will be taken as the problem domain.

This Section has been written based essentially on the teaching materials given in Pfister (1995).

12.1. Sequential Method

The finite difference method (*the grid method*) is most widely used for numerical solving differential equations (see, for instance, Pfister (1995)). Following this approach, the domain D is represented as a discrete (uniform, as a rule) set (*grid*) of points (*nodes*). Thus, for instance, the rectangular grid in the domain *D* can be given as (Figure 12.1)

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih,\ y_i = jh,\ 0 \le i, j \le N+1, \\ \qquad\qquad h = 1/(N+1), \end{cases}$$

Where the value N defines the grid node number for each coordinate in the region D.

Let us denote by u_{ij} the values of the function $u(x, y)$ at the points (x_i, y_j). Then using *the five-point template* (see Figure 12.1) we can present the Poisson equation in the *finite difference form*:

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

This equation can be rewritten in the following form:

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

It makes possible to determine the value u_{ij} from the given values of the function $u(x, y)$ in the neighboring nodes of the used grid. This result can be used as a basis for developing various *iteration schemes* for solving the Dirichlet problem. At the beginning of calculations in these schemes an initial approximation for values u_{ij} is formed, and then these values are sequentially recalculated in accordance with the given formula. Thus, for instance, *the Gauss-Seidel method* uses the following rule for updating values of approximations:

$$u_{ij}^k = 0.25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}),$$

according to which the sequent *k* approximation of the value u_{ij} is calculated from the last *k* approximation of the values $u_{i-1,j}$ and $u_{i,j-1}$ and the next to last (*k-1*) approximation of the values $u_{i+1,j}$ and $u_{i,j+1}$. Calculations continue till the variations of values u_{ij}, which are evaluated at the end of each iteration, become less than a certain given value (*the required accuracy*). The described procedure convergence (obtaining the solution of any desired accuracy) is the subject of intensive investigation (see, for instance, Pfister (1995)). It should be also noted that the sequence of approximations obtained by this method converges to the Dirichlet problem solution, while the solution error is the order h^2.

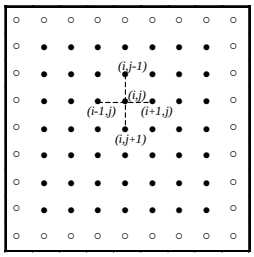

Figure 12.1. Rectangular grid in the region D (the dark points represent the internal nodes, the nodes are numbered from left to right in rows and top to bottom in columns)

The above considered algorithm (the Gauss-Seidel method) can be presented in the following way in pseudocode based on the algorithmical language C++.

```
// Algorithm 12.1
// Sequential Gauss-Seidel method
do {
  dmax = 0; // maximal variation of the values u
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

Algorithm 12.1. Sequential Gauss-Seidel algorithm

(recall that the values u_{ij} with indexes $i, j = 0, N+1$ are boundary values, defined in the problem statement and do not vary in the course of calculations).

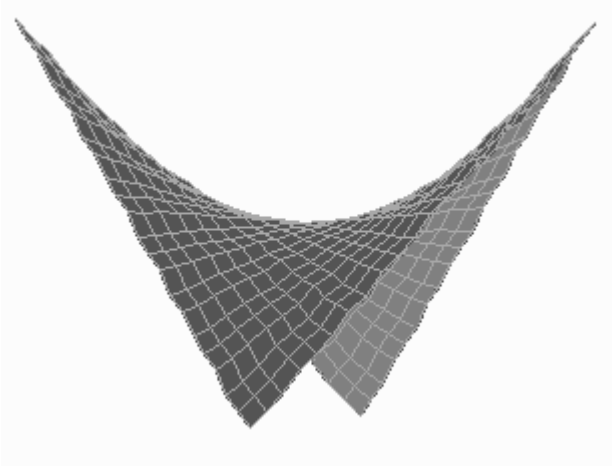

Figure 12.2. Function $u(x, y)$ in the example for the Dirichlet problem

As an example, Figure 12.2 shows the function $u(x, y)$ obtained for the Dirichlet problems with the following boundary conditions:

$$\begin{cases} f(x,y) = 0, & (x,y) \in D, \\ 100 - 200x, & y = 0, \\ 100 - 200y, & x = 0, \\ -100 + 200x, & y = 1, \\ -100 + 200y, & x = 1, \end{cases}$$

The total amount of the Gauss-Seidel method iterations is 210 with the solution accuracy $eps = 0.1$ and $N = 100$ (the values generated by a random-number generator in the range of numbers [-100,100] were used as an initial approximation of values u_{ij}).

12.2. Parallel Computations for Shared Memory Systems

As it follows from the given description, the above considered method is characterized by significant computing efforts

$T_1 = kmN^2$,

where N is the grid node number for each coordinate of region D, m is the number of operations performed by the method at each iteration, k is the number of method iterations before meeting the accuracy requirements.

12.2.1. Parallel Programming with OpenMP

Let us consider the possible ways to develop parallel programs for the grid methods on multiprocessor computer systems with shared memory. We will assume that the available processors possess the same performance and have uniform access to shared memory. We also assume what if several processes need access to the same memory element then access synchronization are provided at hardware level. Multiprocessor systems of this type are usually called *symmetric multiprocessors, SMP*.

The conventional approach of providing parallel computations for such systems is development of new parallel methods on the basis of existed sequential programs. The parts of the program, which are independent of each other and can be executed in parallel, can be selected either automatically by a compiler or directly by a programmer. The second approach prevails as the possibilities of program automatic analysis for generating parallel computations are rather restricted. In this case to develop parallel programs it can be used both new algorithmic languages assigned for parallel programming and the already available programming languages extended by a set of operations for parallel computations.

Both the mentioned approaches lead to the necessity to considerably reprocess the existing software. This fact significantly hinders the wide spread of parallel computations. As a result, one more approach to parallel

program development has recently been worked out. In this approach the instructions of the program developer concerning parallel computations are added to the program with the help of some extralinguistic means of the programming language. For instance, they may be added as directives or comments which are processed be special preprocessor before the program is compiled. In this case the original program code remains the same. If there's no preprocessor, the compiler can use it to construct the original sequential executable program. The preprocessor, if it is used, transforms the parallelism directives into some additional program code (as a rule in the form of calling the procedures of a parallel library).

The above discussed approach forms the basis of *OpenMP* technology (see, for instance, Chandra, et al. (2000)). This technology is currently widely used for developing parallel programs on multiprocessor systems with shared memory. The parallel directives are used for indicating *the parallel regions*, in which the executable code can be divided into several separate instruction *threads*. These threads can be further executed in parallel on different processors of the computer system. As a result of this approach the program is represented as a set of sequential *one-thread* and parallel *multi-thread* fragments of the code (see Figure 12.3). This principle of parallism implementation is called fork-join or *pulsatile* parallelism. More information on OpenMP technology can be found in the various resources - see, for instance Chandra, et al. (2000) or in the Internet information resources. In this section we will discuss the OpenMP capabilities necessary to demonstrate possible ways of developing parallel programs for solving the Dirichlet problem.

12.2.2. Computation Synchronization

The first variant of the Gauss-Seidel parallel algorithm may be obtained if random order of recalculating the values u_{ij} is accepted. The program for this computation method may be represented in the following way:

```
//Algorithm 12.2
// Parallel Gauss-Seidel method (first variant)
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
  dmax = 0; // maximum variation of values u
#pragma omp parallel for shared(u,n,dmax) \
                         private(i,temp,d)
  for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,n,dmax) \
                         private(j,temp,d)
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      omp_set_lock(dmax_lock);
        if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // end of inner parallel region
  } // end of outer parallel region
} while ( dmax > eps );
```

Algorithm 12.2. The first variant of the parallel Gauss-Seidel algorithm

It should be noted that the program is derived from the original sequential code by means of adding the directives and the operators of calling the functions of the OpenMP library (these additional lines in the program are marked by the dark color, the reverse slash at the end of the directives means that the directives will be continued in the following lines of the program).

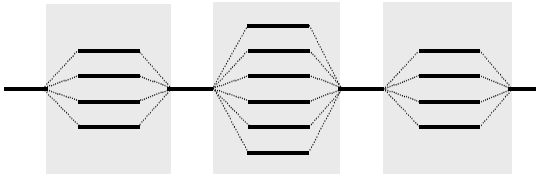

Figure 12.3. Parallel regions created by OpenMP directives

As it follows from the program code, the parallel regions in the given example set by the directive **parallel for** are nested. They include the *for* loop operators. The compiler supporting OpenMP technology divides the execution of the loop iterations among several program threads. The number of the threads is usually the same as the number of processors in the computational system. The parameters (*clauses*) of the directives **shared** and **private** define the data availability in the program threads. The variables described as shared are common for the threads. For the variables described as private separate copies are created for each thread. These copies may be used in the threads independently of each other.

|The availability of shared data provides the possibility of thread interaction. In this aspect the shared variables may be considered as thread *shared resources*. As a result, the definite rules of *mutual exclusion* must be observed in their use (it means that access of a thread for modifying shared data has to block accessing to these data for the other threads). The shared resource in our example is the value **dmax**; the access of threads to this value is regulated by the variable of special type (*semaphore*) **dmax-lock** and the functions **omp_set_lock** (access blocking) and **omp_unset_lock** (access unblocking). Such parallel program implementation guarantees the uniqueness of thread access for changing the shared data. The fragments of the program code (the blocks between calling the functions *omp_set_lock* and omp_unset_lock**),** for which mutual exclusion is provided, are usually called *critical sections*.

The results of computation experiments are given in Table 12.1 (hereinafter the four-processor server of the Nizhni Novgorod State University cluster with Pentium III, 700 Mhz, 512 RAM processors was used for the parallel programs developed with the use of OpenMP technology).

Table 12.1. The results of computation experiments for the Gauss-Seidel parallel algorithm (***p***=4)

Grid size	Gauss-Seidel sequential method (algorithm 12.1)		Parallel algorithm 12.2			Parallel algorithm 12.3		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03
900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

(***k*** – number of iterations, ***T*** – time in sec., ***S*** – speed-up)

Let us discuss the obtained result. The developed parallel algorithm is correct, i.e. it provides the solution of the problem as well as a sequential algorithm. The used approach guarantees achieving the practically maximum possible parallelism. Up to N^2 processors can be used for program execution. Nevertheless, the result can not be called satisfactory as the program will be working more slowly the original sequential implementation. The main reason for this is the excessively high synchronization of program parallel threads. In our example each parallel thread after averaging the values u_{ij} must check (and may be change) the value **dmax**. The permission for using the variable can be obtained be one thread only. The other threads must be blocked.

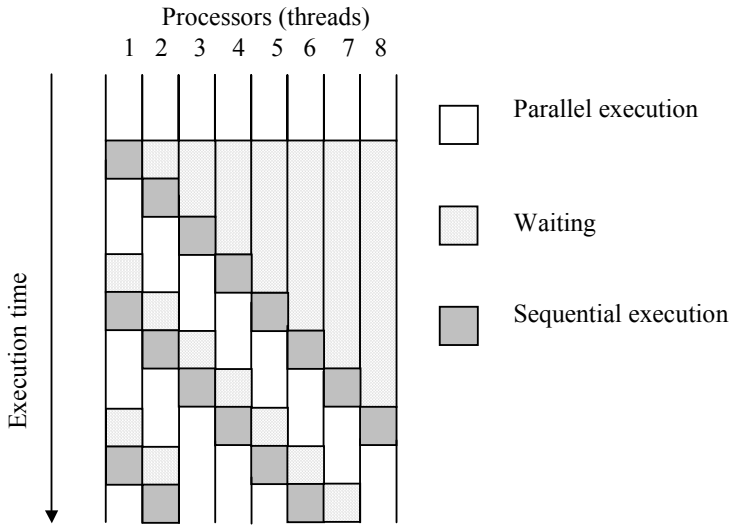

Figure 12.4. Thread serialization due to excessive synchronization

After the shared variable is released the next thread may get control etc. As access synchronization is necessary a multithread parallel program turns practically into a sequentially executable code, which is additionally less efficient that the original sequential variant because synchronization leads to additional computation costs (see Figure 12.4). It should be noted that despite the ideal distribution of computation load among processors for the given in Figure 6.4 combination of parallel and sequential computations, only two processors simultaneously can be executed at any given moment of time. This effect of parallelism degradation caused by intense synchronization of parallel threads is usually called *serialization*.

As the above discussion shows, the way of achieving efficient parallel computation is decreasing the necessary synchronization of parallel program fragments. Thus, in our case we may parallelize only one outer **for** loop. Besides to decrease possible blocking we may use a multilevel calculation scheme to estimate the maximum error. Let each parallel thread initially form its private estimation *dm* only for its own processed data (one or several grid rows). Then as calculations are completed each thread should compare its estimation *dm* with the shared general error estimation *dmax*.

The new variant of the Gauss-Seidel parallel algorithm looks as follows:

```
// Algorithm 12.3
// Parallel Gauss-Seidel method (second variant)
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // maximum variation of values u
#pragma omp parallel for shared(u,n,dmax)\
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // end of parallel region
} while ( dmax > eps );
```

Algorithm 12.3. The second variant of the parallel Gauss-Seidel algorithm

As a result of these modifications in computational scheme the number of accessing to the shared variable *dmax* decreases from N^2 to N times. It should lead to a considerable decrease in costs of thread synchronization and a decrease of computation serialization effect. The results of the experiments with this variant of parallel algorithm given in Table 12.1 show a remarkable change of the situation. The speed up in a number of experiments appears to be even greater than the number of processors used (this *superlinear speedup effect* is achieved due to the fact that each processor of the computation server has its own fast cash memory). It should be also noted that the improvement of parallel algorithm parameters is achieved by decreasing the maximum possible parallelism (no more than *N* processors may be used for executing the program).

12.2.3. Computation Indeterminacy

The latter variant of the parallel program provides practically maximum possible speedup of parallel calculations. Thus, the given speedup reached the value 5.9 in the executed experiments when a four-processor computation server was used. However, it should be noted that the developed computational calculation scheme has an essential feature - the sequence of grid node processing generated by the parallel method may vary at different program executions even if the initial parameters of the solved problem remain the same. This effect can be appeared due to some change of program execution conditions (computational loads, the algorithm of thread synchronization etc.). Such changeable environment may influence the time correlation among threads (see Figure 12.5). As a result the structure of thread locations in calculation domain may be different. Some threads may pass ahead the others and vice versa, a part of threads may remain behind (moreover the thread locations may change in the course of calculations). This behavior of parallel threads is usually called *the race condition*.

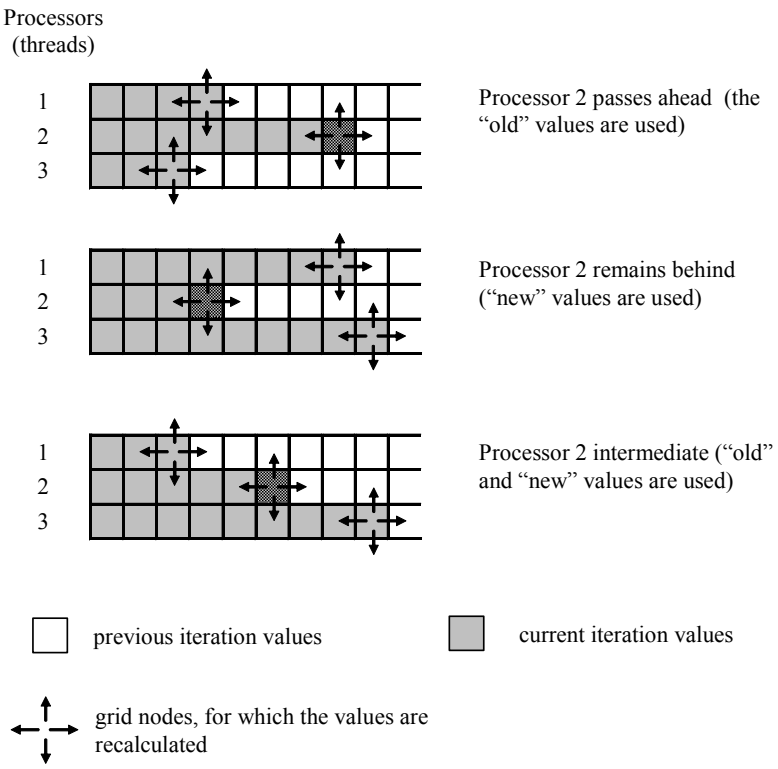

Figure 12.5. Various thread locations in the presence of race condition

In our case depending on the execution conditions the different (from the precious or the current iterations) values of the neighboring vertical nodes may be used for calculating the new values u_{ij}. Thus, the number of method iterations before meeting the accuracy requirements and, what is more important, the obtained problem solution may differ at several repeated program executions. The obtained estimations of the values u_{ij} will correspond to the precise problem solution within the range of the required accuracy but nevertheless they may be different. The computational scheme based on such calculations for grid algorithms got the name of *the chaotic relaxation method*.

In general case the indeterminacy of the results is not desirable as it disturbs the main basis of complicated software development which consist in the repeatability of the results of program executions with the same values of initial data. As a result seeking calculation determinacy we can state the important principle of parallel

programming, which requires that the time dynamics of parallel threads should not influence executing parallel programs.

12.2.4. Computation Deadlock

A possible approach for achieving the uniqueness of the execution results (escaping the thread race condition) may be the restriction of access to the grid nodes, which are processed in parallel program threads. A set of semaphores *row_lock[N]* may be introduced for this purpose, which will allow the threads to block the access to their grid rows:

```
// the thread is processing the grid row i
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// <processing the grid row i>
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```

Having blocked the access to its data, a parallel thread will not depend on the execution dynamics of the other parallel program threads. The thread computation result is unambiguously defined by the data values at the time of calculation start.

This approach allows us to demonstrate one more problem, which may appear in the course of parallel computations. The essence of the problem is that a conflict between parallel threads may appear in accessing to multiple shared variables, and this conflict may not be solved successfully. Thus, in the given fragment of the program code when the threads process any two sequential rows (for instance, rows 1 and 2) this situation may arise when each thread block first its row 1 and 2 correspondingly and only then pass over to blocking the rest of required rows (see Figure 12.6). In this case the access to the necessary rows will be provided for neither of the threads. The situation is insolvable. It is usually called a *deadlock*. As it be shown the necessary condition of the deadlock is the presence of a cycle in the resource request and distribution graph. For our case the strict sequent sequence of row blocking may help to escape the cycle:

```
// the thread is processing the grid row i
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i-1]);
// <processing the grid row i>
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i-1]);
```

(it should be noted that this scheme of row blocking may also lead to a deadlock if we consider the modified Dirichlet problem, the horizontal borders of which are identified.

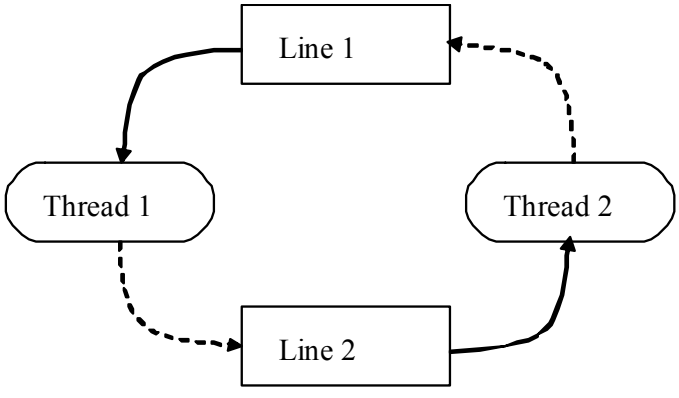

Figure 12.6. Deadlock situation at accessing rows of the grid (the thread 1 possesses row 1 and requests row 2, the thread 2 possesses row 2 and requests row 1)

12.2.5. Indeterminacy Elimination

The approach discussed in 12.2.4 decreases the effect of thread races but it does not guarantee the solution uniqueness at recurrent calculations. To achieve a determinacy it is necessary to use additional computational schemes.

The possible method which is widely used in practice is to separate the memory for storing the computation results at the previous and the current iterations of the grid method. The scheme of this approach may be represented as follows:

```
// Algorithm 12.4
// Parallel Gauss-Jacobi method
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // maximum change of the values u
#pragma omp parallel for shared(u,n,dmax) \
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      un[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-un[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // end of parallel region
  for ( i=1; i<N+1; i++ ) // data updating
    for ( j=1; j<N+1; j++ )
      u[i][j] = un[i][j];
} while ( dmax > eps );
```

Algorithm 12.4. Parallel implementation of the Gauss-Jacobi method

As it follows from the given algorithm the results of the previous iterations are stored in the array **u** , and new calculation values are stored in the additional array **un**. As a result, the value u_{ij} from the previous method iterations are always used for calculations regardless of the calculation execution order. This scheme of the algorithm implementation is usually called *the Gauss-Jacobi method*. It guarantees the result determinacy regardless of the parallelizing method. But it should be noted that this method requires the great additional memory amount and possesses a lesser (in comparison to the Gauss-Seidel algorithm) rate of convergence. The results of the calculations with the sequential and parallel variants of the method are given in the Table 12.2.

Table 12.2. Results of computational experiments for the Gauss-Jacobi algorithm (***p***=4)

Mesh size	Sequential Gauss-Jacobi method (algorithm 12.4)		Parallel method developed on the analogy with algorithm 12.3		
	k	***T***	***k***	***T***	***S***
100	5257	1,39	5257	0,73	1,90
200	23067	23,84	23067	11,00	2,17
300	26961	226,23	26961	29,00	7,80
400	34377	562,94	34377	66,25	8,50
500	56941	1330,39	56941	191,95	6,93
600	114342	3815,36	114342	2247,95	1,70
700	64433	2927,88	64433	1699,19	1,72
800	87099	5467,64	87099	2751,73	1,99
900	286188	22759,36	286188	11776,09	1,93

1000	152657	14258,38	152657	7397,60	1,93
2000	337809	134140,64	337809	70312,45	1,91
3000	655210	247726,69	655210	129752,13	1,91

(***k*** – the number of iterations, ***T*** – time in sec., ***S*** – speed up)

Another possible approach to eliminate the mutual dependences of parallel threads is to apply *the red/black row alteration scheme*. In this scheme each iteration of the grid method execution is subdivided into two sequential stages. At the first stage only the rows with even numbers are processed, and then at the second stage the rows with odd numbers (see Figure 12.7). The given scheme may be generalized for simultaneous application to the rows and the columns (checkerboard decomposition scheme) of the calculation domain.

The discussed scheme of row alteration does not require any additional memory in comparison to the Jacobi method and guarantees the solution determinacy at multiple program executions. But it should be noted that both the approaches considered in this Subsection can have the results that do not coincide with the Dirichlet problem solution, which is obtained by means of the sequential algorithm. Besides these computational schemes have a smaller domain and a worse convergence rate than the original variant of the Gauss-Seidel method.

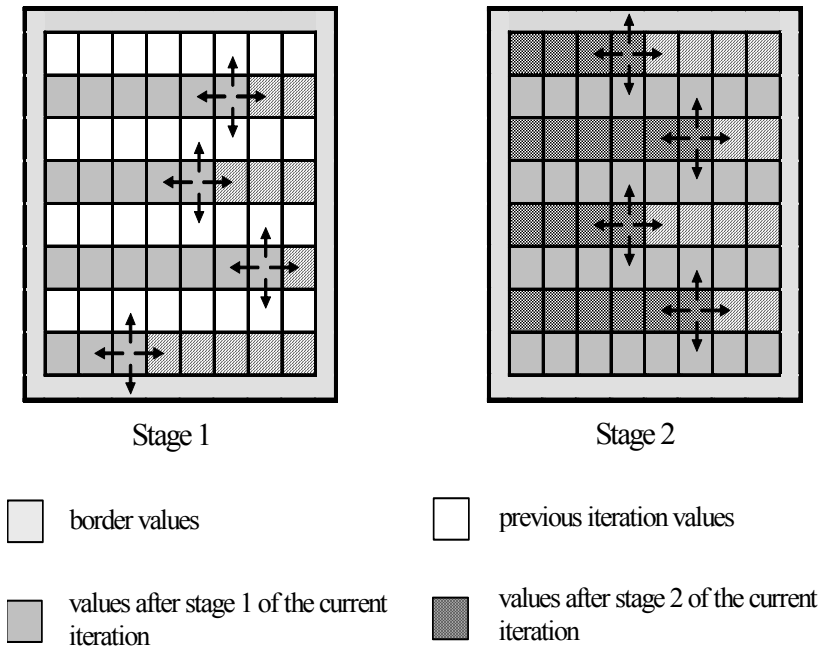

Figure 12.7. The red/black row alteration scheme

12.2.6. Wavefront Computation Scheme

Let us now consider the possibilities of developing such parallel algorithm, which would execute only those computations that the sequential method would (may be in a slightly different order). As a result, such an algorithm would provide obtaining exactly the same solution of the original problem. As it has already been noted above, each sequent *k* approximation of the value u_{ij} is calculated according to the last *k* approximation of the values $u_{i-1,j}$ and $u_{i,j-1}$ and the next to last (*k-1*) approximation of the values $u_{i+1,j}$ and $u_{i,j+1}$. As a result, if the results of the sequential computational schemes and parallel computational schemes calculations must coincide at the beginning of each method iteration only one value u_{11} may be recalculated (there is no possibility for parallelizing). But when the value u_{11} has been recalculated, the computations may be executed in two grid nodes u_{12} and u_{21} (the conditions of the sequential scheme are met in these nodes). Then after the recalculation of u_{12} and u_{21}, calculations may be performed at nodes u_{13}, u_{22} and u_{31} etc.

Summing it up, it is possible to see that the grid method iteration execution may be divided into a number of steps. At each step the nodes of the bottom-up grid diagonal with the numbers defined by the stage number (see Figure 12.8) will be ready for calculations. The set of grid nodes that can be recalculated in accordance with this computational scheme got the name of *a wave* or *calculation front*. The algorithms developed on the basis of

such calculations were called *wavefront* or *hyperplane methods*. It should be noted that the wave size (*the degree of possible parallelism*) changes dynamically in the course of calculations. The wave reached its peak and then decays approaching the right lower grid node.

Table 12.3. The results of the experiments for the parallel Gauss-Seidel algorithm with the wavefront computational scheme (***p***=4)

Grid size	Sequential Gauss-Seidel method (algorithm 12.1)		Parallel algorithm 12.5			Parallel algorithm 12.6		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,30	0,21	210	0,16	0,40
200	273	0,34	273	0,86	0,40	273	0,59	0,58
300	305	0,88	305	1,63	0,54	305	1,53	0,57
400	318	3,78	318	2,50	1,51	318	2,36	1,60
500	343	6,00	343	3,53	1,70	343	4,03	1,49
600	336	8,81	336	5,20	1,69	336	5,34	1,65
700	344	12,11	344	8,13	1,49	344	10,00	1,21
800	343	16,41	343	12,08	1,36	343	12,64	1,30
900	358	20,61	358	14,98	1,38	358	15,59	1,32
1000	351	25,59	351	18,27	1,40	351	19,30	1,33
2000	367	106,75	367	69,08	1,55	367	65,72	1,62
3000	370	243,00	370	149,36	1,63	370	140,89	1,72

(***k*** – the number of iterations, ***T*** – time in sec., ***S*** – speed up)

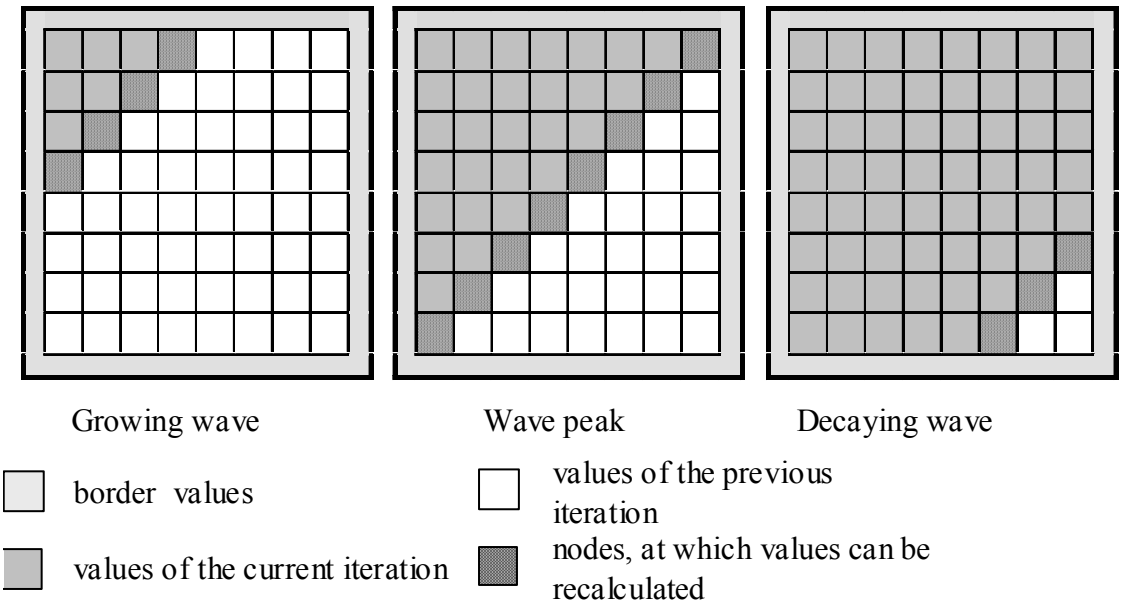

Figure 12.8. The wavefront computational scheme

The possible scheme of the parallel method, which is based on the wavefront computational scheme, can be given in the following way:

```
// Algorithm 12.5
// Parallel Gauss-Seidel method (wavefront computational scheme)
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // maximum change of the values u
  // growing wave (nx – the wave size)
  for ( nx=1; nx<N+1; nx++ ) {
```

```
      dm[nx] = 0;
  #pragma omp parallel for shared(u,nx,dm) \
                           private(i,j,temp,d)
      for ( i=1; i<nx+1; i++ ) {
        j    = nx + 1 – i;
        temp = u[i][j];
        u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                  u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
        d = fabs(temp-u[i][j])
        if ( dm[i] < d ) dm[i] = d;
      } // end of parallel region
    }
    // decaying wave
    for ( nx=N-1; nx>0; nx-- ) {
  #pragma omp parallel for shared(u,nx,dm) \
                           private(i,j,temp,d)
      for ( i=N-nx+1; i<N+1; i++ ) {
        j    = 2*N - nx – I + 1;
        temp = u[i][j];
        u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                  u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
        d = fabs(temp-u[i][j])
        if ( dm[i] < d ) dm[i] = d;
      } // end of parallel region
    }
  #pragma omp parallel for shared(n,dm,dmax) \
                           private(i)
    for ( i=1; i<nx+1; i++ ) {
      omp_set_lock(dmax_lock);
        if ( dmax < dm[i] ) dmax = dm[i];
      omp_unset_lock(dmax_lock);
    } // end of parallel region
  } while ( dmax > eps );
```

Algorithm 12.5. The Gauss-Seidel parallel algorithm with the wavefront computational scheme

As it can be noted the estimation of the solution error may be calculated for each separate row (array *dm*). This array is shared for all executed threads. However, the synchronization for accessing to the elements is not required, as the threads always use different array elements (the calculation wave front contains only one node of each grid row).

After all the wave elements have been processed the maximum error of computation iteration is found among elements of the array *dm*. However, this final part of calculations may appear to be the least efficient due to the high additional synchronization cost. The situation may be improved by increasing the size of sequential fragments and thus, decreasing the necessary interactions of parallel threads.

The possible variant to implement this approach may be the folowing:

```
  chunk = 200; // sequential part size
  #pragma omp parallel for shared(n,dm,dmax) \
                           private(i,d)
    for ( i=1; i<nx+1; i+=chunk ) {
      d = 0;
      for ( j=i; j<i+chunk; j++ )
        if ( d < dm[j] ) d = dm[j];
      omp_set_lock(dmax_lock);
        if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // end of parallel region
```

This technique of enlarging sequential computation fragments for decreasing the synchronization overhead is called *chunking*. The experimental results for this parallel computation variant are given in Table 12.3.

One more aspect in analyzing the efficiency of the developed parallel algorithm is worth mentioning. The calculation wave front does not agree properly with the rules of the efficient cache use – fast cache memory used to store the copies of the most frequently used main memory areas. The use of cache may considerably increase in tens of times the computation speed. Data allocation in cash may occur either preliminarily (when some

algorithm is used for predicting data needs) or at the moment of the data reading from main memory. It should be noted that caching is executed not by single element but by small sequential memory blocks (*cache lines*). The typical cache line sizes are usually equal to 32, 64, 128, 256 bites. As a result, the effect of cache presence will be observed, if the performed calculations use the same data repeatedly (*data processing localization*) and provide accessing to memory elements with sequentially increasing addresses (*sequential access*).

In the considered algorithm data allocation is realized rowwise but the calculation wave front is located on grid diagonal. This leads to low efficiency of cache use. A possible way to remedy the situation is to enlarge computational operations. To follow this intension the procedure of processing some rectangular calculation subdomains (*block*) can be considered as operations that can be distributed among the processors – see Figure 12.9.

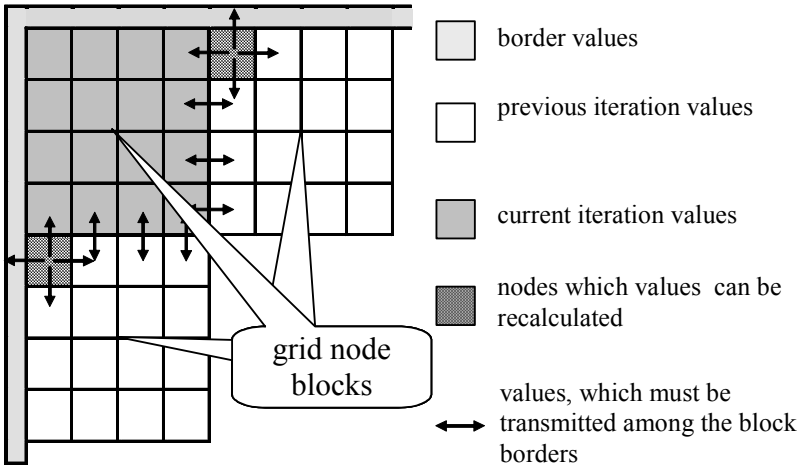

Figure 12.9. Block-oriented decomposition of grid calculation domain

In the most general way the computational method developed on the basis of this approach may be described as follows (the calculation domain is represented as the block grid of size *NBxNB*).

```
// Algorithm 12.6
// Parallel Gauss-Seidel method (block wavefront computational scheme)
do { // NB is the size of row/column of blocks
  // growing wave (wave size is nx+1)
  for ( nx=0; nx<NB; nx++ ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = nx – i;
      // <processing the block with coordinates (i,j)>
    } // end of parallel region
  }
  // decaying wave
  for ( nx=NB-2; nx>-1; nx-- ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = 2*(NB-1) - nx – i;
      // < processing the block with coordinates (i,j)>
    } // end of parallel region
  }
  // <calculation error estimation>
} while ( dmax > eps );
```

Algorithm 12.6. Block-oriented wavefront computational scheme

The calculations in the suggested algorithm are performed in accordance with the wavefront data processing scheme. First calculations are performed only in the upper left block with the coordinates (0,0). Then the blocks with the coordinates (0,1) and (1,0) become available for processing – the experimental results for this parallel method are given in Table 12.3.

The block-oriented approach to the wavefront data processing method considerably changes the situation. It means that processing nodes may be arranged rowwise, access to the data may be executed sequentially along the memory elements, the values transferred to the cache are used repeatedly. Besides, as block processing is performed on different processors and the blocks are mutually disjoint in data, there will be no additional costs for providing cache coherence of different processors.

The best cache utilizations will be achieved if there is enough space in the cache for allocating not fewer than three block rows as just the data of three block rows will be simultaneously used in processing a block row. Thus, having the definite cache size it is possible to define the recommended maximum possible block size. For instance, if cache is 8 Kb and the size of data values is 8 bytes, this size is approximately 300 (8Кб/3/8). It is also possible to define the minimum allowable block size on condition that the cache line and block row sizes are the same. Thus, if the cache line size is 256 bytes and the size of data values is 8 bytes, the block size must be divisible by 32.

The last remark should be made concerning the transmission of border block nodes between processors. Taking into consideration the border transmission, it is appropriate to process the neighboring blocks on the same processors. Otherwise it is possible to define the block sizes in such a way, that the amount of the border data transmitted among processors should be minimal. Thus, if the cache line size is 256 bytes, the size of data values is 8 bytes and the block size is 64x64, the amount of the transmitted data is 132 cache lines; if the block size is 128x32, the amount is only 72 cache lines. Such optimization is of principal importance in slow operations of transmitting data among processor caches, particularly in case of nonuniform memory access systems (NUMA).

12.2.7. Computation Load Balancing

As it has been previously mentioned, the computational load in wavefront data processing changes dynamically in the course of calculations. This aspect should be taken into account in distributing the computational load among processors. Thus, for instance, if the wave front is 5 blocks and 4 processors are used, wavefront processing will require two parallel iterations. During the second iteration only one processor will be employed. The rest will be idle waiting for the end of calculations. The calculation speedup achieved in this case will be equal to 2.5 instead of potentially possible 4. This decrease of processor utilization efficiency becomes less obvious if the wave length is greater. The size of the wave length may be regulated by the block size. As a general result, it may be observed that the block size determines the calculation *granularity level* for parallelizing. Regulating this parameter it is also possible to manage the parallel computation efficiency.

It is possible to use one more approach to provide the uniform processor loads (*load balancing*) which is widely used in parallel computations. In accordance with this approach all the computational operations ready for execution are collected in *a job queue*. In the course of computations the processor, which is already free, may request for the job from the queue. Additional jobs, which appear in the course of data processing, should be placed in the queue as well. This scheme of processor load balancing is simple, obvious and efficient. As a result a job queue may be considered as the general parallel computation management scheme for processor load balancing l in case of share memory computer systems.

The discussed scheme may be also used for the block-oriented wavefront parallel computations. Actually, some blocks of the next calculation wave are gradually formed in the course of current wave front processing. These blocks may be also employed for processing if there is a shortage of computational load for processors.

The general computational scheme with the use of the job queue may be represented as follows:

```
// Algorithm 12.7
// Parallel Gauss-Seidel method (block wavefront computational scheme)
//  job queue is used for processor load balansing
// <data initialization>
// <loading initial block pointer into the job queue >
// pick up the block from the job queue (if the job queue is not empty)
while ( (pBlock=GetBlock()) != NULL ) {
  // <block processing>
  // neighboring block availability mark
  omp_set_lock(pBlock->pNext.Lock); // right neighbour
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 )
      PutBlock(pBlock->pNext);
  omp_unset_lock(pBlock->pNext.Lock);
  omp_set_lock(pBlock->pDown.Lock); // lower neighbour
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 )
```

```
        PutBlock(pBlock->pDown);
      omp_unset_lock(pBlock->pDown.Lock);
    } // end of computations, as the queue is empty
```

Algorithm 12.7. Block-oriented wavefront computational scheme with the use of job queue

To describe the grid node blocks the block descriptor with the following parameters is used in the algorithm:

- **Lock** – semaphore, which synchronizes access to the block descriptor,
- **pNext** – pointer to the right neighboring block,
- **pDown** – pointer to the lower neighboring block,
- **Count** – counter of the block availability for computations (the number of available block borders).

The operations of selecting a block from the queue and inserting a block into the queue are provided correspondingly by the function *GetBlock* and *PutBlock*.

As it follows from the scheme, a processor gets the block for processing from the queue, performs all the necessary calculations for the block and marks the availability of its borders for its right and lower neighboring blocks. If it appears that the neighboring blocks have both the borders available, the processor transmits these blocks into the job queue.

The use of the job queue makes possible to solve practically all the remaining problems of developing parallel computations for shared memory multiprocessor systems. The development of this approach can provide for refinement of the rule of selecting jobs from the queue in accordance with the processor states (it is appropriate to process the neighboring blocks on the same processors). Then in some cases it is useful to create several job queue etc. Additional information on these problems may be found, for instance, in Xu, Hwang (1998), Buyya (1999).

12.3. Parallel Computation for Distributed Memory Systems

The use of processors with distributed memory is another general way to create multiprocessor computer systems. Such systems become more and more acute nowadays as a result of the wide spread of high-performance cluster computer systems (see Section 1).

Many parallel programming problems such as computation race, deadlocks, serialization are common for the systems with shared and distributed memory. The aspect specific for parallel computations with distributed memory is the interaction of parallel program parts on different processors, which may be provided only through message passing.

It should be noted that a distributed memory processor is, as a rule, a more sophisticated computational node than a processor in a multiprocessor shared memory system. In order to take these two differences into account we will further refer to the distributed memory processor as to a computational server. In particular a server may be represented by a multiprocessor shared memory system. In all the experiments described below we used a cluster with 4 computers with Pentium IV, 1300 Mhz, 256 RAM, 100 Mbit Fast Ethernet.

12.3.1. Data Distribution

The first problem to be solved in developing parallel computations in distributed memory systems is choosing the method of distributing the processed data among the computational servers. The efficiency of the distribution is defined by the achieved degree of computation localization on the servers (due to considerable time delays in message passing the intensity of server interactions should be minimal).

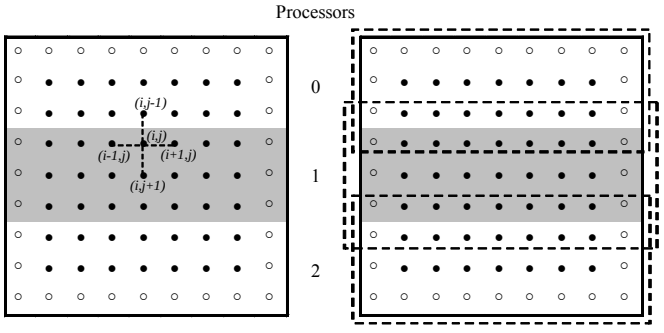

Figure 12.10. Grid block-striped distribution among processors (border nodes are marked by circles)

In the Dirichlet problem discussed here there are two different data distribution methods: *one-dimensional* and *block-striped* scheme (see Figure 12.10) or *two-dimensional* or *checkerboard* computational grid decomposition (see also Section 7). Our discussion will be based on the first method. The checkerboard scheme will be briefly discussed further.

In case of block-striped decomposition the computational grid domain is divided into *horizontal* or *vertical* stripes (without loss of generality we will further consider only the horizontal stripes). The number of stripes can be defined to be equal to the number of processors. The stripe sizes are usually selected the same. The horizontal border nodes (the first and the last lines) are included into the first and the last stripes correspondingly. The bands for processing are distributed among the processors.

The essential aspect of parallel computations with block-striped data distribution is that the border rows of the previous and the next computational grid stripes should be duplicated on the processor, which performs processing of some grid node stripe. The border rows of the previous and the next computational grid stripes can be considered as a result of the stripe expansion (these extended stripes are shown in Figure 12.10 by dotted frames). The duplicated stripe border rows are used only for calculations, the recalculations of the rows should be performed in the stripes of the initial location. Thus, border row duplicating has to be executed prior to the beginning of the execution of each iteration.

12.3.2. Information Communications among Processors

The parallel Gauss-Seidel method based on block-striped data distribution suggests simultaneous stripe processing on all servers as described below.

```
// Algorithm 12.8
// Parallel Gauss-Seidel method
//   distribured memory, band-striped data distribution
// operations executed on each processor
do {
  // <border stripe row exchange with the neighbours >
  // < stripe processing>
  // < calculations of general error computation dmax >}
 while ( dmax > eps ); // eps - accuracy
```

Algorithm 12.8. Parallel Gauss-Seidel algorithm based on block-striped data distribution

For further consideration the following notation is used:

- **ProcNum** – the number of processor, which performs the described operations,
- **PrevProc, NextProc** – the numbers of neighboring processors containing the previous and the following stripes,
- **NP** – the number of processors,
- **M** – the number of rows in a stripe (without considering the duplicated border rows),
- **N** – the number of inner nodes in a grid row (i.e. the total number of row nodes is equal to *N+2*).

To indicate the band rows we will use the enumeration where the rows *0* and *M+1* are the border rows duplicated from the neighboring stripes and own stripe rows of the processor are enumerated from *1* to *M*.

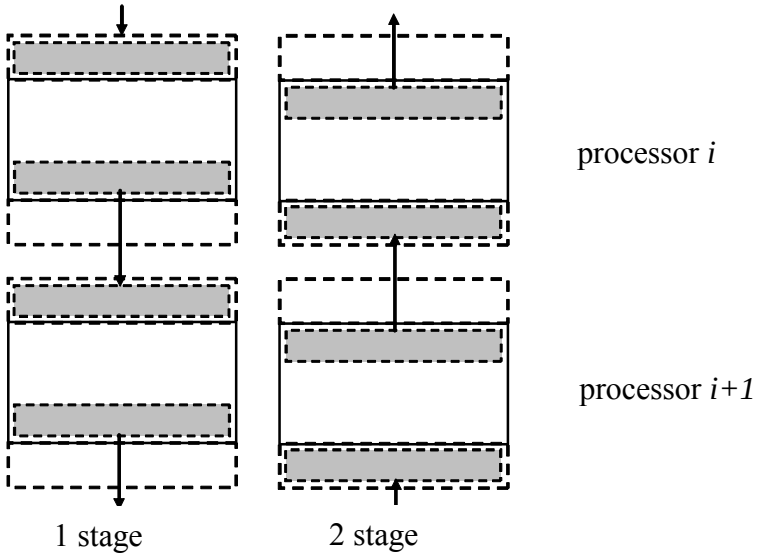

Figure 12.11. Border row transferring among the neighboring processors

The procedure of border row exchange among the neighboring processors may be divided into two sequent stages. During the first stage each processor transmits its lowest border row to the following processor and receives the similar row from the previous processor (see Figure 12.11). The second transmission stage is performed in the reverse order: the processors transmit their upper border rows to the previous neighbors and receive similar rows from the following neighboring processors.

In a general way carrying out such data transmission operations may be represented as follows (for simplicity let us discuss only the first stage of the exchange procedure):

```
// transmission of the lowest border row to the following processor
// and receiving the border row from the previous processor
if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
```

(a format closed to MPI (see Section 4 and, for instance, Group, et al. (1994)) is used the communication functions *Send* and *Receiv*. The first and the second parameters of the functions represent the transmitted data and their amount, and the third parameter defines the addressee (for the function *Send*) or the source (for the function *Receive*).

Two different mechanisms may be employed for data processing. If the first one is used, carrying out the programs, which initiated transmission, is suspended until the termination of all data transmission operations (i.e. until the moment when the addressee processor receives all the transmitted data). The communication operations, which are implemented in this way, are usually called *synchronous* or *blocking*. Another approach – *asynchronous* or *nonblocking* communications – the function of this type only initiates the transmission process and hereon terminate their execution. As a result, the programs can continue the computational operations without waiting for the termination of the long-continued communication operations. When necessary the programs may check the availability of the transmitted data. These two alternatives are widely used in parallel computations and have their advantages and disadvantages. The synchronous communication procedures are, as a rule, simpler for use and more reliable. The nonblocking operations allow for combining the processes of data transmission and computations, but they usually cause the increase of programming complexity. Taking into account the above given discussion, we will use the blocking communication operations in all the following examples.

The above given sequence of the blocking communication operations (first the function *Send*, then the function *Receive*) leads to a strictly sequential scheme of row transmissions, as all the processors address the operation *Send* simultaneously and pass into the waiting mode. The server *NP-1* will be the first processor to be ready for receiving the transmitted data. As a result, the processor *NP-2* will perform the operation of sending its border row and pass over to receiving a row from the processor NP-3 etc. The total number of such steps is equal to *NP-1*. The execution of the second stage of the procedure (transmitting the upper border rows) is done analogously (see Figure 12.11).

The sequential execution of the described data transmission operations is defined by the selected implementation method. To improve the situation the send-receive alternation scheme can be applied for the processors with even and odd numbers:

```
// sending the lowest border row to the following processor and
// receiving the transmitted row from the previous processor
if ( ProcNum % 2 == 1 ) { //  odd processor
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
}
else { //  even processor
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
}
```

This technique makes possible to perform all the necessary data transmission operations in two sequent steps. During the first step all odd processors send data, and the even processors receive the data. During the second step the processors change their roles: the even processors perform the function *Send*, the odd processors perform the function *Receive*.

The described sequence of send-receive operations for the neighboring processor interaction is widely used in practice of parallel computations. As a result, the procedures for supporting such operations are available in many basic parallel program libraries. Thus, the standard MPI (see Section 4, for instance, Group, et al. (1994)) provides for the function **Sendrecv.** This function allows to write the above part of the program code more concisely:

```
// sending the lowest border row to the following processor and
// receiving the transmitted row from the previous processor
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

This integrated function *Sendrecv* is usually used to provide both correct operating on bordering processors (when there is no need to perform all send-receive operations) and executing transmission alteration scheme on the processors for escaping deadlock situations. The function implementation also provides for the possibilities to execute all the necessary data communications in parallel.

12.3.3. Collective Communications

To complete the spectrum of problems related with parallel implementation of grid method for distributed memory systems, it is necessary to discuss the methods of evaluating computation error. A possible approach is to transmit all the local error estimations obtained on separate grid stripes to some processor. The processor will calculate the maximum value and send it to all the processors. However, this scheme is utterly inefficient, as the number of the necessary data communication operations is defined by the number of processors and the operations may be carried out only sequentially. Meanwhile the analysis of the required communication operations shows that the operation of collecting and sending the data may be executed through a data processing cascade scheme, discussed in Section 2.5. Actually, obtaining the maximum value of local errors calculated on each processor may be provided, for instance, by means of preliminary finding the maximum values for separate processor pairs. Such calculations may be performed in parallel. It will be possible then to perform pairwise search of the maximum among the obtained results etc. All in all according to the cascade scheme it is necessary to perform $log_2 NP$ of parallel iterations to achieve the total value of computation error (*NP* is the number of processors).

As the cascade scheme is highly efficient for performing collective communication operations, procedures for such operations is available in the majority of basic parallel program libraries. Thus, the standard MPI (see Section 4 and, for instance, Group, et al. (1994)) provides for the following operations:

- **Reduce(dm,dmax,op,proc)** – the procedure of collecting the final result *dmax* on the processor *proc* among the values *dm*, located on each processor, with the help of operation *op*,
- **Broadcast(dmax,proc)** – the distribution procedure of the value *dmax* from processor *proc* to all the available processors.

With regard to the mentioned procedures the general computation scheme for the Gauss-Seidel parallel method on each processor may be represented as follows:

```
// Algorithm 12.8 (modified variant)
// Parallel Gauss-Seidel method
//   distribured memory, band-striped data distribution
// operations performed on each processor
do {
  // border stripe row exchange with the neighbours
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <stripe processing with error estimation dm >
  // calculations of general error computation dmax
  Reduce(dm,dmax,MAX,0);
  Broadcast(dmax,0);
} while ( dmax > eps ); // eps – accuracy
```

(the variable *dm* in the described algorithm is the local accuracy of computations on a separate processor. Parameter *MAX* sets the operation of maximum value search for collection operation). It should be noted that the standard MPI contains the function *Allreduce*, which combines the operations of data reduction and distribution. The results of the experiments for the given parallel computation variant for the Gauss-Seidel method are given in Table 12.4.

12.3.4. Wavefront Computations

The algorithms described in sections 1-3 define the general scheme of parallel computations for the Gauss-Seidel method in multiprocessor distributed memory systems. This scheme can further be specified by the corresponding implementation of practically all the computational schemes, which are described for shared memory systems (the use of additional memory for the Gauss-Jacobi scheme, the row processing alteration etc.). The development of these variants does not cause any new effects with regard to parallel computations. The analysis of these variants may be used as a topic for individual studies.

Table 12.4. Experimental results for the Gauss-Seidel parallel method (distributed memory, block-striped distribution, p=4)

Grid size	Gauss-Seidel sequential method (algorithm 12.1)		Parallel algorithm 12.8			Parallel algorithm base on the wavefront calculation scheme (see 12.3.4)		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,54	0,11	210	1,27	0,05
200	273	0,35	273	0,86	0,41	273	1,37	0,26
300	305	0,92	305	0,92	1,00	305	1,83	0,50
400	318	1,69	318	1,27	1,33	318	2,53	0,67
500	343	2,88	343	1,72	1,68	343	3,26	0,88
600	336	4,04	336	2,16	1,87	336	3,66	1,10
700	344	5,68	344	2,52	2,25	344	4,64	1,22
800	343	7,37	343	3,32	2,22	343	5,65	1,30
900	358	9,94	358	4,12	2,41	358	7,53	1,32
1000	351	11,87	351	4,43	2,68	351	8,10	1,46
2000	367	50,19	367	15,13	3,32	367	27,00	1,86
3000	364	113,17	364	37,96	2,98	364	55,76	2,03

(***k*** – the number of iterations, ***T*** – time in sec., ***S*** – speed up)

Finally let us analyze such possibilities of parallel computations that would provide for finding the same Dirichlet problem solutions as the use of initial sequential Gauss-Seidel method does. As it has been mentioned previously, this result may be obtained by the wavefront calculation scheme. To form a calculation wave let us represent logically each processor stripe as a set of blocks (the block size may be assumed to be equal to the stripe width) and implement stripe processing blockwise sequentially (see Figure 12.12). To follow the sequential algorithm calculations may be started only for the first block of the first processor stripe. After the block has been processed, two more blocks will be available for calculations – block 2 of the first stripe and block 1 of the second stripe (to process the block of the second stripe it is necessary to transmit the border block row of the first stripe). After these blocks are processed, three more blocks will be available for computations. So we observe the familiar process of wavefront data processing (see the experimental results in Table 12.4).

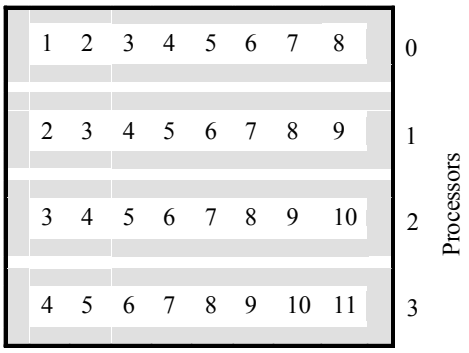

Figure 12.12. Wavefront computational scheme in case of block-striped dada distribution among processors

The attempt to combine border row transmission operations and the operations on data block processing may be an interesting aspect of developing such parallel computation scheme.

12.3.5. Checkerboard Data Distribution

Another approach for distributing calculation grid among processor is the checkerboard block decomposition scheme (see Figure 12.9 and Section 7). This radical change of grid distribution method will not require practically any substantial corrections of the parallel computation scheme, which has already been discussed. The essentially new aspect in block data representation is the increase of the number of the border block rows on each processor (their number becomes equal to 4). It leads correspondingly to a greater number of data communication operations in border row exchange. The analysis of border row transmissions shows that in case of the block-striped scheme 4 data send-receive operations are performed on each processor and *(N+2)*

values are sent in each message. In case of the checkerboard block method 8 transmission operations take place and the size of each message is equal to ($N/\sqrt{NP}+2$) where N is the number of inner grid nodes, NP is the number of processors, the sizes of all blocks are assumed to be equal. Thus, the use of the checkerboard block decomposition scheme is appropriate if the number of grid nodes is sufficiently big. In this case the increase of communication operations compensate by decreasing costs of data transmission due to the reduction of the transmitted message sizes. The experimental results in case of checkerboard data distribution scheme are shown in Table 12.5.

Table 12.5. Experimental results for the Gauss-Seidel parallel method (distributed memory, checkerboard distribution, ***p***=4)

Grid size	Gauss-Seidel sequential method (algorithm 12.1)		Parallel algorithm based on the checkerboard data distribution (see 12.3.5)			Parallel algorithm 12.9		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,71	0,08	210	0,60	0,10
200	273	0,35	273	0,74	0,47	273	1,06	0,33
300	305	0,92	305	1,04	0,88	305	2,01	0,46
400	318	1,69	318	1,44	1,18	318	2,63	0,64
500	343	2,88	343	1,91	1,51	343	3,60	0,80
600	336	4,04	336	2,39	1,69	336	4,63	0,87
700	344	5,68	344	2,96	1,92	344	5,81	0,98
800	343	7,37	343	3,58	2,06	343	7,65	0,96
900	358	9,94	358	4,50	2,21	358	9,57	1,04
1000	351	11,87	351	4,90	2,42	351	11,16	1,06
2000	367	50,19	367	16,07	3,12	367	39,49	1,27
3000	364	113,17	364	39,25	2,88	364	85,72	1,32

(***k*** – the number of iterations, ***T*** – time in sec., ***S*** – speed up)

The wavefront computation method may be also implemented in case of checkerboard block distribution (see Figure 12.13). Let the processors form a rectangular grid, the size of which is $NBxNB$ ($NB=\sqrt{NP}$) and let the processors be enumerated from 0 from left to right according to the rows of the grid.

The general parallel computation scheme in this case looks as follows:

```
// Algorithm 12.9
// Parallel Gauss-Seidel method
//   distribured memory, checkerboard data distribution
// operations executed on each processor
do {
  // obtaining border nodes
  if ( ProcNum / NB != 0 ) { // processor row number is not equal to 0
    // obtaining data from upper processor
    Receive(u[0][*],M+2,TopProc); // upper row
    Receive(dmax,1,TopProc);      // calculation error
  }
  if ( ProcNum % NB != 0 ) { // processor column number is not equal to 0
    // obtaining data from left processor
    Receive(u[*][0],M+2,LeftProc); // left column
    Receive(dm,1,LeftProc);        // calculation error
    If ( dm > dmax ) dmax = dm;
  }
  // <processing a block with error estimation dmax >
  // transmission of border nodes
  if ( ProcNum / NB != NB-1 ) { // processor row is not last one
    //data transmission to the lower processor
    Send(u[M+1][*],M+2,DownProc); // bottom line
    Send(dmax,1,DownProc);        // calculation error
  }
  if ( ProcNum % NB != NB-1 ) { // processor column is not last one
```

```
      // data transmission to the right processor
      Send(u[*][M+1],M+2,RightProc); // right column
      Send(dmax,1, RightProc);       // calculation error
    }
    // synchronization and error dmax distribution
    Barrier();
    Broadcast(dmax,NP-1);
  } while ( dmax > eps ); // eps –accuracy
```

Algorithm 12.9. Parallel Gauss-Seidel algorithm based on checkerboard data distribution

(the function *Barrier* is the collective synchronization operation, i.e. each concurrently executed program part is suspended at the moment of calling this function, calculations can be resumed only after calling the function by all the parts of parallel program).

It should be noted that at the initial moment all the processors (except processor 0) must be in the state of waiting their border nodes (the upper row and the left column). The calculations must be started by the processor with the left upper block. After terminating the block processing the recalculated data of the right column and the lower block row should be transmitted to the right and the lower grid processors correspondingly. These actions provide unblocking the processors of the second processor diagonal (see the left part of Figure 12.13) etc.

The analysis of the wavefront computation efficiency in distributed memory multicomputer systems (see Table 12.5) demonstrates a considerable decrease of processor utilization because processors are involved in data processing only at the moment when their blocks belong to calculation wave front. In case of distributed memory load balancing (data redistribution) is very complicated, as it requires transmitting large amounts of data among processors. A possible way to improve the situation is to allow processing *a multiple calculation wave*. According to this approach the processors after processing the current calculation wave may start processing the wave of the following iteration. Thus, for instance, the processor 0 (see Figure 12.13) after processing its block and transmitting the border data to initiate computations on the processors 1 and 4 may start calculations of the next Gauss-Seidel method iteration. After processing the blocks of the first (the processors 1 and 4) and the second (the processor 0) waves, the following processor groups appear to be ready for computations: for the first wave – the processors 2, 5, 8; for the second wave – processors 1 and 4. Correspondingly at this moment the processor 0 will again be ready to start processing the next data wave. After executing *NB* such steps, *NB* iterations of the Gauss-Seidel method will simultaneously be processed, and all the processors will be involved in calculations. This computational scheme makes possible to consider the grid as a computational pipeline of the Gauss-Seidel method iterations. As previously the calculations will be terminated when maximum computation error meet the accuracy requirement (estimating maximum computation error should be begun only at complete pipeline loading after starting *NB* iterations). It should be noted that the Dirichlet problem solution, which is obtained after meeting the accuracy conditions, will contain the grid node values from different method iterations. It will not coincide with the solution, which is obtained by means of the original sequential algorithm.

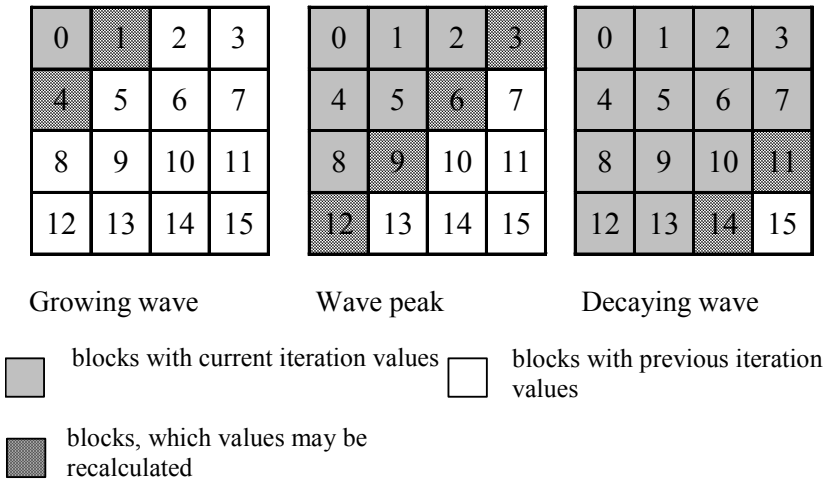

Figure 12.13. Wavefront computational scheme in case of the checkerboard block distribution

12.3.6. Communication Complexity Analysis

The execution time of communication operation considerably surpasses the duration of computing operations. As it has been previously mentioned (see Section 3), the time of communication operations may be

estimated with the help the two main network characteristics: *latency*, which defines the period of preparing data for network transmission, and *bandwidth*, which defines the maximum amount of data that can be transmitted in the network per sec. The problem is discussed in more detail in Section 3.

Fast Ethernet network bandwidth is 10Mbit/second. For the more fast Gigabit Ethernet network the bandwidth is 1000Mbit/second. At the same time the data transmission rate in shared memory systems is usually hundreds and thousands of millions bytes per second. Thus, the use of distribued memory systems leads to decreasing data transmission rate greater than 100 times.

The state of things is even worse in case of latency. For Fast Ethernet this characteristic has values of the order 150 microseconds, for Gigabit Ethernet it is approximately 100 microseconds. For modern computer with the clock frequency higher than 2 GHz/second the difference in performance is greater than 10000-100000 times. To be 90% efficient for solving the Dirichlet problem (i.e. for the data processing to take up not less than 90% of the general duration of computations) on a computer system with the above mentioned characteristics the size of computational blocks should be not less than N=7500 nodes (recall that the amount of calculations for each block is $5N^2$ floating-point operations.

As a result, it is possible to come to the conclusion that parallel computation efficiency in case of distributed memory is defined mostly by the intensity in processors interactions. The necessary analysis of parallel methods and programs may be carried out much easier in the presence of typical widely used data communication operations which time complexity characteristics have been evaluated earlier (see Section 3). Thus, for instance, in the Dirichlet problem solution practically all the data transmission can be reduced to standard communication operations, which have adequate support at the standard MPI (see Figure 12.14):

– sending the number of grid nodes to all the processors is the basic data communication operation from one processor to all processors (the function *MPI_Bcast*);

– distributing stripes or blocks to all the processors is transmitting different data from one processor to all the network processors (the function MPI_Scatter);

– exchanging of border rows or grid columns among the neighboring processors is the basic data communication operation for data distributing among the neighboring network processors (the function *MPI_Sendrecv*);

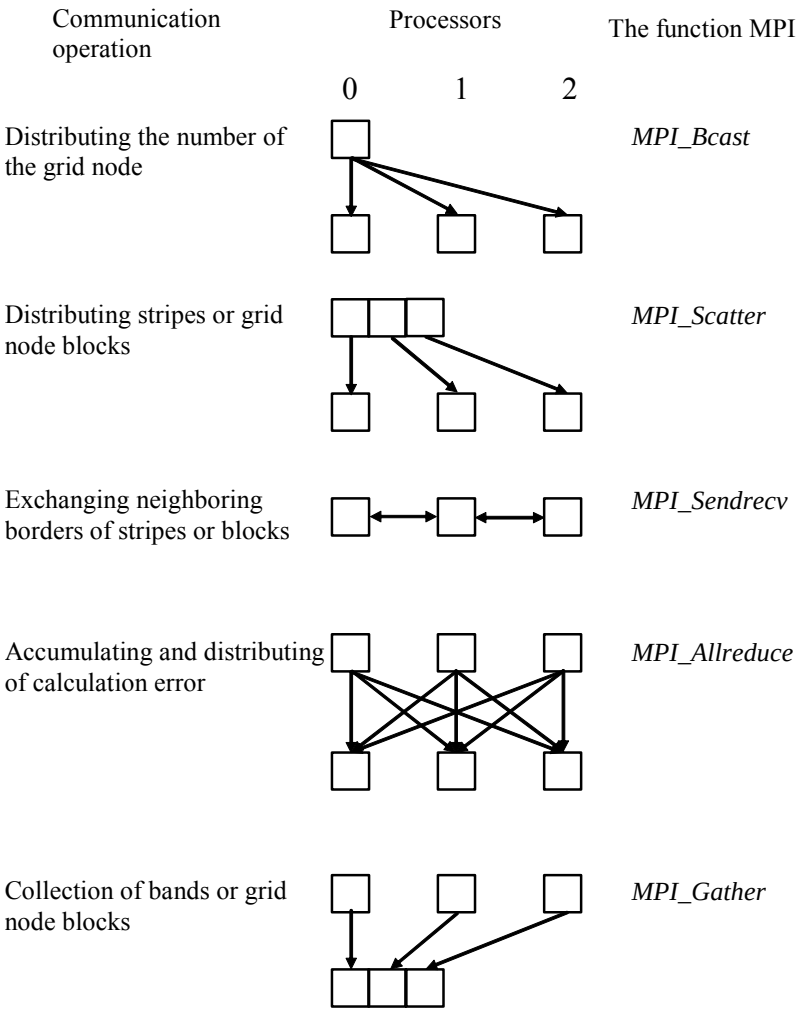

Figure 12.14. Data communications for the Gauss-Seidel method in case of distributed memory

– accumulating the calculation error, finding the maximum calculation error and its distributing among processors is a basic communication operation of data transmission from all processors to all processors (the function *MPI_Allreduce*);

– collecting the problem solution on one processor (all stripes or grid blocks) is a basic operation of data transmission from all the processors to one processor (the function *MPI_Gather*).

12.4. Summary

The Section describes parallel methods for solving partial differential equations. The Dirichlet problem for the Poisson equation is given as an example for consideration. The method of finite differences is one of the most widely used approaches to solving differential equations. The amount of computations in this case is considerable. Possible ways are also considered to develop parallel computations on multiprocessor computer systems with shared and distributed memory.

In Subsection 12.1 the statement of the Dirichlet problem is given. For solving the problem the finite difference methods (in particular the Gauss-Seidel algorithm) are described.

In Subsection 12.2 problems of parallel programming for shared memory multiprocessor systems are discussed. It is noted that in this case one of the most widely used approaches is OpenMP technology. OpenMP provides to develop parallel software on the basis of existed sequential programs by adding directives in the program code for indicating program fragments that can be executed in parallel. As a results the first prototypes of parallel programs can be obtained quite easily but then in the course of improving parallel program efficiency the problems of parallel computations are appeared immediately. Some of these problems are discussed, namely redundancy of computation synchronization, possible deadlock appearance, computation indeterminacy in case of race conditions etc. To overcome the problems various computational schemes are given for solving the Dirichlet problem are given (the red/black row alteration scheme, the wavefront computation scheme etc.). Finally the job queue technique is described as a way to provide processor load balancing.

In Subsection 12.3 the subject is parallel programming for distributed memory multicomputer systems. The discussion is concentrated on the problems of data distribution and information communication among processors. The block-striped decomposition scheme is considered in detail, a general description is given for the checkerboard data partition. For communication operations the variants of synchronous and asynchronous implementations are discussed. Finally the data communication complexity is estimated.

12.5. References

Additional information on the problems of numeric solving partial differential equations and on the method of finite differences can be found in Fox, et al. (1988).

Memory organizing and cache using is described in detail in Culler, et al. (1998).

The information on the problem of processor load balancing may be found in Xu and Hwang (1998).

12.6. Discussions

1. How is the Dirichlet problem for Poisson equation defined?
2. How is the method of finite differences applied for solving the Dirichlet problem?
3. What approaches determine the parallel computations for the grid methods on shared memory multiprocessor systems?
4. What is the essence of parallel computation synchronization?
5. What are the characteristics of the race condition of the parallel program threads?
6. What is the essence of the deadlock problem?
7. Which method guarantees the grid method determinacy but requires a great additional memory for implementation?
8. How does the computation amount change in case of using the wavefront processing methods?
9. How is chunking applied to reduce computation synchronization?
10. What are the ways to increase the efficiency of wavefront data processing methods?
11. In which way does the job queue provide for computational load balancing?
12. What are the problems to be solved in the course of parallel computations on distributed memory systems?
13. What mechanisms can be involved into the process of data transmission?

14. In which way can the multiple wave calculations be applied for increasing the wave computation efficiency in distributed memory systems?

12.7. Exercises

1. Develop the first and the second variants of the Gauss-Seidel parallel algorithm. Carry out the computational experiments and compare the execution time.

2. Implement the parallel algorithm based on the wavefront computation scheme. Develop the implementation of the parallel algorithm, in which the block-oriented approach to the wavefront processing method is applied. Carry out the computational experiments and compare the execution time.

3. Develop the implementation of the parallel computation job queue for shared memory systems. It is necessary to provide processing the neighboring blocks on the same processors.

References

Buyya, R. (Ed.) (1999). High Performance Cluster Computing. Volume1: Architectures and Systems. Volume 2: Programming and Applications. - Prentice Hall PTR, Prentice-Hall Inc.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R. (2000). Parallel Programming in OpenMP. Morgan Kaufmann Publishers.

Culler, D., Singh, J.P., Gupta, A. (1998) Parallel Computer Architecture: A Hardware/Software Approach. - Morgan Kaufmann.

Fox, G.C. et al.(1988). Solving Problems on Concurrent Processors.- Prentice Hall, Englewood Cliffs, NJ.

Group, W., Lusk, E., Skjellum, A. (1994). Using MPI. Portable Parallel Programming with the Message-Passing Interface. –MIT Press.

Pfister, G. P. (1995). In Search of Clusters. - Prentice Hall PTR, Upper Saddle River, NJ (2nd edn., 1998).

Roosta, S.H. (2000). Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag,NY.

Tanenbaum, A. (2001). Modern Operating System. 2nd edn. – Prentice Hall

Xu, Z., Hwang, K. (1998). Scalable Parallel Computing Technology, Architecture, Programming. – Boston: McGraw-Hill.