# 11. Parallel Methods for Graph Calculations



Mathematical models in the form of graphs are widely used for modeling various phenomena, processes and systems. As a result, many theoretical and applied problems may be solved by means of various procedures of graph model analysis. It is possible to select a set of typical algorithms of graph processing among all those procedures. The problems of graph theory, modeling algorithms, analyzing and solving problems based on graphs are discussed in a number of publications. One of the editions that can be recommended for deeper understanding of the graph theory is the book by Cormen, et al. (2001).

Let ***G*** be a graph

$$G = (V, R),$$

for which the set ***V*** of vertices $v_i$, $1 \le i \le n$, is defined , and the list of arcs

$$r_j = (v_{s_j}, v_{t_j}),\ 1 \le j \le m,$$

is defined by the set ***R***. Generally, the arcs may be assigned certain numerical characteristics (weights)) $w_j$, $1 \le j \le m$ (*the weighted graph*). An example of the weighted graph is given in Figure 11.1.

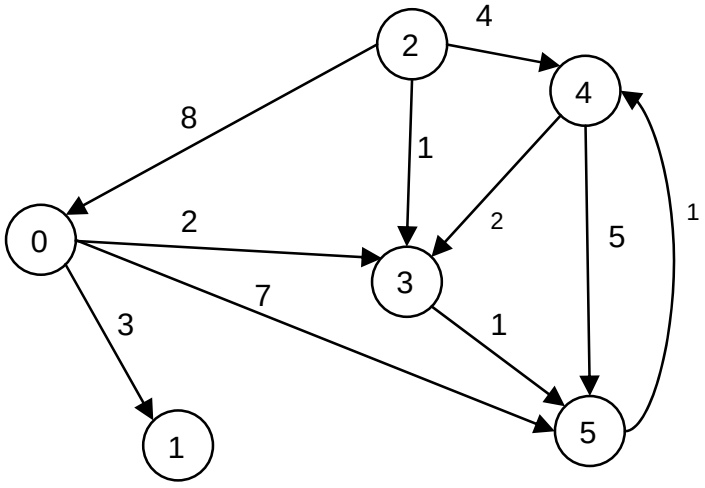


**Figure 11.1.** Example of the weighted oriented graph

There are various ways to represent a graph on a computer. If the number of arcs in the graph is small (i.e. $m << n^2$) it is useful to apply the list enumerating the arcs of the graph. Representing the dense graphs, almost all the nodes of which are linked by arcs (i.e. $m \sim n^2$), may be efficiently achieved by means of *the adjacency matrix*

$$A = (a_{ij}),\ 1 \le i,\ j \le n,$$

the nonzero values of which correspond to the arcs of the graph

$$a_{ij} = \begin{cases} w(v_i, v_j), & if \quad (v_i, v_j) \in R, \\ 0, & if \quad i = j, \\ \infty, & otherwise. \end{cases}$$

(the infinity sign is used in a corresponding position to denote the absence of an arc between the vertices in the adjacency matrix. In computations the infinity sign may be replaced by, for instance, a negative number). For instance, the adjacency matrix, which corresponds to the graph in Figure 11.1, is shown in Figure11.2

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

**Figure 11.2.** The adjacency matrix for the graph in Figure 11.1

It should be noted that the use of the adjacency matrix makes possible to apply the matrix algorithms of data processing in the implementation of the computational procedures of graph analysis.

We will discuss further some approaches to parallel algorithm implementation on graphs using *the problem of searching the shortest paths among all the pairs of vertices* and *the problem of finding the minimum spanning tree*. Besides, we will consider *the problem of optimal graph partition*, which is widely used in parallel computations. To represent graphs in the course of consideration of the above mentioned problems we will use the adjacency matrix.

This Section has been written based essentially on the teaching materials given in Schloegel, et al. (2000) and Quinn (2004).

## 11.1. The Problem of the Search for All the Shortest Paths

The initial information for the problem is the weighted graph $G = (V, R)$, which contains $n$ vertices ($|V| = n$). Each arc of the graph is assigned the non-negative weight. The graph is assumed to be oriented, i.e. if there is an arc from the vertex $i$ to the vertex $j$, it should not be supposed that there is an arc from $j$ to $i$. In case when the vertices are connected by inverse arcs, the weights assigned to the arcs may not coincide. Let us consider the problem, in which we have to find the minimum paths among each pair of the graph vertices for the given graph $G$. As a practical example we may use the problem of working out the transport route between various cities, if the distances between them are given, and all the problems alike.

As a method for solving the problem of searching all the shortest paths, we will further use *the Floyd algorithm* (see, for instance, Cormen, et al. (2001)).

### 11.1.1. The Sequential Floyd Algorithm

The complexity of the algorithm proposed by Floyd for searching the minimum paths between all the pairs of vertices is the order $n^3$. Generally, the algorithm may be given in the following way:

```
// Algorithm 11.1
// The Sequential Floyd Algorithm
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
```

**Algorithm 11.1.** The general scheme of the Floyd algorithm

(the implementation of the minimum value operation *min* must be performed with taking into account the method of describing nonexistent graph arcs in the adjacency matrix). As it can be noted, while the algorithm is executed, the adjacency matrix $A$ changes. After the termination of computations, the required result (the length of all the minimum paths) will be stored in matrix $A$.

Additional information and the proof of the Floyd algorithm correctness may be obtained in Cormen, et al. (2001).

### 11.1.2. Computation Decomposition

As a general scheme of Floyd algorithm suggests, the main computational load in solving the problem of searching the shortest paths is choosing the minimum values (see Algorithm 11.1). It is not worthwhile to parallelize this rather simple operation, as it will not speed up the computation significantly. It is more efficient to update the values of matrix $A$ simultaneously, as it will make parallel computations more effective.

Let us illustrate the correctness of this method of parallelism implementation. For this we have to prove that the operation of updating the values of the matrix $A$ during one iteration of the outer cycle $k$ may be performed independently. In other words, we must prove that the elements $A_{ik}$ and $A_{kj}$ do not change at iteration $k$ for any pair of indices $(i,j)$. Let us analyze the expression, according to which the change of the elements of the matrix $A$ happens:

$A_{ij} \leftarrow \min (A_{ij}, A_{ik} + A_{kj})$.

For $i = k$ we will have

$A_{kj} \leftarrow \min (A_{kj}, A_{kk} + A_{kj})$,

but then value $A_{kj}$ will not change as $A_{kk}$=0.

For $j = k$ the expression is transformed and reduced to

$A_{ik} \leftarrow \min (A_{ik}, A_{ik} + A_{kk})$,

which also shows that values $A_{ik}$ are unchangeable. As a result, the necessary conditions for parallel computations take place. Thus, the operation of updating the elements of the matrix $A$ may be used as the basic computational subtask (to identify subtasks we will use the indices of the elements, which are updated in them).

### 11.1.3. Analysis of Information Dependencies

Computations in the subtasks become possible only if each subtask $(i,j)$ contains elements $A_{ij}, A_{ik}, A_{kj}$ of the matrix $A$, which are necessary for computations. To eliminate data duplicating we will place the only element $A_{ij}$ in the subtask $(i,j)$. Obtaining all the other necessary values may be provided only by means of data transmission. Thus, each element $A_{kj}$ of the row $k$ of the matrix $A$ must be transmitted to all the subtasks $(k,j)$,

$1 \le j \le n$, and each element $A_{ik}$ of the column $k$ of the matrix $A$ must be transmitted to all the subtasks $(i,k)$, $1 \le i \le n$, - see Figure11.3.

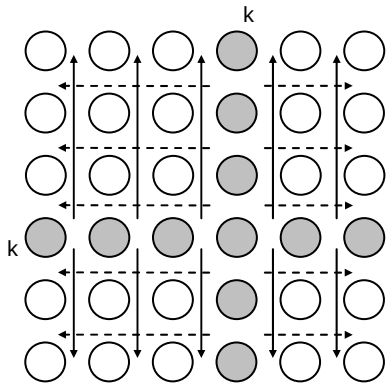


**Figure 11.3.** The information dependencies of the basic subtasks (the arrows show the direction of exchanging values at iteration $k$)

### 11.1.4. Scaling and Distributing Subtasks among Processors

As a rule, the number of available processors $p$ is considerably smaller than the number of basic subtasks $n^2$ ( $p << n^2$ ). The use of the block-striped scheme decomposition of the matrix $A$ is a possible way to aggregate the computations. This approach corresponds to uniting in one basic subtask the computations related with updating the elements of one or several rows (*rowwise* or *horizontal partitioning*) or columns (*columnwise* or *vertical partitioning*) of the matrix $A$. These two decomposition schemes are practically equal. With regard to the fact that for the algorithmic language C arrays are located rowwise, we will further apply only partitioning the matrix $A$ into horizontal stripes.

It should be noted that in case of this method of data decomposition, it is necessary to transmit among the subtasks only the elements of one of rows of the matrix $A$. The network topology for efficient execution of this communications is a hypercube or a complete graph.

### 11.1.5. Efficiency Analysis

Let us analyze the efficiency of the Floyd parallel algorithm. As previously, it will be done in two stages. During the first stage we will estimate the order of the algorithm computational complexity. At the second stage we will specify the estimations and take into account the time complexity of data communications.

The total time complexity of the sequential algorithm, as it has been previously mentioned, is equal to $n^3$. For the parallel algorithm each processor performs updating of the elements of the matrix $A$ at each iteration. There are $n^2/p$ such elements in each subtask. The number of the algorithm iterations is equal to $n$. Thus, the speedup and efficiency characteristics of the Floyd algorithm look as follows:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{and} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \tag{11.1}$$

Thus, the general efficiency analysis gives ideal characteristics of parallel computation efficiency. To specify the obtained relations we will introduce the execution time of the basic operation of choosing the minimum value into the obtained expressions. We will also take into account the overhead of data communications among the processors.

The communication operation performed at each iteration of the Floyd algorithm consists in transmitting a row of the matrix $A$ to all the processors. As it has been shown previously, the execution of this operation may be done in $\lceil \log_2 p \rceil$ steps. With regard to the number of the Floyd algorithm iterations in case when the Hockney model is used, the total duration of data communications may be defined by means of the following expression

$$T_p(comm) = n \lceil \log_2 p \rceil (\alpha + wn/\beta), \tag{11.2}$$

where, as previously, α is the latency of the transmission network, β is its bandwidth, and $w$ is the matrix element size in bytes.

With regard to the obtained relations, the total execution time for the Floyd parallel algorithm may be defined in the following way:

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2(p) \rceil (\alpha + w \cdot n/\beta), \quad (11.3)$$

where $\tau$ is the execution time of choosing the minimum value.

### 11.1.6. Software Implementation

Let us present a variant of the Floyd algorithm parallel implementation. Program code is given for all main modules of the software. The absence of several parts of the parallel program does not hinder the general understanding of parallel computation scheme.

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

```
// Program 11.1
int ProcRank;    // rank of current process
int ProcNum;     // number of processes

// Maximum evaluation function
int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

// Main function
int main(int argc, char* argv[]) {
  int *pMatrix;      // adjacency matrix
  int  Size;         // size of adjacency matrix
  int *pProcRows;    // rows of adjacency matrix
  int  RowNum;       // number of rows on current process

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

  // Data initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  // Data distribituon to all processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // The Floyd parallel algorithm
  ParallelFloyd(pProcRows, Size, RowNum);

  // Result collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);

  // Computation Termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}
```

The function *Min* evaluates the minimum of two integers taking into account the applied method of marking the nonexistent arcs in the adjacency matrix (for instance, in the given implementation we use the value -1).

The function *ProcessInitialization* defines the initial data of the problem being solved (the number of graph vertices), allocates memory for data storage, inputs the adjacency matrix (or forms it by means of any random number generator).

The function *DataDistribution* distributes the data among the processors. Each process receives the horizontal stripe of the adjacency matrix.

The function *ResultCollection* accumulates the horizontal stripes of the result matrix with the shortest paths between any pairs of graph vertices from all the processes.

The function *ProcessTermination* performs the necessary output of the computation results and releases the previously allocated memory.

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 ParallelFloyd**. This function performs the iterative change of adjacency matrix according to the Floyd algorithm.

```
// The Floyd parallel algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // k-ой row distributin among processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Updating the adjacency matrix
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2);
        }
  }

  delete []pRow;
}
```

**3. The function RowDistribution**. This function distributes k-th row of the adjacency matrix among processes:

```
// The row distribution function
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank;
  int ProcRowNum;

  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
     copy(&pProcRows[ProcRowNum * Size], &pProcRows[(ProcRowNum + 1) *
      Size], pRow);

   MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

### 11.1.7. Computational Experiment Results

The computational experiments for estimating the efficiency of the Floyd parallel algorithm for searching all the shortest paths were carried out under the conditions described in 7.6.5. They are described below.

The computational cluster on the basis of processor Intel XEON 4 EM64T 3000 Mhz and Gigabit Ethernet under OS Microsoft Windows Server 2003 Standard x64 Edition was used in the experiments.

To estimate the duration $\tau$ of the basic scalar operation of choosing the minimum value, we solved the problem of searching the shortest paths by means of the sequential algorithm. The calculation time obtained this way was divided by the total number of the executed operations. As a result of those experiments, the value 7.1 nsec was obtained for $\tau$ . The experiments carried out in order to determine the parameters of the data communication network gave the latency value α and the bandwidth value β correspondingly 47 msec and 53,29 Mbyte/sec. All the computations were performed with the numerical values of the type *int*, i.e. value *w* is equal to 4 bytes.

The results of the computational experiments are given in Table 11.1. The experiments were carried out on 2, 4 and 8 processors. The time is given in seconds.

**Table 11.1.** The results of the computational experiments for the parallel Floyd algorithm

| Number of vertices | Sequential algorithm | Parallel algorithm | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 processors | | 4 processors | | 8 processors | |
| | | Time | Speedup | Time | Speedup | Time | Speedup |
| 1000 | 8,0370 | 4,1519 | 1,9357 | 2,0671 | 3,8880 | 0,9407 | 8,5439 |
| 2000 | 59,8119 | 30,3234 | 1,9725 | 15,3752 | 3,8901 | 8,0577 | 7,4229 |
| 3000 | 197,1114 | 99,2642 | 1,9857 | 50,2323 | 3,9240 | 25,6433 | 7,6867 |
| 4000 | 461,7849 | 232,5071 | 1,9861 | 117,2204 | 3,9395 | 69,9457 | 6,6021 |
| 5000 | 884,6221 | 443,7467 | 1,9935 | 224,4413 | 3,9414 | 128,0784 | 6,9069 |

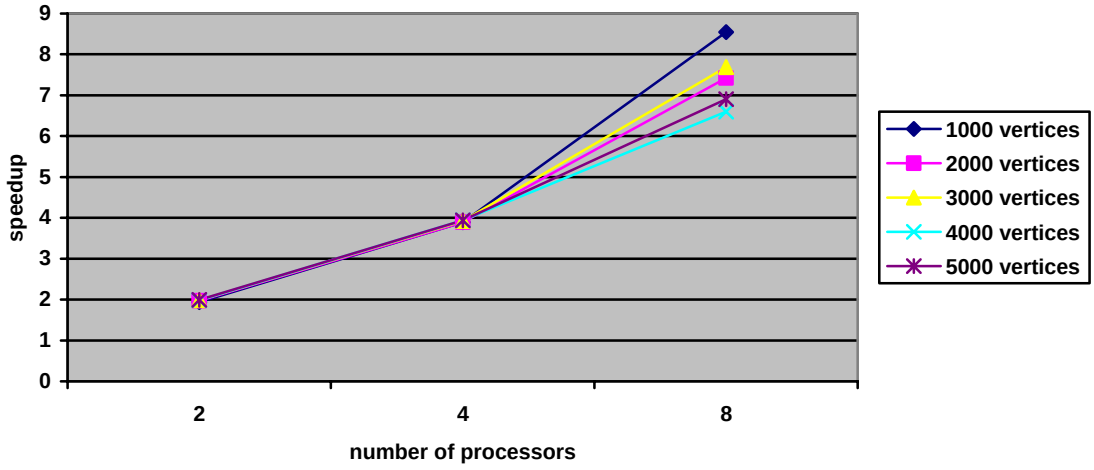


**Figure 11.4.** Speedup of the parallel Floyd algorithm

The comparison of the experiment execution time $T_p^*$ and the theoretical estimation $T_p$ from (11.3) is given in Table 11.2 and in Figure 11.5.

**Table 11.2.** The comparison of the experimental time and the theoretical time of the Floyd algorithm execution

| Number of vertices | Parallel algorithm | | | | | |
|---|---|---|---|---|---|---|
| | 2 processors | | 4 processors | | 8 processors | |
| | $T_2$ (model) | $T_2^*$ | $T_4$ (model) | $T_4^*$ | $T_8$ (model) | $T_8^*$ |
| 1000 | 3,7757 | 4,1519 | 2,1960 | 2,0671 | 1,5090 | 0,9407 |
| 2000 | 29,1234 | 30,3234 | 15,4046 | 15,3752 | 8,8261 | 8,0577 |
| 3000 | 97,4645 | 99,2642 | 50,3361 | 50,2323 | 27,3066 | 25,6433 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 4000 | 230,2200 | 232,5071 | 117,7013 | 117,2204 | 62,3058 | 69,9457 |
| 5000 | 448,8112 | 443,7467 | 228,2108 | 224,4413 | 119,1790 | 128,0784 |

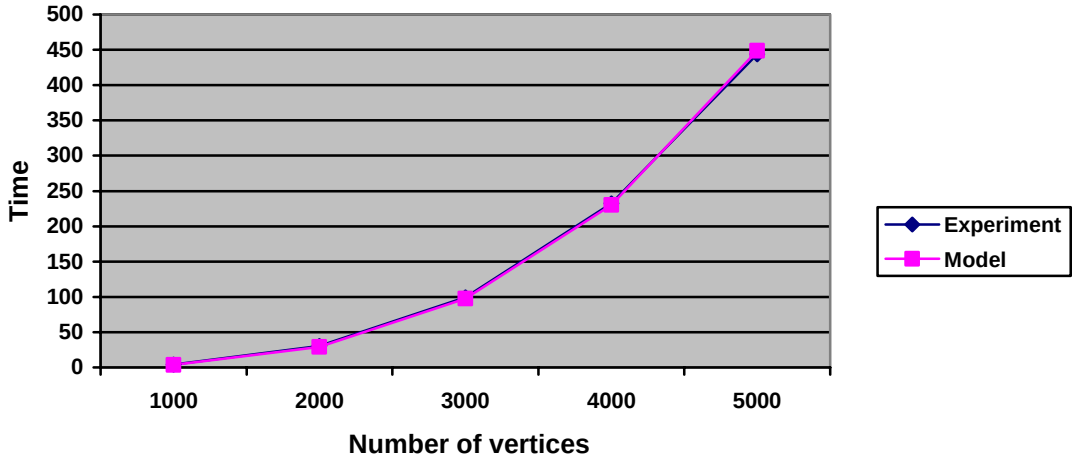


**Figure 11.5.** Experimental and theoretical execution time of the parallel Floyd algorithm for 2 processors

## *11.2. The Problem of Finding the Minimum Spanning Tree*

*The spanning tree (or the skeleton)* of the non-oriented graph $G$ is the subgraph $T$ of the graph $G$, which is a tree and contains all the vertices of graph $G$. The subgraph weight for the weighted graph is equal to the sum of all the weights of the subgraph arcs. Thus, *the minimum spanning tree* (*MST*) $T$ may be defined as the spanning tree of the minimum weight. An applied interpretation of the problem of finding MST may consist in, for instance, practical example of creating a local network of personal computers and connecting them by communication lines of minimum length. An example of the weighted non-oriented graph and the minimum spanning tree, which corresponds to it, is given in Figure 11.6.

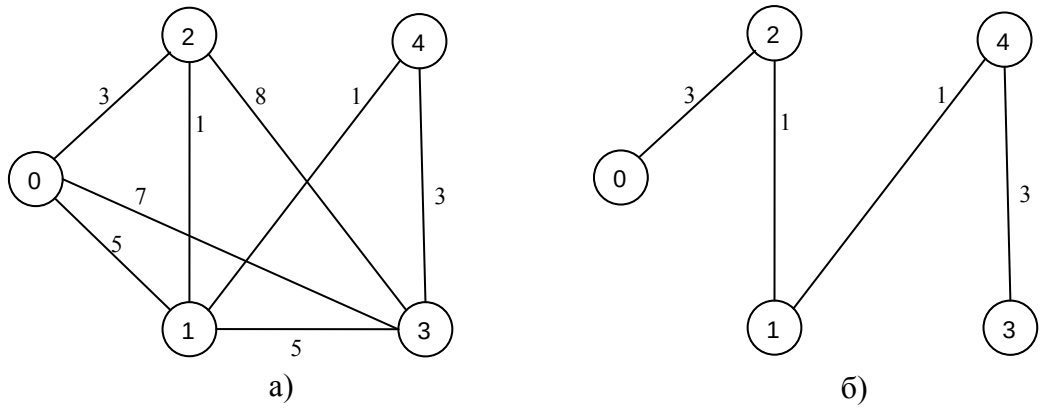


**Figure 11.6.** The examples of (a) the weighted non-oriented graph and (b) its minimum spanning tree

We will further describe the algorithm for solving this problem, which is known as *the Prim method*. More detailed information on the problem may be found in Cormen, et al. (2001).

### 11.2.1. The Sequential Prim Algorithm

The algorithm starts from an arbitrary graph vertex, which is selected as the tree root. In the course of sequentially executed iterations the algorithm expands the tree, which it constructs, up to MST. Let $V_T$ be the

set of the vertices, which are already included into MST by the algorithm. Let the values $d_i$, $1 \le i \le n$, be the minimum arc weights for arcs from the vertices, which are not included into MST yet, to the set $V_T$, i.e.

$$\forall i \notin V_T \Rightarrow d_i = \min\{w(i,u) : u \in V_T, (i,u) \in R\}$$

(if for some vertex $i \notin V_T$ there is no arcs in $V_T$, value $d_i$ may be set the infinity value $\infty$). At the beginning of the algorithm execution the root vertex MST with number *s* is selected and it is assumed that $V_T = \{s\}$, $d_s = 0$.

The calculations, carried out at each Prim algorithm iteration, consist in the following:

- Determining the values $d_i$ for all the vertices not included into MST;
- Finding the vertex $t$ of graph $G$, which has the arc of the minimal weight to the set $V_T$:
  $t : d_t = \min(d_i)$, $i \notin V_T$;
- Including the vertex $t$ into $V_T$.

After executing *(n-1)* iterations of the method, MST will be formed. The tree weight may be obtained by means of the expression:

$$W_T = \sum_{i=1}^{n} d_i .$$

The time complexity of defining MST is evaluated by the square dependence with respect to the number of graph vertices, i.e. is the order $O(n^2)$.

### 11.2.2. Computation Decomposition

Let us estimate the possibilities to execute the algorithm of finding the minimum spanning tree in parallel.

The method iterations should be carried out sequentially. Thus, they cannot be parallelized. On the other hand, the operations, performed at each iteration, are independent and may be executed simultaneously. For instance, defining the values $d_i$ may be done for each graph vertex simultaneously, finding the minimum weight arc may be implemented according to the cascade scheme, etc.

Data distribution among the processors should provide the information independence of the above mentioned Prim algorithm operations. In particular, it may be provided if each graph vertex is located on the processor along with all the information related with the vertex. If follow this principle each processor $P_j$, $1 \le j \le p$, should hold:

- A set of vertices
  $V_j = \{v_{i_j+1}, v_{i_j+2}, ..., v_{i_j+k}\}$, $i_j = k \cdot (j-1)$, $k = \lceil n/p \rceil$,
- The corresponding block of *k* values
  $\Delta_j = \{d_{i_j+1}, d_{i_j+2}, ..., d_{i_j+k}\}$,
- The vertical stripe of k neighboring columns of the adjacency matrix of the graph G
  $A_j = \{\alpha_{i_j+1}, \alpha_{i_j+2}, ..., \alpha_{i_j+k}\}$ ($\alpha_s$ is $s\text{-}th$ column of matrix $A$),
- The common part of the set $V_j$ and the set of vertices $V_T$, which is being formed in the course of the method execution.

As a result, we may conclude that the procedure of computing the block $\Delta_j$ of values for the vertices $V_j$ of the the adjacency matrix $A$ may be the basic computational subtask of the Prim parallel algorithm .

### 11.2.3. Analysis of Information Dependencies

With regards to the choice of the basic subtasks, the general Prim algorithm execution scheme will consist in the following:

- Finding the vertex $t$ of the graph $G$, which has the minimum weight arc to the set $V_T$; it is necessary to carry out the search of minimum in the sets of $d_i$ available on each of the processors in order to choose this vertex, then the obtained values should be accumulated on one of the processors;

- Broadcasting the number of the selected vertex to all the processors for including it into the spanning tree;
- Updating the set of $d_i$ values with regard to the new vertex, which has been added.

Thus, two types of information communications are executed in the course of parallel computations among the processors. They are accumulating the data from all the processors on one of the processors and broadcasting messages from the processor to all the processors.

### 11.2.4. Scaling and Distributing Subtasks among Processors

According to the definition the number of the basic subtasks always corresponds to the number of the available processors. Thus, the problem of scaling for parallel algorithm does not arise.

Distributing subtasks among the processors must be done with regards to the communication operations performed in the Prim algorithm. For efficient implementation of the required information communications among the basic subtasks the network topology must be a hypercube or a complete graph.

### 11.2.5. Efficiency Analysis

The general analysis of the Prim parallel algorithm for finding the minimum spanning tree produces ideal characteristics of parallel computation efficiency:

$$S_p = \frac{n^2}{(n^2/p)} = p \quad \text{and} \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \tag{11.4}$$

It should be noted that the ideal computational load balancing may be violated. In the course of parallel computations the number of the selected vertices of the spanning tree on different processors may appear to be different depending on the type of the initial graph *G*. Thus, the distribution of computations among the processors turns to be unequal (some processors may have no computational load at all in this case). However, such extreme situations will appear rarely enough. Dynamic redistributing the computational load among the processors in the course of computations is a challenging but a very complicated job.

In order to specify the obtained characteristics of parallel computation efficiency, we will estimate more precisely the number of the algorithm computational operations and take into account the expenses on data communications among the processors.

At each iteration of the Prim parallel algorithm each processor finds the number of the nearest vertex from $V_j$ to the spanning tree and performs recalculations of the distances $d_i$ after the expansion of MST. The number of operations executed during each of these computational procedures is bounded from above by a number of vertices available on the processors, i.e. the value $\lceil n/p \rceil$. As a result, with regard to the total number of iterations *n*, the computation execution time for the Prim parallel algorithm may be estimated by the following relation:

$$T_p(calc) = 2n\lceil n/p \rceil \cdot \tau \tag{11.5}$$

(here, as previously, , $\tau$ is the execution time of an elementary computational operation).

All gather communication operation to accumulate data from all the processors may be executed in $\lceil \log_2 p \rceil$ iterations. As a result the general estimation of the data communication duration can be evaluated by the expression (this communication operation is described in more detail in Sections 3 and 7):

$$T_p^1(comm) = n(\alpha \log_2 p + 3w(p-1)/\beta), \tag{11.6}$$

where $\alpha$ is the network latency, , $\beta$ is the network bandwidth, and $w$ is the size of a transmitted data element in bytes (the coefficient 3 in the expression corresponds to the number of the values transmitted among the processors, namely, the weight of the minimum arc and the numbers of the 2 vertices connected by the arc).

The communication operation of transmitting the data from a processor to all the processors may also be executed in $\lceil \log_2 p \rceil$ iterations, and the general time estimation is done in the following way:

$$T_p^2(comm) = n \log_2 p(\alpha + w/\beta). \tag{11.7}$$

With regard to the obtained relations the total execution time for the Prim parallel algorithm is the following:

$$T_p = 2n\lceil n/p \rceil \cdot \tau + n(\alpha \cdot \log_2 p + 3w(p-1)/\beta + \log_2 p(\alpha + w/\beta)) \tag{11.8}$$

### 11.2.6. Computational Experiment Results

The computational experiments for estimation the efficiency of the Prim parallel algorithm were carried out under the same conditions as the experiments described in the previous Sections (see 11.1.7).

To estimate the duration $\tau$ of the basic computational operation we solved the problem of searching for the minimum spanning tree by means of the sequential algorithm. The computation time obtained in such a way was divided into the total number of the executed operations. As a result of the experiments, the value 4,8 nsec was obtained for $\tau$. All computations were performed with the numerical values of the type *int*, i.e. the value $w$ is equal to 4 bytes.

The results of the computational experiments are given in Table 11.3. The experiments were carried out with the use of 2, 4 and 8 processors. The time is given in seconds.

**Table 11.3.** The results of the computational experiments for the parallel Prim algorithm

| Number of vertices | Sequential algorithm | Parallel algorithm | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 processors | | 4 processors | | 8 processors | |
| | Time | Time | Speedup | Time | Speedup | Time | Speedup |
| 1000 | 0,0435 | 0,2476 | 0,1757 | 0,9320 | 0,0467 | 1,5735 | 0,0277 |
| 2000 | 0,2079 | 0,6837 | 0,3041 | 1,7999 | 0,1155 | 2,1591 | 0,0963 |
| 3000 | 0,4849 | 1,4034 | 0,3455 | 2,2136 | 0,2191 | 3,1953 | 0,1518 |
| 4000 | 0,8729 | 1,9455 | 0,6220 | 3,3237 | 0,2626 | 5,4309 | 0,1607 |
| 5000 | 1,4324 | 2,6647 | 0,7363 | 2,9331 | 0,4884 | 4,1189 | 0,3478 |
| 6000 | 2,1889 | 2,8999 | 0,8214 | 4,2911 | 0,5101 | 7,7373 | 0,2829 |
| 7000 | 3,0424 | 3,2364 | 1,0491 | 6,3273 | 0,4808 | 8,8255 | 0,3447 |
| 8000 | 4,1497 | 4,4621 | 1,2822 | 6,9931 | 0,5934 | 10,3898 | 0,3994 |
| 9000 | 5,6218 | 5,8340 | 1,2599 | 7,4747 | 0,7521 | 10,7636 | 0,5223 |
| 10000 | 7,5116 | 6,9902 | 1,2875 | 8,5968 | 0,8738 | 14,0951 | 0,5329 |

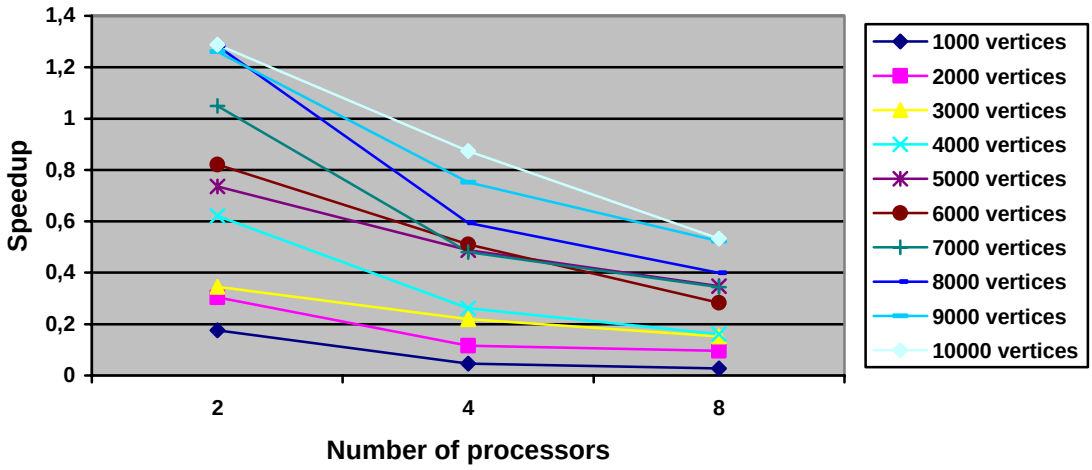


**Figure 11.7.** Speedup of the parallel Prim algorithm

The comparison of experimental execution time $T_p^*$ and the theoretical estimation $T_p$ from (11.3) is given in Table 11.4 and in Figure 11.8.

**Table 11.4.** The comparison of the experimental time and the theoretical time for the parallel Prim algorithm

| Number of vertices | Parallel algorithm | | | | | |
|---|---|---|---|---|---|---|
| | 2 processors | | 4 processors | | 8 processors | |
| | $T_2$ (model) | $T_2^*$ | $T_4$ (model) | $T_4^*$ | $T_8$ (model) | $T_8^*$ |
| 1000 | 0,4054 | 0,2476 | 0,8040 | 0,9320 | 1,2048 | 1,5735 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2000 | 0,8203 | 0,6837 | 1,6128 | 1,7999 | 2,4120 | 2,1591 |
| 3000 | 1,2447 | 1,4034 | 2,4264 | 2,2136 | 3,6216 | 3,1953 |
| 4000 | 1,6786 | 1,9455 | 3,2447 | 3,3237 | 4,8335 | 5,4309 |
| 5000 | 2,1220 | 2,6647 | 4,0678 | 2,9331 | 6,0479 | 4,1189 |
| 6000 | 2,5750 | 2,8999 | 4,8957 | 4,2911 | 7,2646 | 7,7373 |
| 7000 | 3,0375 | 3,2364 | 5,7283 | 6,3273 | 8,4837 | 8,8255 |
| 8000 | 3,5095 | 4,4621 | 6,5656 | 6,9931 | 9,7052 | 10,3898 |
| 9000 | 3,9911 | 5,8340 | 7,4078 | 7,4747 | 10,9290 | 10,7636 |
| 10000 | 4,4821 | 6,9902 | 8,2546 | 8,5968 | 12,1552 | 14,0951 |

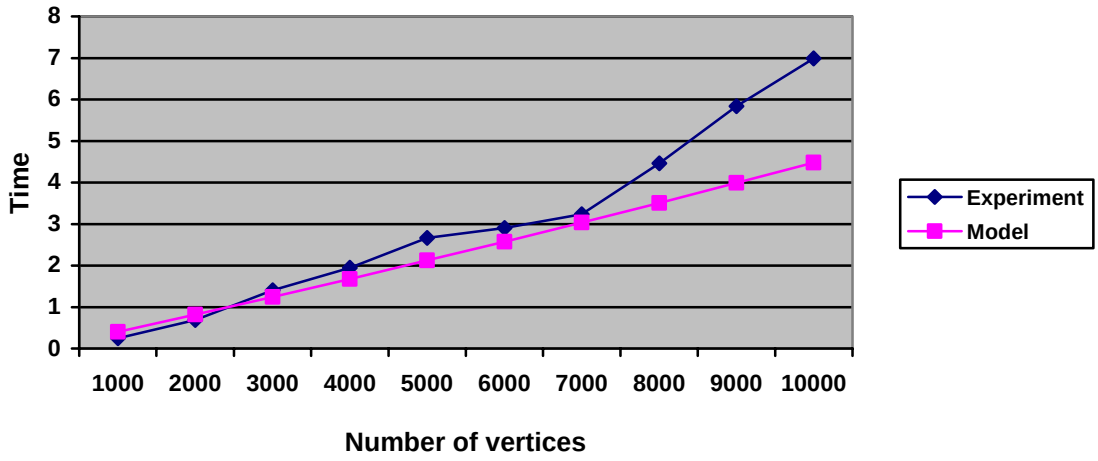


**Figure 11.8.** Experimental and theoretical execution time of the parallel Prim algorithm for 2 processors

As it can be noted from Table. 11.4 and Figure11.8, the theoretical estimations evaluate the execution time of the Prim parallel algorithm with the error, which may be considerable. This divergence is caused by the fact that the Hockney model is less precise in estimating the message communication time, if the amount of the transmitted data is small. In this respect it is necessary to use other more precise model for estimating the time complexity of the communication operations in order to specify the obtained estimations. This problem has been discussed in Section 3.

## *11.3. The Problem of the Optimum Graph Partition*

This problem is one of those, which frequently occur in the various areas, which involve parallel computations. As an example we may use the problem of processing the data when the domains of calculations are presented as two- or three-dimensional grids. As a rule the computations in such problems are reduced to the execution of data processing procedures for each element (a grid node). In the course of computations the results of data processing may be transmitted among the neighboring grid elements, etc. It is evident that for the efficient solution of such problems on multiprocessor systems with distributed memory, the grid should be distributed among the processors so that each of the processors holds approximately the same number of the grid elements, and the interprocessor communications necessary for the information exchange among the neighboring elements are minimal. Figure 11.19 shows the example of an irregular grid divided into 4 parts (the different parts of the grid partitioning are marked by the dark colors of various intensities).

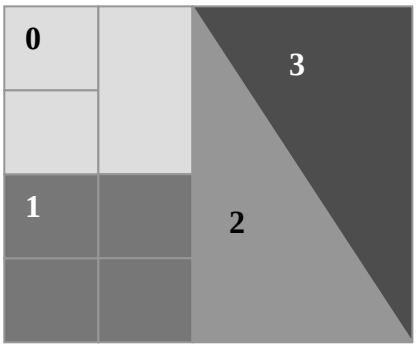


**Figure 11.9.** The example of irregular grid partitioning

It is obvious that such problem of distributing grid among the processors may be reduced to the problem of optimum graph partitioning. This approach is efficient because representing calculation models as graphs facilitates the problems of storing the processed data and gives the opportunity to apply the well known graph algorithms.

If a grid should be represented as a graph, each grid element may be associated with a graph vertex. The graph arcs should be used for reflecting the grid element closeness (for instance, the arcs between the vertices of the graph may be defined only if the corresponding elements of the initial grid are neighbors). In case of this approach the grid shown in Figure 11.9 will correspond to the graph shown in Figure 11.10

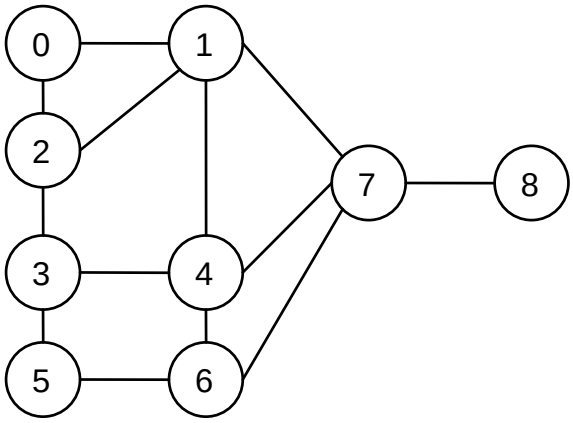


**Figure 11.10.** The example of the graph, which represents the structure of the grid shown in Figure 11.9

Additional information on the problem of the graph partitioning may be found, for instance, in Schlegel, et al. (2000).

The problem of the optimum graph partitioning may itself become the subject of the parallelization. It becomes necessary when the computational power and RAM size of average computers appear to be insufficient for solving such problems efficiently. Parallel algorithms of graph partitioning are the subject of considerations in many publications: Barnard (1995), Gilbert et al. (1987), Heath et al. (1995), Karypis et al. (1998, 1999), Raghavan (1995), Walshaw et al. (1999).

### 11.3.1. Problem Statement

Let there be given the weighted non-oriented graph $G=(V,E)$, each vertex of which $v \in V$ and each arc of which $e \in E$ are assigned a weight. The problem of optimum graph partition consists in partitioning its vertices into nonintersecting subsets with maximum close summary vertex weights and the minimum summary weight of the arcs, passing through the obtained vertex subsets.

It should be noted that the given criteria of graph partitioning may be contradictory. The equilibrium of the vertex subsets may not correspond to the minimality of the neighboring arc weights and vice versa. In most cases a compromise solution is necessary. Thus, if the fraction of communication is small, it appears to be efficient to optimize the arc weights in the solutions, which provide the optimum partitioning of vertices according to their weights.

For the sake of simplicity the vertex weights and the arc weights will further be assumed to be equal to one.

### 11.3.2. Recursive Bisection Method

*The recursive bisection method* may be recursively used for solving the problem of graph partitioning. This method implies dividing the graph into two equal parts at the first iteration. Further during the second step each of the parts is also divided into halves and so on. This method requires $\log_2 k$ recursion levels for dividing the graph into $k$ parts; also it requires $k-1$ operations of dividing in half. In case when the number of required parts $k$ is not a power of two, each division should be done in the corresponding relation.

The example of the graph division into five parts in Figure 11.11 illustrates the scheme of the method application. First, the graph should be divided into two parts as 2:3 (continuous line), then the right part should be divided as 1:3 (dashed line), after that we have to divide two outermost subareas to the left and to the right as 1:1 (dashed pointed line).

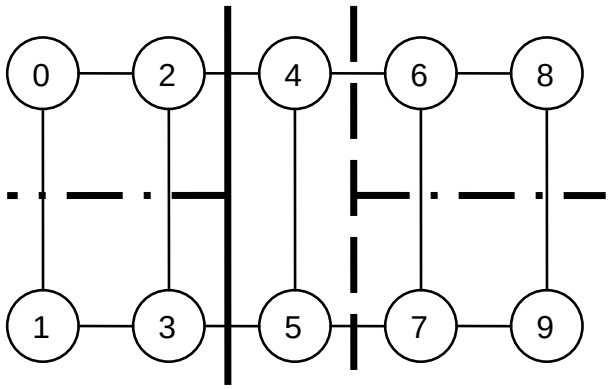


**Figure 11.11.** The example of graph division into five parts by the recursive bisection method

### 11.3.3. The Geometric Methods

*The geometric methods* (see, for instance, Berger, et al. (1987), Gilbert, et al. (1995), Heath, et al. (1995), Miller, et al. (1993), Nour-Omid, et al. (1986), Patra, et al. (1998), Pilkington, et al. (1994), Raghavan (1993)) are used for grid dissection. They are based exclusively on the coordinate information about the grid nodes. As these methods do not take into account the information concerning the grid element connectivity, they cannot explicitly cause the minimization of the summary weight of boundary arcs (in terms of the graph, which corresponds to the grid). To minimize the interprocessor communications the geometric methods should optimize some auxiliary criteria (for instance, the length of the border between the partitioned parts of the grid).

The geometric methods do not usually require a big amount of computations. However, the quality of their partitioning is usually not so high as that of the methods, which take into account the connectivity of the grid elements.

#### 11.3.3.1. Coordinate Nested Dissection

*The coordinate nested dissection algorithm* is the method, which is based on the recursive division of the grid into half according to the longest side. Figure 11.12 illustrates the method. The method of coordinate nested dissection used for partitioning the grid shown in the Figure gives a considerable smaller length of the border between the separated parts in comparison with the case when the grid is partitioned according to shortest (vertical) side.

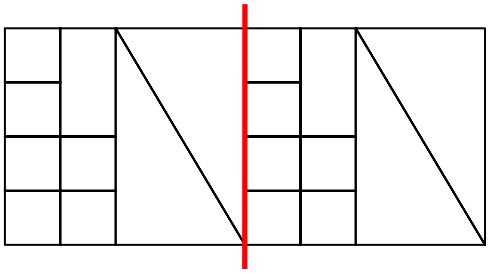

**Figure 11.12.** The example of grid partition by the coordinate nested dissection method (the border line is shown by the bold line)

The general computational scheme of this method is described below. First the centers of mass of the grid elements are computed. The points obtained are projected on the axis, which corresponds to the longest side of the grid being partitioned. Thus, we obtain a well-ordered list of all grid elements. Dividing the list in half (possibly, in the necessary proportion) we obtain the required dissection. The obtained fragments are analogously divided into the required number of parts recursively.

The coordinate nested dissection method operates very quickly and requires a small amount of memory. However, the quality of the obtained partition is lower than that of the more complex and more time consuming computational methods. Besides, if the grid structure is complicated enough the algorithm may produce partition with disconnected subgrids.

#### 11.3.3.2. The Recursive Inertial Bisection Method

The previous scheme could produce a partition of the grid only along the line, which is perpendicular to one of the coordinate axes. In many cases this limitation appears to be crucial for creating well-balanced partition. To make it evident, it is enough to turn the grid in Figure 11.12 at a sharp angle to the coordinate axes (see Figure 11.13). In order to minimize the borders between the subgrids, it is desirable to be able to draw the dissection line with any desired angle of rotation. A possible way to define the angle of deflection, which is used in *the recursive inertial bisection method*, is the use of the main inertial axis (see, for instance, Pothen, A. (1996)). The grid elements are regarded to be mass points. The bisection line, which is orthogonal to the obtained axis, produces, as a rule, the shortest border.

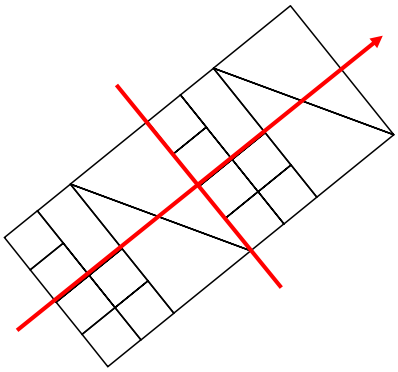

**Figure 11.13.** The example of grid partition by the recursive inertial bisection method (the main inertial axis is marked by the arrow)

#### 11.3.3.3. Grid Dissection by Means of Space-Filling Curve Technique

All the previously described geometric methods have a common drawback. The fact is that at each bisection these methods take into account only one dimension. Thus, the schemes, which take into account more dimensions, are able to provide better partition.

One of such methods puts the elements in order according to the positions of their mass centers along the Peano curves. *The Peano curves* are such curves, which fully fill high dimensional figures (for instance, a square or a cube). The use of the Peano curves provides the closeness of the figure points if they correspond to points, which are close on the curve. After generating the list of all the grid elements sorted according to their position on the Peano curve, it is enough to divide the list into the necessary number of parts according to the established order. The method produced by this approach is referred to as *the space-filling curve technique*. More information about the method can be found in Ou, et al. (1996), Patra, et al. (1998), Pilkington, et al. (1994).

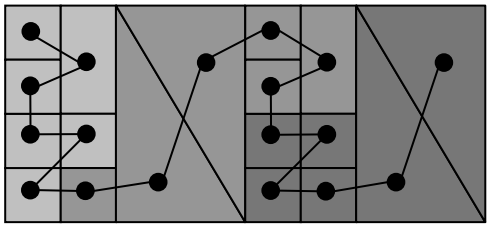

**Figure 11.14.** The example of grid dissection into 3 parts by means of space-filling curve technique

### 11.3.4. The Combinatorial Methods

Unlike the geometric methods *the combinatorial algorithms* (see, for instance, George (1981), Schloegel, et al. (2000)) operate, as a rule, not with the grid but with the graph, constructed on basis of the grid. Correspondingly, these methods make no regard to the information of the closeness of the grid elements. The combinatorial methods are guided by the adjacency of graph vertices. They provide more balanced partition and lesser information relations of the obtained subgrids than the previously described methods. However, such methods tend to operate much longer than the geometric ones.

#### 11.3.4.1. Grid Dissection with Regards to Adjacency

It is clear that in case of graph dissection the information dependencies between the separated subgraphs will be lesser, if the neighboring vertices (the vertices between which there are the graph arcs) will be in the same subgraph. *The levelized nested dissection algorithm* is aimed to implement this intention. The algorithm adds the neighboring vertices to the created subgraph subsequently. The graph is bisected at each algorithm iteration. Thus, partitioning the graph into the required number of parts is achieved by means of the recursive use of the algorithms.

The general computational scheme of the algorithm may be described in the following way.

```
1. Iteration = 0
2. Assigning the number Iteration to the arbitraty graph vertex
3. Assigning the number Iteration + 1 to the unnumbered neighbors of the
   vertices with the number  Iteration
5. Iteration = Iteration + 1
6. If there are any unnumbered neighbors, then proceed to step 3
7. dividing the graph into two equal parts according to the enumeration
```

**Algorithm 11.2.** The general scheme of the graph dissection algorithm with regard to adjacency

To minimize the information dependencies it is reasonable to choose a bordering vertex as the initial one. The search of such vertex may be carried out by the method, which is close to the discussed scheme. Thus, having enumerated the graph vertices according to algorithm 11.2 (starting the enumeration with an arbitrary vertex), we may take any vertex with the maximum number. Obviously, it will be a bordering one.

The example of the algorithm execution is given in Figure 11.15. The digits denote the numbers, which have been assigned to the graph vertices in the course of enumeration. The solid line shows the border, which parts the two subgraphs. The optimum solution is shown by the dashed line. It is obvious, that the partition obtained by means of the algorithm is far from being optimum, as the example contains the solution only with three splitted arcs instead of 5.

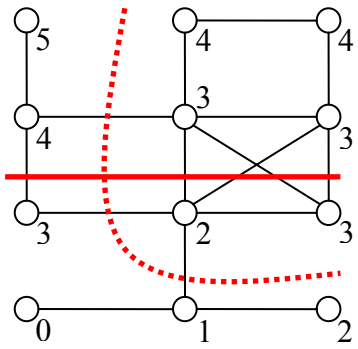


**Figure 11.15.** The example of the graph dissection algorithm with regard to adjacency

#### 11.3.4.2. The Kernighan-Lin Algorithm

*The Kernighan-Lin algorithm* makes use of another approach in order to solve the problem of optimum graph partition. It is assumed from the very beginning, that some initial graph partition already exists. Then the approximation is improved in the course of a number of iterations. The method of improvement used in the Kernighan-Lin algorithm consists in the exchange of vertices among the subsets of the available graph partition

(see Figure 11.16). To form the required number of graph parts it is possible, as previously, to use the recursive bisection method.

The iteration of the Kernighan-Lin algorithm may be described as follows:

1. Forming a set of pairs of vertices for permutation

   The vertices, which have not yet been rearranged at the given iteration, are taken to create all the possible pairs (the pairs should contain vertices from each part of the given graph partition)

2. Creating new variants of graph partition

   Each pair, prepared at step 1, is used by turns for exchangig vertices among the parts of the available graph partition. It is done for obtaining the set of new partition variants.

3. Choosing the optimum variant of graph partition

   The optimum variant is chosen for the set of new variants of graph partition obtained at step 2. This variant is fixed as the new current graph graph partition. The pair of vertices, which corresponds to the selected variant, is marked as being used at the current algorithm iteration.

4. Checking of graph vertex availability

   If there are any graph vertices, which have not been used in permutation, the algorithm iteration is repeated beginning from step 1. Otherwise step 5 follows.

5. The choice of the optimum graph partition

   The best generated variant of graph partition is chosen among all the graph partitions, obtained at step 3 of the carried out iterations.

**Algorithm 11.3.** The iteration of the Kernighan-Lin algorithm

It should be additionally explained that the permutation of each pair of vertices is carried out at step 2 of the algorithm iteration for the same graph partition, which was chosen before the beginning of iteration or was determined at step 3. The total number of carried out iterations can be considered as the algorithm parameter and, as a rule, is fixed beforehand. In should be noted that calculations may be terminated due to the lack of improvement in graph partition at some iteration).

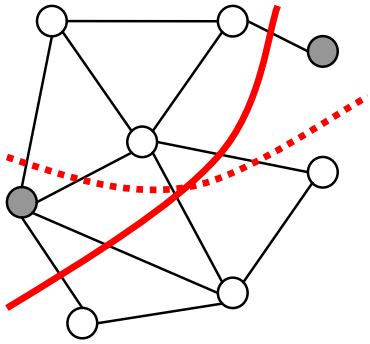

**Figure 11.16.** The example of permutation of two vertices (marked by the grey color)

### 11.3.5. Efficiency Analysis

The above considered algorithms of graph partition differ in accuracy of their solutions, the execution time and the possibilities of parallelizing (accuracy is the value of closeness of the solutions obtained by algorithms to the optimum variants of graph partition). The choice of the appropriate algorithm in each particular case is a complicated and unevident problem. The general characteristics of the algorithms discussed in this Section, which is given in Table 11.5, may be helpful in this respect (see Schloegel, et al. (2000)). Additional information on the problem of graph partition may be obtained, for instance, in Schloegel, et al. (2000).

**Table 11.5.** The comparative table of some graph dissection algorithms

| | Necessity of coordinate | Accuracy | Execution | Possibilities for |
|---|---|---|---|---|

| | | information | | time | parallelizing |
|---|---|---|---|---|---|
| Coordinate nested dissection | | yes | ● | ● | ●●● |
| Recursive inertial bisection | | yes | ●● | ● | ●●● |
| Levelized nested dissection | | no | ●● | ●● | ●● |
| Kernighan-Lin algorithm | 1 iteration | no | ●● | ●● | ● |
| | 10 iterations | no | ●●●◐ | ●●● | ●● |
| | 50 iterations | no | ●●●● | ●●●◐ | ●● |

The column "Necessity of coordinate information" shows whether and how the algorithm uses the coordinate information about the grid elements or graph vertices.

The column "Accuracy" gives the quality characteristic of the closeness of the solutions obtained by the algorithm to the optimum variants of graph partition. Each additional shaded circle signifies approximately 10% improvement of the approximation accuracy (correspondingly, each half shaded circle signifies 5 % improvement of the obtained solution).

The column "Execution time" shows the relative time needed for different partition algorithms. Each additional shaded circle corresponds to 10 times increase of the partition time (a half shaded circle correspondingly marks a 5 times increase).

The column "Possibilities for Parallelizing" characterizes the algorithm features for parallelization. The Kernighan-Lin algorithm in case of carrying out only one iteration is practically not parallelizable. The same algorithm in case of multiple iterations and also the levelized nested dissection method may be parallelized with average efficiency. The coordinate nested dissection algorithm and the recursive inertial bisection method can be efficiently parallelized.

## 11.4. Summary

The Section discusses several algorithms for solving some well known problems of graph calculations. Besides, it presents a review of graph partition methods.

In Subsection 11.1 *the Floyd algorithm* is considered *for solving the problem of search for the shortest paths among all the pairs of graph vertices*. To develop the parallel variant of the Floyd method a complete design cycle is carried out. The sequential computational scheme is described, the possible ways for algorithm parallelizing are discussed, the efficiency of the obtained parallel computations is evaluated, the software implementation is suggested and the results of the computational experiments are given. The approach, which is used for parallelizing the Floyd algorithm, consists in distributing the vertices of the graph among the processors. In this case the information communications at each method iteration consist in broadcasting an adjacency matrix row from a processor to all the processors.

In Subsection 11.2 *the Prim algorithm* is described. The algorithm is used for solving *the problem of finding the minimum spanning tree for a non-oriented weighted graph*. The graph spanning tree is the subgraph, which holds all the vertices of the original graph and has the minimum summary weight. The parallel Prim algorithm is also based on distributing the graph vertices among the processors. The amount of information communications is somewhat bigger than in the case of the Floyd algorithm. At each iteration the gather communication operation is executed. Then the selected graph vertex is distributed to all the processors.

In Subsection 11.3 *the problem of optimum graph partition* is discussed. This problem is essential in various areas, which involve parallel computations. As an example it is shown that the problem of distributing grid computations among the processors may be reduced to the problem of optimum graph partitioning.

To solve the graph partition problem two different types of methods are considered. *The geometric methods* use only coordinate information for graph partitioning. Among these methods *the coordinate nested dissection algorithm*, *the recursive inertial bisection method*, *the space-filling curve techniques* are given. Other approach is exploited in *the combinatorial algorithms*, at which the adjacency of the graph vertices is taken into account. The methods of this type are *the levelized nested dissection method* and *the Kernighan-Lin algorithm*. The comparison of the two approaches is illustrated by analyzing such characteristics as execution time, accuracy of the obtained solution, possibility of parallelizing etc.

## 11.5. References

Additional information on the Floyd method and the Prim algorithm may be obtained, for instance, in Cormen, et al. (2001).

More detailed information on the problem of graph partition may be found in Schloegel, et al. (2000), Berger, et al. (1987), Gilbert, et al. (1995), Heath, et al. (1995), Miller, et al. (1993), Nour-Omid, et al. (1986), Patra, et al. (1998), Pilkington, et al. (1994), Raghavan (1993), George (1981).

Parallel algorithms of graph partition are discussed in Barnard (1995), Gilbert, et al. (1987), Heath, et al. (1995), Karypis, et al. (1998, 1999), Raghavan (1995), Walshaw, et al. (1999).

## 11.6. Discussions

1. Give the definition of the graph. What are the main methods of graph representation on a computer?
2. What does the problem of searching all the shortest paths consist in?
3. Give the general scheme of the Floyd algorithm. What is the time complexity of the algorithm?
4. What approach can be applied to parallelize the Floyd algorithm?
5. What is the essence of the problem of searching the minimum spanning tree? Give an example illustrating how the problem can be used in practice.
6. Give the general scheme of the Prim algorithm. What is the time complexity of the algorithm?
7. What approach can be applied to parallelize the Prim algorithm?
8. What is the difference between the geometric and the combinatorial methods of graph partition? Which of them are preferable and why?
9. Describe the coordinate nested dissection method and the levelized nested dissection algorithm. Which of them is easier to implement?

## 11.7. Exercises

1. Develop the Floyd parallel algorithm using the given program code. Execute the computational experiments. Formulate the theoretical estimations. Compare the theoretical estimations with the experimental data.
2. Develop software implementation for the Prim parallel algorithm. Execute the computational experiments. Formulate the theoretical estimations. Compare the theoretical estimations with the experimental data.
3. Develop software implementation for the Kernighan-Lin algorithm. Estimate the possibilities of the algorithm parallelizing.

## References


**Berger, M., Bokhari, S.** (1987). Partitioning strategy for nonuniform problems on multiprocessors. - IEEE Transactions on Computers, C-36(5). P. 570-580.

**Cormen, T.H., Leiserson, C. E. , Rivest, R. L. , Stein C.** (2001). Introduction to Algorithms, 2nd Edition. - The MIT Press.

**George, A., Liu, J.** (1981). Computer Solution of Large Sparse Positive Definite Systems. - Prentice-Hall, Englewood Cliffs NJ.

**Gilbert, J., Miller, G., Teng, S.** (1995). Geometric mesh partitioning: Implementation and experiments. - In Proceedings of International Parallel Processing Symposium.

**Heath, M., Raghavan, P.** (1995). A Cartesian parallel nested dissection algorithm. SIAM Journal of Matrix Analysis and Applications, 16(1):235-253.

**Karypis, G., Kumar, V.** (1998). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing, 48(1).

**Karypis, G., Kumar, V.** (1999). Parallel multilevel k-way partitioning scheme for irregular graphs. Siam Review, 41(2): 278-300.

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

**Miller, G., Teng, S., Thurston, W., Vavasis, S.** (1993). Automatic mesh partitioning. - In A. George, John R. Gilbert, and J. Liu, editors, Sparse Matrix Computations: Graph Theory Issues and Algorithms. IMA Volumes in Mathematics and its applications. Springer-Verlag.

**Nour-Omid, B., Raefsky, A., Lyzenga, G.** (1986). Solving finite element equations on concurrent computers. - In A. K. Noor, editor, American Soc. Mech. P. 291-307.

**Ou, C., Ranka, S., Fox G.** (1996). Fast and parallel mapping algorithms for irregular and adaptive problems. - Journal of Supercomputing, 10. P. 119-140.

**Patra, A., Kim, D.** (1998). Efficient mesh partitioning for adaptive hp finite element methods. – In International Conference on Domain Decomposition Methods.

**Pilkington, J., Baden, S.** (1994) Partitioning with spacefilling curves. Technical Report CS94-349, Dept. of Computer Science and Engineering, Univ. of California.

**Pothen, A.** (1996). Graph partitioning algorithms with applications to scientific computing. - In D. Keyes, A. Sameh, and V. Venkatakrishnan, editors, Parallel Numerical Algorithms. Kluwer Academic Press.

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

**Raghavan, P.** (1993). Line and plane separators. - Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61901.

**Raghavan, P.** (1995). Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee.

**Schloegel**, K., Karypis, G., Kumar, V. (2000). Graph Partitioning for High Performance Scientific Simulations.

**Walshaw, C., Cross, M.** (1999). Parallel optimization algorithms for multilevel mesh partitioning. Techical Report 99/IM/44, Uviversity of Greenwich, London, UK.