A Mapping Strategy for MIMD Computers

Jiyuan Yang, Lubomir Bic, Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Technical Report No. 91-35

A Mapping Strategy For MIMD Computers

Jiyuan Yang, Lubomir Bic, Alexandru Nicolau

Department of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

Abstract

In this paper, a heuristic mapping approach which maps parallel programs, described by precedence graphs, to MIMD architectures, described by system graphs, is presented. The complete execution time of a parallel program is used as a measure, and the concept of critical edges is utilized as the heuristic to guide the search for a better initial assignment and subsequent refinement. An important feature is the use of a termination condition of the refinement process. This is based on deriving a lower bound on the total execution time of the mapped program. When this has been reached, no further refinement steps are necessary. The algorithms have been implemented and applied to the mapping of random problem graphs to various system topologies, including hypercubes, meshes, and random graphs. The results show reductions in execution times of the mapped programs of up to 77 percent over random mapping.

Index Terms— Critical edge, heuristic algorithm, mapping problem, parallel processing.

1 Introduction

In order to effectively utilize large-scale parallel computers, the scheduling problem is one of crucial importance. It is possible to divide the general scheduling problem into two parts — independent job scheduling and task scheduling. For independent job scheduling, there are optimal scheduling algorithms [4] for distributed computing systems and multiprocessors. Task scheduling, on the other hand, is much harder than independent job scheduling, since it needs to schedule multiple interrelated tasks in a single program for a parallel computer system. Many researchers have addressed task scheduling in various approaches [5], [6], [7], [12], [13],

[15]. The scheduling problems are usually classified into static and dynamic methods. This paper addresses static task scheduling.

A parallel program is represented by a *problem graph*, an example of which is shown in Fig. 2. The parallel computer system on which the parallel program is to be executed is referred to as a *system graph*. An example is shown in Fig. 5-a. The purpose of the static task scheduling presented in this paper is to minimize the complete execution time of the parallel program.

Usually, the number of nodes in the problem graph, np, is much larger than the number of nodes in the system graph ns, ($np \gg ns$). In order to simplify the scheduling problem, it can be divided into two steps. The first step, called *clustering*, combines np problem nodes into na groups, where $na = ns$. The edges connecting problem nodes within the same group are removed. The resulting graph is called a *clustered problem graph*. The second step, refered to as *mapping*, then maps the na clusters to the ns system nodes. Here, each cluster is treated as a single abstract node and edges connecting two abstract nodes are combined into one abstract edge. Under this abstraction, the second step only deals with graphs having the same number of nodes.

In this paper, we present an approach for performing the mapping of a clustered problem graph onto a system graph. In other words, we assume that an existing technique is first applied to produce a clustering from a given problem graph. The resulting clustered problem graph is then used as input to our algorithms. Note that the problem graph has the same number of nodes as the system graph, as has been done with other approaches. However, in our case, we still use the information about individual tasks within each cluster and their communication.

Since the mapping problem is NP-Complete, various heuristic algorithms have been devel-

A Mapping Strategy For MIMD Computers

Jiyuan Yang, Lubomir Bic, Alexandru Nicolau

Department of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

Abstract

In this paper, a heuristic mapping approach which maps parallel programs, described by precedence graphs, to MIMD architectures, described by system graphs, is presented. The complete execution time of a parallel program is used as a measure, and the concept of critical edges is utilized as the heuristic to guide the search for a better initial assignment and subsequent refinement. An important feature is the use of a termination condition of the refinement process. This is based on deriving a lower bound on the total execution time of the mapped program. When this has been reached, no further refinement steps are necessary. The algorithms have been implemented and applied to the mapping of random problem graphs to various system topologies, including hypercubes, meshes, and random graphs. The results show reductions in execution times of the mapped programs of up to 77 percent over random mapping.

Index Terms— Critical edge, heuristic algorithm, mapping problem, parallel processing.

1 Introduction

In order to effectively utilize large-scale parallel computers, the scheduling problem is one of crucial importance. It is possible to divide the general scheduling problem into two parts — independent job scheduling and task scheduling. For independent job scheduling, there are optimal scheduling algorithms [4] for distributed computing systems and multiprocessors. Task scheduling, on the other hand, is much harder than independent job scheduling, since it needs to schedule multiple interrelated tasks in a single program for a parallel computer system. Many researchers have addressed task scheduling in various approaches [5], [6], [7], [12], [13],

[15]. The scheduling problems are usually classified into static and dynamic methods. This paper addresses static task scheduling.

A parallel program is represented by a *problem graph*, an example of which is shown in Fig. 2. The parallel computer system on which the parallel program is to be executed is referred to as a *system graph*. An example is shown in Fig. 5-a. The purpose of the static task scheduling presented in this paper is to minimize the complete execution time of the parallel program.

Usually, the number of nodes in the problem graph, np, is much larger than the number of nodes in the system graph ns, ($np \gg ns$). In order to simplify the scheduling problem, it can be divided into two steps. The first step, called *clustering*, combines np problem nodes into na groups, where $na = ns$. The edges connecting problem nodes within the same group are removed. The resulting graph is called a *clustered problem graph*. The second step, refered to as *mapping*, then maps the na clusters to the ns system nodes. Here, each cluster is treated as a single abstract node and edges connecting two abstract nodes are combined into one abstract edge. Under this abstraction, the second step only deals with graphs having the same number of nodes.

In this paper, we present an approach for performing the mapping of a clustered problem graph onto a system graph. In other words, we assume that an existing technique is first applied to produce a clustering from a given problem graph. The resulting clustered problem graph is then used as input to our algorithms. Note that the problem graph has the same number of nodes as the system graph, as has been done with other approaches. However, in our case, we still use the information about individual tasks within each cluster and their communication.

Since the mapping problem is NP-Complete, various heuristic algorithms have been developed in the past [1], [2]. They focus primarily on minimizing the communication overhead. Bokhari [1] describes a mapping strategy, where the cardinality, defined as the number of the problem edges that fall on system edges, is used as the measure for evaluating a mapping. Unfortunately, the edges that don't fall on system edges can have a significant effect on the system's performance. Another limitation of this strategy is that all problem edges are assumed to have the same weight. However, in a general problem graph, the communication load carried by the different problem edges may vary significantly. Furthermore, the algorithm assumes $np \leq ns$, which imposes a serious limitation on the number of problems this method can be applied to.

Lee describes another mapping strategy which takes the phase for each problem edge into account, and uses actual distances between the system nodes rather than their nominal distances [2]. However, the assumption that all problem edges have to be activated simultaneously, i.e., all communications in one phase must start at the same time, is too restrictive for most applications. Similar to Bokhari, he also assumes that the number of nodes in the system graph must be equal to or greater than the number of nodes in the problem graph, i.e., $np \leq ns$. In most actual cases, the number of nodes in the problem graph is much larger than the number of nodes in the system graph.

The main drawback of both of the above mapping strategies is that they only consider communication cost but ignore execution time. We will show later that an optimal communication cost may still result with a non-optimal complete execution time.

Another limitation is inherent to the process of deriving a solution using these approaches, which is based on iterative improvement, i.e., repeatingly modifying assignments and comparing their results. Unfortunately, this process can't be terminated until a predetermined number of moves have been performed. Hence the search may continue long after the optimal solution has already been found.

Finally, neither approach considers any data dependencies among nodes. The assumed problem graphs are not directed, which means that they only consider the communications among the tasks but not their precedences.

The above limitations indicate the need for a better mapping strategy, which would consider data dependency, had a more realistic measure of how good a mapping is, and a better termination method. In this paper, we present such a mapping strategy. The complete execution time is used as the measure and the data dependencies are taken into account in the mapping process. The most important merit of this strategy is that, in some cases, it can detect when the optimal solution has been reached and thus no further refinement attempts are necessary. This reduces the total search space and time.

The paper is organized as follow. The terminology and the quality measure are introduced in section 2. The internal problem representation and mapping algorithms are described in sections 3 and 4, respectively. Experimental results are discussed in section 5. Finally, conclusions are given in section 6.

2 Terminology and Measure of Quality

In this section, we shall introduce some terms and discuss the measure for evaluating the goodness of a mapping.

2.1 Terms

The algorithms will use the following five graphs:

A problem graph $G_p = \{V_p, E_p\}$,

A clustered problem graph $G_c = \{V_c, E_c\}$,

An abstract graph $G_a = \{V_a, E_a\}$,

An ideal graph $G_i = \{V_i, E_i\}$,

A system graph $G_s = \{V_s, E_s\}$,

where V_p, V_c, V_a, V_i, V_s are sets of nodes and E_p, E_c, E_a, E_i, E_s are sets of edges in the respective graphs. The numbers of nodes in each graph are given by $np = \mid V_p \mid$, $nc = \mid V_c \mid$, $na = \mid V_a \mid$, $ni = \mid V_i \mid$ and $ns = \mid V_s \mid$, where $np = nc = ni$ and $na = ns$.

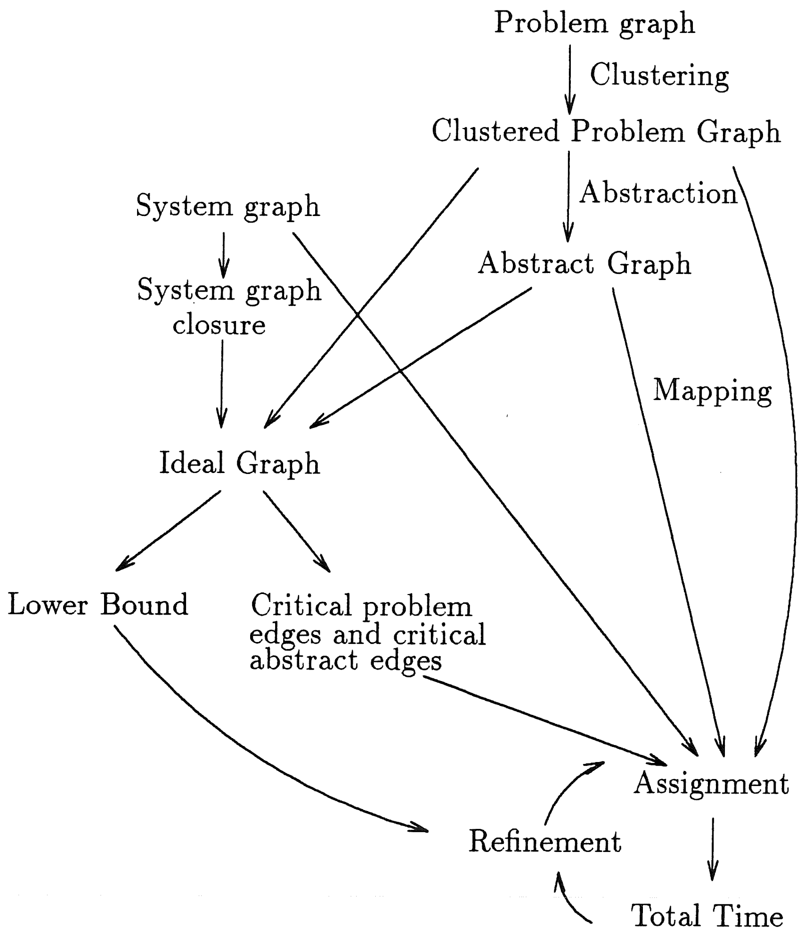


Fig. 1 Relationships among the graphs

The relationships among the graphs are shown in Fig. 1. The *problem graph* describes the tasks and their interactions. Fig. 2 is an example of a problem graph, where each node has an ID and a weight to indicate the number of time units for executing the task. Each edge also has a weight which represents the communication time. The *clustered problem graph* is derived from the problem graph by combining the problem nodes into groups. An example of a clustered problem graph derived from the problem graph in Fig. 2 is shown in Fig. 3. As mentioned earlier, we assume that an existing technique for clustering a given problem graph is used. Examples of such techniques may be found in [8], [9], [10], [11].

The *abstract graph* is the result of treating each cluster as one abstract node and collapsing edges between the same abstract nodes into one. Fig. 4 is the abstraction of the clustered problem graph in Fig. 3. The main purpose of the abstract graph is to be able to talk about all edges between two clusters as one. In particular, we need to know if a given abstract edge is critical. This information is used to guide the mapping.

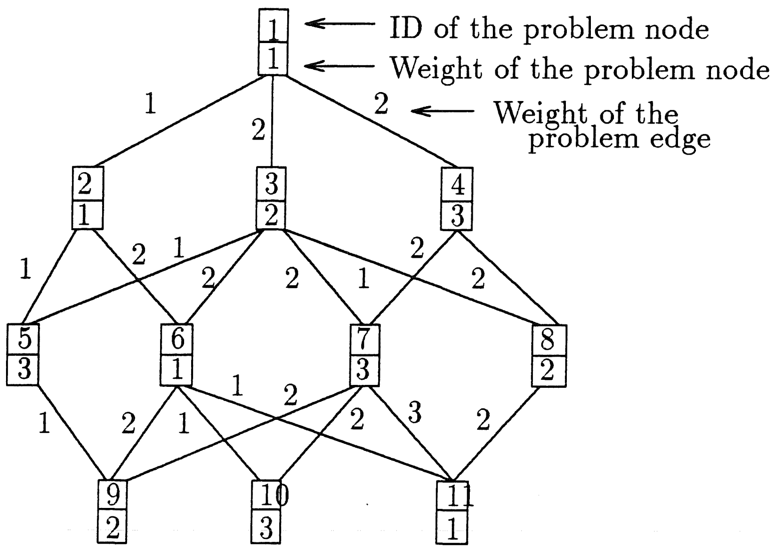


Fig.2 Problem Graph

The *system graph* describes the topology interconnecting homogeneous processing elements of a parallel computer system. A *system graph closure* is the fully connected superset of the system graph. Fig. 5-b shows the closure for the system graph of Fig. 5-a. The concept of the system graph closure is used to derive the *ideal graph*, which is a mapping of the clustered problem graph onto a fully connected system graph. This mapping is unique and is easily derivable, since the graphs have the same numbers of nodes and the communication cost between any two system nodes is identical (due to its full connectivity).

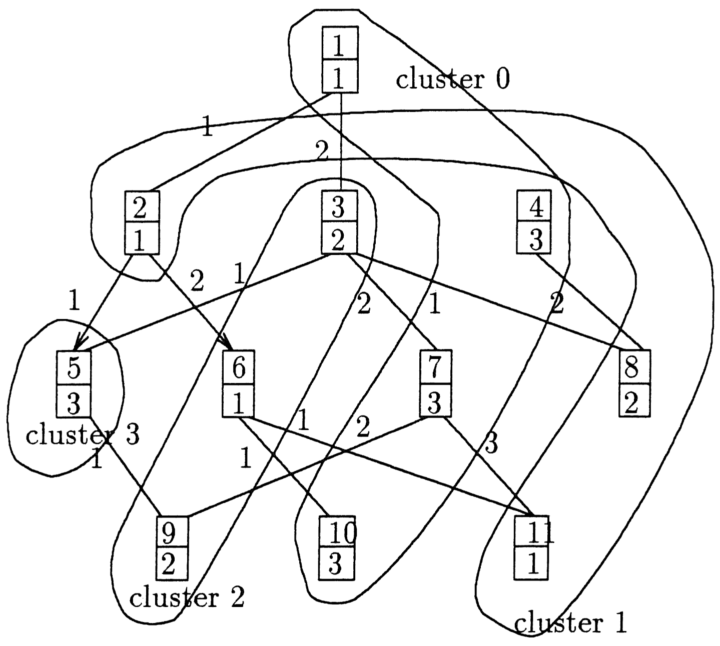


Fig. 3 Clustered problem graph

The purpose of deriving the ideal graph is to obtain a *lower bound* on the complete execution time of the parallel program. It is also used to derive the *critical problem edges* and *critical abstract edges* (see below) which are used to guide the mapping of the clustered problem graph to the actual system graph.

An example of the ideal graph resulting from mapping the clustered problem graph of Fig.

3 to the closure of Fig. 5-b is shown in Fig. 6. Note that the ideal graph carries the same information as the clustered problem graph, but is depicted in a different format to visually capture the time line of execution. In particular, the node weights are unchanged, each shown inside the corresponding node. The edge weights, on the other hand, are not shown explicitly as numbers attached to edges but as the time units that separate the nodes on the vertical axis. For example, the weight on the edge (1,3) is 2 (Fig. 3), which corresponds to a 2-time-unit delay between the end of node 1 and the beginning of node 3 (Fig. 6).

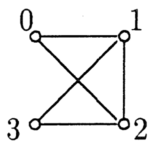


Fig. 4 Abstract graph

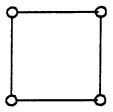
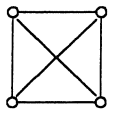



Fig. 5-a System graph Fig. 5-b System graph closure

The main difference between the clustered problem graph and the ideal graph, however, is that some edge weights become longer due to data dependencies. For example, the edge (6,11) carries the weight 1 in Fig. 3, but it results in 7 time units (weight 7) in Fig. 6. This is due to its dependency on node 7, which, in turn, depends on node 3, and so on. In other words, the ideal graph may be viewed as the topologically sorted form of the clustered problem graph.

When the abstract graph is mapped to the system graph, rather than its closure, an *assignment* is produced. Based on the assignment and the information in the clustered problem graph, the complete execution time of the parallel program can be derived. If the result is not equal to the lower bound, a refinement of the assignment is attempted, as illustrated in Fig. 1.

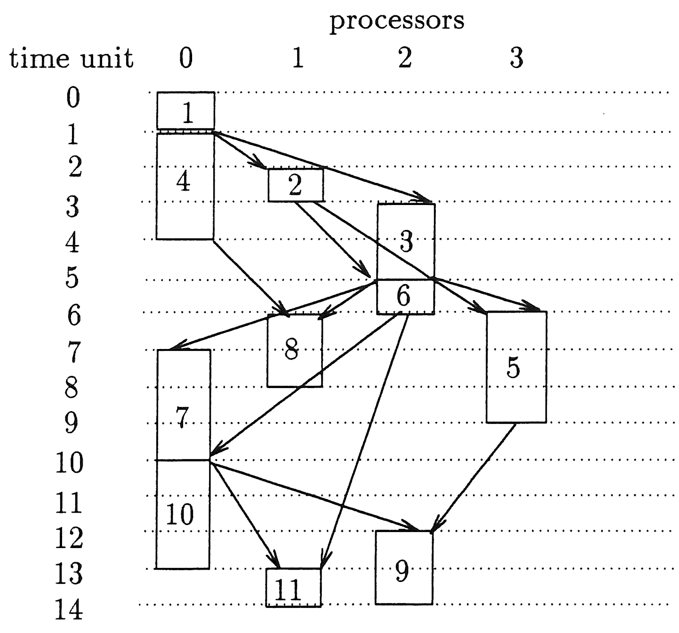


Fig. 6 Ideal graph

We assume that the execution time of each task (problem node) and the communication times are measured in time units. For any two problem nodes connected by an edge, we assume the worst case where each communication takes place between the end of the sending task and the beginning of the receiving task.

We will define several terms based on the graphs mentioned before, which will be used by the algorithms of section 4.

1. *Latest task* — This is the task (problem node) which terminates last. For example, in Fig. 6, tasks 9 and 11 are the latest tasks.

2. *Critical problem edge* — An edge in the ideal graph is critical if increasing the weight of the corresponding edge in the clustered problem graph by any amount will lengthen the complete execution time of the ideal graph. For example, the problem edge e_{i79} in Fig.

6 is critical, since any increase in the weight of the clustered problem edge e_{c79} (Fig. 3) must increase the weight of the ideal edge e_{i79} and thus delay the start time of the latest task 9. On the other hand, edge e_{i59} is not critical, since increasing the weight of the clustered problem edge e_{c59} will not necessarily increase the weight of the ideal edge e_{i79} (Only when the increase is by more than 2, will the ideal graph edge be affected).

3. *Critical abstract edge* — An abstract edge e_{akl} is critical if there is at least one critical problem edge e_{iij}, where ideal node v_{ii} is mapped onto abstract node v_{ak} and ideal node v_{ij} is mapped onto abstract node v_{al}. The abstract edges e_{a01}, e_{a02} in Fig. 4, for example, are critical.

4. *Critical degree of an abstract node* — This is the sum of the weights of all critical abstract edges directly connected to that abstract node. For example, there are two critical abstract edges e_{a01}, e_{a02} that connect to abstract node v_{a0}. Hence, the critical degree of v_{a0} is the sum of the weights of the two critical abstract edges.

5. *Critical abstract node* — An abstract node is critical if it is connected to a critical abstract edge which has been mapped to a single system edge. This is used in the algorithms for initial assignment and refinement (see 4.3.2 and 4.3.3).

6. *Total time* — This is the complete execution time of the parallel program.

2.2 Measure of Quality

The goodness of a mapping may be measured in different ways for different applications but in most cases, total time is the most important criterion. Other measures are based on the assumption that the total time would be minimized indirectly, as a result of minimizing (or maximizing) other characteristics, such as communication time. Unfortunately, this is not always the case. We demonstrate this by considering two other measures used by existing mapping strategies and showing that optimal mappings measured by other characteristics may be far from optimal in terms of the total time.

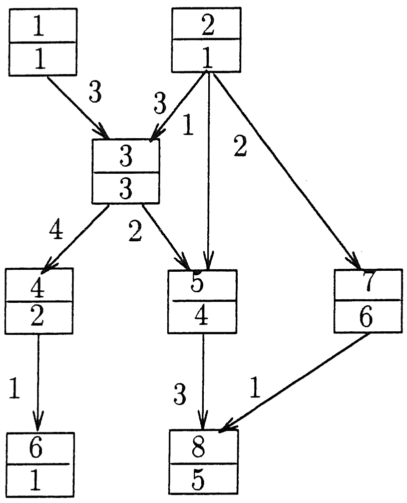


Fig. 7 Problem graph

For the convenience of comparison with Bokhari's approach [1], a problem graph, consisting of 8 nodes, is given in Fig. 7. Since $np = na = ns$, the clustered problem graph is the same as the problem graph. The system graph, which also has 8 nodes, is given in Fig. 8.

Every node in the system graph has degree 3. However, in the problem graph, node 3 has degree 4. Therefore, at least one problem edge which connects problem node 3 has to be mapped to two non-adjacent system nodes. Following Bokhari's cardinality measure, assignment A1 in Fig. 9 maps eight out of nine problem edges to a single system edge each while one problem edge, in this case e_{p35}, must be mapped on two system edges. It is easy to prove that A1 with cardinality 8 is the optimal solution according to the cardinality measure.

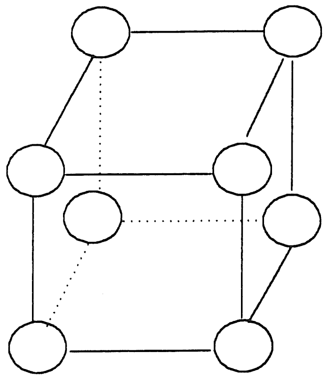


Fig. 8 System graph
(1,2,3,4,5,6,7,8)

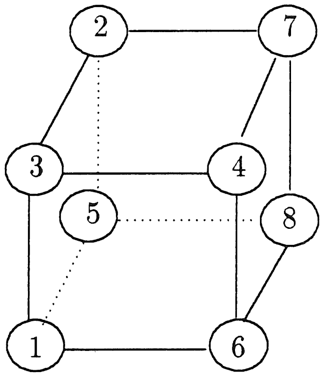


Fig. 9 Assignment A1
(1,3,5,2,6,4,8,7)

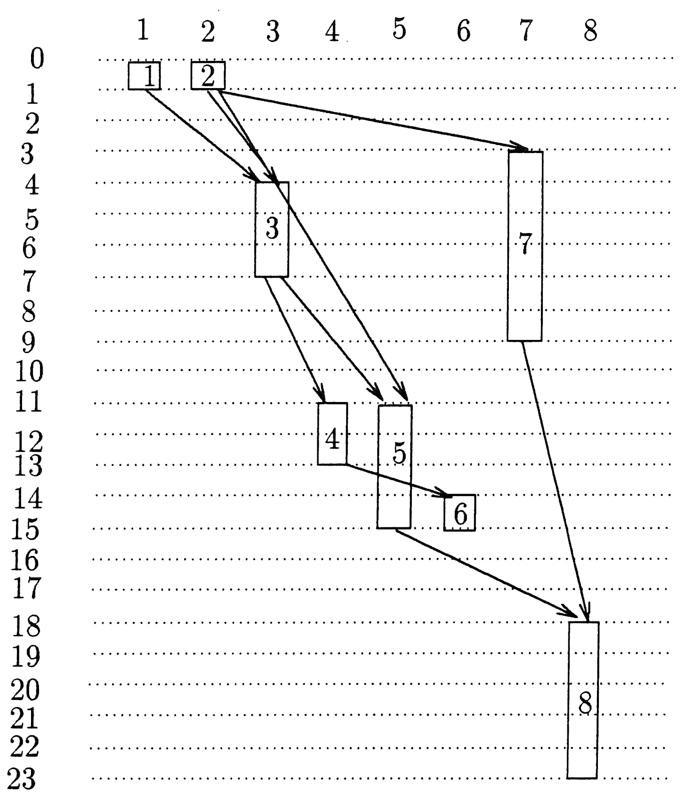


Fig. 10

Consider now the total time of the assignment A1, which is 23 time units, as shown in Fig. 10. On the other hand, in assignment A2, shown in Fig. 11, only seven problem edges are mapped to a single system edge each. Nevertheless, the total time under this assignment is 21 time units, as shown in Fig. 12, which is less than the total time under assignment A1 (23 units). In other words, assignment A1 with optimal cardinality does not have the minimum total time.

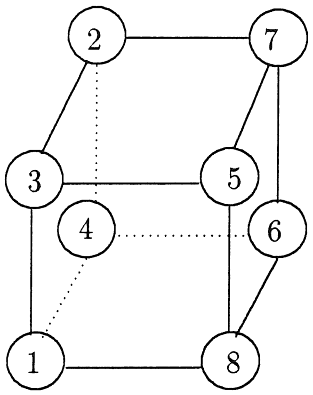


Fig. 11 Assignment A2

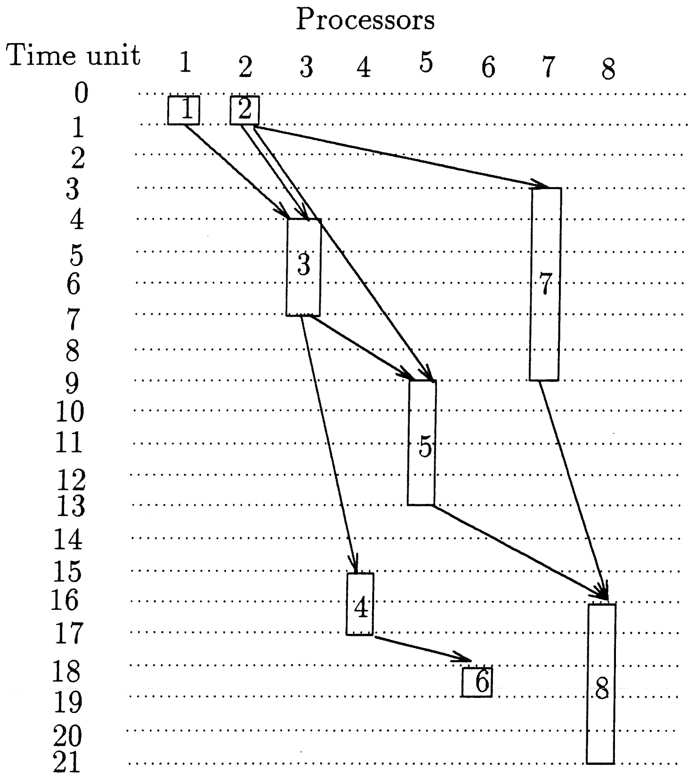


Fig.12 Execution time of Assignment A2

Another measure which has been used frequently to evaluate the goodness of a mapping is communication cost. We present a case where this measure also does not yield the optimal total time. Fig. 13 shows a problem graph that is to be mapped on the same system graph of Fig. 8. Based on the objective function and the phase to evaluate the communication cost developed by Lee [2], an optimal assignment A3 is given in Fig. 14. Four phases are identified. According to Lee's algorithm, all communications in the same phase are assumed to start at the same time. The communication cost in each phase is represented by the largest one among all the communication costs in that phase, and overall communication cost is the sum of the communication costs of all phases. The total communication cost for assignment A3 is 11 units, as shown in Fig. 15. It is easy to prove that assignment A3 has the minimum communication cost, but its total time is 23 units (Fig. 15). On the other hand, assignment A4 with 15 units of communication cost has 21 units of total time (Fig. 17), which is less than the total time of assignment A3 with optimal communication cost.

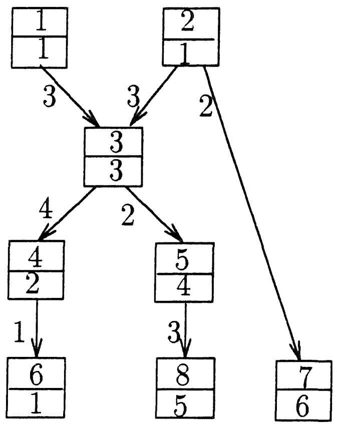


Fig. 13 Problem graph

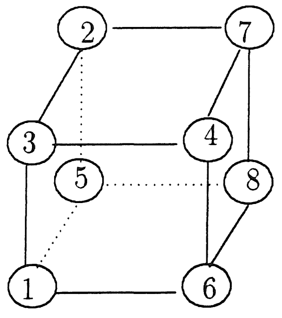
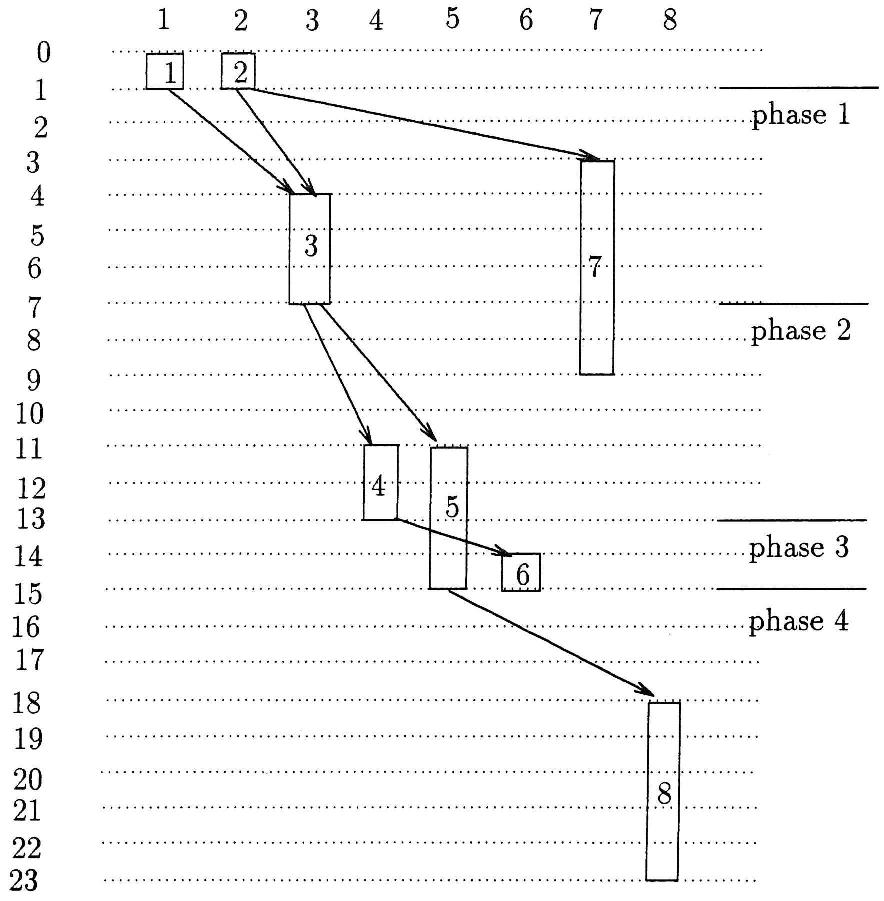


Fig. 14 Assignment A3



	edges and communication costs	max-comm-cost	
phase 1	(1,3)=3, (2,3)=3, (2,7)=2	3	
phase 2	(3,4)=4, (3,5)=4	4	
phase 3	(4,6)=1	1	
phase 4	(5,8)=3	3	+

sum of commu. cost = 11 units

Fig. 15

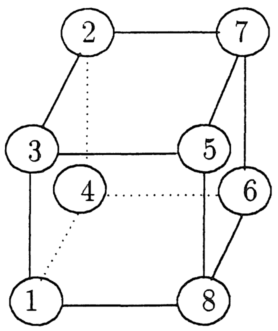
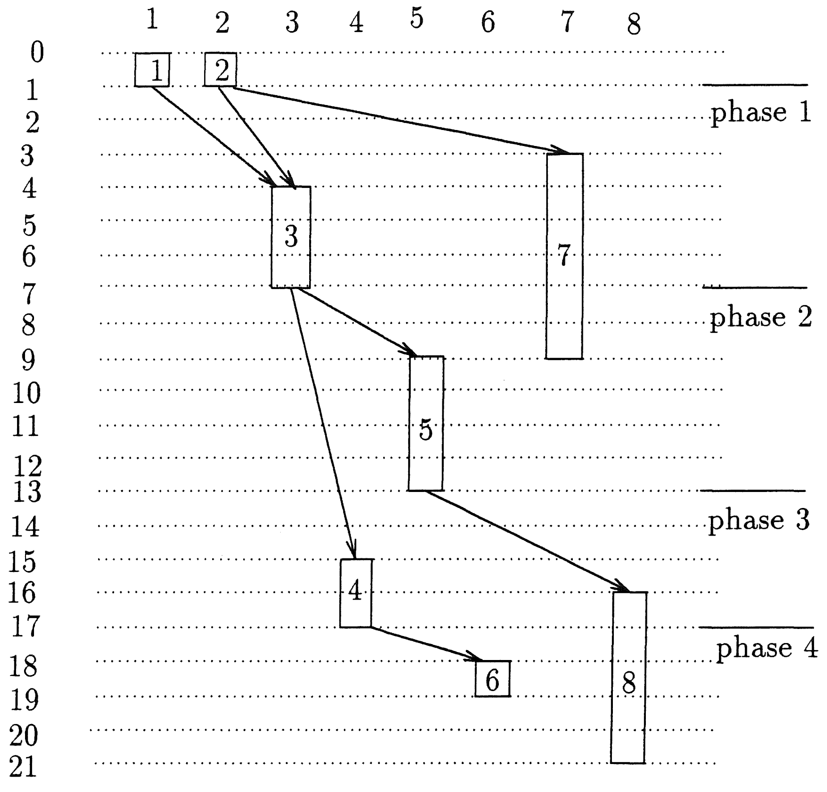


Fig. 16 Assignment A4



	edges and communication costs	max-comm-cost
phase 1	(1,3)=3, (2,3)=3, (2,7)=2	3
phase 2	(3,4)=8, (3,5)=2	8
phase 3	(5,8)=3	3
phase 4	(4,6)=1	+ 1
	sum of commu. cost	=15 units

Fig. 17

These examples illustrate that indirect measures do not always properly reflect the total execution time of a parallel program. Hence to achieve an optimal mapping, the total time will be used by our approach directly as the only measure.

3 Internal Representation

So far we have talked about all graphs of Fig. 1 at only a conceptual level. In this section we introduce the internal representation of each graph, and other auxiliary data structures, needed by the mapping algorithms presented in subsequent sections.

1. Problem graph

   (a) A problem edge matrix, $prob\_edge[np][np]$, describes the edges in the problem graph. Each element is denoted by $prob\_edge[i][j]$ where the value of $prob\_edge[i][j]$ is the weight of the problem edge. A problem edge matrix for the problem graph in Fig. 2 is given in Fig. 18.

   (b) $task\_size[np]$ is a one-dimension matrix which describes the execution time for each task, i.e., the weight of each node.

2. Clustered problem graph

   (a) A clustered edge matrix, $clus\_edge[np][np]$, contains the weights of the edges in the clustered problem graph. It is derived from the problem edge matrix by omitting the problem edges whose nodes are in the same abstract node. For example, the problem edge $prob\_edge[1][4]$ is eliminated in the clustered edge matrix in Fig. 19-a, since task 1 and task 4 are in the same abstract node v_{a0}.

(b) A cluster matrix *clus_pnode*$[na][np]$ describes which problem nodes are in which cluster. For example, in Fig. 19-b, the value of *clus_pnode*$[2][3]$, 9, is the ID of the third problem node in cluster 2. Note that the horizontal dimension is 11, which is the maximum size of any cluster, i.e., the total number of nodes in the graph.

3. Abstract graph

   (a) An abstract edge matrix, *abs_edge*$[na][na]$, represents the edges of the abstract graph. An element *abs_edge*$[i][j]$ contains 0 if there is no edge between node i and node j; otherwise, it contains a 1. Fig. 20-a is an example of the matrix representation of the abstract graph shown in Fig. 4.

   (b) A critical abstract edge matrix, *c_abs_edge*$[na][na+1]$, represents the critical abstract edges. The value of each element, except the elements in the last column, is the weight of the critical abstract edge. The last element of each row is the *critical degree*, defined as the sum of all the numbers in that row (see section 2.1). Fig. 20-b shows the critical abstract edge matrix corresponding to Fig. 20-a and Fig. 6.

   (c) The matrix, *mca*$[na]$, represents the communication intensity of the abstract nodes. Each element *mca*$[i]$ is the sum of the weights of all clustered problem edges which directly connect to abstract node i. For example, *mca*$[2] = 13$, as shown in Fig. 20-c, says that the sum of the weights of all edges that connect to abstract node 2 is 13.

4. System graph

   (a) The system graph is represented by a matrix *sys_edge*$[ns][ns]$. An example of a system graph and its matrix representation are given in Fig. 5-a and 21-a, respectively.

(b) A shortest path matrix, $shortest[ns][ns]$, represents the shortest path between any pair of system nodes. The shortest path matrix corresponding to the system graph of Fig. 5-a is shown in Fig. 21-b.

(c) A node degree matrix, $deg[ns]$, gives the degree of each system node. The node degree matrix corresponding to the system graph matrix in Fig. 21-a is shown in Fig. 21-c.

5. System graph closure

The system graph closure is fully connected. Its matrix contains all ones except the diagonal elements. Thus, it is not necessary to explicitly represent this closure by a matrix.

6. Ideal graph

(a) A matrix $i\_edge[np][np]$ is used to represent the ideal graph edges. The value of element $i\_edge[i][j]$ is the weight of an ideal edge. Each $i\_edge[i][j]$ is always equal to or greater than $clus\_edge[i][j]$. The reason is data dependencies (as explained in section 2). An example of the ideal graph edge matrix is shown in Fig. 22-a.

(b) Matrices $i\_start[np]$ and $i\_end[np]$ are used to represent the start time and end time of each task, respectively. Fig. 22-b shows two examples of the matrices. $i\_start[4] = 1$ means that start time of task 4 is at unit 1. $i\_end[4] = 4$ means that end time of task 4 is at unit 4.

(c) A matrix $crit\_edge[np][np]$ is used to describe the critical problem edges. $crit\_edge[i][j]$ is the weight of a critical problem edge between task i and task j. In Fig. 6, the edge $i\_edge[7][9]$ is critical, since task 9 terminates last and $i\_edge[7][9] = clus\_edge[7][9]$. The critical problem edge matrix corresponding to the ideal graph of Fig. 6 is shown in Fig. 22-c.

7. Assignment

Four matrices are used for representing the assignment of abstract nodes to system nodes: the assignment itself, the communication time between each pair of tasks, the start time, and the end time of each task.

(a) Assignment matrix, $assi[ns]$, expresses the assignment from abstract nodes to system nodes. The value of $assi[i]$ is the ID number of the abstract node that is mapped to system node i. Fig. 23-a is an example of an assignment. The numbers in brackets are the IDs of the system nodes, and the numbers without brackets are the IDs of the abstract nodes. For example, the abstract node 3 is mapped to system node 2. Fig. 23-b is an example of the assignment matrix corresponding to Fig. 23-a.

(b) A communication matrix $comm[np][np]$ describes the communication between any pair of problem nodes under a given assignment. When a clustered problem graph is mapped to a system graph, instead of the fully connected closure, a clustered problem edge may be mapped to more than one system edge. $comm[i][j]$ equals to $clus\_edge[i][j] * n_e$, where n_e is the number of system edges of the shortest path between two system nodes on which task i and task j are allocated. Fig. 23-c represents a communication matrix under the assignment A_i shown in Fig. 23-b. For example, the expression in the third row and the eighth column is $1 * 2$, which means that the communication with weight 1 between the two tasks 3 and 8 will pass two system edges.

(c) Two matrices *start*$[np]$ and *end*$[np]$ represent the start time and end time of each task, respectively. Examples of start time and end time matrices corresponding to the communication matrix *comm*$[np][np]$ are shown in Fig. 23-d.

$$\begin{array}{c|ccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
1 & & 1 & 2 & 2 & & & & & & & \\
2 & & & & & 1 & 2 & & & & & \\
3 & & & & & 1 & 2 & 2 & 1 & & & \\
4 & & & & & & & 2 & 2 & & & \\
5 & & & & & & & & & 1 & & \\
6 & & & & & & & & & 2 & 1 & 1 \\
7 & & & & & & & & & 2 & 2 & 3 \\
8 & & & & & & & & & & & 2 \\
9 & & & & & & & & & & & \\
10 & & & & & & & & & & & \\
11 & & & & & & & & & & &
\end{array}$$

Fig.18 Problem edge matrix

$$\begin{array}{c|ccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
1 & & 1 & 2 & & & & & & & & \\
2 & & & & & 1 & 2 & & & & & \\
3 & & & & & 1 & & 2 & 1 & & & \\
4 & & & & & & & & 2 & & & \\
5 & & & & & & & & & 1 & & \\
6 & & & & & & & & & & 1 & 1 \\
7 & & & & & & & & & 2 & & 3 \\
8 & & & & & & & & & & & \\
9 & & & & & & & & & & & \\
10 & & & & & & & & & & & \\
11 & & & & & & & & & & &
\end{array}$$

Fig.19-a
Clustered problem edge matrix

$$\begin{array}{c|ccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
1 & 1 & 4 & 7 & 10 & & & & & & & \\
2 & 2 & 8 & 11 & & & & & & & & \\
3 & 3 & 6 & 9 & & & & & & & & \\
4 & 5 & & & & & & & & & &
\end{array}$$

Fig.19-b Cluster matrix

$$\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 \\
2 & 1 & 1 & 0 & 1 \\
3 & 0 & 1 & 1 & 0
\end{array}$$

Fig. 20-a Abstract edge matrix

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 3 & 6 & 0 & 9 \\
1 & 3 & 0 & 0 & 0 & 3 \\
2 & 6 & 0 & 0 & 0 & 6 \\
3 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Fig. 20-b Critical abstract edge matrix $c\_abs\_edge[na][na+1]$

$$\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\hline
13 & 11 & 13 & 3
\end{array}$$

Fig. 20-c Matrix mca[4]

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 1 & 0 & 1 & 0 \end{array}$$

Fig. 21-a Matrix of system graph

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 1 \\ 1 & 1 & 0 & 1 & 2 \\ 2 & 2 & 1 & 0 & 1 \\ 3 & 1 & 2 & 1 & 0 \end{array}$$

Fig. 21-b Shortest path matrix of system graph

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 2 & 2 & 2 & 2 \end{array}$$

Fig. 21-c Node degree matrix

$$\begin{array}{c|ccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 1 & & 1 & 2 & & & & & & & & \\ 2 & & & & & 3 & 2 & & & & & \\ 3 & & & & & 1 & & 2 & 1 & & & \\ 4 & & & & & & & & 2 & & & \\ 5 & & & & & & & & & 3 & & \\ 6 & & & & & & & & & & 4 & 7 \\ 7 & & & & & & & & & 2 & & 3 \\ 8 & & & & & & & & & & & \\ 9 & & & & & & & & & & & \\ 10 & & & & & & & & & & & \\ 11 & & & & & & & & & & & \end{array}$$

Fig. 22-a Ideal edge matrix

$$\begin{array}{c|ccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 1 & & & 2 & & & & & & & & \\ 2 & & & & & & & & & & & \\ 3 & & & & & & & 2 & & & & \\ 4 & & & & & & & & & & & \\ 5 & & & & & & & & & & & \\ 6 & & & & & & & & & & & \\ 7 & & & & & & & & & 2 & & 3 \\ 8 & & & & & & & & & & & \\ 9 & & & & & & & & & & & \\ 10 & & & & & & & & & & & \\ 11 & & & & & & & & & & & \end{array}$$

Fig. 22-c Critical problem edge matrix

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 0 & 2 & 3 & 1 & 6 & 7 & 7 & 7 & 12 & 10 & 13 \end{array}$$

Start time *i_start*[11]

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 1 & 3 & 5 & 4 & 9 & 8 & 10 & 9 & 14 & 13 & 14 \end{array}$$

End time *i_end*[11]

Fig. 22–b Start time and end time of each task in the ideal graph

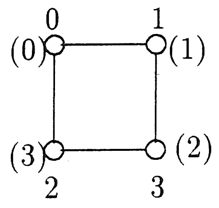


Fig.23-a Assignment

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 3 & 2 \end{array}$$

Fig.23-b Assignment matrix

$$\begin{array}{c|ccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
1 & & (1*1) & (2*1) & & & & & & & & \\
2 & & & & & (1*1) & (2*2) & & & & & \\
3 & & & & & (1*1) & & (2*1) & (1*2) & & & \\
4 & & & & & & & & (2*1) & & & \\
5 & & & & & & & & & (1*1) & & \\
6 & & & & & & & & & & (1*1) & (1*2) \\
7 & & & & & & & & & (2*1) & & (3*1) \\
8 & & & & & & & & & & & \\
9 & & & & & & & & & & & \\
10 & & & & & & & & & & & \\
11 & & & & & & & & & & &
\end{array}$$

Fig.23-c Communication matrix
comm[11][11]

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 0 & 2 & 3 & 1 & 6 & 7 & 7 & 7 & 12 & 10 & 13 \end{array}$$

start[11]

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 1 & 3 & 5 & 4 & 9 & 8 & 10 & 9 & 14 & 13 & 14 \end{array}$$

end[11]

Fig.23-d Start time and end time matrices
of each task under a given assignment

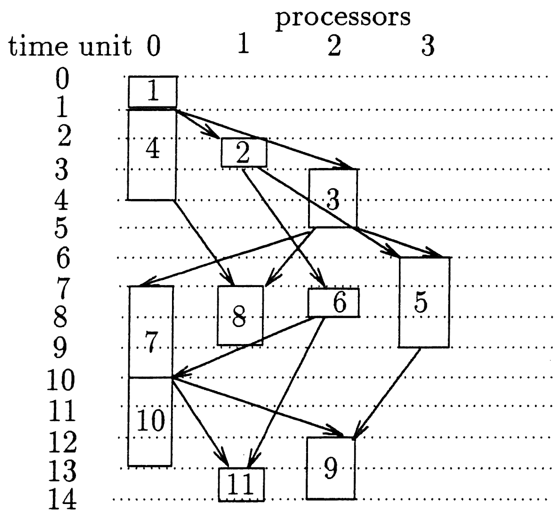


Fig. 24

4 Mapping Algorithms

In this section, we will introduce the algorithms which perform the transitions of Fig. 1.

4.1 Deriving the Ideal Graph

If we know the start/end time of each task, deriving the ideal graph, represented by the ideal edge matrix $i\_edge[i][j]$, is easy, since each ideal edge $i\_edge[i][j]$ is the difference between the start time of task j and the end time of task i. If we want to obtain the start time of a task, we have to know the communication times between the task and its predecessors. However, the ideal graph is the result of mapping a clustered problem graph to a fully connected closure, and thus the communication time between each pair of tasks is the same as the weight of the corresponding clustered problem edge; there is no need to derive the communication time for each pair of tasks. Instead, we can derive the start time, represented by matrix $i\_start[i]$, and end time, represented by matrix $i\_end[i]$, of all tasks directly from the clustered problem edge matrix $clus\_edge[np][np]$. Hence, deriving the ideal graph consists of two steps: first, derive start time and end time of each task; then, derive ideal edges.

The start time of a task is derived by adding together the end time of each predecessor and the corresponding communication time, and taking the maximum over all sums. This is because a task can only be executed after all its predecessors have finished. The end time of a task is the sum of its start time and its weight. Note that we can't find a task's predecessor only from matrix $clus\_edge[np][np]$, because two tasks may be in the same abstract node and the problem edge connecting the task and its predecessor is removed in the clustered problem graph. For example, we can't find the predecessor of task 4 from the matrix $clus\_edge[np][np]$ in Fig. 19-a. Instead, we have to check the problem edge matrix $prob\_edge[np][np]$ in Fig. 18, which tells us that task 1 is the predecessor of task 4. This yields the algorithm below.

I Algorithm for deriving the start and end time of each task in ideal graph

Do the following until all tasks have been visited:

1. For an unvisited task i, find its predecessors from matrix $prob\_edge[np][np]$.
2. If it has a predecessor, go to 3. Otherwise, do the following:
   (a) $i\_start[i] = 0$
   (b) $i\_end[i] = i\_start[i] + task\_size[i]$
   (c) Mark task i as visited.
3. If any one of the end times of all the predecessors of task i is unknown, go to 1. Otherwise, do the following:
   (a) For each predecessor j, derive $i\_start[i] = max_j(i\_end[j] + clus\_edge[j][i])$
   (b) $i\_end[i] = i\_start[i] + task\_size[i]$
   (c) Mark task i as visited.

The following explains how some elements of the start and end time matrices are obtained. In the matrix $prob\_edge[np][np]$ (Fig. 18), all the elements in column 1 are 0s, which means that no predecessor of task 1 exists. Therefore, task 1 can be executed first. Its start time, $i\_start[1]$, is 0 and its end time, $i\_end[1]$, is the sum of its start time and its weight, which is 1. In column 4 of Fig. 18, only the first row has a non-zero element, 2, Which means that only task 1 is the predecessor of task 4. Since the end time of task 1 is known, $i\_start[4] = i\_end[1] + clus\_edge[1][4]$. Then, $i\_end[1] = 1$ and $clus\_edge[1][4] = 0$, since task 1 and task 4 are in a same abstract node. The end time of task 4, $i\_end[4]$, consequently, is 4. In the same way, task 9 has three predecessors, 5, 6, and 7. So, the start time of task 9 is the maximum sum of the end times of its predecessors and the weights of the clustered problem edges between the predecessors and the task 9, i.e., 12. Its end time is 14. The complete start time and end time matrices derived from the problem edge matrix in Fig. 18 and the clustered problem edge matrix in Fig. 19-a using the above algorithm are shown in Fig. 22-b.

II Algorithm for deriving the lower bound

The lower bound is derived simply from the end time matrix:

$$lower\_bound = i\_end[l]$$

where l is the latest task.

After we obtain the start time and end time of each task, it is easy to derive the ideal edge matrix $i\_edge[np][np]$ which will be used to derive critical edges.

III Algorithm for finding ideal edge matrix $i\_edge[np][np]$

Do the following for each pair of tasks i and j:

1. If there is an edge between i and j in the clustered problem graph, and j is the predecessor of i, i.e., $clus\_edge[j][i] > 0$, then $i\_edge[j][i] = i\_start[i] - i\_end[j]$.
2. All other elements remain 0.

An example of the ideal edge matrix derived from the start and the end time matrices in Fig. 22-b is shown in Fig. 22-a.

4.2 Finding Critical Abstract Edges

To find the critical abstract edges we first need to find all critical problem edges. Following is the basic idea needed for finding the latter.

Theorem 1 An edge e_{iij} in the ideal graph is critical if its weight, $i\_edge[i][j]$, is equal to the weight of the corresponding edge, $clus\_edge[i][j]$, in the clustered problem graph and it directly connects to the latest task.

Proof : Since $i\_edge[i][j] = clus\_edge[i][j]$, any increase in the weight of the corresponding edge in the clustered problem graph, $clus\_edge[i][j]$, must increase the weight of the ideal edge $i\_edge[i][j]$. Increasing the weight of any ideal edge, $i\_edge[i][j]$, will delay the start time of the task j. Since the edge $i\_edge[i][j]$ directly connects to the latest task, increasing the weight of the clustered problem edge will lengthen the start time of the latest task, and consequently, the total time of the program. Hence, by definition, the edge e_{iij} is critical. $\square$

The following lemmas are needed to prove Theorem 2 below.

Lemma 1 If an ideal edge, $i\_edge[i][j]$, is critical, any delay of the start time of the task i will delay the start time of the task j by the same amount.

Proof : Since $i\_edge[i][j]$ is critical, $clus\_edge[i][j] = i\_edge[i][j]$. From the definition of the start time we know that $i\_start[j] = i\_start[i] + task\_size[i] + i\_edge[i][j]$. If $i\_start[i]$ increases by Δ, from the above equation, $i\_start[j]$ must also increase by the same Δ. $\square$

Lemma 2 If an ideal edge, $i\_edge[i][j]$, is critical, there exists a path from task i to the latest task l in which all the edges are critical.

Proof : We can distinguish the following three cases: the path from task i to the latest task consists of (1) one edge, (2) two edges, and (3) more than two edges. For first case, task j is the latest task. Therefore, the critical edge $i\_edge[i][j]$ is the path from task i to the latest task.

For the second case, we assume that the edge $i\_edge[j][l]$ is not critical and show that, under this assumption, $i\_edge[i][j]$ is also not critical, which contradicts the condition of the lemma.

First, from the assumption it follows that we can find an amount Δ such that increasing the weight $clus\_edge[j][l]$ by that amount will not increase the weight of the ideal edge, $i\_edge[j][l]$, i.e., $i\_edge[j][l] \geq clus\_edge[j][l] + \Delta$ will still hold. Next, if $i\_edge[i][j]$ is critical then, by definition, increasing the corresponding clustered problem edge by Δ will increase the total time. The total time before the increase is $total\_time = i\_end[j] + i\_edge[j][l] + task\_size[l]$. After the increase, $i\_end[j]' = i\_end[j] + \Delta$, and the total time is $total\_time' = i\_start[l]' + task\_size[l]$, where $i\_start[l]'$ is derived from the definition of the start time:

$$\begin{aligned} i\_start[l]' &= i\_end[j]' + clus\_edge[j][l] && (1)\\ &= i\_end[j] + \Delta + clus\_edge[j][l] && (2) \end{aligned}$$

Therefore,

$$\begin{aligned} total\_time' &= i\_start[l]' + task\_size[l] && (3)\\ &= i\_end[j] + \Delta + clus\_edge[j][l] + task\_size[l] && (4) \end{aligned}$$

From the inequality

$$i\_edge[j][l] \geq clus\_edge[j][l] + \Delta$$

derived above and the equation for total time, we obtain the following inequality

$$\begin{aligned} total\_time' &= i\_end[j] + \Delta + clus\_edge[j][l] + task\_size[l] && (5)\\ &\leq i\_end[j] + i\_edge[j][l] + task\_size[l] && (6)\\ &= total\_time && (7) \end{aligned}$$

Hence, increasing the weight of the clustered problem edge, $clus\_edge[i][j]$, doesn't lengthen the total time, and thus the ideal edge, $i\_edge[i][j]$ is not critical. This contradicts the given condition and, therefore, the edge $i\_edge[j][l]$ is critical.

For the third case we assume that there is no path $(i, ..., l)$ such that all edges on this path are critical and derive a contradiction from this assumption. Suppose that there is exactly one ideal edge, $i\_edge[k_1][k_2]$, on the path $(i, j, ..., k_1, k_2, ..., l)$, which is not critical. Similar to the second case, we derive the inequality $i\_edge[k_1][k_2] \geq clus\_edge[k_1][k_2] + \Delta$. The total time before the increase is $total\_time = i\_end[k_1] + i\_edge[k_1][k_2] + (i\_end[l] - i\_start[k_2])$. Note that all edges from node k_2 to the latest task l are critical. From lemma 1, the difference, $(i\_end[l] - i\_start[k_2])$, is a constant if the weights of the corresponding clustered problem edges are not changed. Thus, we have

$total\_time = i\_end[k_1] + i\_edge[k_1][k_2] + C$

Similar to the second case, we increase the weight of the clustered problem edge, $clus\_edge[i][j]$ by some amount Δ. The weight of the corresponding ideal edge, $i\_edge[i][j]'$, will increase by the same amount, since it is critical. This will delay the start time of task j by Δ. Since all the edges from task i to task k_1 on the path are critical, by lemma 1, the delay, Δ, will be transferred to the task k_1, i.e., $i\_end[k_1]' = i\_end[k_1] + \Delta$. From the definition of the start time of a task, we obtain $i\_start[k_2]' = i\_end[k_1]' + clus\_edge[k_1][k_2]$. Thus, we have

$$\begin{aligned} total\_time' &= i\_start[k_2]' + C && (8) \\ &= i\_end[k_1] + \Delta + clus\_[k_1][k_2] + C && (9) \\ &\leq i\_end[k_1] + i\_[k_1][k_2] + C && (10) \\ &= total\_time && (11) \end{aligned}$$

As in case two, increasing the weight of the clustered problem edge, $clus\_edge[i][j]$ doesn't lengthen the total time, and thus the ideal edge, $i\_edge[i][j]$ is not critical. This is contrary to the given condition. Therefore, the edge $i\_edge[k_1][k_2]$ and all other edges on the path are critical. □

Lemma 3 If an ideal edge, $i\_edge[i][j]$, is critical, any delay of the start time of the task i will postpone the total time.

Proof : From lemma 2, there exists a path from i to the latest task l in which all the edges are critical. Then by applying lemma 1 repeatedly, the delay of the start time of the task i will postpone the total time. □

Theorem 2 An edge in the ideal graph, $i\_edge[i][j]$, is critical if its weight is equal to the weight of the corresponding edge in the clustered problem graph, $clus\_edge[i][j]$, and it is the predecessor of a critical problem edge $i\_edge[j][k]$.

Proof: Increasing the weight of the edge $clus\_edge[i][j]$ will delay the start time of task j, because $i\_edge[i][j] = clus\_edge[i][j]$. Since the task j is connected by a critical problem edge, the delay of the start time of the task j will lead to lengthening of the end time of the latest task (total time), based on lemma 3. Consequently, this edge is critical. □

Based on the above theorems, we can find all critical problem edges in an ideal graph recursively. This is the main idea of the following algorithm.

I Algorithm for finding critical problem edges

1. Find the set, LS, of latest tasks v_i in the ideal graph from the end time matrix $i\_end[np]$. LS can have one or more elements.

2. Repeat until LS is empty.

   For each v_i in LS do the following:

(a) Find the predecessors of v_i in the matrix $clus\_edge[np][np]$.

(b) For each predecessor v_j, compare the weight of the corresponding edge in the clustered problem graph, $clus\_edge[j][i]$, with the weight of the edge in the ideal graph, $i\_edge[j][i]$. If they are equal, then the edge $i\_edge[j][i]$ is critical, and $crit\_edge[j][i] = clus\_edge[j][i]$.

(c) Include all predecessors of v_i that are connected to v_i by a critical problem edge(s) in LS.

3. All other elements of $crit\_edge[np][np]$ remain 0.

An example of a critical problem edge matrix, resulting from the clustered problem matrix (Fig. 19-a) and the ideal edge matrix (Fig. 22-a), is shown in Fig. 22-c.

Since an abstract edge is derived by collapsing clustered problem edges between the same abstract nodes into one, by definition, if there is at least one critical problem edge among those clustered problem edges, this abstract edge is critical. Therefore, the algorithm for finding the critical abstract edge is to detect whether there is a critical problem edge in an abstract edge. We have the following algorithm.

II Algorithm for finding the critical abstract edges

1. Find all critical problem edges $crit\_edge[i][j]$ (above algorithm).

2. For abstract nodes v_{al}, v_{am}, where $(0 \leq l, m \leq na - 1)$, find all critical problem edges $crit\_edge[i][j]$ where, problem nodes i, j are included in abstract nodes v_{al}, v_{am}, respectively. Assign the sum of the weights of the critical problem edges as the weight of this critical abstract edge.

$$c\_abs\_edge[v_{al}][v_{am}] = \sum(crit\_edge[i][j] + crit\_edge[j][i])$$

Based on the fact that the critical degree of each abstract node is the sum of the weights of all critical abstract edges directly connected to the abstract node, we have the following algorithm.

III Algorithm for finding the critical degree of each abstract node

For all i $(0 \leq i \leq na - 1)$, do

$$c\_abs\_edge[i][na] = \sum_{j=0}^{na-1} c\_abs\_edge[i][j]$$

An example of the critical abstract edge matrix is shown in Fig. 20-b.

4.3 The Mapping Algorithm

The mapping is performed in two stages: initial assignment and refinement. How to refine the initial solution and when to stop the refinement are the two most important aspects.

4.3.1 The Termination Condition

The system graph closure provides a means to derive a lower bound on the total time of the problem program. This is based on the following theorem.

Theorem 3 If the total time of any assignment is equal to the total time of the ideal graph, this assignment is an optimal mapping.

Proof: The communication time between any pair of system nodes can't be lower than that of the corresponding pair of the nodes in the closure, since the latter is fully connected. Therefore, the total time of any assignment can't be lower than the total time of the ideal graph, which is the result of mapping the abstract graph to the system graph closure. If the total time of an assignment is equal to the total time of the ideal graph, then the optimal mapping is reached. □

(a) Find the predecessors of v_i in the matrix $clus\_edge[np][np]$.

(b) For each predecessor v_j, compare the weight of the corresponding edge in the clustered problem graph, $clus\_edge[j][i]$, with the weight of the edge in the ideal graph, $i\_edge[j][i]$. If they are equal, then the edge $i\_edge[j][i]$ is critical, and $crit\_edge[j][i] = clus\_edge[j][i]$.

(c) Include all predecessors of v_i that are connected to v_i by a critical problem edge(s) in LS.

3. All other elements of $crit\_edge[np][np]$ remain 0.

An example of a critical problem edge matrix, resulting from the clustered problem matrix (Fig. 19-a) and the ideal edge matrix (Fig. 22-a), is shown in Fig. 22-c.

Since an abstract edge is derived by collapsing clustered problem edges between the same abstract nodes into one, by definition, if there is at least one critical problem edge among those clustered problem edges, this abstract edge is critical. Therefore, the algorithm for finding the critical abstract edge is to detect whether there is a critical problem edge in an abstract edge. We have the following algorithm.

II Algorithm for finding the critical abstract edges

1. Find all critical problem edges $crit\_edge[i][j]$ (above algorithm).

2. For abstract nodes v_{al}, v_{am}, where $(0 \leq l, m \leq na - 1)$, find all critical problem edges $crit\_edge[i][j]$ where, problem nodes i, j are included in abstract nodes v_{al}, v_{am}, respectively. Assign the sum of the weights of the critical problem edges as the weight of this critical abstract edge.

$$c\_abs\_edge[v_{al}][v_{am}] = \sum(crit\_edge[i][j] + crit\_edge[j][i])$$

Based on the fact that the critical degree of each abstract node is the sum of the weights of all critical abstract edges directly connected to the abstract node, we have the following algorithm.

III Algorithm for finding the critical degree of each abstract node

For all i $(0 \leq i \leq na - 1)$, do

$$c\_abs\_edge[i][na] = \sum_{j=0}^{na-1} c\_abs\_edge[i][j]$$

An example of the critical abstract edge matrix is shown in Fig. 20-b.

4.3 The Mapping Algorithm

The mapping is performed in two stages: initial assignment and refinement. How to refine the initial solution and when to stop the refinement are the two most important aspects.

4.3.1 The Termination Condition

The system graph closure provides a means to derive a lower bound on the total time of the problem program. This is based on the following theorem.

Theorem 3 If the total time of any assignment is equal to the total time of the ideal graph, this assignment is an optimal mapping.

Proof: The communication time between any pair of system nodes can't be lower than that of the corresponding pair of the nodes in the closure, since the latter is fully connected. Therefore, the total time of any assignment can't be lower than the total time of the ideal graph, which is the result of mapping the abstract graph to the system graph closure. If the total time of an assignment is equal to the total time of the ideal graph, then the optimal mapping is reached. □

From theorem 3 we derive the following termination condition:

Termination Condition If the total time of an assignment is equal to the total time of the ideal graph, terminate the refinement process.

4.3.2 Initial Assignment

Initial assignment tries to achieve the smallest possible total time of a given clustered problem graph and a system graph. As mentioned before, the critical edges influence the goodness of the mapping most significantly. Hence the basic idea of the initial assignment algorithm is to map the critical edges to neighboring system nodes or at least as close as possible. This usually yields a very good initial assignment, as has been verified by our experiments.

The initial assignment algorithm consists of the following three steps:

1. (a) Using the matrix $deg[i]$, select node v_s from V_s such that $deg[v_s]$ has the maximum degree. In the event of a tie, select any qualifying node arbitrarily. Mark v_s as visited.

   (b) Select node v_a from V_a such that $c\_abs\_edge[v_a][na]$ has the maximum critical degree (see section 3.3.(b)). In the event of a tie, select any qualifying node arbitrarily. Mark v_a as visited.

   (c) $assi[v_s] = v_a$. Mark v_a as a critical abstract node.

2. Repeat until all the abstract nodes that have critical abstract edges have been visited.

   (a) Using the matrix $c\_abs\_edge[i][j]$, select node v_a from V_a such that v_a is unvisited, $c\_abs\_edge[v_a][na]$ has the maximum critical degree, v_a is a neighbor of some marked node v'_a in V_a and the abstract edge $abs\_edge[v_a][v'_a]$ is critical, i.e., $c\_abs\_edge[v_a][v'_a] > 0$. Mark v_a as visited.

(b) If a node v_s from V_s can be selected such that v_s is unvisited, $deg[v_s]$ has the maximum degree and v_s is a neighbor of some marked node v'_s in V_s ($sys\_edge[v_s][v'_s] > 0$), such that $assi[v'_s] = v'_a$, then $assi[v_s] = v_a$, mark v_s as visited, mark v_a as critical abstract node and go to (a). Otherwise, go to (c).

(c) Select node v_s from V_s such that v_s is unvisited, v_s is the closest node from some marked node v'_s in V_s, i.e., $shortest[v_s][v'_s]$ is smallest, and $assi[v'_s] = v'_a$, then $assi[v_s] = v_a$ and mark v_s as visited.

3. Repeat until all nodes in V_a have been visited.

   (a) Select node v_a from V_a such that v_a is unvisited, $mca[v_a]$ has the largest communication intensity and v_a is a neighbor of some marked node v'_a in V_a ($abs\_edge[v_a][v'_a] > 0$). Mark v_a as visited.

   (b) If a node v_s from V_s can be selected such that v_s is unvisited, $deg[v_s]$ has the largest degree and v_s is a neighbor of some marked node v'_s in V_s, ($sys\_edge[v_s][v'_s] > 0$), such that $assi[v'_s] = v'_a$, then $assi[v_s] = v_a$, mark v_s as visited and go to (a). Otherwise go to (c).

   (c) Select node v_s from V_s such that v_s is unvisited, v_s is the closest node of some marked node v'_s in V_s, i.e., $shortest[v_s][v'_s]$ is smallest, and $assi[v'_s] = v'_a$, then $assi[v_s] = v_a$, and mark v_s as visited.

4.3.3 Refinement

The initial assignment which uses the critical abstract edges to guide the mapping process is usually quite good, but subsequent refinement is likely to improve the mapping further.

An iterative improvement technique has been chosen to refine the mapping.

We wish to preserve the mappings of the critical abstract nodes to the system nodes, as performed by the initial assignment, since they map critical abstract edges to a single system edge each. Hence, we don't touch the critical abstract nodes, but only change the mapping of the non-critical abstract nodes. This is achieved by performing random changes to the assignment and keeping the new mapping if it is better than the current one. A total of ns changes are allowed. We make use of the termination condition at each iteration. It has been verified by our experiment that this method works better than pairwise exchanges [2]. The refinement procedure is described below.

1. Derive an initial assignment A1 (algorithm in section 4.3.2).
2. Evaluate the total time (see section 4.3.4).
3. If the total time of A1 is equal to the total time of the ideal graph, stop. Otherwise, go to 4.
4. Repeat the following ns times
   (a) Randomly assign the non-critical abstract nodes to the system nodes which are not occupied by critical abstract nodes.
   (b) Evaluate the total time of the changed assignment, A2.
   (c) If the total time of the changed assignment A2 is equal to the total time of the ideal graph, stop; the optimal solution has been reached.
   (d) If the total time of A2 is less than that of A1, assign A2 to be the current assignment; else keep A1.

It is very easy to obtain the time complexity of the former algorithms. The highest order is $O(np^2)$. In the refinement algorithm, procedure 4-(b) determines the time complexity. Because the time complexity of the algorithm for evaluating total time is also $O(np^2)$ and ns changes are allowed, the worst case of the complete procedure is $O(ns * np^2)$ time.

4.3.4 Evaluating Total Time

Mapping of the clustered problem graph to the system graph closure is similar to mapping it to the system graph. Hence, evaluating the total time is similar to deriving the lower bound, as described in section 4.1. The main difference is that the system graph usually is not fully connected. To derive the start and end time matrices, we first generate the communication matrix $comm[np][np]$, which describes the communication between any pair of problem nodes under a given assignment.

When an assignment is obtained, the relationship between each abstract node and each system node has been set up. Hence, the communication time for each pair of problem nodes can be derived by multiplying the weight of the clustered problem edge between the two nodes by n_{ij}, the length of the shortest path between the two system nodes on which the two problem nodes are allocated. Thus, $comm[i][j] = clus\_edge[i][j] * n_{ij}$. Based on this, we obtain the following algorithm for deriving the communication matrix.

I Algorithm for finding the communication matrix

1. Find the shortest path between any pair of system nodes $shortest[ns][ns]$ (use some existing algorithm [16]).

2. Do the following for each pair of the clustered problem nodes i and j:

   If i and j are in different abstract nodes v_{al}, v_{am}, and $assi[v_{sl}] = v_{al}, assi[v_{sm}] = v_{am}$, then

$$comm[i][j] = clus\_edge[i][j] \times shortest[v_{sl}][v_{sm}]$$

Using the matrix $comm[i][j]$, we can derive the start time and end time of each task by using the following algorithm, which is similar to that used for the ideal graph.

II Algorithm for deriving the start time and end time of each task under a given assignment

Do the following until all tasks have been visited:

1. For an unvisited task i, find its predecessors from matrix $prob\_edge[np][np]$.

2. If it has predecessors, go to 3. Otherwise, do the following:

   (a) $start[i] = 0$

   (b) $end[i] = start[i] + task\_size[i]$

   (c) Mark task i as visited.

3. If the end time of any of the predecessors of task i is unknown, go to 1. Otherwise, do the following:

   (a) For each predecessor j, derive $start[i] = max_j(end[j] + comm[j][i])$

   (b) $end[i] = start[i] + task\_size[i]$

   (c) Mark task i as visited.

III Algorithm for deriving the total time of the program under an assignment

Assign the maximum end time to the total time of the program. The node with the maximum end time is the latest task.

$$total\_time = max_j(end[j])$$

Fig. 24 shows the result of mapping the clustered problem graph from Fig. 3 onto the system graph in Fig. 5-a using the preceding algorithms. Since the total time of this initial assignment is equal to that of the ideal graph, it is an optimal mapping and no further refinement is needed.

5 Experiments

It is hard to compare one heuristic approach with other heuristic approaches. To avoid criticism for having used only several special examples particularly suited to our approach, random mapping was chosen to be compared with our mapping strategy. For this purpose, a random problem graph generator was created and a random clustering program was developed. The weights of the problem nodes and the weights of the problem edges are also produced randomly. The numbers of nodes in a problem graph range from 30 to 300, while the numbers of nodes in a system graph range from 4 to 40. The system topologies are hypercubes, meshes, and random graphs. All algorithms were implemented in C++ and run on a SUN-4 workstation.

Since the problem graphs and the clusterings are produced randomly, the numbers of the problem nodes and system nodes as well as the total time of each experiment fluctuate significantly, thus making it difficult to compare the total times by actual units. To compensate for the variance, we performed several random mappings of the same problem graph to the same system graph and take the average of the total times. The percentages of the total times derived by using our approach and the average total times derived by using random mappings over the lower bound are used to show the improvement.

5.1 Mapping to Hypercube

Table 1

expts	our approach	random	improvement
1	104	148	44
2	115	178	63
3	110	158	48
4	118	147	29
5	105	140	35
6	106	147	41
7	100	158	58
8	100	160	60
8	107	155	48
10	105	159	54

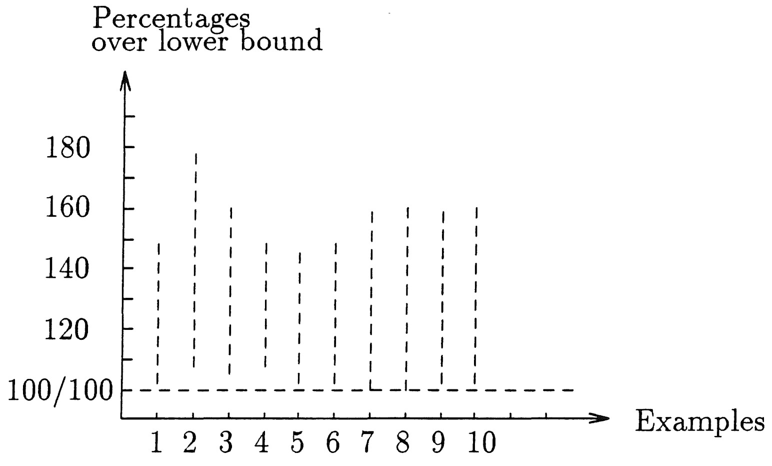


Fig. 25 Mapping to Hypercubes

The comparison between mapping randomly produced abstract graphs to a hypercube topology using our mapping strategy versus random mapping is shown Table 1. The first column is the experiment number. In each experiment, the lower bound is used as the basis for comparisons and is set to 100 percent. The results of using our approach and random mapping are represented by the percentages over the lower bound, as shown in columns 2 and 3, respectively. The fourth column describes the improvement of using our approach over the random mapping. Fig. 25 presents the same results in a graphical form. Each point on the horizontal axis corresponds to one problem graph, i.e., the figure is a histogram. The lower end of each vertical dashed line shows the result of a mapping using our mapping strategy; the higher end shows the random mapping result. For example, a lower end value of 110 and an upper end value of 160 mean that a program mapped by using our approach requires only 10% more time than the lower bound, while a random mapping would result in a 60% increase in total time. The results demonstrate that the improvement between the results of our approach and those of the random mapping range from 29 percent to 63 percent. In 2 out of 10 cases, our results reached the lower bound.

Table 2

expts	our approach	random	improvement
1	100	134	34
2	100	148	48
3	105	142	37
4	100	147	47
5	100	133	33
6	112	153	41
7	100	132	32
8	100	135	35
9	100	133	33
10	103	136	33
11	107	144	37

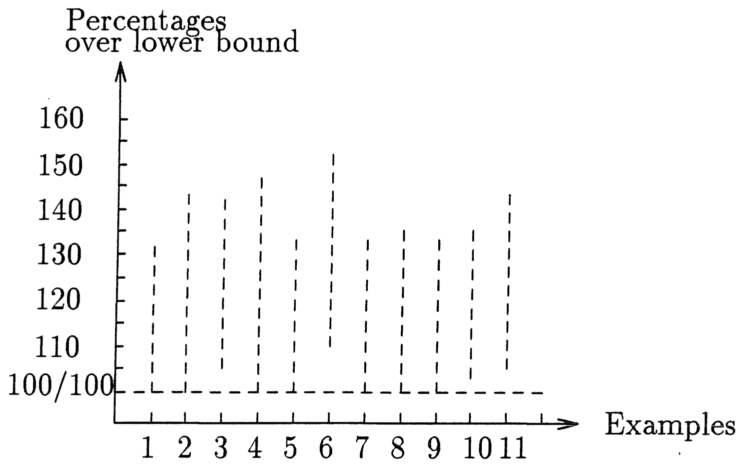


Fig. 26 Mapping to Meshes

Table 3

expts	our approach	random	improvement
1	102	163	61
2	107	178	71
3	105	152	47
4	105	158	53
5	112	180	68
6	104	161	57
7	100	153	53
8	114	182	66
9	108	173	65
10	100	148	48
11	100	149	49
12	105	153	48
13	102	158	56
14	100	177	77
15	100	168	68
16	102	148	46
17	103	147	44

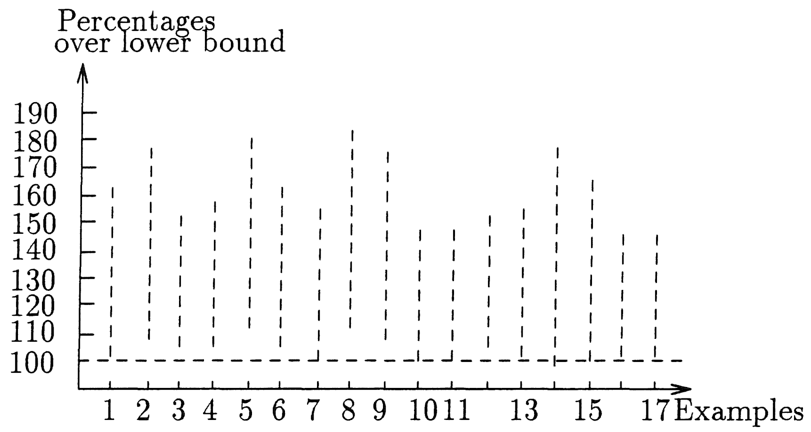


Fig. 27 Mapping to Randomly Produced Topologies

5.2 Mapping to Mesh and Random Topologies

Table 2 and Table 3 are analogous to Table 1, and Fig. 26 and 27 are analogous to Fig. 25. They show the results of mapping random problem graphs to mesh architectures and to randomly produced system architectures, respectively. They show improvements between the results of our approach and those of the random mapping ranging from 33 percent to 77 percent for the total time. The experiments also demonstrate that the termination condition works well. In Fig. 27 there are 4 out of 15 cases where our mapping stops the refinement by the termination condition. In Fig. 26, there are 7 out of 11 such cases.

6 Conclusion

In this paper, we presented a mapping strategy which maps a clustered problem graph to a system graph. This strategy uses the complete execution time of a parallel program, represented by a clustered problem graph, as the measure to evaluate the goodness of the mapping. Through the analysis of the critical edges based on the mapping of the abstract graph to the system graph closure, we obtain two important concepts: critical abstract edges and a lower bound. The former is used to guide the mapping by attempting to assign critical edges to a single system edge each. The latter allows us to derive a termination condition which stops unnecessary refinement and reduce both searching space and mapping time. The algorithms presented in this paper make it possible to map np problem nodes to ns system nodes where $np > ns$. The effectiveness of this approach has been verified empirically, by deriving the mappings of different, randomly generated problem graphs onto hypercube, mesh-connected, and random system graphs. The results have shown improvements ranging from 29 to 77% in total execution time over random mappings.

Acknowledgement

The authors would like to thank Meng-lai Yin for useful discussions.

References

[1] S. H. Bokhari, "On the Mapping Problem", *IEEE Trans. on Computers*, vol. V-30, pp. 207-214, Mar. 1981.

[2] S.-Y. Lee, J.K. Aggarwal, "A Mapping Strategy for Parallel Processing", *IEEE Trans. on Computers*, vol. V-36, pp. 433-442, April, 1987.

[3] S.Kirkpatrck, C.D.Gelatt, M.P. Vecchi, "Optimization by Simulated Annealing", *Science* , vol. V220, pp.671-680, May 13, 1983.

[4] L.M. Ni and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes", *IEEE Trans. on Software Eng.,* vol. SE-11, pp.491-496, May 1985.

[5] K. Fukunaga, S. Yamada, T. Kasai. "Assignment of Job Modules onto Array Processors", *IEEE Trans. on Computers*, vol. V-36, no. n7, pp. 888-891, July 1987.

[6] F. Berman, M. Goodrich, C. Koelbel, W.J. Robison, K. Showell, "Prep-P: A Mapping Processor for CHiP Computers", *Int'l Conf. on Parallel Processing*, pp. 731-733. 1985.

[7] P. Sadayappan, F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Trans. on Computers*, vol. V-36, pp. 1408-1424, Dec., 1987.

[8] A. Gerasoulis, S. Venugopal, T. Yang, "Clustering Task Graphs for Message Passing Architectures", *ACM International Conference on Supercomputing*, June 11-15, 1990, Amsterdam, Holand.

[9] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed systems", *IEEE Computer*, 15(6), 1982, pp.50-56.

[10] A. Gerasoulis, I. Nelken. "Static Scheduling for Linear Algebra DAGs." HCCA4(1989).

[11] M. Cosnard, M. Marrakchi, Y. Robert and D. Trystram. "Parallel Gaussian Elimination on an MIMD Computer." *Parallel Computing*, 6, 1988, pp. 275-296.

[12] J. Baxter, L. H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm", *IEEE Int'l Conf. on Parallel Processing*, pp. II217-222, Aug., 1989.

[13] L. Kim, C. R. Das and W. Lin, "A Processor Allocation Scheme for Hypercube Computers", *IEEE Int'l Conf. on Parallel Processing*, pp. II-231 - II-238, Aug., 1989.

[14] C. Lee, L. Bic, "Comparing Quenching and Slow Simulated Annealing in the Mapping Problem", *IEEE Third Annual Parallel Processing Symposium* April, 1989.

[15] D.T. Peng, K.G. Shin, "Static Allocation of Periodic Tasks With Precedence Constraints in Distributed Real-Time Systems", *9th Int'l Conf. on Distri. Compt. Syst.* , pp. 190-198, CA. June 5-9, 1989.

[16] S. Baase, "Computer Algorithms: Introduction to Design and Analysis", Second Edition, Addison-Wesley Publishing company, 1988.