BIT 28 (1988), 54–68

OPTIMAL ASSIGNMENT OF TASK MODULES WITH PRECEDENCE FOR DISTRIBUTED PROCESSING BY GRAPH MATCHING AND STATE-SPACE SEARCH

LING-LING WANG and WEN-HSIANG TSAI

Institute of Computer Engineering,
National Chiao Tung University,
Hsinchu, Taiwan 30050,
Republic of China

Department of Information Science,
National Chiao Tung University,
Hsinchu, Taiwan 30050,
Republic of China

Abstract.

A graph matching approach to optimal assignment of task modules with varying lengths and precedence relationship in a distributed computing system is proposed. Inclusion of module precedence into the optimal solution is made possible by the use of topological module orderings. Two graphs are defined to represent the processor structure and the module precedence relationship, respectively. Assignment of the task modules to the system processors is transformed into a type of graph matching. The search of optimal graph matching corresponding to optimal task assignment is formulated as a state-space search problem which is then solved by the A^* algorithm in artificial intelligence. Illustrative examples and experimental results are included to show the effectiveness of the proposed approach.

C.R. Categories: D.4.1, I.2.8.

Index Terms: Task assignment, distributed processing, graph matching, topological tree, minimax criterion, task turnaround time, state-space search, A^* algorithm.

1. Introduction.

A distributed computing system consists of two or more arbitrarily and incompletely interconnected processors. A concern in the research of distributed processing is how to utilize the processors evenly in a distributed computing system. This is the so-called task assignment problem [1, 9]. In such a problem, a task consisting of several modules is to be solved on a set of processors with the aim to reduce task turnaround time and to increase system throughput.

The purpose of reducing task turnaround time and increasing system throughput can be achieved by maximizing (or balancing) the utilization of resources and minimizing the communication between processors [2]. While minimizing interprocessor communication tends to assign the whole task to a single processor, load balancing tries to distribute the task modules evenly among the

Received October 22, 1986. Revised February 1987 and December 1987.

processors. Therefore, there exists a conflict between these two criteria and a compromise must be made to find an optimum policy for task assignment.

Several approaches [1–16] have been suggested, including the graph-theoretic approach, the integer 0–1 programming approach, the heuristic approach, the simulated annealing method, and approaches used in the construction of parallel computing elements in VLSI. In most of the approaches, module precedence is neglected. Some have considered this problem but only allow unit-length modules in a task [10]. In this paper, we remove these constraints and consider optimal assignment of task modules with varying lengths and precedence relationship.

Each graph match corresponds to a task assignment. Cost values are defined in terms of a single unit, time, and a state-space search method [19] is used for finding the minimal-cost graph match corresponding to the optimal task assignment.

Inclusion of module precedence into the optimal solution is made possible by the use of topological module orderings constructed from the precedence relationship. The proposed model allows most constraints encountered in practice to be easily incorporated. On the other hand, the proposed approach guarantees to find an optimal solution. This is especially important for those applications where the resulting assignment will be run on a distributed system repeatedly.

In the remainder of this paper, we describe the system assumptions in Sec. 2, and the graph matching model in Sec. 3. In Sec. 4, we describe the minimax criterion and in Sec. 5, we derive the cost function, namely, the task turnaround time. In Sec. 6, the state-space search method is reviewed and applied to finding the optimal solution. Some illustrative examples are given in Sec. 7. Conclusions appear in the last section.

2. System assumptions.

Various assumptions made of the task and the distributed computing system are described in the following.

(1) The processors in the system are heterogeneous, and the modules may have different degrees of preference of the processors.

(2) Nonidentical communication links are used by the processors for message transmission, and module communication processes may have different degrees of preference of the links.

(3) The link between any two processors is symmetric. Therefore, the time to transmit messages from one processor to another is identical to that to transmit the same messages in the reverse direction.

(4) There exists a precedence relationship among the task modules. This means that if module m_1 is a predecessor of module m_2, m_2 cannot be executed before

the result of m_1 is transmitted to m_2. This in turn means that processor idleness may occur in the system.

(5) To transmit messages from one processor to another, the latter must be free from module execution. This means that either of two processors in communation spends an identical amount of time for the communication.

3. Graph matching model.

Based on the purpose or the criterion of matching, there are various types of graph matching [17]. The type we consider is defined as follows.

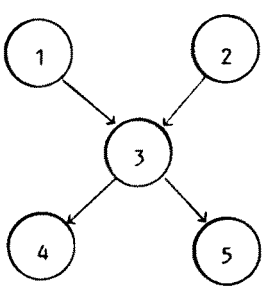
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs where V_1 and V_2 are the vertex sets, E_1 and E_2 are the corresponding edge sets. In this paper G_1 is said to *match* G_2 if there exists a mapping (possibly many-to-one) $M: V_1 \to V_2$ such that if (a, b) is in E_1, then there exists an edge in E_2 connecting $M(a)$ and $M(b)$.

A task submitted to a distributed computing system is first partitioned into suitable modules and then assigned to the processors. Each task can be represented as a directed acyclic graph $T = (V_T, E_T)$, which we call the *task graph*, where (1) V_T is a set of vertices, each of which represents a module of the task; (2) $E_T \subset V_T * V_T$ is a set of directed edges, each of which represents the precedence relationship of the two modules at the two ends of the edge. The module at the head of the edge must be executed before the module at the tail. The execution result is then transmitted from the former to the latter.

If there exists a path from x to y, then x is called a *predecessor* of y, and y a *successor* of x. In case the path from a vertex x to a vertex y is just the edge (x, y), x is called an *immediate predecessor* of y, and y an *immediate successor* of x.

The processors in a distributed computing system can be represented by an undirected graph $P = (V_P, E_P)$, which we call the *processor graph*, where (1) V_P is a set of vertices, each representing a processor in the system; (2) $E_P \subset V_P * V_P$ is a set of edges, each representing a communication link between two processors.

Because two related modules may be assigned to a single processor, we add a self-looping edge to each vertex in the processor graph. For example, shown in Figure 1 is a task graph and in Figure 2 a processor graph.



$T=(V_T, E_T)$

$V_T=\{1, 2, 3, 4, 5\}$

$E_T=\{(1,3), (2,3), (3,4),$
$(3,5)\}$

Fig. 1. A task graph.

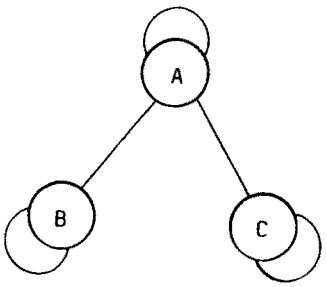


Fig. 2. A processor graph.

Now, considering the assignment of a module m to a processor p as a mapping from m to p, we see that task assignment is equivalent to matching the task graph with the processor graph. Thus, the task assignment problem can be formulated as a graph matching problem which will be solved by the state-space search method described in Sec. 6.

For each task graph, we can generate a linear ordering of the modules such that if m_i is a predecessor of m_j in the task graph, then m_i precedes m_j in the ordering. This ordering is called a *topological ordering* [18].

Considering each topological ordering as a tree path consisting of all the modules as nodes, we may build a topological tree to include all the topological orderings of the task graph. Topological trees will serve as the basis for generating state-space search trees, which we discuss in Sec. 6.

4. Optimization by minimax criterion.

Suppose that a task is partitioned into suitable modules. Let $t_p^e(A)$ denote the total time spent for module execution, $t_p^c(A)$ the total time for interprocessor communication delay, and $t_p^i(A)$ the total idle time, all in a certain processor p according to a certain task assignement A. Let $t_p(A) = t_p^e(A)+t_p^c(A)+t_p^i(A)$ which, called the *processor turnaround time* of p, is the total time spent in processor p for task assingment A. This turnaround time is different for each distinct processor. Let $t(A) = \max_p t_p(A)$ which, called the *task turnaround time* of A, is the total time required to compute the whole task according to assignment A. The smaller $t(A)$ is, the better A is. Therefore, $t(A)$ may be considered as a cost function for task assignment. An optimal task assignment may then be defined as the one A_0 which minimizes the task turnaround time, i.e.,

$$t(A_0) = \min_A t(A) = \min_A \max_p t_p(A).$$

This is the so-called minimax criterion [1]; $t(A_0)$ will be called the *minimum task turnaround time.*

Note that in order to obtain the minimum cost function $t(A_0)$, each component of the function must be converted to a uniform unit that is rational and related to a realistic practical application.

5. Derivation of task turnaround time.

If M is a mapping defined by the task assignment A, then we use $a \to M(a)$ to denote the assignment of module a to processor $M(a)$ and $C_p(a \to M(a))$ to denote the execution time of module a on processor $M(a)$. Similarly, we use $(a, b) \to (M(a), M(b))$ to denote the assignment of module communication between a and b to the communication link between $M(a)$ and $M(b)$, and $C_c((a, b) \to (M(a), M(b)))$ to denote the communication time between a and b using the link $(M(a), M(b))$. If $M(a)$ is equal to $M(b)$, then the time $C_c((a, b) \to (M(a), M(b)))$ is defined to be zero.

The first factor which influences the values of the task turnaround time is obviously the mapping from the modules to the processors defined by the assignment. The assignments of a module to different processors will result in different execution time values, and so different task turnaround time values.

Next, since there exists a precedence relationship in the task modules, processor idleness may occur in the execution process of the task. Such idleness is found in the processors which are waiting to communicate with other processors for sending or receiving messages. Different orderings of module execution may result in different sequences of processor idleness, and so different task turnaround time values.

Additionally, each module m may have multiple immediate predecessors which will transmit messages to module m in a certain order before m is executed. The ordering of message transmission from the immediate predecessors to the module will also influence the final task turnaround time value.

All three factors above can be fully considered if we first obtain all the topological orderings of the modules according to the precedence relationship, and then consider all possible ways to assign the modules to the processors according to each topological module ordering. In the following we consider how to compute the task turnaround time value $t(A)$, given a fixed topological ordering and a fixed assignment A of the modules to the processors. The variation of topological orderings and module mappings will be handled in the next section by state-space search.

First, we have to compute the processor turnaround time $t_p(A)$ for each system processor p because $t(A) = \max_p t_p(A)$. Assume that in the beginning $t_p(A)$ values are set zero for all processors p, and that module m_1 is assigned to processor p_1 in assignment A. Then

$$t_{p_1}(A) = \text{time for processing module } m_1 \text{ at processor } p_1 = C_p(m_1 \to p_1).$$

Next, suppose that m_2 is assigned to processor p_2 in A. If m_2 has an immediate predecessor, which is just m_1, then m_2 cannot be executed on p_2 until p_2 receives the execution result of m_1 from p_1. In this case we have

$$\begin{aligned} t_{p_2}(A) &= \text{idle time for waiting messages from } m_1 \text{ at } p_1 \\ &\quad + \text{time for receiving messages from } m_1 \text{ at } p_1 \\ &\quad + \text{time for processing } m_2 \text{ at } p_2 \\ &= \text{old } t_{p_1}(A) + C_c((m_1, m_2) \to (p_1, p_2)) + C_p(m_2 \to p_2); \end{aligned}$$

$$\begin{aligned} t_{p_1}(A) &= \text{time for processing } m_1 \text{ at } p_1 \\ &\quad + \text{time for transmitting the result of } m_1 \text{ to } m_2 \text{ at } p_2 \\ &= \text{old } t_{p_1}(A) + C_c((m_1, m_2) \to (p_1, p_2)). \end{aligned}$$

Note that when p_2 begins to process m_2, p_1 is just free for another job. Also note that if $p_1 = p_2$, then the communication time $C_c((m_1, m_2) \to (p_1, p_2))$ is zero as defined previously and

$$t_{p_1}(A) = t_{p_2}(A) = C_p(m_1 \to p_1) + C_p(m_2 \to p_2).$$

If m_2 has no predecessor, then m_1 and m_2 can be processed in p_1 and p_2, respectively, in parallel. So we have

$$t_{p_1}(A) = C_p(m_1 \to p_1); \quad t_{p_2}(A) = C_p(m_2 \to p_2).$$

Similarly, if $p_1 = p_2$, then we have

$$t_{p_1}(A) = t_{p_2}(A) = C_p(m_1 \to p_1) + C_p(m_2 \to p_2).$$

A similar analysis can be performed for more general cases. The following algorithm is a summary which can be used to update processor turnaround time values when module m is additionally assigned to processor p in the process of developing a partial task assignment into a complete one A.

ALGORITHM 1. Processor turnaround time updating.

Step 1 If module m has no predecessor, then set $t_p(A) := t_p(A) + C_p(m \to p)$ and stop.

Step 2 Suppose that module m has H immediate predecessors $m_1, m_2, \ldots, m_H$, where m_i transmits messages to m before m_j if $i < j$ for all $1 \le i, j \le H$, and that m_h is assigned to p_h in A for all $1 \le h \le H$. Set $h := 1$.

Step 3 **While** $h \le H$ **do**
begin
set both $t_{p_h}(A)$ and $t_p(A)$ to be
$\max\{t_p(A), t_{p_h}(A)\} + C_c((m_h, m) \to (p_h, p))$,
and set $h := h + 1$.
end.

Step 4 Set $t_p(A) := t_p(A) + C_p(m \to p)$ and stop.

6. State-space search for optimal graph matching.

A. *Basic ideas.*

Briefly speaking, we use state-space search to consider all possible topological module orderings and module mappings. The result of the search is an optimal task assignment.

In order to maintain the precedence relationship, node expansion in the state-space is guided by traversing the topological tree which includes all the topological module orderings. Actually, we can consider the topological tree as a basic structure for generating state-space nodes to form a larger state-space search tree. The traversing starts from the root of the topological tree, which corresponds to the root (i.e., the start node) of the state-space search tree. To expand this tree root, all the sons of the root of the topological tree are collected. Each son m represents a first-executable module in the task. All system processors p are then attached to each son to form a set of candidate pairs (m, p). Each pair represents the possible assignment of module m to processor p. Next all the pairs are checked for their validity (i.e., checked if m is really executable on p based on processor and link characteristics, etc.). Valid pairs are finally kept in the state-space as the generated nodes. This completes the expansion of the root of the state-space search tree.

To expand a non-root tree node (m, p) in the state space, we first identify the corresponding node with label m in the topological tree. Similarly to the expansion of the root as described above, all the sons of m in the topological tree are then collected, and appropriate processors attached to form valid module-processor pairs as the generated nodes.

B. *State-space problem formulation.*

(1) *State description*:

Let two-tuple K = (MPPAIR, PTIME) denote the partially-developed mapping corresponding to a node $n = (k, z)$ in the state-space search tree, where MPPAIR = $\{(i, x) | (i, x)$ is the module-processor pair associated with a node in the path from the start node to node $n\}$ and PTIME = $\{t_p | t_p$ is the current processor turnaround time of system processor p updated at node $n\}$. K can be considered as the state description of the current search space at node n. But for simplicity, only the module-processor pair (k, z) is associated with each node n in the state-space search tree diagram (denoted as $k \rightarrow z$). Let $M_K = \{i | (i, x) \in \text{MPPAIR}\}$ which denotes the set of all the already-assigned modules in the partially-developed mapping. Note that M_K includes k which is the module just assigned at node n.

(2) *Initial state*:

The initial state is K = (MPPAIR, PTIME) with MPPAIR being empty and all t_p in PTIME being set zero.

(3) *Operators*:

An operator applied to a state-space node $n = (k, z)$ adds new module-processor pairs (j, y) to MPPAIR and updates PTIME in K. The procedure is as follows. First, find the node k' in the topological tree which corresponds to k, and form a set of candidate ordered module-processor pairs $D = \{(j, y) | j$ is a son of k' in the topological tree, y is a system processor$\}$. All j included in D are the next-executable modules according to the topological ordering. Next, check all candiate pairs one by one for validity. A candidate pair $(j, y) \in D$ is said to be *valid* if for each $(i, x) \in$ MPPAIR, it is true that if directed edge (i, j) exists in the task graph, then either there exists an edge connecting x and y, or $x = y$. This means that if module j has to communicate with an already-assigned module i which was assigned to processor x, then j has to be assigned either to a processor y which has a communication link to x, or to processor x itself. Let D' be the set of all valid pairs in D. Each pair (j, y) in D' forms a generated node of node n. The operator finally updates K for each generated node $n' = (j, y)$ as $K' =$ (MPPAIR', PTIME') where MPPAIR' = MPPAIR $\cup \{(j, y)\}$ and PTIME' = PTIME updated by Algorithm 1.

Note that the operator is defined in such a way that it always expands the partially-developed mapping by assigning each of the next-executable modules found in the topological tree to all the system processors. It is in this way that developing of the partial mapping can be kept in a proper order until all possible module assignments are exhaustively considered with the precedence relationship correctly maintained.

(4) *Goal state*:

Any state denoted by K with $M_K = V_T$ (the set of all task modules) is a goal state; that is, the search stops when all modules are completely assigned.

C. *Search of optimal task assignment.*

The formulation above just offers a search scheme for finding a mapping corresponding to a task assignment. Based on the search scheme, the A^* algorithm described in Nilsson [19] now can be used to find the optimal mapping. In the A^* algorithm, the cost evaluation function at node n in the state-space is usually defined as $f(n) = g(n) + h(n)$, where the value of $g(n)$ is the minimum path cost from the start node to node n and the value of the heuristic function $h(n)$ is a lower-bounded estimate of the value of $h^*(n)$ which is the minimum path cost from node n to a goal node.

For our case here, since each state-space node n corresponds to a partially-developed mapping as described by $K =$ (MPPAIR, PTIME), we can let $g(n)$ be, according to the minimax criterion, defined as

$$g(n) = \max_{t_p \in \text{PTIME}} t_p$$

which is the time required to process all the modules already assigned (i.e., to process all $i \in M_K$).

On the other hand, nonzero $h(n)$ values should be used for the search to be more efficient. The heuristic function $h(n)$ we use is defined as follows. Let processor p_K be the one with the maximum processor turnaround time found so far, i.e., let p_K be such that

$$t_p = \max_{t_p \in \text{PTIME}} t_p.$$

Let A_K be the set of all the modules assigned to p_K in the partially-developed mapping, and M'_K be the set of all the modules not assigned yet (i.e., M'_K and M_K so defined together form the set V_T of all task modules). Also, define R_K to be the set of all those modules in M'_K which have to communicate with the modules in A_K, i.e., $R_K = \{b | b \in M'_K, (a, b) \in E_T$ (the edge set of the task graph) with $a \in A_K\}$, and define L_K to be the set of all those modules in A_K which have communication with the modules in R_K, i.e., $L_K = \{a | a \in A_K, (a, b) \in E_T$ with $b \in R_K\}$. In fact, $L_K \subseteq A_K$ denotes part of the modules already assigned to p_K, whose unassigned immediate successors form the set $R_K \subseteq M'_K$. A diagram indicating all the relationships between all the module sets defined above is shown in Figure 3.

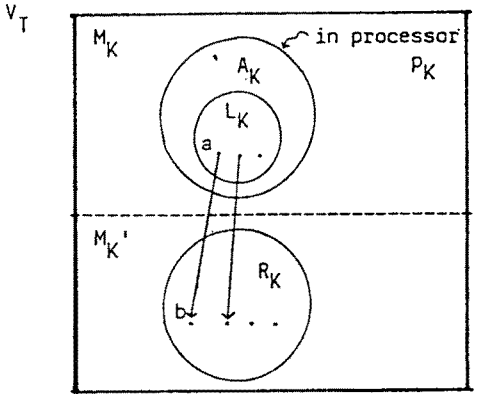


Fig. 3. A diagram showing the relationships between the module sets useful for extracting heuristic information.

Now each module b in R_K in further state-space search will have to be assigned *either* to processor p_K *or* to a processor p connected through a communication link to p_K in order that the communication from the predecessors of b in L_K to b can be accomplished. But these two types of assignment will result in two different amounts of partial processor turnaround time being increased at p_K. If b is assigned to p_K, then p_K will have to spend time *both* to receive

messages from each predecessor of b already assigned to a processor other than p_K *and* to process module b. If b is assigned to a processor p with a link to p_K, each predecessor of b already assigned to p_K will have to send messages to b. So p_K will have to spend time in communication with p. To keep $h(n)$ as a lower bound of $h^*(n)$ for the A^* algorithm to be optimum, it is necessary to find for each module $b \in R_K$ the smaller of the above-mentioned two different amounts of time increased at p_K. We can then let $h(n)$ be the sum of these amounts of time required for all the modules in R_K, as is calculated in Algorithm 2 below.

ALGORITHM 2. Computation of $h(n)$.

Step 1 Find p_K and all the sets shown in Figure 3.

Step 2 For each $b \in R_K$, find $S_K = \{m | (m, b) \in E_T, m \in (M_K - A_K)\}$ (which is the set of the predecessors of b already assigned to a processor other than p_K) and $S'_K = \{m | (m, b) \in E_T, m \in A_K\}$ (which is the set of the predecessors of b already assigned to p_K), and compute

$$t = C_p(b \to p_K) + \sum C_c((m, b) \to (p, p_K))$$

with summation over $m \in S_K$, $m \to p \neq p_K$, and

$$t' = \min_{p \in V_p} \sum_{m \in S'_K} C_c((m, b) \to (p_K, p)).$$

Set $H_b := \min(t, t')$.

Step 3 Set $h(n) := \sum_{b \in R_K} H_b$.

The following algorithm follows that in [1] and is used to find an optimal mapping between two graphs, a task graph and a processor graph.

ALGORITHM 3. Find the optimal task assignment.

Step 1 Put the initial node n on a list called OPEN, and set the evaluation function value $f(n) = 0$.

Step 2 Remove from OPEN the node n with the smallest f value and put it on a list called CLOSED.

Step 3 If n satisfies the goal state, then backtrack to the start node to find the corresponding graph matching as the desired task assignment and exit. Otherwise, continue.

Step 4 Expand node n, using operators applicable to n, and update the set PTIME by Algorithm 1 for each generated son n' of n to get $g(n')$. Also compute $h(n')$ according to Algorithm 2. Set $f(n') = g(n') + h(n')$ for each n'. Insert the nodes generated to the OPEN list.

Step 5 Go to Step 2.

In Step 4, if $h(n')$ is always set 0, then Algorithm 3 is a uniform-cost search [19]. Otherwise, it is an A^* algorithm.

7. Illustrative examples.

First, we give an example to illustrate the search of the optimal mapping from a task graph to a processor graph using Algorithm 3. The effectiveness of the proposed heuristic function to reduce the number of generated state-space nodes will also be shown. The task graph T and the processor graph P are shown in Figure 4 and Figure 5, respectively.

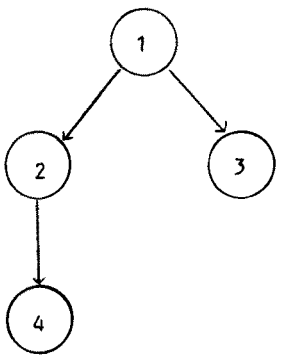


Fig. 4. Task graph T.

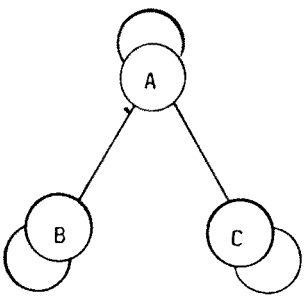


Fig. 5. Processor graph P.

The intermodule communication time and the module processing time are listed in Table 1 and Table 2, respectively.

Table 1. *Intermodule communication time.*

		module			
		1	2	3	4
module	1	0	10	20	0
	2	0	0	0	15
	3	0	0	0	0
	4	0	0	0	0

Table 2. *Module processing time.*

		processor		
		A	B	C
module	1	70	20	85
	2	10	60	50
	3	20	100	130
	4	120	90	35

We show in Figure 6 a topological module tree to specify all possible module expansion orderings.

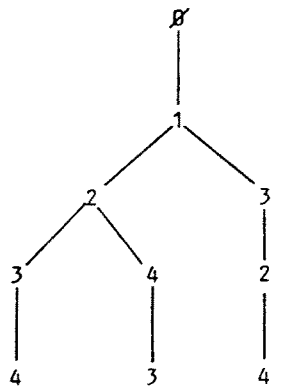


Fig. 6. Topological module tree.

Then, using Algorithm 3 as an ordered-search algorithm, we can find an optimal task assignment and compute the minimum task turnaround time. Figure 7, shows the resulting search tree. The optimal mapping as shown is $1 \rightarrow B$, $2 \rightarrow A$, $4 \rightarrow C$, $3 \rightarrow A$, and the minimum task turnaround time is 95.

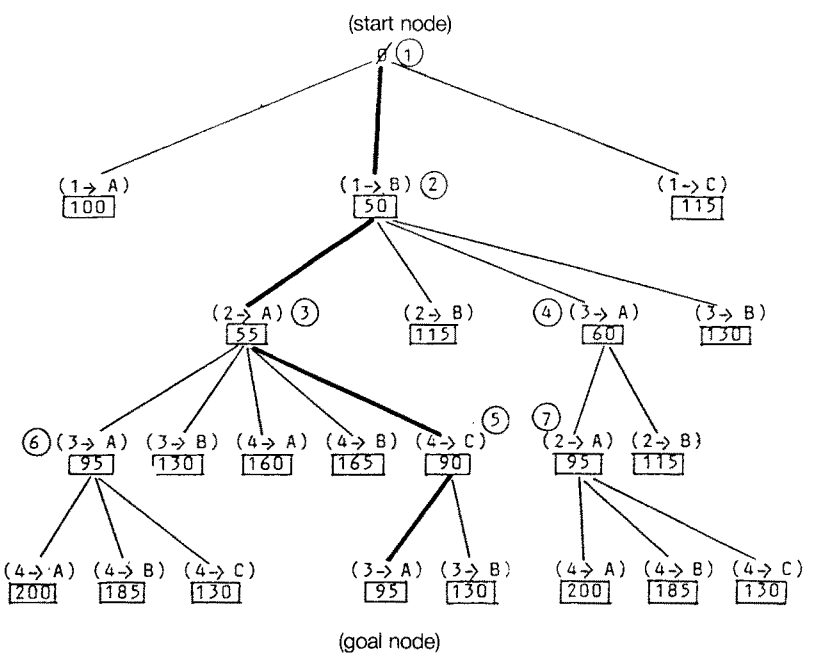


Fig. 7. State-space search tree of the illustrative example with $h(n) \neq 0$ (circled numbers indicate node expansion order, and squared numbers indicate $f(n)$ values).

As shown in Figure 7, totally only 7 state-space nodes are expanded and 23 nodes generated before the goal node is found. To prove that this result comes from the use of the heuristics proposed previously, we further reuse Algorithm 3 as a uniform-cost algorithm (setting $h(n)$ zero for all nodes) to find the optimal assignment again. Totally 12 state-space nodes are expanded in this case with 46 nodes being generated. Note that the following assignment also found in the search tree of Figure 7 is similar to the above optimal assignment: $1 \rightarrow B$, $2 \rightarrow A$, $3 \rightarrow A$, $4 \rightarrow C$. The only difference between them is the topological module ordering (or the order of module processing and communication). But the task turnaround time of this assignment is 130. This shows that the topological module ordering influences the task turnaround time value. Finally, we show in Table 3 the running process of each processor for the optimal task assignment.

Table 3. *Running process of the optimal task assignment.*

<table>
<tr><th>time</th><th>processor A</th><th>processor B</th><th>processor C</th></tr>
<tr><td>20</td><td>idle</td><td>run m_1</td><td rowspan="2">idle</td></tr>
<tr><td>10</td><td>receive massage from m_1 for m_2</td><td>transmit result of m_1 to m_2</td></tr>
<tr><td>10</td><td>run m_2</td><td rowspan="2">idle</td><td rowspan="2">receive message from m_2 for m_4</td></tr>
<tr><td>15</td><td>transmit result of m_2 to m_4</td></tr>
<tr><td>20</td><td>receive message from m_1 for m_3</td><td>transmit result of m_1 to m_3</td><td rowspan="2">run m_4</td></tr>
<tr><td>20</td><td>run m_3</td><td>idle</td></tr>
<tr><td colspan="3"></td><td>idle</td></tr>
</table>

More examples with different task graphs, processor graphs, intermodule communication times and module processing times have been tested to check the effectiveness of the heuristic function $h(n)$ used in Algorithm 3. The optimal solutions of these examples are listed in Table 4. It is observed that the heuristics we use in general cut the number of generated nodes down to about a half.

8. Conclusions.

As a nontrivial extension of Shen and Tsai [1], module precedence consideration has been successfully incorporated in this study into the state-space search of optimal task assignment. This is made possible by transforming

Table 4. *Results of some more examples.*

Example	Number of modules	Number of processors	# of nodes generated		Optimal assignment	cost
			h(n) = 0	h(n) ≠ 0		
1	4	3	163	60	2→C 1→B 3→C 4→A	310
2	6	3	427	199	1→B 2→A 6→C 4→A 5→A 3→A	378
3	6	4	325	233	1→A 3→C 4→D 2→B 5→A 6→A	682
4	7	4	1025	517	1→A 2→D 5→C 3→B 6→A 7→B 4→D	297

the precedence relationship into a topological tree which is used for module execution and message transmission ordering. The ordering of message transmission from the immediate predecessors of a module to the module itself is taken care of when processor turnaround time values are updated by Algorithm 1. Module execution ordering is taken care of when the topological tree is used as the basic structure for search tree expansion.

In addition, heuristics for speeding up the search are also proposed (computed by Algorithm 2). The heuristic information is collected from the predecessor and successor relationship created by module precedence. Experimental results show the effectiveness of the proposed heuristics. Note that speedup in solution search is useful for real-time applications.

REFERENCES

1. C. C. Shen and W. H. Tsai, *A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion*, IEEE Trans. Computers, Vol. C-34, No. 3, pp. 197–203, Mar. 1985.

2. K. Efe, *Heuristic models of task assignment scheduling in distributed systems*, Computer, Vol. 15, No. 6, pp. 50–56, June 1982.
3. W. W. Chu, L. J. Holloway, M. T. Lan and K. Efe, *Task allocation in distributed data processing*, Computer, Vol. 13, No. 11, pp. 57–69, Nov. 1980.
4. H. S. Stone and S. H. Bokhari, *Control of distributed processes*, Computer, Vol. 11, No. 7, pp. 97–106, July 1978.
5. H. S. Stone, *Multiprocessor scheduling with the aid of network flow algorithms*, IEEE Trans. Software Eng., Vol. SE-3, No. 1, pp. 85–93, Jan. 1977.
6. T. C. K. Chow and J. A. Abraham, *Load balancing in distributed systems*, IEEE Trans. Software Eng., Vol. SE-8, No. 4, pp. 401–412, July 1982.
7. S. H. Bokhari, *On the mapping problem*, IEEE Trans. Computers, Vol. C-30, No. 3, pp. 207–214, March 1981.
8. Y. C. Chow and W. Kohler, *Models for dynamic load balancing in a heterogeneous multiple processor system*, IEEE Trans. Computers, Vol. C-28, No. 5, pp. 354–361, May 1979.
9. Shyue B. Wu and Ming T. Liu, *Assignment of tasks and resources for distributed processing*, Proc. Distributed Processing (IEEE COMPCON), pp. 655–662, Washington D.C., Sept., 1980.
10. Danny Dolev and M. K. Warmuth, *Scheduling precedence graphs of bounded height*, Journal of Algorithms, Vol. 5, pp. 48–59, 1984.
11. P. R. Ma, E. Y. S. Lee, and M. Tsuchiya, *A task allocation model for distributed computing systems*, IEEE Trans. Computers, Vol. C-31, No. 1, pp. 41–47, Jan. 1982.
12. S. Kirkpatrick, *Optimization by simulated annealing*, Science, Vol. 220, No. 4598, pp. 671–680, May 1983.
13. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, IBM Research Report RC9355 (41093), April 1982.
14. S. M. Jacobs, L. V. Johnson, and O. Khedr, *A technique for systems architecture analysis and design applied to the satellite ground systems (SGS)*, Proc. 4th Int. Conf. Distributed Computing Systems, San Francisco, CA, 14–18, May 1984.
15. M. J. Chung, E. J. Toy, and A. A. Lobo, *A parallel computer based on cube connected cycles (an abstract)*, Proc. Army Research Office Workshop on Future Directions in Computer Architecture and Software, Charleston, SC, 5–7, May 1986.
16. M. J. Chung, et al., *A parallel computer based on cube connected cycles*, RPI CIE Report, Rensselaer Polytechnic Institutes Center for Integrated Electronics, Troy, NY, Mar. 1986.
17. W. H. Tsai, *Graph matching problems: a survey and a tutorial*, Proc. 1st Conf. Computer Algorithms, Hsinchu Taiwan 300, Republic of China, pp. 16.1–16.2, July 1982.
18. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science, Woodland and Hills, California, 1982.
19. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
20. E. M. Reingold, Jurg Nievergelt, and Narsingh Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.
21. R. C. Read and D. G. Corneil, *The graph isomorphism disease*, J. Graph Theory, Vol. 1, pp. 339–363, 1977.
22. L. Babai, P. Erdös, and S. M. Selkow, *Random graph isomorphism*, SIAM J. Computing, Vol. 9, No. 3, pp. 628–635, Aug. 1980.
23. W. H. Tsai and K. S. Fu, *Subgraph error-correcting isomorphisms for syntactic pattern recognition*, IEEE Trans. Syst., Man, Cybern., Vol. SMC-13, No. 1, pp. 48–62, Jan./Feb. 1983.
24. R. M. Haralick and J. S. Kartus, *Arrangements, homomorphisms, and discrete relaxation*, Proc. IEEE Conf. Pattern Recog. and Image Processing, Chicago, IL, U.S.A., May 1978.