A BRANCH-AND-BOUND-WITH-UNDERESTIMATES ALGORITHM FOR THE TASK ASSIGNMENT PROBLEM WITH PRECEDENCE CONSTRAINT

Gen-Huey Chen+ and **Jyh-Shiarn Yur**++

+*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China*
++*Matsushita Electric Institute of Technology (Taipei) Co., Ltd., Taipei, Taiwan, Republic of China*

ABSTRACT

We consider the problem of finding an optimal assignment of task modules with precedence relationship in a distributed computing system. The objective of task assignment is to minimize the task turnaround time. This problem is known to be NP-complete for more than three processors. To solve the problem, a well-known state space reduction technique, branch-and-bound-with-underestimates, is applied and two underestimate functions are defined. Through experiments, their effectiveness is shown by comparing the proposed algorithm with both Wang and Tsai's algorithm and A^* algorithm with $h(x)=0$.

C.R. Categories: D.4.1, I.2.8.
Index Terms: Task assignment, branch-and-bound-with-underestimates, distributed processing, state-space search, precedence relationship, minimax criterion, task turnaround time.

1. Introduction

The rapid progress of microprocessor technology has made the distributed computing systems economically attractive for many computer applications. In a distributed computing system, a task (program) may be distributed among processors to speedup the execution by taking advantage of system computation abilities and resources. However, the overall system performance is dependent on many factors; among them, the most crucial one is the assignment of task modules to processors. In general, a task can be suitably divided into a set of interdependent *task modules* (*modules*, for short) that can be executed on the processors of the distributed computing system. The cost of executing a module may vary from processor to processor. During task execution, some control messages and intermediate data are required to be transmitted among modules. Two communicating modules that are executed on different processors consume system's communication resources and thus incur a communication cost. Here cost values are defined in terms of a single unit, time. Hence, the total time, called the *task turnaround time*, required to finish the execution of the entire task is composed of *module execution time* (*MET*), *intermodule communication time* (*ICT*), and *processor idle time* (*PIT*).

Our attention for the task assignment is focused on finding an optimal assignment that minimizes the task turnaround time. To achieve this objective, we need to balance the computation loads of the processors and at the same time to minimize the intermodule communication overheads. The task assignment problem for more than three processors is known to be NP-complete (2). Solution methods already suggested for the problem can be roughly classified into four categories: graph-theoretic approaches (11), (15), (16), integer 0-1 programming approaches (5), (12), (13), (19), heuristic approaches (6), (8), and simulated annealing approaches (18).

Wang and Tsai (19) formulated the task assignment problem as a graph matching problem and then presented an A^* algorithm (10) to search for an optimal assignment. In this paper we propose a new algorithm for the task assignment problem that behaves very well in that case. In the proposed algorithm, a well-known state space reduction technique, *branch-and-bound-with-underestimates* (BBU), is applied and two underestimate functions, f_{METU} and f_{ATU}, are defined. To show the effectiveness of the proposed algorithm, the execution time are measured for the proposed algorithm, Wang and Tsai's algorithm, and A^* algorithm with $h(x)=0$ through experiments. Parameters considered in the experiments include the number of modules, the shapes of task graphs, and the ratio of average intermodule communication time to average module execution time.

The remainder of this paper is organized as follows. In Section 2, system assumptions are stated and the task assignment problem is formulated. In Section 3, two underestimate functions, f_{METU} and f_{ATU}, are defined and a BBU algorithm is proposed. Experimental results are shown in Section 4. Finally, concluding remarks are given in Section 5.

2. Assumptions and Problem Statement

2.1 Assumptions

The task assignment problem we consider in this paper has the same assumptions as Wang and Tsai have made in ref. 19.

(1) The processors in the distributed computing system are heterogeneous.
(2) All processors can communicate with each other through the communication subnetwork.
(3) All communication links are symmetric. That is, transmission on both directions of a communication link takes the same time. But, transmission on different communication links may take different times.
(4) Synchronization between two communicating processors is necessary before starting message transmission (i.e., message transmission and module execution can not be overlapped). This means that the two communicating processors spend the same amount of communication time, but one of them may incur additional idle time.
(5) There exists a precedence relationship among modules. It specifies the feasible execution sequences of modules. No cyclic precedence relationship is allowed among modules.

2.2 Problem Statement

There are m modules $M_1, M_2, ..., M_m$ contained in a given

CH2878-7/90/0000/0494$01.00 © 1990 IEEE

task. The task can be conveniently represented by an acyclic directed graph, called the *task graph*, as follows. Each module of the task is uniquely represented by a vertex of the task graph and there is an arc from M_i to M_j if and only if message transmission is needed from M_i to M_j (i.e., M_i precedes M_j) during the task execution.

If there exists a path from M_i to M_j in the task graph, then M_i is called a *predecessor* of M_j, and M_j is called a *successor* of M_i. If there exists an arc from M_i to M_j, then M_i is called an *immediate predecessor* of M_j, and M_j is called an *immediate successor* of M_i. A module without any successor (predecessor) is called a *sink module(source module)*. A module is not allowed to start execution until all its immediate predecessors have finished execution.

There are n processors $P_1, P_2, ..., P_n$ in the distributed computing system. Let $MET(i, j)$ denote the module execution time required for executing M_i on P_j and $ICT(a, b, i, j)$ denote the intermodule communication time required for the pair of modules M_a and M_b when they are assigned to P_i and P_j respectively. $ICT(a, b, i, j) = 0$ if $i = j$.

Let $PT(i)$ denote the processor turnaround time of P_i, which is the total time consumed on P_i. The maximal processor turnaround time, $\max_{i=1,...,n} \{PT(i)\}$, is the task turnaround time. The task assignment problem is to find a mapping from the task graph to the distributed computing system, subject to the precedence constraint, which minimizes the task turnaround time. Since the task turnaround time can be viewed as the latest finish time of all sink modules, minimizing the task turnaround time is equivalent to minimizing the maximal finish time of all sink modules.

Suppose that an unassigned module M_x has k immediate predecessors $M_{m_1}, M_{m_2}, ..., M_{m_k}$, if we decide to assign module M_x to processor P_y, then the processor turnaround times of those processors where M_x and its predecessors are resident should be updated. The detailed procedure for updating processor turnaround times can be found in ref. 20.

3. State Space Search Reduction

In this section, a branch-and-bound-with-underestimates (BBU) algorithm is presented to find an optimal solution for the task assignment problem. The state space graph of a BBU algorithm is a search tree whose nodes each, except the root node, corresponds to an assignment of a module to a processor. Associated with each node x in the search tree is a partial assignment A_x that consists of all the module-to-processor assignments of the nodes along the path from the root to x. Associated with each node x is also an underestimation $f(x) = g(x) + h(x)$ of the minimal task turnaround time caused by the complete assignments that include A_x as a part. The value $g(x)$ is the maximal processor turnaround time caused by A_x and $h(x)$ is an underestimation of the minimal processor turnaround time that will be incurred from node x to a goal node. A *goal node* is a node that represents a complete assignment. The accuracy of $h(x)$ greatly affects the efficiency of a BBU algorithm. Moreover, an upper bound cost (UC) is along with a BBU algorithm and it represents an upper bound on the minimal task turnaround time.

A list, called *unexpanded list*, is necessary for a BBU algorithm to store all unexpanded search nodes from which to find an optimal assignment is still possible. Initially, the unexpanded list is empty. The BBU algorithm begins with placing the root node into the unexpanded list. The root node corresponds to the null state (no module assigned). During the state space search, a search node x with minimal underestimation $f(x)$ is always selected from the unexpanded list to be expanded next. Let those unassigned modules whose predecessors have all been assigned be referred to as *ready modules*. If x is not a goal node, n possible assignments: M_i to P_j , $j = 1, ..., n$, for each ready module M_i are checked for their feasibilities and a child node is generated for each of the feasible assignments. Then, for each generated child node y, the underestimation $f(y)$ is computed. If $f(y) < UC$, node y is inserted into the unexpanded list. Otherwise, node y is fathomed. If the selected node x is a goal node, the algorithm terminates.

In the rest of this section, we first briefly review Wang and Tsai's algorithm [19] and then introduce two underestimate functions f_{METU} and f_{ATU}.

3.1 A Brief Review of Wang and Tsai's Algorithm

The essence of Wang and Tsai's algorithm [19] is to underestimate the minimal task turnaround time from the viewpoint of bottleneck processor.

For a partial assignment A_x, let us define the following notations:

- P_b : the bottleneck processor;
- L_i : the set of modules assigned to processor P_i;
- Q : the union of L_i's, i.e., the set of all assigned modules;
- Q' : the set of all unassigned modules;
- S : the set of modules in Q' that communicate with modules in L_b.

Wang and Tsai's algorithm computes $h(x)$ as the summation of H_q for all M_q in S, where

$$H_q = min\{t, t'\},$$

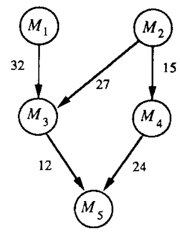
$$t = MET(q, b) + \sum_{\substack{r \in Q-L_b \text{ and} \\ TSK(r,q)=1}} ICT(r, q, A_x(r), b), \text{ and}$$

$$t' = \min_{p=1,...,n} \{ \sum_{\substack{r \in L_b \text{ and} \\ TSK(r,q)=1}} ICT(r, q, b, p)\}.$$

In essence, Wang and Tsai's algorithm computes $h(x)$ from the viewpoint of processors, which is the main reason of a poor underestimation as the intermodule communication time is relatively small (compared with the module execution time).

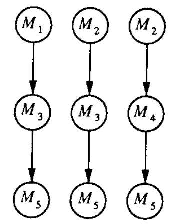
3.2 Minimal Execution Time Underestimate (METU)

Given a task graph, the task starts execution from source modules and terminates after all sink modules are finished. A directed path from a module M_i to a sink module M_j is called an *execution path*. Moreover, if M_i is a source module, then it is called a *complete execution path*. The execution time of an execution path from M_i to M_j is defined to be the time length from the time when M_i starts execution to the time when M_j finishes execution. The execution time of an execution path contains the module execution time, the intermodule communication time, and the processor idle time. With respect to a mapping from the task graph to the set of processors, we define the *critical complete execution paths* as those complete execution paths whose execution times are equal to the task turnaround time. In Figure 1, an example is shown where the



(a) A task graph and the intermodule communication times.

	p_1	p_2	p_3
M_1	1	69	76
M_2	16	89	92
M_3	71	88	84
M_4	86	98	24
M_5	63	16	38



(b) Module execution times (c) Three complete execution paths

Figure 1. An illustrative example.

given task graph contains three complete execution paths: (M_1, M_3, M_5), (M_2, M_3, M_5), and (M_2, M_4, M_5). Also, note that uniform intermodule communication times are assumed in Figure 1. That is, for two communicating modules M_a and M_b, $ICT(a, b, i, j)$'s are the same for any $i \neq j$.

Based on the concept of execution paths, two underestimate functions, f_{METU} and f_{ATU}, are therefore proposed.

For an arbitrary execution path (M_{i1}, M_{i2}, ..., M_{ik}) extended from M_{i1}, the summation $\sum_{l=1}^{k} \min_{p=1,\ldots,n} \{ MET(i_l, p) \}$ is an underestimation of the execution time for the execution path. For each module M_i, we define $MAXET(i)$ to be the maximum of the underestimated execution times for all the execution paths that are extended from the immediate successors of M_i. Clearly, if M_i is a sink module, $MAXET(i) = 0$. Otherwise, $MAXET(i)$ is computed recursively as

$$\max_{M_j \text{ is an immediate successor of } M_i} \{ MAXET(j) + \min_{p=1,\ldots,n} \{ MET(j, p) \} \}.$$

All the values $MAXET(i)$'s are determined prior to the execution of the BBU algorithm. In Table I, we show the values of $MAXET(i)$'s for the example of Figure 1.

Let us consider a partial assignment A_x that is associated with a search node x during the execution of the BBU algorithm. With respect to A_x, also denote the set of all assigned modules by Q and the set of all unassigned modules by Q'. Since the value $MAXET(i)$ is an underestimation of the time required to finish the execution of all successors of M_i, we can define an underestimate function f'_{METU} as follows:

$$f'_{METU}(x) = \max_{\substack{M_i \text{ is in } Q \text{ and all} \\ \text{immediate successors of} \\ M_i \text{ are in } Q'}} \{ PT(A_x(i)) + MAXET(i) \}.$$

i	1	2	3	4	5
$MAXET(i)$	87	87	16	16	0

Table I. The values of $MAXET(i)$'s for the example of Figure

In the above formula, $PT(A_x(i))$ is the current processor turnaround time of the processor where M_i is resident. It also represents the time when the execution of M_i and all its predecessors is finished. The computation of $f'_{METU}(x)$ is to underestimate the task turnaround time with respect to the partial assignment A_x by underestimating the time required to finish the execution of all successors of M_i as $MAXET(i)$. Note that since $MAXET(i)$ is defined for all immediate successors of M_i, they must be unassigned in the computation of $f'_{METU}(x)$.

The computation of $f'_{METU}(x)$ ignores the processor synchronization and the intermodule communication time caused by M_i and its immediate successors. To obtain a more accurate estimation of the task turnaround time, we have to take these two factors into consideration. Hence, the assignment of the immediate successors of M_i should be considered. The resulting underestimate function is $f_{METU}(x)$, defined as follows:

$$f_{METU}(x) = \max_{M_i \text{ is in } Q} \{ \max_{\substack{M_j \text{ is an immediate} \\ \text{successor of } M_i \text{ and} \\ M_j \in Q'}} \{ \min_{p=1,\ldots,n} \{ \max\{ PT(A_x(i), PT(p) \} + ICT(i, j, A_x(i), p) + MET(j, p) + MAXET(j) \} \} \}.$$

In the above formula, the term $\max\{ PT(A_x(i), PT(p) \}$ indicates the synchronization between the two communicating processors where M_i and M_j are assigned respectively. The value $\min_{p=1,\ldots,n} \{ \max\{ PT(A_x(i), PT(p) \} + ICT(i,j, A_x(i), p) + MET(j, p) + MAXET(j) \}$ is an underestimation of the time required to finish the execution of all predecessors of M_i, M_i, M_j, and all successors of M_j. The computation of $f_{METU}(x)$ is performed for each M_i in Q and each immediate successor M_j of M_i and then takes the maximum as an underestimation of the task turnaround time with respect to the partial assignment A_x. If M_i is a sink module, the value of the term $\max\{ \min\{ \max\{ \ldots \} + \ldots \}\}$ is computed as $PT(A_x(i))$.

3.3 Assignment Tree Underestimate (ATU)

The underestimate function f_{METU} does not consider the intermodule communication time that will be spent along an execution path. We take this factor into consideration in the underestimate function f_{ATU}. In essence, f_{ATU} determines how to assign the modules along a complete execution path such that the sum of module execution time and intermodule communication time is minimized. Thus, finding an optimal assignment of modules along each complete execution path forms the central part of the f_{ATU} function.

Before defining the f_{ATU} function, we describe the construction of *execution trees* from a task graph. There are the same number of execution trees as sink modules. The execution trees are rooted at sink modules and grow upward. Each node of the execution trees represents (probably not uniquely) a module.

Module M_i is an immediate predecessor of module M_j in the task graph if and only if a corresponding node of M_i is a child node of a corresponding node of M_j in execution trees. Thus, each path from a leaf node to a root node in the execution trees forms a complete execution path. The execution tree for the example of Figure 1 is shown in Figure 2.

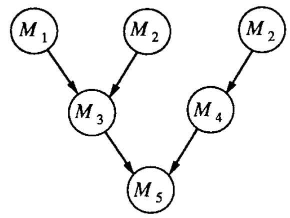


Figure 2. The exection tree for the task graph of Figure 1.

Based on the execution trees, we can build *assignment trees*. Each assignment tree is built from an execution tree by considering the assignment of the corresponding modules of the nodes in the execution tree. Each assignment tree is almost the same as the *assignment graph* that has been described in [2]. Each node of the assignment trees considers the n possible assignments of its corresponding module (each node of the assignment trees corresponds to a layer of the assignment graph). Each edge in the execution trees is replaced by $n \times n$ links in the assignment trees. These links represent all possible assignments of two communicating modules. In Figure 3, a part

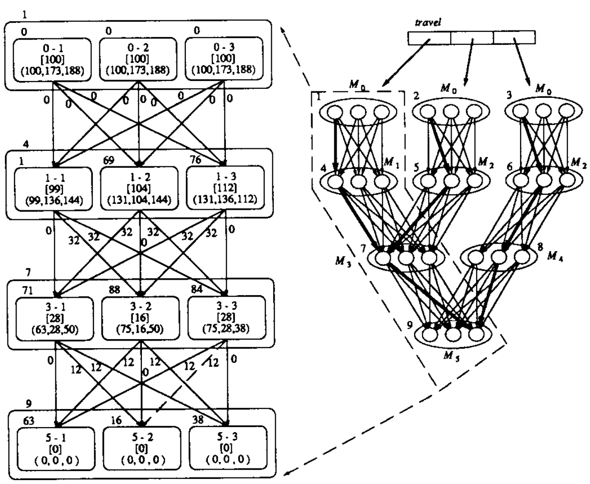


Figure 3. Part of the assignment tree built from Figure 2.

(corresponding to the complete execution path (1, 3, 5)) of the assignment tree built from Figure 2 is shown, where the notation "i - j" represents "assigning module M_i to processor P_j". For example, the dashed line connecting node 7 and node 9 means that M_3 and M_5 are assigned to P_3 and P_2 respectively.

Associated with each node in the assignment trees are some variables which are necessary in defining the underestimate function f_{ATU}. For the convenience of the description, we collect these variables in a *C*-type data structure as follows.

```
typedef struct node
{
    int          module;
    int          no_child;
    unsigned     exe_time[NO_PROC][NO_PROC];
    unsigned     min_exe_time[NO_PROC];
    struct node  *parent;
} NODE;
```

The identifier *NO_PROC* is a constant denoting the number of processors in the distributed computing system. The identifier *module* is a variable denoting the module represented by the node. The module M_0 is considered a *dummy module* and a node representing a dummy module is considered a *dummy node*, e.g., node 1 in Figure 3. The dummy node is like the source node of the assignment graph [2] and acts as the head of a complete execution path. The identifier *no_child* is a variable giving the number of child nodes (equal to the number of immediate predecessors of the associated module). The identifer *parent* is a pointer to the parent node. A node representing a sink module has its *parent* equal to NULL. The identifiers *exe_time* and *min_exe_time* will be explained later.

From Figure 3, it is seen that the assignment trees consider all possible assignments of modules along each complete execution path. Therefore, a specific assignment of modules along a complete execution path corresponds to a path from a dummy node to a root node in the assignment trees. Like the assignment graph, the links of the assignment trees are weighted with intermodule communication times and the nodes of the assignment trees are weighted with module execution times. All links incident to a dummy node have their weights equal to zero. Unlike the execution time of an execution path in the task graph, let us define the execution time of a path from a node to a root node in the assignment trees as the sum of the module execution times and the intermodule communication times along that path, excluding the module execution time of the starting node. For example, in Figure 3, the execution time of the path (0 - 3, 1 - 3, 3 - 3, 5 - 2) is $0 + MET(1, 3) + 0 + MET(3, 3) + ICT(3, 5, 3, 2) + MET(5, 2) = 188$. Note that the intermodule communication time of two communicating modules is 0 if they are assigned to the same processor.

Consequently, determining an optimal assignment of modules along a complete execution path which minimizes the sum of the module execution times and the intermodule communication times is equivalent to determining a shortest path from a dummy node to a root node in the assignment trees, which can be done by the aid of the values *min_exe_time*[i]'s and *exe_time*[i][j]'s that are stored in nodes of the assignment trees.

For each node in Figure 3, the values in parentheses, represented by variables *exe_time*[i][j], denote the execution time of the shortest path from the node to the root node if its associated module and the module associated with its parent node are assigned to P_{i+1} and P_{j+1} respectively (note that the array index of C language starts from 0). And the values in square brackets, represented by variables *min_exe_time*[i], denote the execution time of the shortest path from the node to the root node if its associated module is assigned to P_{i+1}. Clearly, $min\_exe\_time[i] = \min_{j=0,\ldots,n-1} \{ exe\_time[i][j] \}$. For example, node 4 in Figure 3 considers the assignment of module M_1 along the complete execution path (1, 3, 5). The value 136 in the left parenthesis is the content of *exe_time*[0][1] and it represents the execution time of the shortest path from node 4 to the root node if M_1 and its immediate successor M_3 are assigned to P_1 and P_2 respectively. The value 99 in the left bracket is the content of *min_exe_time*[0] and it represents the execution time of the shortest path from node 4 to the root node if M_1 is assigned to P_1.

The assignment trees are established before the BBU algorithm starts execution. By applying Bokhari's shortest tree algorithm [(2)], the values *min_exe_time[i]*'s and *exe_time[i][j]*'s can be computed. These values can be used to find a shortest path from an arbitrary node to a root node in the assignment trees (equivalent to determining an optimal assignment of modules along an execution path), which is the most essential step in computing $f_{ATU}(x)$.

Since the assignment trees are obtained from the execution trees, they also retain the precedence relationship among modules. Let us consider a complete execution path in the task graph. Assigning modules along the complete execution path can be regarded as traversing a path from a dummy node to a root node in the assignment trees. A complete (partial) assignment along the complete execution path corresponds to a travelling tour that contains the entirety (a part) of the corresponding path in the assignment trees. Moreover, a complete assignment can be regarded as a tree embedded in the assignment tree. For example, the bold lines in Figure 3 represent the assignment of M_1 to P_1, M_2 to P_2, M_3 to P_1, M_4 to P_2, and M_5 to P_3.

Since any node x in the search tree represent a partial assignment A_x, we can associate an array of pointers, named *travel*, with the node x to represent the travelling tours that correspond to A_x. In the proposed BBU algorithm, each pointer in *travel* always points to the frontier of a travelling tour, that is, the node (of the assignment trees) whose associated module was assigned last along a dummy node to root node path. For example, let us consider the example of Figure 1. If three modules: M_1, M_2, and M_4 have been assigned in the partial assignment A_x, then the pointers in *travel* of node x must point to nodes 4, 5, 8 respectively in the assignment tree.

At the beginning of the BBU algorithm, the pointers in *travel* of the root node point to the dummy nodes of the assignment trees since all modules are unassigned. During the execution of the BBU algorithm, whenever a search node x corresponding to, for example, the assignment of module M_a to processor P_b, is generated, the array *travel* of node x is constructed as follows. A pointer in *travel* is moved down to the next node (in the assignment trees) toward the root node if the module associated with the next node is M_a. If multiple pointers point to the same node, only one of them is kept. For example, let us consider the example of Figure 1 again. Suppose three modules: M_1, M_2, and M_4 have been assigned in the partial assignment A_x and the pointers in *travel* of node x point to nodes 4, 5, 8 respectively in Figure 3. If a node y that corresponds to the assignment of M_3 is generated as a child node of x during the execution of the BBU algorithm, then the array *travel* of node y is constructed as follows. Those two pointers to nodes 4 and 5 respectively are moved down to node 7 since the module associated with node 7 is M_3. Further, since they both point to the same node after movement, only one of them is kept. The pointer to node 8 remains unchanged.

A more detailed description about constructing the array *travel* for a newly generated search node x is shown in *Algorithm 2*.

Algorithm 2: /* Construct the array *travel* for a newly generated search node x. Assume that the node x corresponds to the assignment of module M_a to processor P_b. The variable t saves the number of pointers in *travel*. The array *no_pred* is a global variable and *no_pred[i]* denotes the number of immediate predecessors of module M_i. */

```
for (i = 1, j = 0; i <= t ; i ++)
{
    next = travel[i]->parent ;
    if (next != NULL && next ->module == a)
    {
        next ->no_child - - ;
        /* Are there multiple pointers to node x ? */
        if (next ->no_child >= 1) continue ;
        travel[i] = next ;
        /* Restore the value of no_child */
        next->no_child = no_pred[next->module];
    }
    /* Pack the travel pointers */
    travel[++j] = travel[i] ;
}
t = j ;
```

Based on the above discussion, we can define an underestimate function $f'_{ATU}(x)$ for a partial assignment A_x that is represented by a search node x.

$$f'_{ATU}(x) = \max_{i=1,\dots,t} \{ PT(A_x(travel[i]\text{->}module)) + ravel[i]\text{->}min\_exe\_time [A_x(travel[i]\text{->}module) - 1] \}$$

In the above formula, the value t denotes the number of pointers in *travel*, and decreasing the index of *min_exe_time* by 1 is due to the array index of C language starting from 0. If *travel[i]->module* is a dummy module, then $PT(A_x(travel[i]\text{->}module))$ is set 0 and $A_x(travel[i]\text{->}module)$ can be any of 1, 2, ..., n. If *travel[i]->module* is not a dummy module, say M_k, then $PT(A_x(k))$ is the time when M_k and its immediate successors can start message transmission (by the time the execution of M_k and all its predecessors is finished). The value $travel[i]\text{->}min\_exe\_time[A_x(k) - 1]$ is taken as an underestimation of the time required to finish the execution of all successors of M_k along the path from the node pointed by *travel[i]* to the root node. The computation of $f'_{ATU}(x)$ is to underestimate the task turnaround time with respect to the partial assignment A_x by taking $travel[i]\text{->}min\_exe\_time[A_x(k) - 1]$ as an underestimation of the execution time of the path from the node pointed by *travel[i]* to the root node. For example, let us consider Figure 3 again. If only module M_3 and all its predecessors have been assigned, then there is a pointer, say *travel[i]*, to node 7. Now, $PT(A_x(travel[i]\text{->}module)) = PT(A_x(3))$ is the time when the execution of M_3 and all its predecessors is finished and $travel[i]\text{->}min\_exe\_time[A_x(3) - 1]$ is an underestimation of the time required to finish the execution of M_5. Thus, $PT(A_x(3)) + travel[i]\text{->}min\_exe\_time[A_x(3) - 1]$ is an underestimation of the time required to finish the execution of all predecessors of M_3, M_3, and M_5.

Note that the computation of $f'_{ATU}(x)$ ignores the processor synchronization and the intermodule communication time caused by the module *travel[i]->module* and its immediate successor *travel[i]->parent->module*. To make a more accurate estimation of the task turnaround time, we have to take these two factors into consideration. Hence, the assignment of the module *travel[i]->parent->module* should be considered. The resulting underestimate function is $f_{METU}(x)$, defined as follows:

$$f_{ATU}(x) = \max_{i=1,\dots,t} \{ \min_{p=1,\dots,n} \{ \max\{PT(A_x(travel[i]\text{->}module)), PT(p)\} + travel[i]\text{->}exe\_time [A_x(travel[i]\text{->}module) - 1][p - 1] \} \}$$

In the above formula, P_p is the processor where the module *travel*[*i*]->*parent*->*module* is attempted to be assigned. The term $max\{PT(A_x(travel[i]\text{->}module)), PT(p)\}$ indicates the synchronization between the two communicating processors where the module *travel*[*i*]->*module* and the module *travel*[*i*]->*parent*->*module* are resident. If *travel*[*i*]->*module* is a dummy module, then $PT(A_x(travel[i]\text{->}module))$ is set 0 and $A_x(travel[i]\text{->}module)$ can be any of 1, 2, ..., *n*. If *travel*[*i*]->*module* is a sink module, then no immediate successor of the module *travel*[*i*]->*module* exists and $PT(p)$ is set 0. The value *travel*[*i*]->*exe_time*[A_x(*travel*[*i*]->*module*) - 1)][*p* - 1] is taken as an underestimation of the time required to finish the execution of all successors of the module *travel*[*i*]->*module* along the path from the node pointed by *travel*[*i*] to the root node.

3.4 An Initial Solution

For a BBU algorithm, a good enough initial solution can save much computation and memory by fathoming nodes at the beginning of the state space search. For the task assignment problem we consider, there is a trivial solution, i.e., assigning all modules to the same processor. In fact, our experiments show that the trivial solution is almost an optimal solution when the intermodule communication time is much greater than the module execution time. On the other hand, the trivial solution is bad when the module execution time is greater than the intermodule communication time. For the latter case, an algorithm using the concept of f_{ATU} is applied to find a good enough initial solution. A similar algorithm using the concept of f_{METU} can also be derived easily. For the sake of the limit of space, we do not describe them here, but the detailed desciption can be found in ref. 20.

The proposed BBU algorithm using the underestimate function f_{ATU} will select the better one of the trivial solution and the solution obtained from our proposed algotiyhm as an initial solution and set the task turnaround time of the initial solution as the initial value of *UC*.

4. Experimental Results

In this section we compare the performance of the proposed algorithm with that of Wang and Tsai's algorithm and A^* algorithm with $h(x)$=0. The execution time of 200 tested instances is measured for performance evaluation. In general, the performance of the proposed algorithm is affected by many factors. Among them, four factors: number of processors, number of modules, the ratio of average intermodule communication time to average module execution time (called *C:P ratio*), and the shapes of task graphs are considered in the experiments. The shapes of task graphs, which was neglected in [19], reflect the precedence relationship among all modules, and they will affect the accuracy of the estimation made by an underestimate function. In order to investigate the effect of the shapes of task graphs on the performance of the proposed algorithm, instead of generating tested task graphs randomly, we consider six types of task graphs in the experiments: linear, convergence, X_type, tree, ladder, and mesh (see Figure 4).

A task graph is of *linear type* if it forms a linear chain. A task whose execution consists of several serial phases has a linear-typed task graph. A task graph is of *convergence type* if it is a tree with the root at the bottom. A task has a convergence-typed task graph if its modules can be partitioned into several disjoint subsets $S_1, S_2, ..., S_r$ with $|S_1| \geq |S_2| \geq ... \geq |S_r|$ such that the precedence relationship only exists between S_i and S_{i+1}, $1 \leq i \leq r-1$. The *tree-typed task graph* is similar to the convergence-typed task graph except that the root of the tree is at

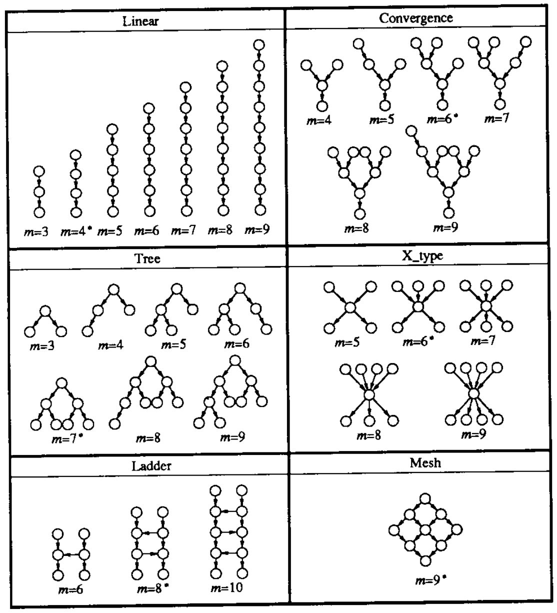


Figure 4. Six types of task graphs.

execution paths.

A task graph with a look similar to one of these six types of task graphs is expected to have similar experimental results.

In our experiments Wang and Tsai's algorithm and $A*$ algorithm with $h(x)$=0 are provided with the trivial initial solution. The intermodule communication times are assumed uniform. Module execution times and intermodule communication times are generated randomly according to the given *C:P* ratio. The *C:P* ratios considered in our experiments are from 0.01 to 100 (or from -2 to 2 with logarithmic values based 10).

In the rest of this section, experimental results about execution time are shown. The experiments are carried out for different numbers of modules, different *C:P* ratios, and different types of task graphs. For each tested case, 200 randomly generated instances are run, and the total execution times is measured. The experimental results versus the number of modules is given by taking the average with $\log_{10}(C:P)$ ranged from -2 to 2.

4.1 Execution Time

General speaking, number of search nodes and maximal queue length are two important criteria for evaluating the performances of a BBU algorithm since they are machine independent and program independent. However, they do not take the computational complexity of the underestimate function into consideration. A heavy computation of the underestimate on each search node may offset the gains from reducing the search space. Hence, execution time is the most reliable measure to prove the effectiveness of a BBU algorithm. In our experiments all the tested algorithms are programmed in *C* language to measure their execution times. The experimental results are shown in Figures 5-6.

Figure 5 shows the execution time of 200 randomly generated instances as a function of $\log_{10}(C:P)$ for the proposed algorithm, Wang and Tsai's algorithm, and $A*$ algorithm with $h(x)$=0. The curves labeled with "ATU" and "METU" represent the results of the proposed algorithm using the underestimate functions f_{ATU} and f_{METU} respectively. The curves labeled with "W&T" and "h(x)=0" represent the results of Wang and Tsai's algorithm and A^* algorithm with $h(x)=0$ respectively.

It can be observed from Figure 5(a) that the proposed algorithm performs better than the other two algorithms

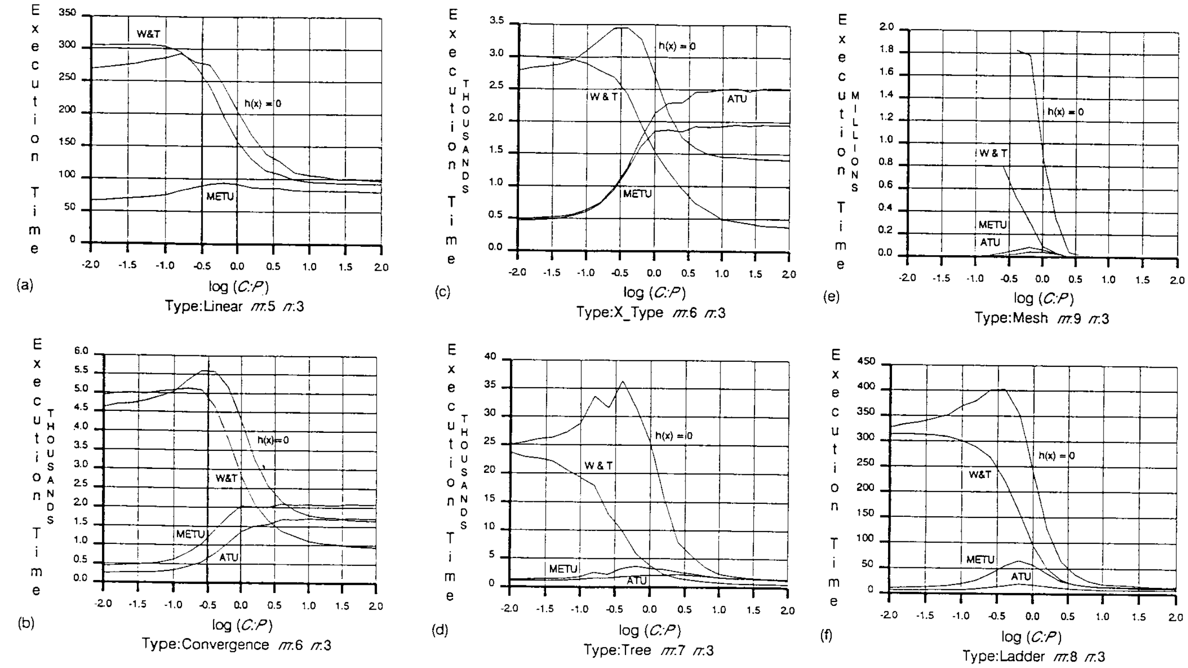


Figure 5. Execution time of 200 tested instances versus $\log_{10}(C:P)$.

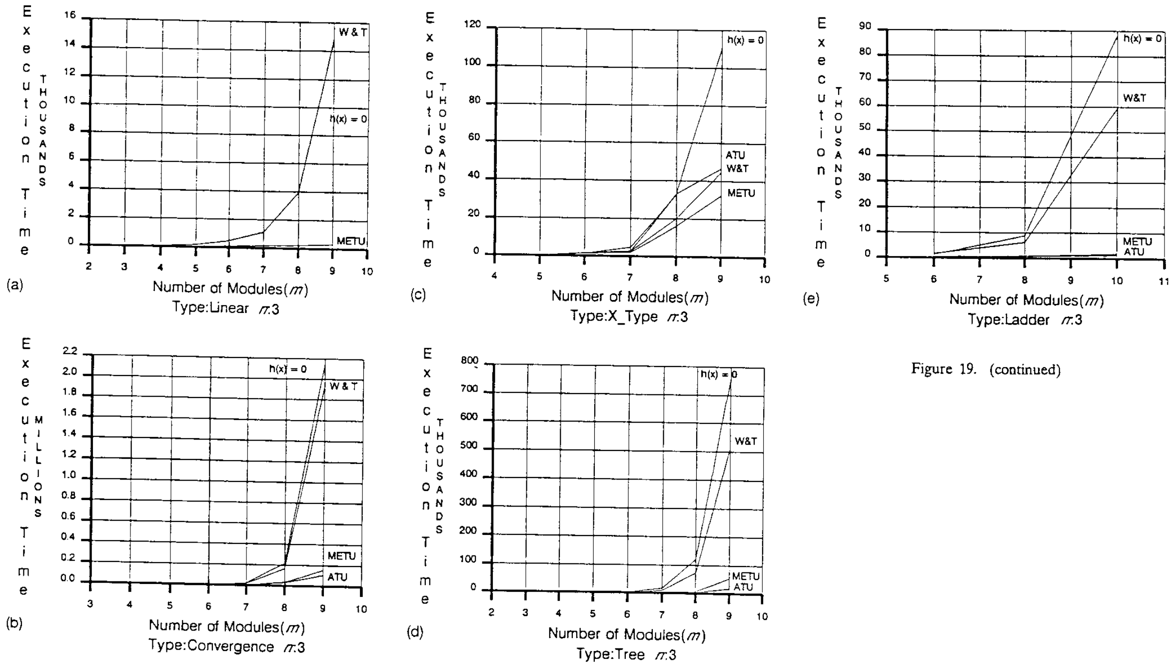


Figure 19. (continued)

Figure 6. Execution time of 200 tested instances versus number of modules.

everywhere for the linear-typed task graph of m=5. Wang and Tsai's algorithm has a bad performance, even worse than the A^* algorithm with $h(x)$=0, as $\log_{10}(C:P) < -0.8$. This is due to the potential weakness of their algorithm in estimating the minimal task turnaround time for a "slim" and "long" task graph. Also note that the curve labeled with "W&T" drops drastically as the $C:P$ ratio > -0.8, which is mainly due to the high accuracy of the trivial initial solution as the $C:P$ ratio is high, not Wang and Tsai's algorithm itself.

Figure 5(b) shows experimental results for the convergence-typed task graph of m=6. The curve labeled with "W&T" is higher than the curve labeled with "h(x)=0" as $\log_{10}(C:P) < -1$. The proposed algorithm performs better than the other two algorithms as $\log_{10}(C:P) < 0.2$. As $\log_{10}(C:P) > 0.5$, Wang and Tsai's algorithm has the best performance.

Figure 5(c) shows experimental results for the X_typed task graph of m=6. The proposed algorithm performs worst for the X_typed task graph among all six types of task graphs. Even so, the proposed algorithm has a satisfactory performance as $\log_{10}(C:P) < 0$.

Figures 5(d)-(f) show experimental results for tree-, mesh-, and ladder-typed task graphs respectively. Because of strict memory limitation in experiment environment, Figure 5(e) shows only partial curves of "h(x)=0" and "W&T". The proposed algorithm performs well for these three types of task graphs. Moreover, it can be observed that for all six types of task graphs but the X-type, the performance of the proposed algorithm is stable in the range of $C:P$ ratios.

Figure 6 shows the execution time of 200 test instances for different numbers of modules. The proposed algorithm has the best performance almost everywhere, except for the X-typed task graph. Because of memory limitation, experimental results for the mesh-typed task graph are not shown here.

4.2 Other performance criteria

In addition to the execution time, in the experiment, we also consider the average number of search nodes and the maximal queue length of the unexpanded list during state-space search under different parameter combinations mentioned above. Because of the limit of space, we do not give them here, but describe in ref. 20. As was expected, the experimental results have the similar trend to it with respect to the execution time described in Section 4.1. Moreover, the deviation of the initial solutions found by our proposed algorithms to the optimal solution is also given in ref. 20.

5. Concluding Remarks

In this paper we have proposed a BBU algorithm for the task assignment problem, which was solved by Wang and Tsai (19). The essence of Wang and Tsai's algorithm is to underestimate the minimal task turnaround time from the viewpoint of bottleneck processor. This causes their algorithm a poor underestimation as the $C:P$ ratio is low. On the other hand, the proposed algorithm underestimates the minimal task turnaround time from the viewpoint of execution paths. Experimental results provide us with a complete comparison among the proposed algorithm, Wang and Tsai's algorithm, and A^* algorithm with $h(x)$=0. The proposed algorithm is stable in performance and has the best performance in most tested cases. Wang and Tsai's algorithm degenerates rapidly as the $C:P$ ratio decreases and its instability in performance makes it less attractive in practical applications.

In order to investigate the effect of the shapes of task graphs on the performance of the proposed algorithm, we consider six types of task graphs: linear, convergence, X_type, tree, ladder, and mesh in the experiments.

REFERENCES

[1] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, pp. 341-349, 1979.

[2] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 583-589, 1981.

[3] S. H. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207-214, 1987.

[4] S. H. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Transactions on Computers*, vol. C-37, no.1, pp. 48-57, 1988.

[5] M. S. Chern, G. H. Chen, and P. Liu, "An LC branch-and-bound algorithm for module assignment problem," to appear in *Information Processing Letters*.

[6] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, vol. 15, pp. 50-56, 1982.

[7] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland, 1976.

[8] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. C-37, no. 11, pp. 1384-1397, 1988.

[9] P. Y. R. Ma, E. Y. S. Lee, and J. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 41-47, 1982.

[10] N. J. Nilsson, "Problem solving methods in artificial intelligence," McGraw-Hill, New York, 1977.

[11] G. S. Rao and H. S. Stone, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Transactions on Computers*, vol. C-28, no.4, pp. 291-299, 1979.

[12] C. Shen and W. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Transactions on Computers*, vol. C-34, no.3, pp.197-203, 1985.

[13] J. B. Sinclair, "Efficient computation of optimal assignments for distributed tasks," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 342-361, 1987.

[14] J. B. Sinclair, "Optimal assignments in broadcast networks," *IEEE Transactions on Computers*, vol. C-37, no. 5, pp. 521- 531, 1988.

[15] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transactions on Software Engineering*, vol. SE-3, no.1, pp. 85-93, 1977.

[16] H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 254-258, 1978.

[17] L. C. Thomas and G. K. Jon, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 2, pp. 141-154, 1988.

[18] P. J. M. van Laarhoven and E. H. L. Aarts, Simulated Annealing: Theory and Applications, Kluwer Academic Publishers, 1987.

[19] L. Wang and W. Tsai, "Optimal assignment of task modules with precedence for distributed processing by graph matching and state-space search," *BIT*, vol. 28, pp. 54-68, 1988.

[20] J. S. Yur, "Optimal assignment of task modules with precedence in distributed computing systems," Department of Information Engineering, Tatung Institute of technology, Taipei, Taiwan, 1989.