Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
J. Parallel Distrib. Comput. 64 (2004) 1223–1240
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Branch-and-bound task allocation with task clustering-based pruning

Yung-Cheng Ma*, Tien-Fu Chen, Chung-Ping Chung

Department of Computer Science and Information Engineering, National Chiao-Tung University, 1001 Ta Hsueh Road, Hsinchu 30050, Taiwan

Received 8 June 2000; received in revised form 5 July 2004

Abstract

We propose a task allocation algorithm that aims at finding an optimal task assignment for any parallel programs on a given machine configuration. The theme of the approach is to traverse a state–space tree that enumerates all possible task assignments. The efficiency of the task allocation algorithm comes from that we apply a pruning rule on each traversed state to check whether traversal of a given sub-tree is required by taking advantage of dominance relation and task clustering heuristics. The pruning rules try to eliminate partial assignments that violate the clustering of tasks, but still keeping some optimal assignments in the future search space. In contrast to previous state–space searching methods for task allocation, the proposed pruning rules significantly reduce the time and space required to obtain an optimal assignment and lead the traversal to a near optimal assignment in a small number of states. Experimental evaluation shows that the pruning rules make the state–space searching approach feasible for practical use.
© 2004 Published by Elsevier Inc.

Keywords: Task allocation; Branch-and-bound; Pruning rule; Dominance relation; State–space searching

1. Introduction

Advances in hardware and software technologies have led to the use of parallel and distributed computing systems. To execute a parallel program efficiently, the mapping of program tasks to processors should consider both load balancing and reducing communication overhead. This paper studies such a task allocation problem.

Several research works have been done for the task allocation problem. Although the task allocation problem has been shown to be NP-complete [3], a set of heuristics have been proposed [4,8,9,11,14,15,19,23]. A drawback of these heuristics is the poor quality on the assignment found [5]. On the other hand, [1,2,7,12,13,16–18,20] proposed state–space searching methods with differences in the problem formulation for various applications and machine configurations. The state–space searching approach finds an optimal assignment at the cost of intractable time and space complexity. Ahmad and Kwok [1] proposed pruning rules and parallelization method to reduce the time to find an optimal solution of assigning precedence-constrained graphs. In this paper, we follow the task graph mode of [18], which models a set of parallel processes without precedence constraint, and propose pruning rules to improve the efficiency of state–space searching method.

The key idea of the proposed pruning rule is to detect task clustering in the task graph. We observe that tasks can be grouped such that a group is a set of heavily communicated tasks and inter-group communication weights are relatively small. While traversing the state–space, our proposed algorithm detects task clustering from traversal history and tries to prune partial assignments that violate the detected task clustering. We prove that the proposed pruning rule will reserve some optimal assignment in the future search space. This guarantees the optimality of the solution found. Moreover, our experiment shows that the proposed algorithm traverses only a low-order polynomial number of states to reach a near optimal assignment. Hence, when time and space is limited, a near optimal assignment can be obtained. This makes our proposed algorithm feasible for practical use.

* Corresponding author. Fax: +886-3-5724176.
E-mail address: ycma@csie.nctu.edu.tw (Y.-C. Ma).

0743-7315/$ - see front matter © 2004 Published by Elsevier Inc.
doi:10.1016/j.jpdc.2004.08.002

This paper is organized as follows. Section 2 models the task allocation problem as a state–space searching problem. Section 3 describes the basic idea of the proposed pruning rule. Section 4 describes the dominance relation, which is the basis to derive our pruning rule. Section 5 described the proposed pruning rule Section 6 describes the proposed task allocation algorithm and the space management policy. Section 7 presents the experiment to show the effectiveness of our proposed pruning rules. Finally, a conclusion is given in Section 8.

2. Modeling task allocation problem

In this section, we present how the task allocation problem is formulated and transformed into state–space searching problem. This section defines the terminologies used in this paper and gives the framework of our proposed task allocation algorithm.

2.1. Formulating task allocation problem

We follow [4,9,18] to formulate the task allocation problem. This formulation assumes that there are little or no precedence relationships and synchronization requirements so that processor idleness is negligible. Contentions on communication links are also ignored.

The optimization problem is formulated as follows. The input to a task allocation algorithm is a *task graph* G and a *machine configuration* M. The output, called a *complete assignment*, is a mapping that maps the set of tasks T to the set of processors P. An *optimal assignment* is a complete assignment with minimum *cost*. The cost of an assignment is the *turn-around time* of the last processor finishing its execution. To find an optimal assignment, the branch-and-bound algorithm will go through several *partial assignments*, where only a subset of the tasks has been assigned. We define the above terminology to formulate the task allocation problem.

A parallel program is represented as a *task graph* $G(T, E, e, c)$. The vertex set of the task graph is the set of tasks $T = \{t_0, t_1, \ldots, t_{n-1}\}$. Each task $t_i \in T$ represents a program module. The edge set E of the task graph represents communication between tasks. Two tasks t_i and t_j are connected by an edge if t_i communicates with t_j. For each task $t_i \in T$, a weight $e(t_i)$ is associated with it to represent the execution time of the task t_i. For each edge $(t_i, t_j) \in E$, a weight $c(t_i, t_j)$ is given to represent the amount of data transferred between tasks t_i and t_j.

An example task graph is depicted in Fig. 1. Each vertex is a task and the number on each task is the execution weight $e(t_i)$ for the task t_i. Associated with the number on edge (t_i, t_j) is the communication weight $c(t_i, t_j)$. Throughout this article, we will use this task graph to demonstrate the idea behind our algorithm.

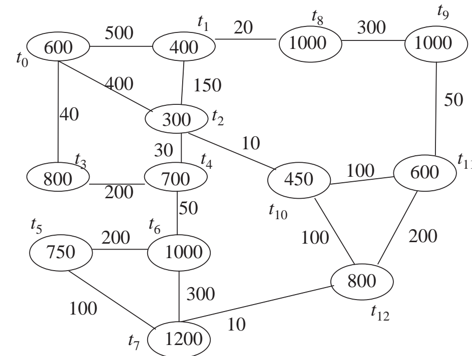


Fig. 1. Example of a task graph.

The *machine configuration* is represented as $M(P, d)$. $P = \{p_0, p_1, \ldots, p_{m-1}\}$ is the set of all processors. For each pair of processors $p_k, p_l \in P$, $k \neq l$, a distance $d(p_k, p_l)$ is associated to represent the latency of transferring one unit of data between p_k and p_l. If two tasks t_i and t_j are assigned to different processors p_k and p_l, respectively, the time required for task t_i to communicate with t_j is estimated to be $c(t_i, t_j)^* d(p_k, p_l)$. The communication time between two tasks within the same processor is assumed to be zero.

A machine configuration example is depicted in Fig. 2. We take the hierarchical architecture as an example. The machine consists of two subnets. It takes 5 units of time to transfer a unit of data for two processors in the same subnet and 20 units for two processors in different subnets. Throughout this paper, we will use the hierarchical architecture to demonstrate the idea of our task allocation algorithm. However, our proposed algorithm can also be applied to other machine configurations with non-uniform distances between processors.

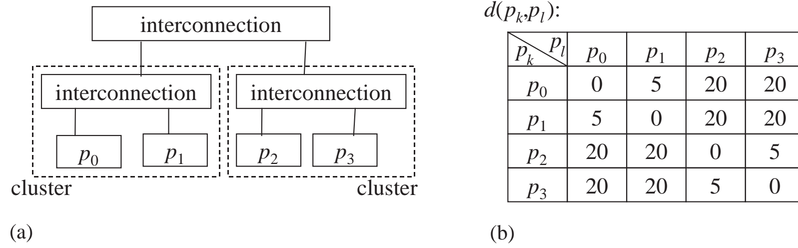
A *complete assignment* A_c is a mapping that maps the set of tasks T to the set of processors P. To find a complete assignment, our task allocation algorithm will examine several *partial assignments*. A *partial assignment* A is a mapping that maps Q, a proper subset of T, to the set of processors P.

The *turn-around time* of processor p_k, denoted $TA_k(A)$, under a partial/complete assignment A is defined to be the time to execute all tasks assigned to p_k plus the time that these tasks communicate with other tasks not assigned to p_k. That is,

$$TA_k(A) = \sum_{t_i : A(t_i) = p_k} e(t_i) + \sum_{t_i : A(t_i) = p_k} \sum_{t_j : A(t_j) \neq p_k} \times c(t_i, t_j)^* d(p_k, A(t_j)). \quad (1)$$

The *cost* of a partial/complete assignment is the turn-around time of the last processor finishing its execution:

$$cost(A) = \max_{\text{processor } p_k} TA_k(A). \quad (2)$$



p_k \ p_l	p_0	p_1	p_2	p_3
p_0	0	5	20	20
p_1	5	0	20	20
p_2	20	20	0	5
p_3	20	20	5	0

Fig. 2. Example of a machine configuration: (a) the clustered architecture and (b) the distance matrix ($d(p_k, p_l)$).

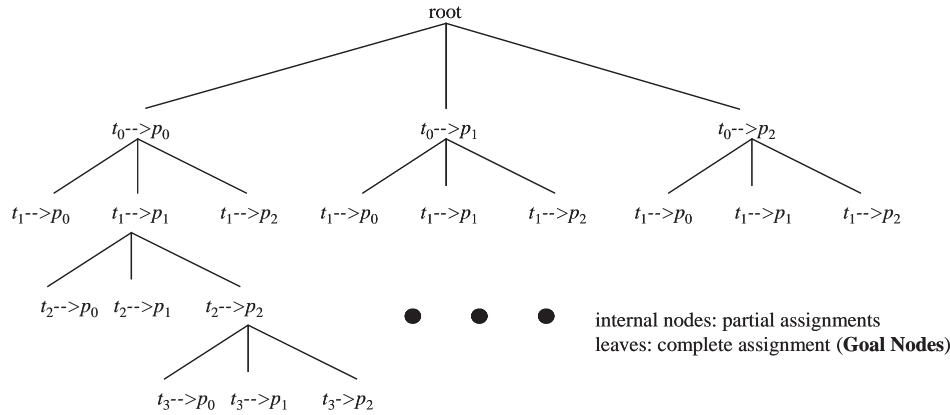


Fig. 3. State–space tree.

An *optimal assignment* A_{opt} is a complete assignment with minimum cost:

$$\begin{aligned} &cost(A_{\text{opt}}) \\ &\quad = \min\{cost(A_c) | A_c \text{ is a complete assignment}\}. \end{aligned} \tag{3}$$

2.2. *Transforming to the state–space searching problem—A*-algorithm*

We solve the task allocation problem by state–space searching with pruning rules. Shen and Tsai [18] proposed a state–space search algorithm without pruning to solve the task allocation problem. This state–space search method is known as the A*-algorithm [6], which has been proven to guarantee the optimality of the solution obtained. Based on the A*-algorithm, we add a pruning rule to reduce the search space to be traversed. In our experiment, this A*-algorithm will be used as a baseline for comparison with our branch-and-bound algorithm.

As illustrated in Fig. 3, the *state–space tree* represents all possible task assignments. We use an $(n+1)$-level m-ary tree to enumerate all possibilities of assigning n tasks to m processors. In the literature of branch-and-bound method, a node in the state–space tree is called a *branching state*. In this study, a branching state represents either a partial or a complete assignment, depending on whether the branching state is an internal node or a leaf node in the state–space tree. In the remaining of this article, we will use the terms branching states and partial/complete assignments interchangeably.

The traversal proceeds as follows. During the traversal, an *active set* [10] (also called the *open set* in some literature [6]), denoted *ActiveSet*, is used to keep track of all partial/complete assignments that have been explored but not visited. In each iteration during the traversal, the following operations are performed:

Step 1: Remove a partial/complete assignment A_v from *ActiveSet* and visit A_v.
Step 2: If A_v is a complete assignment, terminate the traversal and return A_v as the output.
Step 3: Check if the sub-trees derived from A_v need further traversal by using the *pruning rule*.
Step 4: If the sub-tree of A_v needs further traversal, put each child node of A_v in the state–space tree into *ActiveSet*.

For simplicity, we use $ActiveSet^{(k)}$ to denote the contents of the *ActiveSet* at the beginning of the kth iteration, and $A_v^{(k)}$ to denote the partial/complete assignment visited in the kth iteration.

We follow the approach in Shen and Tsai [18] to determine the traverse order. For each partial/complete assignment A, a lower-bound (denoted $L(A)$) on all complete assignments extended from A (or A itself in case that A is a complete assignment) is estimated. In each iteration during the traversal, the partial/complete assignment A_v with minimum $L(\bullet)$ is removed from *ActiveSet* and visited. $L(A)$ is

computed according to the *additional cost* of assigning tasks not assigned in A.

Given a partial assignment A in which $Q \subseteq T$ has been assigned, we define $AC_k(t_j \to p_l, A)$ to reflect the *additional cost* on processor p_k if task t_j is assigned to processor p_l:

$$AC_k(t_j \to p_k, A) = e(t_j) + \sum_{t_i : A(t_i) \neq p_k} c(t_i, t_j)^* d(p_k, A(t_i)) \quad \text{if } p_k = p_l, \tag{4}$$

$$AC_k(t_j \to p_l, A) = \sum_{t_i : A(t_i) = p_k} c(t_i, t_j)^* d(p_k, p_l) \quad \text{if } p_k \neq p_l. \tag{5}$$

For a partial assignment A, the *cost lower-bound* $L(A)$ for all complete assignments extended from A is estimated to be

$$L(A) \equiv \max_{\text{processor } p_k} \left(TA_k(A) + \sum_{t_i : \text{not assigned in } A} \times \left(\min_{\text{prcoessor } p_l} AC_k(t_i \to p_l, A) \right) \right). \tag{6}$$

Without pruning rules, the method presented so far is known as A*-algorithm [6], which was originally proposed by Shen and Tsai [18] for task allocation. The A*-algorithm traverses all partial assignments with $L(\bullet)$ less than the optimal cost. We propose a pruning rule to reduce the state–space size to be traversed.

3. Basic idea of the proposed pruning rule

The development of the pruning rule is based on the clustering of tasks. As shown in Fig. 4, tasks are grouped such that each group contains heavily communicating tasks. The key observation is that a group may contain a set of tasks suitable to be placed in the same processor, or a set of tasks suitable to be placed in the same subnet in the hierarchical architecture. While traversing the state–space tree, our branch-and-bound algorithm detects the clustering of tasks and tries to prune those partial assignments that violate the clustering heuristic. The effectiveness of the pruning rule thus depends on whether the tasks can be clearly clustered into groups.

The development of the pruning rule consists of two phases. In Section 4, we first develop a *dominance relation*. This dominance relation is effective only when a small cut is met. In Section 5, we further integrate the detection of clustering of tasks with the dominance relation to form an enhanced pruning rule.

4. Pruning search space by dominance relation

We first develop a *dominance relation* to serve as the basis for developing the pruning rule. We pick two partial assignments A_1 and A_2 in which the same set of tasks has been assigned. Suppose $cost(A_1) \leqslant cost(A_2)$. We call A_1 the winner and A_2 the loser. Let $A'_{1\text{-best}}$ and $A'_{2\text{-best}}$ be the complete assignments with a minimum cost in the subtree below A_1 and A_2, respectively. We want to be able to check whether it is possible that the winner–loser relationship will be changed, that is, $cost(A'_{1\text{-best}}) \geqslant cost(A'_{2\text{-best}})$. Our proposed dominance relation claims that what may reverse the winner–loser relationship is the weights of edges between assigned and un-assigned tasks in the task graph. The dominance relation is effective in pruning the search space when the weights between assigned and un-assigned tasks are small.

4.1. Formalization of dominance relation

Definition 1 (Dominance relation). Let A_1 and A_2 be two partial assignments. We say A_1 *dominates* A_2 if we can

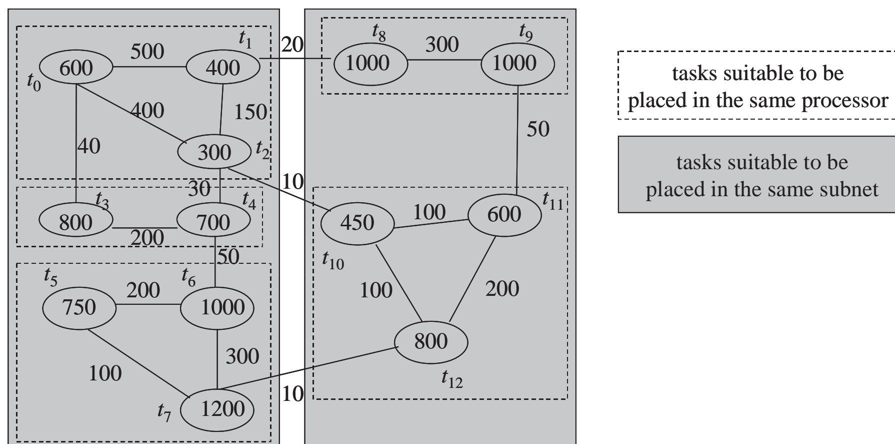


Fig. 4. Sample clustering of tasks according to communication weights.

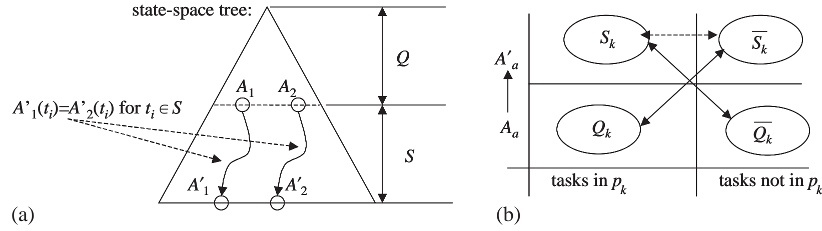


Fig. 5. Idea behind deriving the dominance relation: (a) selection of partial/complete assignments and (b) classifications on tasks.

guarantee that $cost(A'_{1\text{-best}}) \leqslant cost(A'_{2\text{-best}})$, where $A'_{1\text{-best}}$ and $A'_{2\text{-best}}$ are complete assignments with minimum cost extended from A_1 and A_2, respectively.

The inference rule we use to derive a dominance relation is as follows. We omitted the proof since it is a direct consequence from Definition 1.

Corollary 1 (Inference rule for deriving the dominance relation). *Let A_1 and A_2 be two partial assignments. A_1 dominates A_2 if for any complete assignment A'_2 extended from A_2, there exists a complete assignment A'_1 extended from A_1, such that $TA_k(A'_2) - TA_k(A'_1) \geqslant 0$ for each processor p_k.*

The idea to derive a dominance relation is depicted in Fig. 5. The assignments A_1, A_2, A'_1, and A'_2 concerned in Corollary 1 are shown in Fig. 5(a), where $S = T - Q$. A'_1 and A'_2 are chosen such that A_1 and A_2 have the same future extension. We rewrite the turn-around time equation according to the task classification shown in Fig. 5(b). In addition to $TA_k(A_2) - TA_k(A_1)$, the communication time between assigned and to-be-assigned tasks in $A_1(A_2)$ also contribute to $TA_k(A'_2) - TA_k(A'_1)$. This gives a lower bound estimation on $TA_k(A'_2) - TA_k(A'_1)$. The proposed dominance relation checks whether A_2 can be pruned or not according the estimated *turn-around time difference lower-bound*.

We introduce the following notations:

- $Execution(R) = \sum_{t_i \in R} e(t_i)$, where R is a set of tasks.
- $Communication(R_1, R_2) = \sum_{t_i \in R_1} \sum_{t_j \in R_2} c(t_i, t_j)^* d(A'_a(t_i), A'_a(t_j))$, where R_1 and R_2 are sets of tasks.

Following the classification on tasks shown in Fig. 5(b), we rewrite the turn-around time equation in the following lemma. The proof is omitted since it is a trivial computation from the turn-around time formula.

Lemma 1 (Reformulating the turn-around time). *Let A_a be a partial assignment and A'_a be a complete assignment extended from A_a. Q is the set of tasks assigned in A_a and S is the set of tasks not assigned in A_a. Then*

$$\begin{aligned} TA_k(A'_a) = {} & TA_k(A_a) + Execution(S_k(A_a)) \\ & + Communication(Q_k(A_a), \overline{S_k(A_a)}) \\ & + Communication(\overline{Q_k(A_a)}, S_k(A_a)) \\ & + Communication(S_k(A_a), \overline{S_k(A_a)}), \end{aligned} \tag{7}$$

where

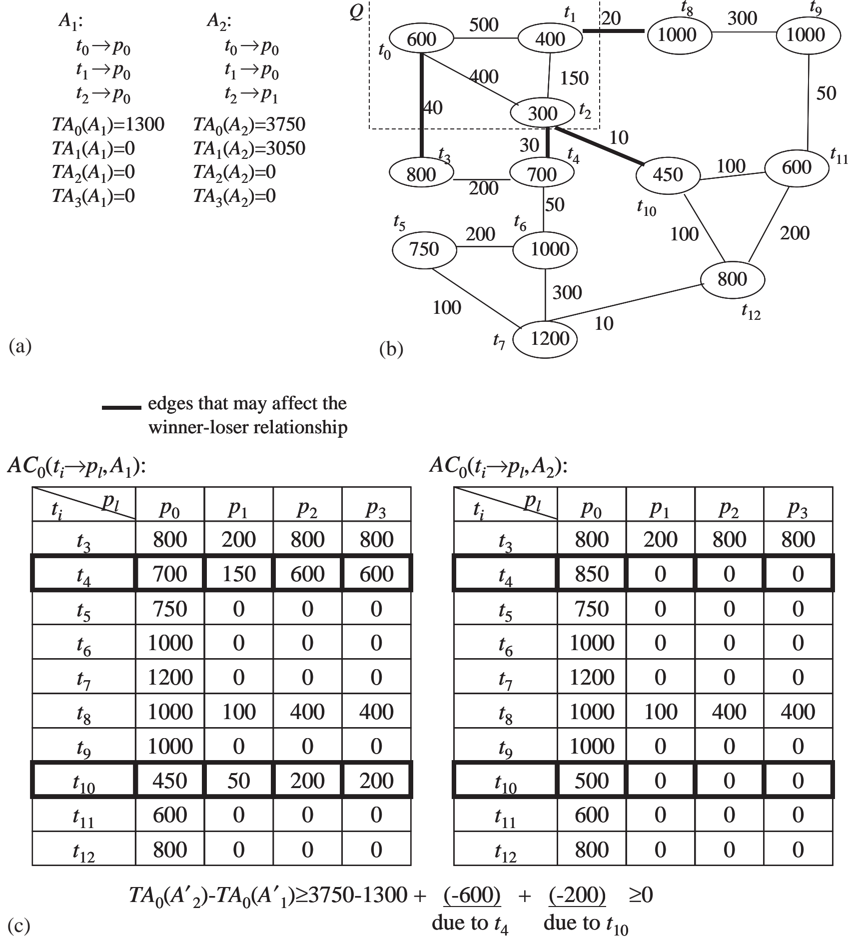
- $Q_k(A_a) = \{t_i \in Q | A_a(t_i) = p_k\}$ *and* $\overline{Q_k(A_a)} = Q - Q_k(A_a)$,
- $S_k(A'_a) = \{t_i \in S | A'_a(t_i) = p_k\}$ *and* $\overline{S_k(A'_a)} = S - S_k(A_a)$.

Before stating the dominance relation, we state the *turn-around time difference lower-bound* $TADL_k(A_1, A_2)$. Let A_1 and A_2 be two partial assignments with the same set of tasks Q being assigned, and $S = T - Q$. $TADL_k(A_1, A_2)$ is a lower bound on $TA_k(A'_2) - TA_k(A'_1)$, where A'_1 and A'_2 are arbitrary complete assignments extend from A_1 and A_2, respectively, such that $A'_1(t_i) = A'_2(t_i)$ for each task $t_i \in S$. $TADL_k(A_1, A_2)$ is estimated to be

$$\begin{aligned} & TADL_k(A_1, A_2) \\ & \quad \equiv TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} \\ & \qquad \times \left(\min_{p_l \in P} (AC_k(t_i \to p_l, A_2) \right. \\ & \qquad \left. - AC_k(t_i \to p_l, A_1)) \right). \end{aligned} \tag{8}$$

We then check whether A_2 can be pruned or not by computing $TADL_k(A_1, A_2)$ for each processor p_k. If $TADL_k(A_1, A_2)$ is greater than or equal to zero for each processor p_k, it indicates that $TA_k(A'_2) - TA_k(A'_1) \geqslant 0$ for each processor p_k and hence we can prune A_2. This is stated in the following theorem.

Theorem 1 (Dominance relation for space pruning). *Let A_1 and A_2 be two partial assignments containing the same set of tasks. If $TADL_k(A_1, A_2) \geqslant 0$ for each processor p_k, then A_1 dominates A_2.*



$AC_0(t_i \to p_l, A_1)$:

t_i \ p_l	p_0	p_1	p_2	p_3
t_3	800	200	800	800
t_4	700	150	600	600
t_5	750	0	0	0
t_6	1000	0	0	0
t_7	1200	0	0	0
t_8	1000	100	400	400
t_9	1000	0	0	0
t_{10}	450	50	200	200
t_{11}	600	0	0	0
t_{12}	800	0	0	0

$AC_0(t_i \to p_l, A_2)$:

t_i \ p_l	p_0	p_1	p_2	p_3
t_3	800	200	800	800
t_4	850	0	0	0
t_5	750	0	0	0
t_6	1000	0	0	0
t_7	1200	0	0	0
t_8	1000	100	400	400
t_9	1000	0	0	0
t_{10}	500	0	0	0
t_{11}	600	0	0	0
t_{12}	800	0	0	0

$TA_0(A'_2)\text{-}TA_0(A'_1) \geq 3750\text{-}1300 + \underbrace{(-600)}_{\text{due to } t_4} + \underbrace{(-200)}_{\text{due to } t_{10}} \geq 0$

Fig. 6. Example to illustrate the dominance relation: (a) partial assignments in consideration, (b) the task graph and (c) effects on p_0 for all possible extensions.

Proof. To draw a dominance relation by Corollary 1, we pick the complete assignment A'_1 extended from A_1 such that $A'_1(t_i) = A'_2(t_i)$ for each $t_i \in S$. The pattern is depicted in Fig. 5(a). We want to show that $TA_k(A'_2) - TA_k(A'_1) \geqslant 0$ for each p_k.

We decompose both $TA_k(A'_2)$ and $TA_k(A'_1)$ as stated in Lemma 1. Since $A'_1(t_i) = A'_2(t_i)$ for each $t_i \in S$, we have

- $Execution(S_k(A'_2)) - Execution(S_k(A'_1)) = 0$, and
- $Communication(S_k(A'_2), \overline{S_k(A'_2)}) - Communication(S_k(A'_1), \overline{S_k(A'_1)}) = 0$.

Hence, we have

$$\begin{aligned}
&TA_k(A'_2) - TA_k(A'_1) \\
&= TA_k(A_2) - TA_k(A_1) \\
&\quad + (Communication(S_k(A'_2), \overline{Q_k(A_2)}) \\
&\quad - Communication(S_k(A'_1), \overline{Q_k(A_1)})) \\
&\quad + (Communication(\overline{S_k(A'_2)}, Q_k(A_2)) \\
&\quad - Communication(\overline{S_k(A'_1)}, Q_k(A_1))) \\
&= TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} \\
&\quad \times (AC_k(t_i \to A'_2(t_i), A_2) \\
&\quad - AC_k(t_i \to A'_2(t_i), A_1)). \qquad (9)
\end{aligned}$$

Taking a lower bound on the turn-around time difference, we have

$$\begin{aligned}
&TA_k(A'_2) - TA_k(A'_1) \\
&\geqslant TA_k(A_2) - TA_k(A_1) \\
&\quad + \sum_{t_i \in S} \min_{p_l \in P} (AC_k(t_i \to p_l, A_2) \\
&\quad - AC_k(t_i \to p_l, A_1)).
\end{aligned}$$

The right-hand side of above inequality is the $TADL_k(A_1, A_2)$ defined previously. Hence if $TADL_k(A_1, A_2) \geqslant 0$ for each p_k, it implies A_1 dominates A_2. □

4.2. Example of the dominance relation

We use the task graph in Fig. 1 and the machine configuration in Fig. 2 to illustrate the idea of the dominance relation given in Theorem 1. The partial assignments concerned are

A_1 and A_2 shown in Fig. 6(a). A_1 is the winner and A_2 is the loser in this comparison. We apply Theorem 1 to guarantee that the winner–loser relationship will not be reversed.

We use the example in Fig. 6 to explain the key idea of exploiting task clustering. In the task graph in Fig. 6(b), $\{t_0, t_1, t_2\}$ is a group of heavily communicating tasks and should be assigned to the same processor. In Fig. 6(a), A_1 is a partial assignment obeying the task clustering and A_2 is a partial assignment that violates the task clustering. The dominance relation examines the "cut", edges between assigned tasks $\{t_0, t_1, t_2\}$ and remaining tasks (bolded edges in Fig. 6(b)), to test whether A_2 can be pruned or not. The examination finds that edges from assigned tasks to t_4 and t_{10} are the only possible causes for A_2 to win back what it has lost (cf. Fig. 6(c)). The edge weights in the cut are relative small and hence positive $TADL_k(A_1, A_2)$ values are obtained. This results in A_2 been pruned. Enumerating heavily communicated tasks in consecutive order ensures that a cut with light-weighted edges can be met and improves the pruning efficiency of the dominance relation.

5. Pruning search space by task clustering

The dominance relation proposed in Section 4 is effective only when a small cut can be found. To relieve this constraint, we develop a further pruning rule that considers both the detection of clustering of tasks and the dominance relation.

How well the pruning rule works depends on the task enumeration order. We assume that tasks are enumerated in an order such that heavily communicated tasks will be enumerated first. We will see how such an enumeration order is obtained in Section 6. With this assumption, a task assignment has the following properties:

- A complete assignment obtained by a greedy search policy reflects the clustering of tasks.
- The first partial assignment of assigning a sub-graph visited reflects the clustering of tasks in the sub-graph.

With these properties, we obtain (1) partial assignment A_k—called the *killer*—reflecting the clustering of tasks, and (2) complete assignment A_u served as an upper bound on the optimal cost to test whether a candidate partial assignment A can be pruned. These are the inputs to our pruning rule.

We use the task graph in Fig. 1 and the machine configuration in Fig. 2 to illustrate how the pruning rule works as depicted in Fig. 7. The killer A_k is a partial assignment with more tasks than the candidate A has. In the Fig. 7 example, A_k reflects the clustering of tasks by showing that $\{t_0, t_1, t_2\}$ should be placed in the same processor and $\{t_0, t_1, t_2, t_3, t_4\}$ should be placed in the same subnet. We are thus given the guidelines to extend A: (i) t_2 should be assigned to p_0, (ii) t_3, t_4 should be assigned to either of p_0 and p_1.

Complete assignments extended from A can be classified into two categories: extensions following or violating the guidelines. For extensions violating the guidelines, we estimate the cost lower bound and exclude those extensions whose costs are guaranteed to be greater than or equal to $cost(A_u)$. For extensions following the guidelines, we find a *dominator* A_d from the killer A_k that dominates these extensions. These observations lead us to propose the pruning rule, whose criteria for pruning the search space is stated as follows.

Pruning criteria: Let A_d and A be two partial assignments in which the same set of tasks has been determined, and A_u be a complete assignment. We prune A if for any complete assignment A' extended from A, either (i) $cost(A') \geqslant cost(A_u)$ or (ii) there exists a complete assignment A'_d extended from A_d such that $cost(A'_d) \leqslant cost(A')$.

5.1. Predicting clustering of tasks

Fig. 8 presents the procedure Compute_$PA(A, A_k)$ to predict the clustering of tasks. The result of this detection is a set of *possible assignments*, denoted PA_is, for each task t_i not assigned in A. Each PA_i is a set of processors which we can assign task t_i to PA_is are determined according to a killer A_k. That is, the killer should reflect the clustering of tasks in a task graph. How such a killer can be obtained will be explained in Section 5.4.

To generate a guideline to extending A, we sketch a distance hierarchy on processors centralized at the "central processor" p_c and map the tasks to the distance hierarchy. Let t_a be the last task assigned in A. We take p_c to be the one t_a is assigned to in A_k (cf. Step 1 in Fig. 8). For each task t_i assigned in A_k but not in A, we let PA_i be the set of all processors with distance less than or equal to $d(p_c, A_k(t_i))$ (cf. Step 2 in Fig. 8). If t_i is not assigned in A_k, no prediction is made and PA_i is set to be the set of all processors.

5.2. Examining partial assignment using pruning rule

Fig. 9 presents the procedure PruneTest to test whether a partial assignment can be pruned. Procedure PruneTest calls Compute_PA to predict the guidelines to extending the candidate A. From there, the remaining work is to examine whether the sub-tree of A needs further traversal using the pruning rule.

We first test the correctness of the prediction outcome PA_is. The test is performed by estimating a *turn-around time lower-bound* for extensions violating the guidelines, denoted $TAL_k(A, \text{violate } PA_i)$, stated as follows:

$$TAL_k(A, \text{violate } PA_i) \equiv TA_k(A) + \sum_{\substack{t_j \text{ not assigned in } A \\ \text{and } t_j \neq t_i}} \times \left(\min_{\text{processor } p_l} AC_k(t_j \to p_l, A) \right) + \min_{\text{processor } p_l \notin PA_i} AC_k(t_i \to p_l, A). \quad (10)$$

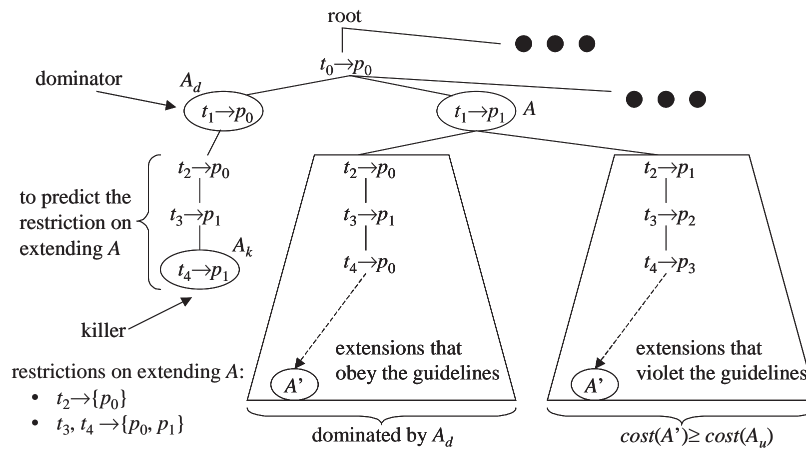


Fig. 7. Pruning based on task clustering.

Algorithm Compute_PA(A,A_k)

- **input**:
  - A, A_k : partial assignments, number of tasks assigned in $A_k \geq$ number of tasks assigned in A
- **output**:
  - $PA_i \subseteq P$ for each task t_i not assigned in A (P is the set of all processors)
- **method**:

  1) $p_c \leftarrow A_k(t_a)$ where t_a is the last task assigned in A

  2) **for** each task t_i not assigned in A **do**

      if t_i is assigned in A_k **then** $PA_i \leftarrow \{$ processor $p_k | \, d(p_k, p_c) \leq d(A_k(t_i), p_c) \}$

      else $PA_i \leftarrow P$

Fig. 8. Algorithm to predict the clustering of tasks.

Algorithm PruneTest(A,A_k,A_u)

- **input**:
  - A, A_k: partial assignments.
    - depth(A_k)≥depth(A)
  - A_u: a complete assignment
- **output**:
  - prune=True if A can be pruned, otherwise prune=False
- **method**:

  1) perform Compute_PA(A, A_k) to determine PA_i for each task t_i not assigned in A

  2) /* exclude extensions violating PA */

      2.1) success←False

      2.2) **for** each processor p_k **do**

          if $TAL_k(A$, violate $PA) \geq cost(A_u)$ **then**

              success ←True

              break

      2.3) **if** success=False **then** $PA_i \leftarrow P$

  3) $A_d \leftarrow$the ancestor of A_k in the same level with A

  4) prune←True

  5) /* dominate extensions obeying PA */

      for each processor p_k **do**

          if $TADL_k(A_d, A, PA) < 0$ **then**

              prune←False

              break

  6) **return** prune

Fig. 9. Algorithm to examine the partial assignment using the pruning rule.

Lemma 2. *Let A be a partial assignment and A' be a complete assignment extended from A. If there exists a task t_i not assigned in A such that $A'(t_i) \notin PA_i$, then $TA_k(A') \geqslant TAL_k(A, \text{violate } PA_i)$ for each processor p_k.*

Proof. The proof is similar to the estimation of the cost lower bound $L(\bullet)$ in [18]. The only difference is that when taking minimum on the sum of additional cost to obtain a lower bound on $TA_k(A')$, the possibilities of assigning t_i to processors in PA_i are excluded. □

After excluding extensions violating the guidelines, we then check the dominance imposed on the remaining extensions. The *dominator* A_d is the ancestor of A_k in the state–space tree at the same level with A. Similar to the procedure in Section 4, we estimate a *turn-around time difference lower-bound* between A_d and A, denoted $TADL_k(A_\mathrm{d}, A, PA)$, assuming that A_d and A have the same future extensions and following the guidelines for each task t_i not assigned in $A(A_\mathrm{d})$. We estimate $TADL_k(A_\mathrm{d}, A, PA)$ as follows:

$$\begin{aligned} TADL_k(A_\mathrm{d}, A, PA) &= TA_k(A) - TA_k(A_\mathrm{d}) \\ &\quad + \sum_{t_i \text{ not assigned}} \left(\min_{p_l \in PA_i} (AC_k(t_i \to p_l, A) \right. \\ &\quad \left. - AC_k(t_i \to p_l, A_\mathrm{d})) \right). \end{aligned} \tag{11}$$

Compared to the $TADL_k(A_\mathrm{d}, A)$ defined in Section 4, these two quantities are estimated in similar ways. The difference is that the future extensions of A_d and A have been restricted to be in PA_is in estimating $TADL_k(A_\mathrm{d}, A, PA)$. And $TADL_k(A_\mathrm{d}, A) = TADL_k(A_\mathrm{d}, A, PA)$ if each PA_i contains all of the processors.

Theorem 2 (Pruning rule). *Let A_d and A be two partial assignments in which the same set of tasks has been determined, and A_u be a complete assignment. PA_i's are guidelines to extend A for each task t_i not assigned in A. If*

(i) *For each task t_i not assigned in A, there exists a processor p_k such that $TAL_k(A, violate\ PA_i) \geqslant cost(A_\mathrm{u})$. And*
(ii) *$TADL_k(A_\mathrm{d}, A, PA) \geqslant 0$ for each processor p_k.*

Then the pruning criteria is satisfied and A can be pruned.

Proof. By Lemma 2, hypothesis (i) implies that complete assignments extended from A violating the guidelines PA_is will have a cost greater than or equal to $cost(A_\mathrm{u})$. The remainder of the proof is to estimate a lower bound on $TA_k(A') - TA_k(A'_\mathrm{d})$. This is similar to Theorem 1, but the possibilities of extending A to an assignment that violate the guidelines PA_is are ignored. The lower bound of $TA_k(A') - TA_k(A'_\mathrm{d})$ is thus estimated to be $TADL_k(A_\mathrm{d}, A, PA)$ as defined before. This proves the theorem. □

The procedure PruneTest uses Theorem 2 to test whether A can be pruned or not. Hypothesis (i) of Theorem 2 is guaranteed by Step 2. Step 5 in the procedure PruneTest checks whether hypothesis (ii) of Theorem 2 holds. This test then returns the result indicating whether A can be pruned or not.

The advantage of using the pruning rule in Theorem 2 instead of the dominance relation in Theorem 1 is that the space can be pruned earlier during the traversal. For the example given in Fig. 7, this advantage is shown in Fig. 10. If we use the dominance relation given in Theorem 1 as the pruning rule, the bolded partial assignments will be traversed. The reduced search space is an exponential function of the depth of the clustering of tasks that we can detect.

5.3. Obtaining an upper bound on the optimal cost

To check whether a partial assignment A can be pruned, the procedure PruneTest uses two additional inputs: (1) a complete assignment A_u served as an upper bound on the optimal cost and (2) a killer A_k reflecting the clustering of tasks. Another use of such an A_u is to serve as an "imperfect solution" once the "perfect solution" cannot be found. The task allocation problem is well known to be NP-complete [2]. Once the optimal assignment cannot be found subject to time and space constraints, an "imperfect solution"—a complete assignment that may not be optimal—would be returned as the output. In this section, we describe how such an A_u can be obtained.

We use a greedy search approach to obtain a complete assignment A_u. A pointer p is used to indicate the status of the greedy search. At the beginning, p points at the starting node (the partial assignment currently visited) in the state–space tree. In each step, we move p down to one of its children with the minimum cost. The procedure terminates when (1) p points at a partial assignment with a cost greater than that of the present A_u, or (2) p points at a complete assignment. A_u is then updated if a better complete assignment is found.

The reason we use greedy search is because not only of its simplicity but also the fact that a low cost complete assignment can be obtained if a careful task enumeration order is applied. Assume the tasks are enumerated in an order such that heavily communicated tasks will be enumerated consecutively. The complete assignment obtained will reflect the clustering of tasks and is likely to have a low cost.

To illustrate the idea, we take the task graph in Fig. 1 and machine configuration in Fig. 2 as an example. Consider the greedy search starts from the partial assignment $\{t_0 \to p_0, t_1 \to p_0\}$. Part of the greedy search path is shown in Fig. 11. The greedy search will assign t_2 to p_0 next since it is the child of $\{t_0 \to p_0, t_1 \to p_0\}$ with the lowest cost. This selection indicates that t_0, t_1, and t_2 may need be placed in the same processor. Similarly, t_3 will be assigned to p_1 following

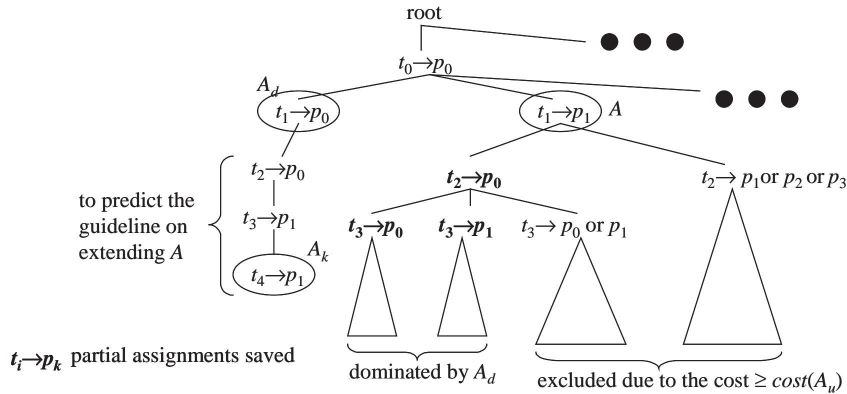


Fig. 10. Space saved by the pruning criteria.

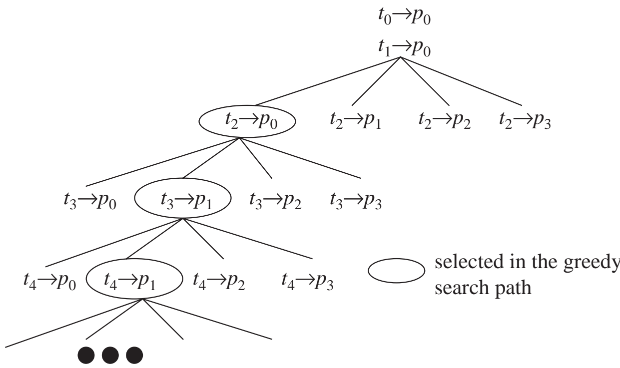


Fig. 11. Greedy search on the state–space tree.

the parent partial assignment $\{t_0 \to p_0, t_1 \to p_0, t_2 \to p_0\}$, also reflecting the clustering of tasks. Following the same procedure, we obtain a complete assignment that obeys the task clustering guideline.

5.4. Obtaining killers reflecting clustering of tasks

In addition to the complete assignment A_u, a partial assignment A_k reflecting the clustering of tasks is also helpful to enhance the pruning rule. To increase the possibility of pruning a partial assignment, we may find multiple killers to form a *KillerSet*, instead of only one killer. The procedure PruneTest is then performed for each killer in the *KillerSet* to test whether a partial assignment can be pruned.

Partial assignments reflecting clustering of tasks can be obtained by the proposed task enumeration order and the state–space tree traverse order. A partial assignment covers a sub-graph of the task graph. With the assumption that heavily communicated tasks are enumerated consecutively, we can capture part of the clustering of tasks in the sub-graph. Since we traverse the task graph in the minimum $L(\bullet)$ first order, the first partial assignment containing the sub-graph visited is the one with minimum $L(\bullet)$ among all partial assignments containing the sub-graph. The first partial assignment of containing a sub-graph visited indicates the clustering of tasks, otherwise it will have a large $L(\bullet)$.

We follow the principle that the first partial assignment indicates clustering of tasks to obtain killers. We assess that a candidate partial assignment A will be pruned if it violates the clustering of tasks somewhere in the path from root to the branching state in the state–space tree. Partial assignments having taken advantage of clustering of the tasks assigned by A are those partial assignments each of which (1) have a common ancestor with A in the state–space tree, (2) are visited earlier than A, and (3) are deeper than A in the state–space tree such that the sub-graph contained in A is also contained in them. This leads to the design of our heuristic scheme to obtain the killers.

To realize the scheme, a link to the deepest descendant node is associated with each visited partial assignment. For each partial assignment A_a, we associate a pointer $deep(A_a)$ pointing at the deepest partial assignment visited in the sub-tree of A_a. If two or more partial assignments at the same level of the state–space tree are visited, $deep(A_a)$ points at the first one visited, which has the smallest cost lower bound estimate ($L(\bullet)$) on all its extensions. The *KillerSet* is the set of all $deep(A_a)$ for each ancestor of A along with the complete assignment A_u.

$$KillerSet(A) = \{deep(A_a) | A_a \text{ is an ancestor of } A\} \cup \{A_u\}.$$

The determination of the *KillerSet* is depicted in Fig. 12. The number in each node is the $L(\bullet)$ of the partial assignment represented by the node. For each visited node A_a, the dashed link represents the deepest link $deep(A_a)$. When a partial assignment A is visited, we follow the deepest link along all ancestors of A to obtain the *KillerSet*. In this example, the *KillerSet* to be used for pruning A is $\{A_6, A_4\}$ plus A_u. That is, for each sub-tree (of the state–space tree) containing A, we pick the best branching state visited in the sub-tree to try to prune A.

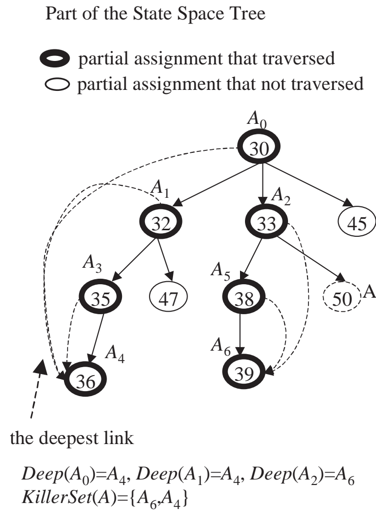


Fig. 12. Deepest link to determine the KillerSet.

6. Branch-and-bound task allocation with preprocessing

We now present the task allocation algorithm using the pruning rules. We present how a good enumeration order is obtained in Section 6.1. In Section 6.2, the branch-and-bound algorithm along with the correctness proof will be presented.

6.1. Preprocessing to determine the task enumeration order

We have seen the importance of the task enumeration order in previous sections. For the following reasons, tasks should be enumerated in such an order that tasks with high communication are enumerated first:

- To arrive at a small cut to exploit the dominance relation before the space overflow.
- To obtain killers that take advantage of the clustering of tasks.
- To obtain a low cost complete assignment serving as an upper bound on the optimal cost.

The task enumeration order is determined by applying the max-flow min-cut algorithm recursively to partition the task graph. Each time the max-flow min-cut procedure is applied, the set of tasks is decomposed into two partitions connected by a minimum cut. We repeat the partitioning recursively until each partition contains only one task. The partitioning process can be represented by a tree. Each leaf in the tree represents a group containing only one task. The enumeration order is thus the order of all leaf nodes in depth first traversal. For instance, the partitioning process for the task graph in Fig. 1 is depicted in Fig. 13. Following this result, we obtain the enumeration order that has been used for illustration in previous discussion.

6.2. The optimal branch-and-bound algorithm

The branch-and-bound algorithm is shown in Fig. 14. This is based on the A* traversal scheme with the addition of the pruning rules and related implementation code presented in Section 5. We now show that an optimal assignment can be obtained by the proposed algorithm if neither time-out nor overflow of the *ActiveSet* occurs.

To be convenient, we introduce some terminologies and notations. A complete assignment A_c is said to be in the *future search space* of $ActiveSet^{(k)}$ if either $A_c \in ActiveSet^{(k)}$ or there exists a partial assignment $A_a \in ActiveSet^{(k)}$ such that A_c can be derived from A_a. On the other hand, we say A_c is *lost* from $ActiveSet^{(k)}$ if A_c is not in the future search space of $ActiveSet^{(k)}$. The depth of a partial/complete assignment A, denoted $depth(A)$, is the length of the path from the root to the branching states representing A in the state–space tree.

The difficulty of showing the correctness of the algorithm is that the pruning rules may remove some partial assignments that can lead to optimal assignments. Fortunately, it can be guaranteed that there exists other optimal assignments in the future search space after pruning. When an optimal assignment is pruned, we always can find another optimal assignment survived in the future search space, as shown in Fig. 15. Provided that some optimal assignments survived in the future search space, we show that the termination condition implies the optimality of the solution obtained.

Lemma 3. *Assume that no overflow in the ActiveSet occurs. Then, during the traversal, there are always some optimal assignments survived in the future search space.*

Proof. We prove this by induction on the number of iterations i. The induction hypothesis is that

- for any optimal assignment $A_{\text{opt-}0}$ not in the future search space, there exists another optimal assignment $A_{\text{opt-}k}$ survived in the future search space such that $depth(A'_k) \geqslant depth(A'_0)$, where A'_0 and A'_k are the last visited ancestors of $A_{\text{opt-}0}$ and $A_{\text{opt-}k}$, respectively.

Lemma 3 holds in the beginning since no optimal assignment is lost at initialization. Assuming the induction hypothesis holds at the beginning of certain iteration. Suppose there is a partial assignment A'_0 been pruned in this iteration and A'_0 can be extended to some optimal assignment $A_{\text{opt-}0}$. The proof is to find the $A_{\text{opt-}k}$ and A'_k described in the induction hypothesis.

In this case, A'_0 must have been pruned by some dominator A_1, which can also be extended to an optimal assignment $A_{\text{opt-}1}$ (otherwise the pruning criteria is violated). Let A'_1 be the last visited ancestor of $A_{\text{opt-}1}$. By the pruning rule, part of the sub-tree below A_1 must be traversed and hence $depth(A'_1) \geqslant depth(A_1) = depth(A'_0)$. If A'_1 is not pruned, then $A_{\text{opt-}1}$ survives in the future search space and

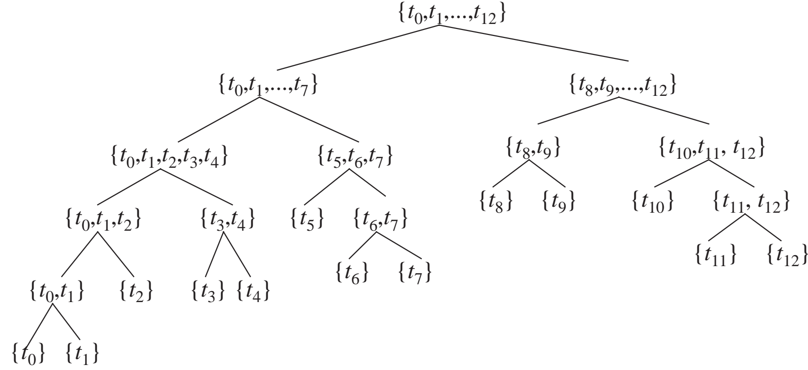


Fig. 13. Determining the task enumeration order.

Algorithm BB-Alloc(G,M)

- /* initialization phase */
  - L(root of the state-space tree) ←0
  - *ActiveSet*←{root of the state-space tree}
  - Obtain A_u by performing greedy search starting at the root of the state-space tree
- **while** not time-out **do** /* traversal phase */
  1) remove a partial/complete assignment A_v with minimum $L(\bullet)$ from *ActiveSet* and perform the following to visit(A_v)
     1.1) /* update deepest link for all ancestor of A */
          deep(A)←A
          for each A_a: ancestor of A in the state-space tree **do**
               if *depth*(A)>*depth*(*deep*(A_a)) **then** *deep*(A_a)←A
     1.2) /* try to improve A_u */
          perform greedy search starting from A to obtain a complete assignment A_c
          if cost(A_c)<cost(A_u) **then** A_u← A_c
  2) **if** A_v is a complete assignment **then** A_u← A_v and terminate the traversal by **return** A_u
  3) /* check if the sub-tree of A needs further traversal */
     KillerSet←{*deep*(A_a)| A_a is an ancestor of A_v in the state-space tree}∪{A_u}
     prune ←False
     for each A_k∈*KillerSet* **do**
          prune←PruneTest(A_k, A_u,A_v)
          if prune=True **then break**
  4) /* exploit children of A if the sub-tree of A needs further traversal */
     if prune=False **then**
          for each child A'_v of A_v in the state-space tree **do** compute $L(A'_v)$ and insert A'_v into *ActiveSet*

Fig. 14. The branch-and-bound algorithm for task allocation.

hence the induction hypothesis holds for the next iteration (cf. Fig. 15(a)). In case that $A_{\text{opt-1}}$ is lost, the induction hypothesis states that there exists a survived optimal assignment $A_{\text{opt-}k}$ with the last visited ancestor A'_k such that $depth(A'_k) \geqslant depth(A'_1) \geqslant depth(A_1) = depth(A'_0)$ (cf. Fig. 15(b)). And hence we obtain the required $A_{\text{opt-}k}$ and A'_k for $A_{\text{opt-0}}$ and A'_0. This proves the lemma. □

Theorem 3 (Correctness of our proposed algorithm). *Our proposed branch-and-bound algorithm will end up with an optimal assignment if neither space overflow in the ActiveSet nor time-out occurs.*

Proof. If not timed-out, some complete assignment A_c will be removed from the *ActiveSet* in the last iteration during the traversal. The complete assignment returned is this A_c. We want to show that A_c is optimal.

We prove this by contradiction. Suppose A_c is not optimal. Consider the contents of $ActiveSet^{(j)}$ for the last iteration j. Lemma 3 states the existence of an optimal assignment A_{opt} in the future search space of $ActiveSet^{(j)}$. Thus, we have $cost(A_c) > cost(A_{\text{opt}})$ since A_{opt} is optimal. Let A_a be the ancestor of A_{opt} (or A_{opt} itself) in $ActiveSet^{(j)}$. By the definition of $L(\bullet)$, $L(A_a) \leqslant cost(A_{\text{opt}})$. And hence $L(A_a) \leqslant cost(A_{\text{opt}}) < cost(A_c) = L(A_c)$. However, A_c is the one with minimum $L(\bullet)$ in $ActiveSet^{(j)}$. This means

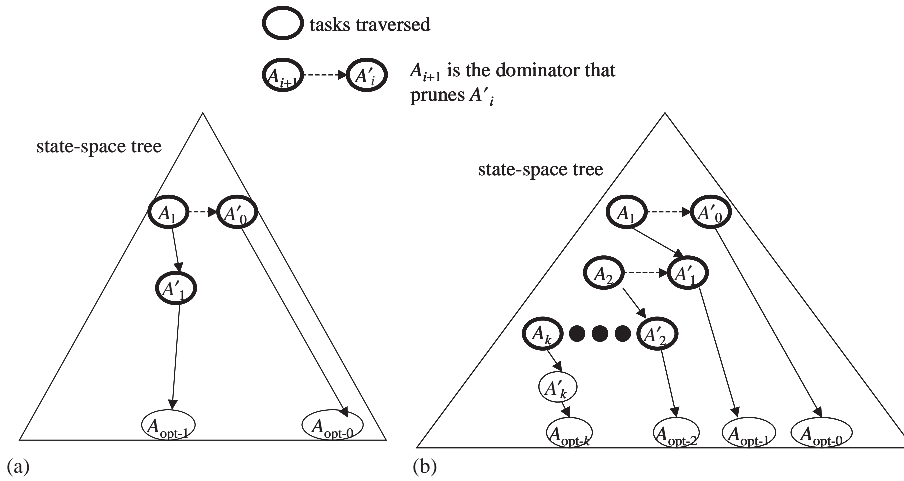


Fig. 15. Finding an optimal assignment survived in the future search space.

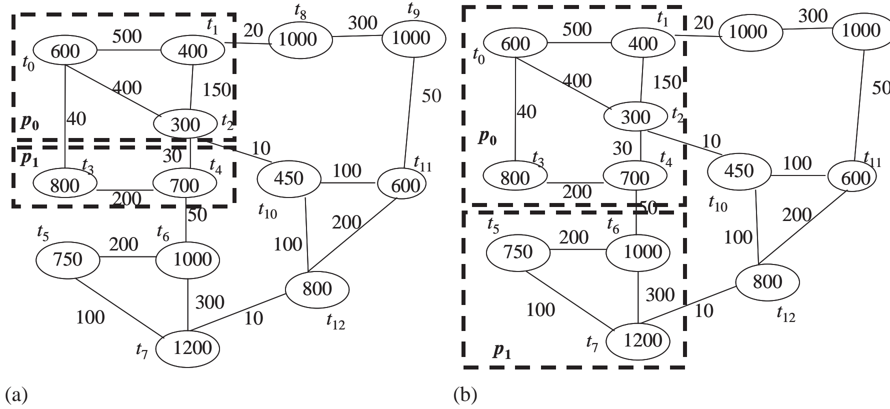


Fig. 16. Unfair comparison: assigning different sets of tasks: (a) partial assignment A_1 and (b) partial assignment A_2.

$L(A_c) \leqslant L(A_a)$. This produces a contradiction and hence proves this theorem. □

6.3. Space-efficient ActiveSet organization

The remaining problem in designing the task allocation algorithm is the design of *ActiveSet* such that (1) the partial/complete assignment with minimum $L(\bullet)$ can be easily removed, and (2) a near optimal assignment can be obtained once overflow occurs. A simple solution is to implement the *ActiveSet* as a heap and drop the partial/complete assignment with maximum $L(\bullet)$ when overflow occurs, because such an assignment is unlikely to be extended to an optimal assignment. However, this scheme has certain drawbacks. We identify two situations that will reduce the effectiveness of the victim selection scheme:

- Unfair comparisons between partial assignments containing different sets of tasks.
- Unfair comparisons between partial assignments using different numbers of processors.

Fig. 16 depicts an example of unfair comparison between partial assignments assigning different sets of tasks. Consider mapping the task graph in Fig. 1 to the machine configuration in Fig. 2. Fig. 16 depicts two partial assignments A_1 and A_2 containing different sub-graphs and $L(A_1) < L(A_2)$. However, A_2 can be extended to an optimal assignment but A_1 cannot. A partial assignment containing less number of tasks usually has lower cost and $L(\bullet)$, but this does not mean it has a better future extension. Our solution is to keep partial assignments assigning different number of tasks in different heaps.

Fig. 17 depicts an example of unfair comparison between partial assignments using different number of processors. We have two partial assignments A_1 and A_2 with $L(A_1) < L(A_2)$. A_1 is the best assignment to assign the sub-graph containing tasks $\{t_0, t_1, t_2, t_3, t_4\}$. However, A_2 can be extended to an optimal assignment but A_1 cannot. The assignment lacks knowledge of future load to be assigned and hence A_1 uses too many processors for tasks $\{t_0, t_1, t_2, t_3, t_4\}$. To avoid this drawback, we keep partial as-

- $L(A_1)<L(A_2)$ but A_2 can be extended to an optimal assignment

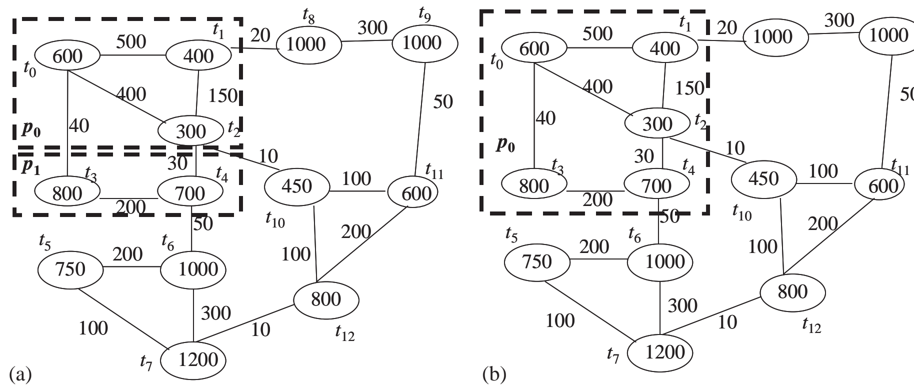


Fig. 17. Unfair comparison: using different number of processors: (a) partial assignment A_1 and (b) partial assignment A_2.

signments using different number of processors in different heaps.

We implement the *ActiveSet* as an array of heaps to avoid these two types of unfair comparisons. To assign n tasks to m processors, the *ActiveSet* is a two-dimensional array heap$[i][j]$ for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$. A (partial) assignment assigning tasks $\{t_0, t_1, \ldots, t_{i-1}\}$ using j processors is placed in heap$[i][j]$. The complexity of the branch-and-bound algorithm is controlled by the size of heap$[i][j]$, denoted size(i, j), which is a polynomial function of i and j. When the number of (partial) assignments in the *ActiveSet* containing $\{t_0, t_1, \ldots, t_{i-1}\}$ and using j processors exceeds size(i, j), the one in heap$[i][j]$ with maximum $L(\bullet)$ will be dropped. The future search space is thus extended from the best size(i, j) partial assignments containing tasks $\{t_0, t_1, \ldots, t_{i-1}\}$ and using j processors for all $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$. The complexity of the proposed algorithm is controlled by setting the heap size. By setting the size of heap$[i][j]$ to be k, the space complexity of the proposed algorithm is $O(n * m * k)$. To control the time complexity, we implemented the algorithm such that no new partial assignment will be inserted into heap$[i][j]$ after the first time heap$[i][j]$ is full. That is, at most k partial assignments that assigns $\{t_0, t_1, \ldots, t_{i-1}\}$ to j processors will be traversed. This makes the time complexity of the proposed algorithm to be also $O(n * m * k)$.

7. Experiments and evaluation

We evaluate the proposed task allocation algorithm by feeding it with several configuration samples generated randomly. The test samples cover many possibilities that may affect the effectiveness of the pruning rule.

7.1. Test samples generation

We randomly generate a set of task graphs and map the task graphs to some randomly selected hierarchical machine architectures. In generating task graphs, the distribution of weights and edge densities are chosen to cover various degrees of clustering of tasks. In selecting the machine configuration, the processor distances are chosen such that the parallelism in optimal assignments ranges from using a few processors to using all processors in the machine. The effectiveness of our proposed pruning rules is evaluated under various situations.

Following the idea in [4], we generate task graphs by hierarchically combining small sub-graphs. At the lowest level is a set of small complete graphs, each containing 1–4 tasks. The lowest level sub-graphs are then randomly combined to form a set of intermediate-level sub-graphs. The intermediate-level sub-graphs are then randomly combined to become a final task graph.

Randomly combining sub-graphs are guided by two parameters, the computation-to-communication weight ratio (denoted E/C ratio) and the edge density, defined as follows:

- $E/C = \dfrac{\text{Average execution weight of all tasks}}{\text{Average communication weight of all edges}}$.
- edge density = Probability of two vertices in different sub-graphs being connected by an edge.

In the process of randomly combining sub-graphs, each pair of tasks in different sub-graphs is examined. Whether there is an edge connecting these two tasks is determined according to the edge density. Once an edge is formed, the weight on the edge is determined according to the E/C ratio.

We denote the attributes of a task graph as a tuple of E/C ratio and an edge density. Task pair at each level or cross level has its own E/C ratio. For example, a task graph may be so generated: (1) select sub-graphs with $E/C = 1$ as the lowest level sub-graph, (2) combine lowest level sub-graphs to form a intermediate-level sub-graph with $E/C = 5$ and edge density=20%, (3) combining intermediate-level sub-graphs to complete a task graph with $E/C = 10$ and edge density=20%. We denote such a task graph with E/C : $(1, 5, 10)$ and edge density=20%.

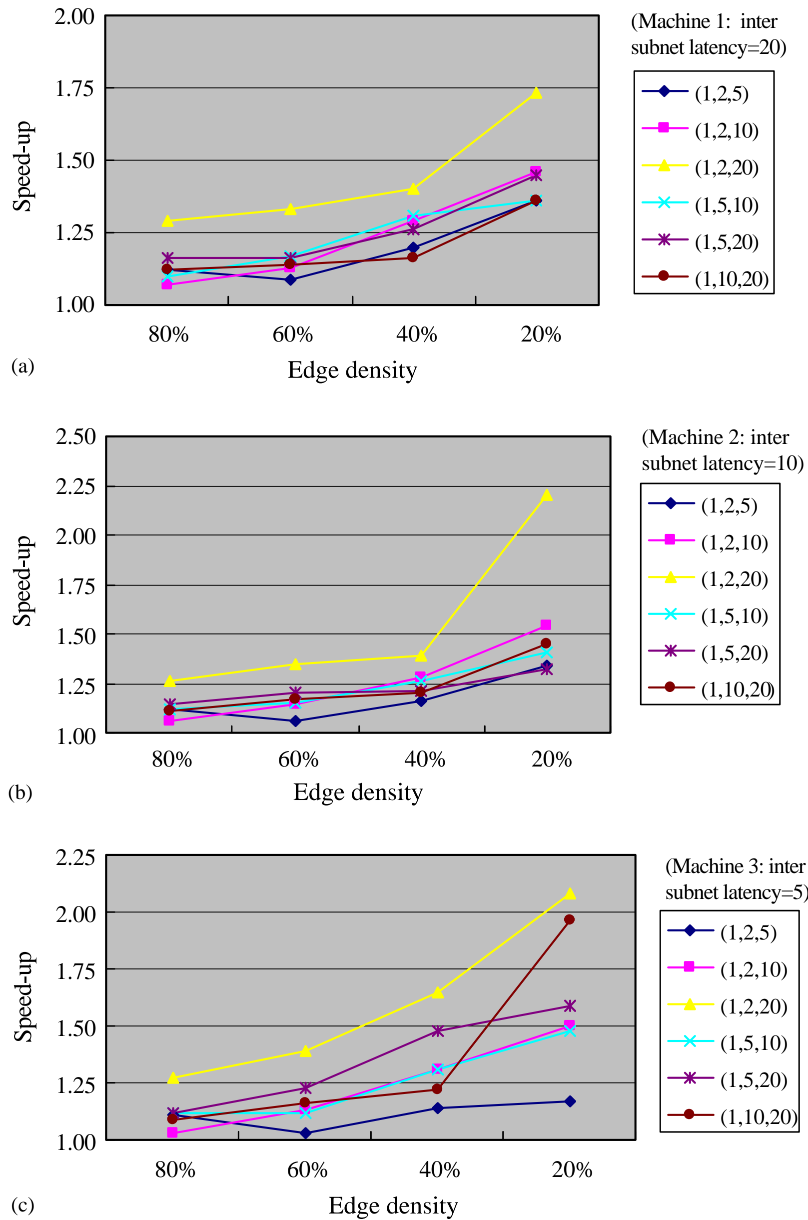


Fig. 18. Performance of the proposed task allocation algorithm: (a) performance comparison in Machine 1, (b) performance comparison in Machine 2 and (c) performance comparison in Machine 3.

The degree of clustering of tasks is controlled through selecting the E/C ratio and the edge density. The set of tasks can be clearly clustered into groups when (1) the gap between E/C ratios of adjacent levels is large, and (2) the sub-graphs are connected with relatively low edge density. In the experiment, the E/C ratio ranges from 1 to 20 and the edge density varies from 20% to 80%.

Another input to the task allocation program is the machine configuration. The machine configuration of interest is hierarchically similar to Fig. 2(a) but with a larger size and different latency. In the experimentation, each machine consists of three subnets, and each subnet consists of three processors. We fix the intra-subnet communication latency to be one. The inter subnet latency varies from 5 to 20. In mapping the same task graph to different machines, the parallelism in optimal assignments ranges from using processors in only one subnet to using processors across subnets. The parallelism decreases as the inter subnet latency increases.

7.2. Evaluation metrics

We evaluate both performance and allocation quality of our task allocation algorithm. We use the term performance to refer to the execution time to obtain an optimal solution. The performance is compared to the A*-algorithm [18] as

		allocation quality			
	assignment to Machine1 (intra subnet latency=20)				
		density=80%	density=60%	density=40%	density=20%
1	E/C:(1, 2, 5)	1.00	1.01	1.02	1.10
2	E/C:(1, 2, 10)	1.01	1.01	1.04	1.12
3	E/C:(1, 2, 20)	1.00	1.00	1.02	1.03
4	E/C:(1, 5, 10)	1.05	1.03	1.01	1.06
5	E/C:(1, 5, 20)	1.03	1.04	1.04	1.09
6	E/C:(1, 10, 20)	1.01	1.01	1.02	1.02
	assignment to Machine 2 (intra subnet latency=10)				
		density=80%	density=60%	density=40%	density=20%
1	E/C:(1, 2, 5)	1.00	1.02	1.05	1.11
2	E/C:(1, 2, 10)	1.03	1.01	1.04	1.10
3	E/C:(1, 2, 20)	1.00	1.00	1.03	1.01
4	E/C:(1, 5, 10)	1.06	1.05	1.05	1.08
5	E/C:(1, 5, 20)	1.05	1.06	1.05	1.07
6	E/C:(1, 10, 20)	1.03	1.01	1.04	1.05
	assignment to Machine3 (intra subnet latency=5)				
		density=80%	density=60%	density=40%	density=20%
1	E/C:(1, 2, 5)	1.00	1.03	1.05	1.14
2	E/C:(1, 2, 10)	1.03	1.04	1.05	1.10
3	E/C:(1, 2, 20)	1.00	1.02	1.02	1.04
4	E/C:(1, 5, 10)	1.06	1.06	1.06	1.08
5	E/C:(1, 5, 20)	1.07	1.07	1.06	1.02
6	E/C:(1, 10, 20)	1.02	1.00	1.03	1.02

Fig. 19. Allocation quality subject to time and space constraints.

follows:

$$\text{Speed-up} = \frac{\text{Number of branching states traversed by the A}^*\text{-algorithm}}{\text{Number of branching states traversed by the proposed branch-and-bound algorithm}}$$

We use the term allocation quality to refer to how good the complete assignment found under limited time and space, formulated as follows:

$$\text{Allocation quality} = \frac{\text{Cost of the complete assignment returned}}{\text{Cost of an optimal assignment}}.$$

7.3. Experimental results

The performance and allocation quality are evaluated using 240 task graphs and three hierarchical machine configurations. The task graphs are generated according to six different E/C tuples and four different edge density values, resulting in 24 different sets of task graphs. We generate 10 task graphs per set. The three machine configurations differ in the inter subnet latencies, varying from 5 to 20. The combinations of task graphs and machine configurations cover all degrees of clustering of tasks and parallelism to test the effectiveness of the pruning rule.

Fig. 18 shows the evaluation results of the performance in finding an optimal assignment. Experimental results on different machine configurations are depicted in different charts. We take the harmonic mean of the speed-ups for each set of ten task graphs generated under the same E/C tuple and edge density. The speed-ups ranges from 1.03 to 2.20, depending on the degree of clustering of tasks and parallelism. As expected, the curves show that the pruning rule is effective when the tasks can be clearly clustered into groups and the parallelism becomes large.

The allocation quality subject to restricted time and space is also evaluated. Time and space complexity are controlled with *ActiveSet* size and time-out threshold. In the experiment, the time-out threshold is set to be $n * m$, where n is the number of tasks and m is the number of processors, and the size of heap$[i][j]$ is set to be $i * j$.

The set of task graphs and machine configurations used to evaluate the allocation quality are the same as those used in evaluating the performance. The result is shown in Fig. 19.

We take harmonic mean of each set of 10 task graphs generated with the same E/C ratios and edge density. As shown in the figure, near optimal assignment can be found for each task graph and machine configuration.

8. Conclusion

In this paper, we have proposed a task allocation algorithm aiming at finding an optimal assignment. The key idea to the efficient task allocation is pruning, which take advantage of a combination of dominance relation and task clustering heuristic. This research shows that solving the task allocation problem by state–space searching approach is an attractive way. Previous state–space searching methods [2,17,18,20] find the optimal assignment in the cost of un-tractable time and space complexity. Our proposed pruning rule (1) reduces the time and space required to obtain an optimal assignment, and (2) makes the traversal reach a near-optimal assignment within a small number of traversal steps. This makes the state–space searching approach feasible in real-world applications.

References

[1] I. Ahmad, Yu-Kwong Kwok, Optimal and near-optimal allocation of precedence-constrained tasks to parallel processors: defying the high complexity using effective search techniques, in: Proceedings of 1998 International Conference on Parallel Processing, 1998, pp. 424–431.

[2] A. Billionnet, M.C. Costa, A. Sutter, An efficient algorithm for a task allocation problem, J. Assoc. Comput. Mach. 39 (3) (1992) 502–518.

[3] S.H. Bokhari, A shortest tree algorithm for optimal assignment across space and time in distributed processor system, IEEE Trans. Software Eng. 7 (11) (November 1981) 583–589.

[4] N.S. Bowen, C.N. Nikolaou, A. Ghafoor, On the assignment problem of arbitrary process systems to heterogeneous distributed systems, IEEE Trans. Comput. 41 (3) (March 1992) 257–273.

[5] H.C. Chou, C.P. Chung, An optimal instruction scheduler for superscalar processor, IEEE Trans. Parallel Distributed Systems 6 (3) (1995) 303–313.

[6] P.H. Hart, N.J. Nilsson, B. Rapheal, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Systems Cybernet. 4 (2) (July 1968) 100–107.

[7] C.J. Hou, K.G. Shin, Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems, IEEE Trans. Comput. 48 (12) (December 1997) 1336–1356.

[8] C.-W. Hsueh, K.-J. Lin, Scheduling real-time systems with end-to-end timing constraints using the directed pinwheel model, IEEE Trans. Comput. 50 (1) (January 2001) 51–66.

[9] C.C. Hui, S.T. Chanson, Allocating task interaction graphs to processors in heterogeneous networks, IEEE Trans. Parallel Distributed Systems 8 (9) (September 1997) 908–925.

[10] W.H. Kohler, K. Steiglitz, Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems, J. Assoc. Comput. Mach. 21 (1) (1974) 140–156.

[11] Y.K. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distributed Systems 7 (5) (May 1996) 506–521.

[12] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surveys 31 (4) (December 1999) 406–471.

[13] C.H. Lee, K.G. Shin, Optimal task assignment in homogeneous networks, IEEE Trans. Parallel Distributed Systems 8 (2) (1997) 119–128.

[14] V.M. Lo, Heuristic algorithms for task assignment in distributed systems, IEEE Trans. Comput. 37 (11) (November 1988) 1384–1397.

[15] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, J. Parallel Distributed Comput. 59 (2) (November 1999) 107–131.

[16] D.T. Peng, K.G. Shin, Optimal scheduling of cooperative tasks in a distributed system using an enumerative method, IEEE Trans. Software Eng. 19 (3) (1993) 253–267.

[17] P.Y. Richard, E.Y.S. Lee, M. Tsuchiya, A task allocation model for distributed computing systems, IEEE Trans. Comput. C-31 (1) (1982) 41–47.

[18] C.C. Shen, W.H. Tsai, A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion, IEEE Trans. Comput. 34 (3) (March 1985) 197–203.

[19] H.J. Siegel, S. Ali, Techniques for mapping tasks to machines in heterogeneous computing systems, J. Systems Architecture 46 (8) (May 2000) 627–639.

[20] J.B. Sinclair, Efficient computation of optimal assignment for distributed tasks, J. Parallel Distributed Comput. 4 (1987) 342–362.

[21] H.S. Stone, Multiprocessor scheduling with the aid of network flow algorithms, IEEE Trans. Software Eng. SE-3 (1) (January 1979) 85–94.

[22] P.A. Tom, C.S.R. Murthy, Algorithms for reliability-oriented module allocation in distributed computing systems, J. Systems Software 40 (1998) 125–138.

[23] C.M. Woodside, G.G. Monforton, Fast allocation of processes in distributed and parallel systems, IEEE Trans. Parallel Distributed Systems 4 (2) (1993) 164–174.

Yung-Cheng Ma received the B.S. and Ph.D. degree in Computer Science and Information Engineering from the National Chiao-Tung University (NCTU), Hsinchu, Taiwan, in 1994 and 2002, respectively. After graduating from NCTU, he joined computers and communication laboratory of Industry Technology Research Institute (ITRI) in Taiwan as an engineer. His research interests include computer architecture, parallel and distributed systems, information retrieval, and SOC system design.

Tien-Fu Chen received the B.S. degree in Computer Science from National Taiwan University in 1983. After completing his military services, he joined Wang Computer Ltd, Taiwan as a software engineer for 3 years. From 1988 to 1993 he attended the University of Washington, receiving the M.S. degree and Ph.D. degrees in Computer Science and Engineering in 1991 and 1993, respectively. He is currently a Professor in the Department of Computer Science and Information Engineering at National Chung Cheng University, Chiayi, Taiwan. In the past, he had published several widely cited papers on dynamic hardware prefetching algorithms and designs. In recent years, he has made contributions to processor design and SOC design methodology. Recent research results of his research team include 8-bit CISC/RISC microcontrollers, a 32-bit RISC processor core, a dual-core (RISC and DSP) SOC platform, and low-power architecture techniques as well as related support tools and verification environment. His current research interests are Computer Architectures, System-on-Chip Design, Design Automation, and Embedded Software Systems.

Chung-Ping Chung received the B.E. degree from the National Cheng-Kung University, Tainan, Taiwan, Republic of China in 1976, and the M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering. He was a lecturer in Electrical Engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Hsinchu, Taiwan, Republic of China, where he is a professor. From 1991 to 1992, he was a visiting Associate Professor of Computer Science at the Michigan State University, MI, USA. From 1998, he joined the Computer and Communications Laboratories, Industrial Technology Research Institute, ROC as the Director of the Advanced Technology Center, and then the Consultant to the General Director until 2002. He also served as the Editor-in-Charge in the Information Engineering Section of the Journal of the Chinese Institute of Engineers, ROC, since 2000. His research interests include computer architecture, parallel processing, and embedded system and DSP design.