A dominance relation enhanced branch-and-bound task allocation

Yung-Cheng Ma *, Chung-Ping Chung

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

Received 13 May 2000; received in revised form 21 September 2000; accepted 11 October 2000

Abstract

We investigate the task allocation problem of allocating a parallel program on parallel processors with non-uniform communication latencies. A branch-and-bound algorithm with a dominance relation is proposed to obtain an optimal task assignment. The key observation on deriving the dominance relation is that tasks can be clustered according to communication weights. The dominance relation is effective to prune the search space when the task clustering boundary – a small cut – is met. The proposed algorithm is compared to the A*-algorithm for task allocation. Experiment shows that our proposed algorithm achieves a speed-up ranging from 1.02 to 1.68, depending on the degree of task clustering and parallelism. This shows the effectiveness of the proposed dominance relation. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Task allocation; Branch-and-bound algorithm; Dominance relation

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, a scheduler should distribute the parallel program tasks to processors such that (1) the processor loads are balanced, and (2) the inter-processor communication time is minimized. This raises the task allocation problem.

We investigate the problem of allocating a parallel program on a multiprocessor with non-uniform inter-processor communication latencies. A parallel program is modeled as a node- and edge- weighted undirected graph, called a task graph. The task allocation problem becomes a problem of mapping the set of tasks to the set of processors such that the completion time is minimized, considering both processor load and communication overhead.

A set of work has been done for the task allocation problem. The task allocation problem has been shown to be NP-complete (Garey and Johnson, 1979) and a set of heuristics have been proposed (Lo, 1988; Bowen et al., 1992; Woodside and Monforton, 1993; Hui and Chanson, 1997). A drawback on these heuristics is the poor quality on the assignment found. On the other hand, (Richard et al., 1982; Shen and Tsai, 1985; Sinclair, 1987; Peng and Shin, 1993; Hou and Shin, 1997; Lee and Shin, 1997) 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 un-tractable time and space complexity.

In this paper, we propose a branch-and-bound algorithm with a dominance relation to obtain an optimal assignment. We follow the problem formulation defined in (Shen and Tsai, 1985) to investigate the task allocation problem. The key idea to the efficient task allocation is that a dominance relation is proposed to reduce the time- and space-complexity required for obtaining an optimal assignment. We compare the performance with A*-algorithm (Shen and Tsai, 1985) to demonstrate the effectiveness of the dominance relation.

The key observation on deriving the dominance relation is that tasks in the task graph can be clustered into groups according to communication weights. A group may consist of tasks suitable to be placed in the same processor or in the same subnet in a hierarchical architecture. A small cut in the task graph implies the boundary of a task group in the clustering. The dominance relation prunes task assignments violating the clustering on tasks according to the cut.

This paper is organized as follows. In Section 2, we formulate the optimization problem and model it as a state-space searching problem. We then present the
dominance relation in Section 3 and derive the task allocation algorithm using the dominance relation in Section 4. The experiment results are shown in Section 5. Finally, a conclusion is given in Section 6.

2. Problem modeling

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 (Shen and Tsai, 1985; Hui, 1997; Bowen et al., 1992) to formulate the task allocation problem for mapping parallel processes to processors. This model 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 $G(T, E, e, c)$ and a machine configuration $M(P, d)$. 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.

The machine configuration is represented as $M(P, d)$. $P = \{p_0, p_1, \ldots, p_{n-1}\}$ 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

---

**Fig. 1.** Example of a task graph.

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

\[
TA_k(A) = \sum_{t_i : A(t_i) = p_k} e(t_i) + \sum_{t_i : A(t_i) = p_k} \sum_{t_j : A(t_j) \neq p_k} c(t_i, t_j) * d(p_k, A(t_j)).
\]  

(1)

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

\[
\text{cost}(A) = \max_{\text{processor } p_k} TA_k(A).
\]  

(2)

An optimal assignment \( A_{opt} \) is a complete assignment with minimum cost

\[
\text{cost}(A_{opt}) = \min \{ \text{cost}(A_c) | A_c \text{ is a complete assignment} \}.
\]  

(3)

2.2. Transforming to the state-space searching problem – \( A^* \)-algorithm

We solve the task allocation problem by state-space searching with a dominance relation. Shen and Tsai (Shen and Tsai, 1985) proposed a state-space searching algorithm without dominance relation to solve the task allocation problem. This state-space search method is known as the \( A^* \)-algorithm (Hart et al., 1968), which has been proven to guarantee the optimality of the solution obtained. Based on the \( A^* \)-algorithm, we apply a dominance relation to reduce the number of states 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 (Kohler and Steiglitz, 1974) (also called the open set in some literature (Hart et al., 1968)), 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:

1. **Step 1.** Remove a partial/complete assignment \( A_c \) from ActiveSet and visit \( A_c \).
2. **Step 2.** If \( A_c \) is a complete assignment, terminate the traversal and return \( A_c \) as the output.
3. **Step 3.** Generate children of \( A_c \) in the state-space tree.
4. **Step 4.** Put each child node of \( A_c \) not pruned by the dominance relation into ActiveSet.

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

We follow the approach in Shen and Tsai (1985) 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 \( A_c \) with minimum \( L(\cdot) \) 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 \( A \) assigning tasks \( Q \subseteq T \), we define \( AC_k(t_j \rightarrow p_i, A) \) to reflect the additional cost on processor \( p_i \) if task \( t_j \) is assigned to processor \( p_i \):

\[AC_k(t_j \rightarrow p_i, A) = \sum_{t_i : A(t_i) = p_k} e(t_i) + \sum_{t_i : A(t_i) = p_k} \sum_{t_j : A(t_j) \neq p_k} c(t_i, t_j) * d(p_k, A(t_j)).\]  

Fig. 3. State-space tree.
\[ AC_k(t_j \rightarrow p_k, A) = e(t_j) + \sum_{t_i \in A(t_j) \neq p_k} c(t_i, t_j) \cdot d(p_k, A(t_i)) \]

if \( p_k = p_l \), \hspace{1cm} (4)

\[ AC_k(t_j \rightarrow p_l, A) = \sum_{t_i \in A(t_j) = p_l} c(t_i, t_j) \cdot d(p_l, p_i) \]

if \( p_k \neq p_l \), \hspace{1cm} (5)

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

\[ L(A) \equiv \max_{\text{processor } p_k} \left( TA_k(A) + \sum_{t_i \text{ not assigned in } A} \min_{\text{processor } p_l} AC_k(t_i \rightarrow p_l, A) \right). \]

(6)

Without dominance relations (Kohler and Steiglitz, 1974), the method presented so far is known as \( \text{A}^* \)-algorithm (Hart et al., 1968), which was originally proposed by Shen and Tsai (1985) for task allocation. The \( \text{A} \)-algorithm traverses all partial assignments with \( L(\cdot) \) less than the optimal cost. In the remaining sections, we improve the algorithm by adding a dominance relation to reduce the number of states traversed.

3. Dominance relation for space pruning

We propose a dominance relation to prune the search space. We pick two partial assignments \( A_1 \) and \( A_2 \) in which the same set of tasks has been assigned. Suppose \( \text{cost}(A_1) \leq \text{cost}(A_2) \). 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 sub-tree 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 reversed, that is, \( \text{cost}(A_{1\text{-best}}') \geq \text{cost}(A_{2\text{-best}}') \). Our proposed dominance relation claims that what may reverse the winner-loser relationship is the cut the weights of edges between assigned and un-assigned tasks in the task graph.

3.1. 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 guarantee that \( \text{cost}(A_{1\text{-best}}') \leq \text{cost}(A_{2\text{-best}}') \) 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 \( TA_k(A_1') - TA_k(A_1) \geq 0 \) for each processor \( p_k \).

The idea to derive a dominance relation is depicted in Fig. 4. The assignments \( A_1, A_2, A_1', \) and \( A_2' \) concerned in Corollary 1 are shown in Fig. 4(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. 4(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 according to the estimated turn-around time difference lower-bound.

We introduce the following notations:
- \( \text{Execution}(R) = \sum_{t_i \in R} e(t_i) \) where \( R \) is a set of tasks.
- \( \text{Communication}(R_1, R_2) = \sum_{t_i \in R_1} \sum_{t_j \in R_2} c(t_i, t_j) \cdot d(A'_2(t_j), A'_1(t_i)) \), where \( R_1 \) and \( R_2 \) are sets of tasks.

Following the classification on tasks shown in Fig. 4(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 \( A_2 \) be a partial assignment and \( A_2' \) be a complete assignment extended from \( A_2 \). \( Q \) is the set of tasks assigned in \( A_2 \) and \( S \) is the set of tasks not assigned in \( A_2 \). Then

---

Fig. 4. Idea behind deriving the dominance relation: (a) selection of partial/complete assignments; (b) classifications on tasks.
\[ TA_k(A'_1) = TA_k(A_1) + \text{Execution}(S_k(A_1)) + Communication(Q_k(A_1), S_k(A_1)) + Communication(Q_k(A_2), S_k(A_2)) + Communication(S_k(A_1), S_k(A'_2)), \tag{7} \]

where

- \( Q_k(A_1) = \{ t_i \in T | A_1(t_i) = p_k \} \) and \( Q_k(A_2) = Q - Q_k(A_1) \)
- \( S_k(A'_1) = \{ t_i \in S | A'_1(t_i) = p_k \} \) and \( S_k(A'_2) = S - S_k(A'_1) \)

Before stating the dominance relation, we state the \textit{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 \( Q \), and \( S = T - Q \). TADL\(k(A_1, A_2)\) is a lower bound on \( TA_k(A'_1) - TA_k(A'_2) \), where \( A'_1 \) and \( A'_2 \) are any 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

\[ TADL_k(A_1, A_2) = TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} \left( \min_{p_i \in P} (AC_k(t_i \rightarrow p_i, A_2) - AC_k(t_i \rightarrow p_i, A_1)) \right). \tag{8} \]

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\(k(A_1, A_2) \geq 0 \) for each processor \( p_k \), it indicates that \( TA_k(A'_2) - TA_k(A'_1) \geq 0 \) 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\(k(A_1, A_2) \geq 0 \) for each processor \( p_k \), then \( A_1 \) dominates \( A_2 \).

**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. 4(a). We want to show that \( TA_k(A'_2) - TA_k(A'_1) \geq 0 \) 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

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

Hence, we have

\[ TA_k(A'_2) - TA_k(A'_1) = TA_k(A_2) - TA_k(A_1) + \left( \text{Communication}(S_k(A'_2), Q_k(A_2)) - \text{Communication}(S_k(A'_1), Q_k(A_1)) \right) + \left( \text{Communication}(S_k(A'_2), Q_k(A_2)) - \text{Communication}(S_k(A'_1), Q_k(A_1)) \right). \]

Procedure **D**ominate**T**est\(k(A_2, A_1)\)

**input:**
- \( A_2, A_1 \): two partial assignments assigning the same set of tasks

**output:**
- returns True if \( A_2 \) can be pruned, otherwise returns False

**method:**

1) prune ● True

2) for each processor \( p_k \) do

   if TADL\(k(A_1, A_2) = 0 \) then

   prune ● False

   break

3) return prune

Fig. 5. Examining partial assignments using the dominance relation.

\[ = TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} (AC_k(t_i \rightarrow A'_2(t_i), A_2) - AC_k(t_i \rightarrow A'_1(t_i), A_1)). \tag{9} \]

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

\[ TA_k(A'_2) - TA_k(A'_1) = TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} (AC_k(t_i \rightarrow p_i, A_2) - AC_k(t_i \rightarrow p_i, A_1)). \]

The right-hand side of above inequality is the TADL\(k(A_1, A_2) \) defined previously. Hence if TADL\(k(A_1, A_2) \geq 0 \) for each \( p_k \), it implies \( A_1 \) dominates \( A_2 \). \( \square \)

The procedure to detect whether a partial assignment dominates another is depicted in Fig. 5. The procedure computes TADL\(k(A_1, A_2) \) for each processor to determine whether \( A_2 \) can be pruned or not.

### 3.2. 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.

Assuming that \( A_1 \) and \( A_2 \) have the same future extension, the weights of the bolded edges in Fig. 6(b) are the only factors that may reverse the winner-loser relationship. Following the decomposition of turn-around time stated in Lemma 1, execution time of tasks \( \{t_3, t_4, \ldots, t_{12}\} \) and communication time between these tasks will not reverse the winner-loser relationship. Fig. 6(c) shows the effects on processor \( p_0 \) for all possibilities of extending \( A_1 \) and \( A_2 \). Theorem 1 states that assignments of tasks \( t_4 \) and \( t_{10} \) are the only possible causes to reverse the winner-loser relationship since both communicate with task \( t_2 \) – the primary
difference between $A_1$ and $A_2$. The sum of communication weights between tasks \{t_2, t_4\} and \{t_2, t_{10}\} is small compared to the turn-around time lost in $A_2$. This means that it is impossible for $A_2$ to win back what it has lost to $A_1$ on processor $p_0$. Similar check applied to other processors also leads to the same conclusion. It indicates that the traversal of the subtree rooted at $A_2$ can be ignored. This is the primary idea of the dominance relation.

4. Task allocation algorithm

We now present the task allocation algorithm. We present how a good enumeration order is obtained in Section 4.1. In Section 4.2, the branch-and-bound algorithm along with the complexity analysis will be presented. We prove that an optimal assignment will be found by the branch-and-bound algorithm in Section 4.3.

4.1. Preprocessing stage to determine the task enumerating order

The enumeration order plays an important role on the performance of the branch-and-bound algorithm. Our proposed dominance relation is effective when a small cut between assigned and unassigned tasks is met. To exploit the effectiveness of the dominance relation before space overflow, the tasks should be enumerated in an order such that heavily communicated tasks will be enumerated consecutively.

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 whole 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 be determined as a sequence of all leaf nodes by performing depth first traversal. For instance, the partitioning process for the task graph in Fig. 1 is depicted in Fig. 7. Following this result, we obtain the enumeration order that has been used for illustration in previous discussion.

4.2. The optimal branch-and-bound algorithm and its complexity

The branch-and-bound algorithm is shown in Fig. 8. This is based on the A* traversal scheme with addition of the dominance relation for space pruning. The dominance relation is applied to test whether a child of the currently visited partial assignment A_i can be pruned or not (cf. Step 4 in Fig. 8). We pick the child of A_i with minimum L(·) as the killer to examining remaining child of A_i (cf. Step 3.3 in Fig. 8). The partial assignment child[k] is pruned and will not be inserted into the ActiveSet if the killer dominates child[k].

The time complexity encountered by both A*-algorithm and our proposed algorithm to visit a branching state is as follows. Let n be the number of tasks and m be the number of processors. In the implementation, AC_k(t_i \rightarrow p_i, A') is for all children of the visited partial assignment A_i are calculated and stored in the AC-tables first. Remaining quantities (TA, TAL, L, TADL) can then be calculated by looking up the AC-table. By examining each edge in the task graph and accumulating communication costs into the AC-table, the AC-table for a partial assignment can be established in O(n^2 \cdot m) and AC-tables for all children can be established in O(n^2 \cdot m^2). The quantity TAL_k(·) can be calculated in O(m \cdot n) and L(·) for a partial assignment can be calculated in O(m^2 \cdot n). The time complexity to compute L(·) for all children is O(m^3 \cdot n). The ActiveSet is implemented as a heap and the time complexity to insert all children is O(m \cdot n \cdot (log m)). Both A*-algorithm and our proposed algorithm encounters the O(n^2 \cdot m^2) + O(m^3 \cdot n) time complexity to visit a branching state.

Our proposed algorithm encounters an additional time complexity to test for the dominance relation. In general, the time complexity to compute TADL_k(A_1, A_2) is O(m \cdot n). But we notice that, in the proposed algorithm, the two partial assignments A_1 and A_2 for testing differ in only the assignment of one task, say t_a. For processor p_k that A_1(t_a) ≠ p_k and A_2(t_a) ≠ p_k, we have AC_k(t_i \rightarrow p_i, A_1) = AC_k(t_i \rightarrow p_i, A_2) for p_i ≠ p_k. The quantity TADL_k(A_1, A_2), for p_k : A_1(t_a) ≠ p_k and A_2(t_a) ≠ p_k, can be written as

$$
TADL_k(A_1, A_2) = TA(A_2) - TA(A_1) + \sum_{t_i \text{ not assigned}} (AC_k(t_i \rightarrow p_k, A_2) - AC_k(t_i \rightarrow p_k, A_2)).
$$

An implementation of procedure DominantTest with time complexity O(n \cdot m) can be achieved and the time complexity to test the dominance relation for all children of A_i is O(n \cdot m^2). The time complexity for the proposed algorithm to visit a branching state is thus O(n^2 \cdot m^2) + O(m^3 \cdot n) + O(n \cdot m^2) = O(n^2 \cdot m^2) + O(m^3 \cdot n). Since the additional complexity is an order lower than the base complexity required, it is worthwhile to spend the additional complexity for reducing number of states traversed.

The space complexity required by the proposed algorithm is the same as that of the A*-algorithm. Besides the ActiveSet, the AC-tables for all children of a partial assignment occupy a space of O(n \cdot m^2). Note that AC-tables can be reused among different branching states and the space does not grow with the number of branching states traversed.

4.3. Correctness proof of the proposed branch-and-bound algorithm

We now show that our proposed algorithm returns an optimal assignment. We first show that the traversal procedure reserves some optimal assignments in the future search space. A complete assignment A_c is said to be in the future search space of ActiveSet(i) if either A_c ∈ ActiveSet(i) or there exists a partial assignment
$A_c \in ActiveSet^{(t)}$ such that $A_c$ can be extended from $A_s$. Provided that some optimal assignments survived in the future search space, we show that the terminate condition implies the optimality of the solution obtained.

**Lemma 2.** During the traversal, there are always some optimal assignments survived in the future search space.

**Proof.** If the pruned partial assignment can be extended to be an optimal assignment, the definition of the dominance relation implies that the killer can also be extended to be an optimal assignment. The branch-and-bound algorithm always inserts the killer into the ActiveSet (cf. Step 3.4 in Fig. 8) and hence there are always some optimal assignments that survive in the future search space. □

**Theorem 2** (Correctness of our proposed algorithm). Our proposed branch-and-bound algorithm will end up with an optimal assignment.

**Proof.** A 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 $ActiveSet^{(t)}$ for the last iteration. Lemma 2 states the existence of an optimal assignment $A_{opt}$ in the future search space of $ActiveSet^{(t)}$. Thus, we have $cost(A_c) > cost(A_{opt})$ since $A_{opt}$ is optimal. Let $A_s$ be the ancestor of $A_{opt}$ (or $A_{opt}$ itself) in $ActiveSet^{(t)}$. By the definition of $L(\cdot)$, $L(A_s) < cost(A_{opt})$. And hence $L(A_s) < cost(A_{opt}) < cost(A_c) = L(A_c)$. However, $A_c$ is the one with minimum $L(\cdot)$ in $ActiveSet^{(t)}$. This means $L(A_s) < L(A_c)$. This produces a contradiction and hence proves this theorem. □

5. Experiments and evaluation

We evaluate the proposed task allocation algorithm by feeding it with several configuration samples generated randomly. The test samples cover different degrees of task clustering and parallelism to test the effectiveness of the dominance relation.

5.1. Test samples generation

We randomly generate a set of task graphs and map the task graphs to selected hierarchical machine architectures. In generating task graphs, the distribution on weights and edge densities are chosen to cover all degrees of clustering and 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 the dominance relation is tested among various degrees of task clustering and parallelism.

Following the scheme in (Bowen et al., 1992), 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 middle-level sub-graph. The middle-level sub-graphs are then randomly combined as a final task graph.

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

- $\frac{E}{C} = \frac{\text{Average execution weight among all tasks}}{\text{Average communication weight among all edges}}$
- edge density = Probability that two vertices in different sub-graphs are 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 decided according to the edge density. Once an edge is really chosen, the weight on the edge is determined according to $E/C$ ratio.

We denote the attributes of a task graph as a tuple of $E/C$ ratio and an edge density. Combination at each level has its own $E/C$ ratio. For example, a task graph may be generated as follows: (1) selecting sub-graphs with $E/C = 1$ as the lowest level sub-graph, (2) combining lowest level sub-graphs to form a middle level sub-graph with $E/C = 5$ and edge density = 20%, (3) combining middle 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%.

The degree of clustering on 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 on $E/C$ ratio between adjacent levels is large, and (2) the sub-graphs are combined in 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 for the task allocation program is the machine configuration. The machine configuration for experiments is hierarchical machine similar to Fig. 2(a) but with a larger size and different latency. In the experiment, each machine consists of three subnet, and each subnet consists of three processors. We fix the intra-subnet latency to be one. The inter-subnet latency varies from 5 to 20. On 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.

5.2. Evaluation metrics

We compare the performance of the proposed task allocation algorithm to A*-algorithm. Let $n$ be the number of tasks and $m$ be the number of processors. According to Section 4.2, we estimate the time for the A*-algorithm to visit a branching state as $C_1 \times n^2 \times m^2 + C_2 \times n \times m^3$, where $C_1$ and $C_2$ are some constants. Similarly, the time for the proposed algorithm to visit a branching state is $C_1 \times n^2 \times m^2 + C_2 \times n \times m^3 + C_3 \times n \times m^2$ for some constant $C_3$. Let $ST$ be the number of states traversed by the proposed algorithm and the number of states traversed by the A*-algorithm is $R$ times $ST$. The speed-up is thus

$$\text{Speed-up} = \frac{(C_1 \times n^2 \times m^2 + C_2 \times n \times m^3) \times R \times ST}{(C_1 \times n^2 \times m^2 + C_2 \times n \times m^3 + C_3 \times n \times m^2) \times ST} = \frac{R}{1 + \frac{C_3}{C_1 \times n + C_2 \times m}}. $$

We estimate the constants on a PC with PentiumPro processor and find that $C_1 \gg C_3$. This is because establishing AC-tables is mainly the multiplication operations and testing the dominance relation is mainly table lookup with the add operations. The speed-up can be approximated by the ratio on traversed states when $n$ and $m$ exceed certain threshold. We thus take the metric to evaluate the effectiveness of the dominance relation as follows.

$$\text{speed-up} \approx R \frac{\text{number of states traversed by A*-algorithm}}{\text{number of states traversed by the proposed algorithm}}.$$

5.3. Experiment result

The performance is 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, hence resulting in 24 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 degree of clustering on tasks and parallelism to test the effectiveness of the dominance relation.

Fig. 9 shows the experiment results. Experiment results on different machine configurations are depicted in different charts. We take the harmonic mean on the speed-up for each set of 10 task graphs generated under the same $E/C$ tuple and edge density. The speed-up ranges from 1.02 to 1.68, depending on the degree of clustering on tasks and parallelism. As expected, the curves show that the dominance relation is effective when the tasks can be clearly clustered into groups and the parallelism becomes large.

6. Conclusion

In this paper, we proposed a two-stage task allocation algorithm to obtain an optimal solution for the task allocation problem. The first stage is a recursive partitioning procedure for determining the task enumeration order to exploit the effectiveness of the dominance relation. The second stage is a branch-and-bound algorithm using the dominance relation to prune the search space such that the time and space required can be significantly reduced. We evaluate the performance of our proposed branch-and-bound algorithm by comparing it to the A*-algorithm for task allocation (Shen and Tsai, 1985).
The dominance relation is the key to the efficient task allocation. The key observation is that tasks can be clustered according to the communication weights. When the boundary of task clustering—a small cut—is met, whether a sub-tree in the state-space tree needs further traversal becomes clear. The dominance relation determines whether a partial assignment can be pruned according to the edge weights contributed to the cut.

The proposed task allocation algorithm is evaluated on randomly generated task graphs. Compared to the $A^*$-algorithm for task allocation (Shen and Tsai, 1985), the speed-up of our proposed algorithm ranges from 1.02 to 1.68, depending on whether the tasks can be clearly clustered into groups. This shows the effectiveness of the proposed dominance relation.

7. For further reading


References


Yung-Cheng Ma received the B.S. degree in computer science and information engineering from the National Chiao-Tung University, Hsinchu, Taiwan, Republic of China in 1994. Currently he is pursuing the Ph.D. degree in computer science and information engineering at the National Chiao-Tung University, Hsinchu, Taiwan, Republic of China. His research interests include computer architecture, parallel and distributed systems, and information retrieval.

Chung-Ping Chung received the B.E. degree from the National Cheng-Kung University, Hsinchu, 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. From 1998, he joined the Computer and Communications Laboratories, Industrial Technology Research Institute, ROC as the Director of the Advanced Technology Center, and then became the Consultant to the General Director. He is expected to return to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.