[Paper-1999-12-14]

行政院國家科學委員會專題計畫成果報告
標竿程式分析、測試與效能評估
Benchmarking and Performance Evaluation

計畫編號：NSC- 88-2213-E-009-040
執行期限：87 年 8 月 1 日至 88 年 7 月 31 日

主持人：鍾崇斌　　交通大學資訊工程學系

中文摘要--作為整合型計劃"單晶片多處理機設計之研究"的一個子計劃，本計劃提出一工作配置方法，針對以多處理機晶片為核心之多處理機系統，尋求平行程式之最佳配置。我們採用狀態空間搜尋法，並提出修剪規則(pruning rule)。此修剪規則由 dominance relation 與分群方法所構成。搜尋程序依據此修剪規則，將不合分群原則的部分解(partial solution)排除，並將最佳解保留於未來將搜索的狀態空間中。與過去的狀態空間搜尋方法相比，此修剪規則有效的減少搜尋最佳解所需的時間與空間，並使搜尋程序能在短時間內，逼近至近似最佳解。實驗結果顯示，所提的修剪規則，使狀態空間搜尋方法，成為實用的方法。

Abstract – As part of the joint project "Study of Single Chip Multiprocessors Design," we propose a task allocation algorithm that aims at finding an optimal task assignment for any parallel programs on the MP-chip based multiprocessor systems. The main theme of our approach is to traverse a state-space tree that enumerates all possible task assignments. The key idea of the efficient task allocation algorithm is that we apply two pruning rules 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 on 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.

1. Introduction

As part of the joint project "Study of Single-Chip Multiprocessor Design," the goal of this project is to optimize the benchmark program for the MP-chip based system. We investigate the task allocation problem of mapping a parallel program to a multiple MP-chip systems. A parallel program is modeled as a node- and edge- weighted undirected graph, called 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.

The main theme of our approach is to traverse a state-space tree that enumerates all possible task assignments. The key idea of the efficient task allocation algorithm is that we apply two pruning rules 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 on 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. Experiment shows that our proposed pruning rule makes state-space searching approach feasible for practical use.

2. Modeling the Task Allocation Problem

2.1. Formulating the task allocation problem

We follow [1][2][3] to formulate the task allocation problem.

The input of a task allocation algorithm is a *task graph* $G(T,E,e,c)$ and a *machine configuration* $M(P,d)$. A parallel program is represented as a *task graph* $G(T,E,e,c)$ in which a node represents a program module, called a *task*, and an edge represents communication between tasks. Weight on a task, denoted $e(t_i)$, represents the execution time of the task and weight on an edge, denoted $c(t_i, t_j)$, represents the amount of data transferred between the two tasks. The *machine configuration* is represented as $M(P,d)$. $P=\{p_0,p_1,\dots p_{m-1}\}$ is the set of all processors. For each pair of processors $(p_k,p_l)\in P$, 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.

The output of the task allocation algorithm, 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. 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. The cost of an partial/complete assignment A is the *turn-around time* of the last processor finishes its execution. The *turn-around time* of processor p_k, denoted $TA_k(A)$, is 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, defined as follows:

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

2.2. Transforming to the state-space searching problem

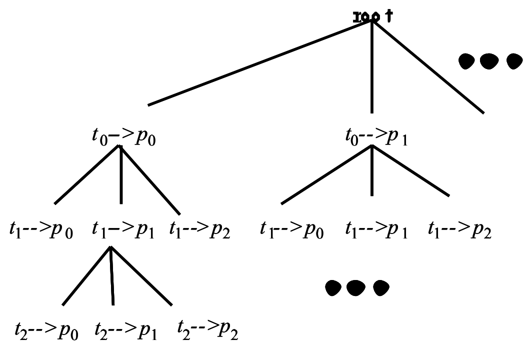

Figure-1. State-space tree

We traverse the state-space tree, as illustrated in Figure-1, to find an optimal assignment. During the traversal, an *active set*, denoted *ActiveSet*, is used to keep track of all partial/complete assignments that have been explored but not visited. We follow the approach in Shen and Tsai[1] 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. The partial/complete assignment in *ActiveSet* with minimum $L(\bullet)$ is removed for visiting in each iteration. $L(A)$ is computed according to the *additional cost* of assigning tasks not assigned in A, defined as follows:

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

$$AC_k(t_j \rightarrow p_l, A) = \sum_{t_i:A(t_i)=p_k} c(t_i,t_j) * d(p_k, p_l) \text{ if } p_k \neq p_l$$

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

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

3. Dominance Relation for State-Space Pruning

We first develop a *dominance relation* [6] to serve as the basis for developing pruning rules. The proposed dominance relation checks whether a partial assignment can be pruned or not according the estimated *turn-around time difference lower-bound*:

$$TADL_k(A_1, A_2) \equiv TA_k(A_2) - TA_k(A_1) + \sum_{t_i \in S} \left(\min_{p_l \in P} (AC_k(t_i \rightarrow p_l, A_2) - AC_k(t_i \rightarrow p_l, A_1)) \right)$$

Theorem 1 (Dominance relation for space pruning). Let A_1 and A_2 be two partial assignments assigning 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.

4. Space Pruning by Detecting the Clustering on Tasks

The dominance relation proposed in Section 3 is only effective when a small cut can be found. To overcome this drawback, we develop a further pruning rule that integrates the detection of clustering on tasks as well as the dominance relation.

```
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
        $PA_i$
    2)  /* exclude extensions violating PA */
        2.1) success←False
        2.2) for each processor $p_k$ do
                 if $TAL_k$($A$, violate $PA$)≥$cost$($A_u$) then
                     success ←True
                     break
        2.3) if success=False then $PA_i$←$P$
    3) $A_d$←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
```

Figure-2. Algorithm to examine the partial assignment using the pruning rule

```
Algorithm Compute_PA($A$,$A_k$)
•   input:
    –   $A$, $A_k$ : partial assignments,
        depth($A_k$)≥depth($A$)
•   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 $p_k$ | $d(p_k, p_c) \le d(A_k(t_i), p_c)$ }
        else $PA_i \leftarrow P$
```

Figure-3. Algorithm to predict the clustering on tasks

Figure-2 presents the algorithm to examining a partial assignment A. It calls procedure Compute_PA, presented in Figure-3, to detect the task clustering. Two additional inputs are required: (1) partial assignment A_k—called the **killer**—reflecting the clustering on tasks, and (2) complete assignment A_u serve as an upper bound on the optimal cost, which is obtained by the greedy search.

We determine whether the candidate partial assignment A can be pruned or not according to the following quantities:

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

$$TADL_k(A_d, A, PA) = TA_k(A) - TA_k(A_d) + \sum_{t_i \text{ not assigned}} \left(\min_{p_l \in PA_i} \left(AC_k(t_i \to p_l, A) - AC_k(t_i \to p_l, A_d) \right) \right)$$

The killers are obtained as follows. To increase the possibility of pruning a partial assignment, we may find multiple killers, called a *KillerSet*, instead of only one killer. To obtain the killers, a link to the deepest descendant node is associated with each visited partial assignment. For each visited partial assignment A_a, we associate a pointer *deep*(A_a) pointing to the deepest partial assignment visited in the sub-tree of A_a. If two or more partial assignments in the same level of the state-space tree are visited, *deep*(A_a) points to the first one visited. The *KillerSet* is the set of all *deep*(A_a) for each ancestor of A along with the complete assignment A_u.

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

5. Task Allocation using Branch-and-Bound Method with Preprocessing Stage

```
Algorithm BB-Alloc($G$,$M$)
•   /* initialization phase */
    –   $L$(root of the state-space tree) ← 0
    –   $ActiveSet \leftarrow$ {root of the state-space tree}
    –   Obtain $A_u$ by perform 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 $A$ */
            $deep(A) \leftarrow A$
            for each $A_a$: ancestor of $A$ in the
                state-space tree do
                if depth($A$)>depth($deep(A_a)$)
                    then $deep(A_a) \leftarrow A$
       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 $A_u \leftarrow A_c$
    2) if $A_v$ is a complete assignment then $A_u \leftarrow A_v$
       and terminate the traversal by return $A_u$
    3) /* check if the sub-tree of $A$ needs further
       traversal */
       $KillerSet \leftarrow \{deep(A_a) \mid A_a$ is an ancestor
           of $A_v$ in the state-space tree$\} \cup \{A_u\}$
       prune ← False
       for each $A_k \in KillerSet$ 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
                $ActiveSet$
```

Figure-4. The branch-and-bound algorithm for task allocation

To exploit the effectiveness of the pruning rule, tasks should be enumerated in such an order that tasks with high communication are enumerated first. This can be achieved by performing the max-flow min-cut algorithm recursively.

The branch-and-bound algorithm is described in Figure-4. Optimal assignment will be obtained if no overflow on the time and space required.

Theorem 2 (Correctness of our proposed algorithm). Our proposed branch-and-bound algorithm will end up with an optimal assignment if neither overflow on the *ActiveSet* nor time-out occurs.

The *ActiveSet* is implemented as an array of heaps. To assign n tasks to m processors, the *ActiveSet* is a two dimensional array heap[i][j] for $1 \le i \le n$ and $1 \le j \le m$. A (partial) assignment assigning tasks $\{t_0, t_1, \ldots, t_{i-1}\}$ to j of the m 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* assigning $\{t_0, t_1, \ldots, t_{i-1}\}$ using j processors exceeds size(i,j), the one in heap[i][j] with maximum $L(\bullet)$ will be dropped.

6. Experimental Evaluation

The performance and allocation quality are evaluated using 240 task graphs and three hierarchical machine configurations. On generating task graphs, the distribution on weights and edge density are chosen to cover all degree of clustering on tasks. On selecting the machine configuration, the processor distances are chosen such that the parallelism in optimal assignments ranges from using a few processors within a MP-chip to using all processors across multiple MP-chip.

We use the term performance to refer to the execution time that the task allocation algorithm spends to obtain an optimal assignment without time and space constraint. The metric is:

Speed-up=(number of states traversed by the A*-algorithm)/(number of states traversed by our proposed algorithm)

The evaluation shows that the speed-up ranges from 1.03-2.20, depending on the degree of clustering on tasks and parallelism.

We use the term allocation quality to refer to how good the complete assignment returned by the task allocation algorithm is subject to time and space constraint. The metric is:

Allocation quality=(cost of the complete assignment returned)/(cost of the optimal assignment)

Time and space complexity are controlled by setting *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 $i*j$. Each test yields an allocation quality within 1.14.

7. Conclusion

In this report, we proposed a two-stage task allocation algorithm that aims at finding an optimal assignment. The first stage is a recursive partitioning procedure to form a task enumerating order such that we can exploit the task clustering property. The second stage is a branch-and-bound algorithm with pruning rule to traverse the state-space tree. The pruning rules keep some optimal assignments in the future search space and hence an optimal assignment will be obtained if neither time-out nor overflow on the *ActiveSet* occurs.

The key idea to the efficient task allocation is the pruning rule, which is a combination of a dominance relation and task clustering heuristic. The pruning rule reduces the time and space required to obtain an optimal assignment. Moreover, cooperated with the space efficient *ActiveSet* design, the traversal procedure can reach a near optimal assignment within a low order polynomial number of states.

The task allocation algorithm is evaluated on randomly generated task graphs. The experiment shows that our proposed pruning rule is effective to prune the search space and lead the traversal to a near optimal assignment within a low order polynomial number of states. This makes the state-space searching approach feasible for practical use.

Reference:

[1] C. C. Shen and W. H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transactions on Computers*, 34(3):197-203, March 1985.

[2] C. C. Hui and S. T. Chanson, "Allocating Task Interaction Graphs to Processors in Heterogeneous Networks," *IEEE Transactions on Parallel and Distributed Systems*, 8(9): 908-925, September 1997.

[3] N. S. Bowen, C. N. Nikolaou, A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Systems," *IEEE Transactions on Computers*, 41(3): 257-273, March 1992.

[4] D. T. Peng and K. G. Shin, "Optimal Scheduling of Cooperative Tasks in a Distributed System Using an Enumerative Method," *IEEE Transactions on Software Engineering*, 19(3):253-267, 1993.

[5] C. C. Hui and S. T. Chanson, "Allocating Task Interaction Graphs to Processors in Heterogeneous Networks," *IEEE Transactions on Parallel and Distributed Systems*, 8(9):908-925, 1997.

[6] W. H. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *Journal of the Association for Computing Machinery* 21(1): 140-156, 1974