A Branch-and-Bound Algorithm for Scheduling of Data-Parallel Tasks

Yang Liu, Lin Meng, Ittetsu Taniguchi, Hiroyuki Tomiyama
Department of Electronic and Computer Engineering Ritsumeikan University
1-1-1 Noji-Higashi, Kusatsu, Shiga 525-8577, Japan

Abstract — This paper studies a task scheduling problem which schedules a set of data-parallel tasks on multiple cores. Unlike most of previous literature where each task is assumed to run on a single core, this work allows individual tasks to run on multiple cores in a data-parallel fashion. Since the scheduling problem is NP-hard, a couple of heuristic algorithms which find near-optimal schedules in a short time were proposed so far. In some cases, however, exactly-optimal schedules are desired, for example, in order to evaluate heuristic algorithms. This paper proposes an exact algorithm to find optimal schedules in a reasonable time. The proposed algorithm is based on depth-first branch-and-bound search. In the experiments, the proposed algorithm could successfully find optimal schedules for task-sets of 50 tasks in a practical time.

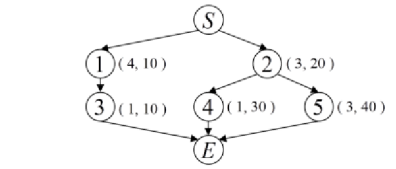
Keywords — task scheduling; multicore; data parallelism; branch-and-bound

I. INTRODUCTION

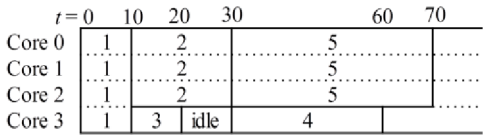
Due to the wide deployment of multicore architecture not only in general-purpose processors but also in embedded processors, task scheduling has now become a more important problem than ever. Given a set of tasks with data dependency, a task scheduling problem decides when and on which core each task is executed in such a way that the overall schedule length is minimized, while meeting constraints on flow dependency among tasks and the number of available cores.

In order to fully utilize the potential parallelism of multicore architectures, both task parallelism (i.e., inter-task parallelism) and data parallelism (i.e., intra-task parallelism) need to be exploited [1][2]. Task parallel execution is achieved by executing multiple independent tasks on different cores simultaneously. On the other hand, data parallel execution is achieved by executing the same task with different data on multiple cores simultaneously. This paper addresses a task scheduling problem which takes into account both task- and data-parallelisms.

In general, task scheduling problems belong to the class of NP-hard [3], and there exists no polynomial-time algorithm which always yields optimal solutions, unless P = NP. In the past, several heuristic algorithms were proposed for scheduling of data parallel tasks [4][5]. In some occasions, however, it is still desirable to obtain optimal schedules, for example, in order to evaluate heuristic algorithms. For a task scheduling problem which only considers task parallelism, several exact algorithms to find optimal schedules were developed [6][7][8]. These algorithms assume that individual tasks run on a single core. To the best of our knowledge, no algorithm was proposed for the task scheduling with both task- and data-parallelisms.



(a) A task graph



(b) An optimal schedule

Figure 1. A scheduling example

In this paper, we propose an exact algorithm for task scheduling with both task- and data-parallelisms. Unlike previous exact scheduling algorithms, this work allows individual tasks to run on multiple cores in a data-parallel fashion. The proposed algorithm is based on a branch-and-bound strategy. A set of rules are proposed to efficiently prune branches.

This paper is organized as follows. Section II formally describes a scheduling problem addressed in this paper, and Section III proposes a scheduling algorithm. Experiments are presented in Section IV.

II. PROBLEM DEFINITION

This section defines a task scheduling problem addressed in this paper.

A. Problem Description

This work assumes homogeneous multicore processors. An application is modeled as an acyclic directed graph (DAG), so called a task graph, where a node represents a task and a directed edge represents a flow dependency between two tasks. Figure 1 (a) shows an example of a task graph. In this graph, tasks labeled "*S*" and "*E*" are dummy tasks which do not perform any meaningful computation. Tasks *S* and *E* denote a start point and an exit point of the application, respectively. Two integer values are associated with each task. The first number denotes the degree of data parallelism of the task. In other words, the number denotes the number of cores which are necessary to run the task. We assume that the degree of data parallelism is decided by programmers, and how to decide it is out of scope of this paper. The latter number on each node denotes the execution time of the task. For example, task 1 runs on 4 cores, and it takes 10 time units to complete the task.

Given a task graph, task scheduling decides when and on which core each task is executed in such a way that the overall schedule length is minimized, while meeting constraints on flow dependency among tasks and the number of available cores. Figure 1 (b) shows one of optimal schedules on four cores for the task graph in Figure 1 (a).

B. ILP Formulation

The task scheduling problem described above can be formulated as an integer linear programming (ILP) problem.

Let $time_i$, $start_i$, and $finish_i$ denote the execution time, start time and finish time of task *i,* respectively. par_i denotes the data parallelism, meaning that task *i* must be mapped onto par_i cores. $flow_{i1,i2}$ denotes a flow dependency between tasks $i1$ and $i2$. $flow_{i1,i2}$ is 1 if task $i1$ must precede task $i2$, otherwise 0. $map_{i,j}$ denotes mapping of tasks on cores. $map_{i,j}$ is 1 if task *i* is mapped to core *j*, otherwise 0.

Then, the task scheduling problem is formally defined as follows: Given $time_i$, par_i and $flow_{i1,i2}$, decide $start_i$, $finish_i$and $map_{i,j}$ which minimize the objective function (1), while meeting the constraints (2), (3), (4) and (5).

$$\text{Minimize:} \quad \text{Max}(finish_i) \tag{1}$$

Subject to:

$$\forall i \quad \sum_j map_{i,j} = par_i \tag{2}$$

$$\forall i \quad finish_i = start_i + time_i \tag{3}$$

$$\forall i1, i2, j \quad map_{i1,j} + map_{i2,j} \leq 1 \vee finish_{i1} \leq start_{i2} \vee finish_{i2} \leq start_{i1} \tag{4}$$

$$\forall i1, i2 \quad flow_{i1,i2} = 1 \rightarrow finish_{i1} \leq start_{i2} \tag{5}$$

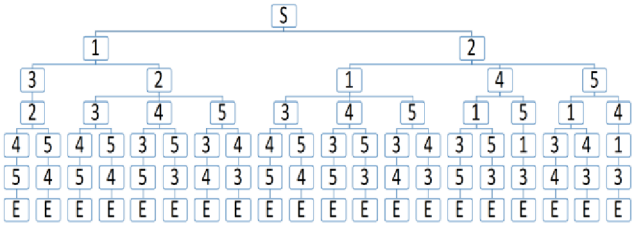


Figure 2. A branching tree

Optimal scheduling results can be obtained by solving the ILP formulas, but it is not practical for large task sets. In the next section, we propose an efficient branch-and-bound algorithm to find the optimal schedules.

III. The Proposed Algorithm

This section proposes a branch-and-bound algorithm for the scheduling problem defined in the previous section. The proposed algorithm basically explores all possible solutions by a depth-first search, and prunes non-optimal solution spaces during the search.

A. Depth-First Search

Our algorithm uses a branching tree to systematically enumerate all possible schedules. For example, Figure 2 shows a branching tree for the task graph in Figure 1 (a). In the tree, each node represents a task, and a branch between two nodes denotes that the parent task is scheduled no later than the child task. A path from the root to a leaf denotes a schedule. For example, a path $(\text{S} \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow \text{E})$ in Figure 2 denotes the schedule shown in Figure 1 (b) [1].

Our algorithm travels the branching tree from the root to leaves in a depth-first order. However, traveling all nodes in the branching tree has time complexity of $O(n!)$, which is not practical for large task graphs. The rest of this section present four rules to prune unnecessary branches.

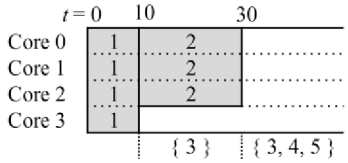
B. Pruning Partial Schedules with Same Tasks

Let us consider the branching tree in Figure 2. Assume that our algorithm already visited partial schedule $(1 \rightarrow 2)$ and now we have reached $(2 \rightarrow 1)$. Note that the two partial schedules contain the same tasks with different orders. If we compare the two partial schedules, we can figure out that $(2 \rightarrow 1)$ cannot be better than $(1 \rightarrow 2)$, and thus, we can prune further branches under $(2 \rightarrow 1)$.

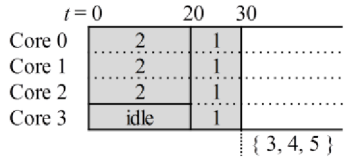
How to compare the two partial schedules is as follows.

Figure 3 (a) and (b) show time charts of partial schedules $(1 \rightarrow 2)$ and $(2 \rightarrow 1)$, respectively. In Figure 3 (a), one of the four cores is available at time 10, and then, task 3 is schedulable. Here, a task is schedulable if both of the following two conditions hold:

[1] Paths $(\text{S} \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow \text{E})$, $(\text{S} \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow \text{E})$ and $(\text{S} \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow \text{E})$ also result in the same schedule as shown in Figure 1 (b).



(a) Partial schedule $(1 \rightarrow 2)$



(b) Partial schedule $(2 \rightarrow 1)$

Figure 3. Partial schedules with same tasks

- All flow dependencies are solved.
- The number of available cores is enough to run the task.

Similarly, tasks 3, 4 and 5 are schedulable at time 30 in Figure 3 (a). In Figure 3 (b), tasks 3, 4 and 5 are schedulable at time 30. Before time 30, no task is schedulable since no core is available.

Now, we see that, at any time point, a set of schedulable tasks in partial schedule $(2 \rightarrow 1)$ is a subset of that in partial schedule $(1 \rightarrow 2)$. For example, at time 10, a set of schedulable tasks in partial schedule $(2 \rightarrow 1)$ is empty, which is a subset of $\{3\}$. Then, it is guaranteed that no schedule under partial schedule $(2 \rightarrow 1)$ is better than the best schedule under $(1 \rightarrow 2)$, and therefore, branches under $(2 \rightarrow 1)$ can be pruned.

In our algorithm, when we visit a new partial schedule, in other words, when we visit a new node in the branching tree, we look-up previously-visited partial schedules with same tasks, and compare their schedulable task sets. If the schedulable task set of one partial schedule is always a subset of the other, we prune the former partial schedule.

C. Scheduling Exclusive Task First

Let us consider the task graph in Figure 1 again. Initially, either task 1 or 2 is schedulable at time 0. In this case, scheduling task 1 first leads to an optimal schedule in the following reason.

Since task 1 requires all of four cores, this task cannot be executed in parallel with any other tasks. We refer to a task as an *exclusive* task if the task cannot run in parallel with any other tasks which are not yet scheduled. Task 1 is an exclusive task. On the other hand, task 2 is not exclusive since task 2 can run in parallel with task 3.

Delaying execution of exclusive tasks which can be scheduled at the earliest cannot minimize the schedule length. Our algorithm schedules exclusive tasks as early as possible. When visiting a node, and if one of the branches goes to an exclusive task with the earliest start time, branches to the other tasks are pruned.

D. Reducing Meaningless Idle Time

Let us consider partial schedule $(1 \rightarrow 2)$ in the branching tree shown in Figure 2. There are three branches from task 2, going to tasks 3, 4 and 5. If we look at the time chart in Figure 3 (a), it is obvious that the branch to task 3 is the best among the three. The earliest start time of task 4 and that of task 5 are both time 30 because of the flow dependencies. On the other hand, the earliest finish time of task 3 is time 20, which is earlier than the earliest start time of the other tasks. Therefore, delaying execution of task 3 produces meaningless idle time.

During traveling a branching tree, if the earliest finish time of a child task is earlier than or equal to the earliest start time of the other children, only the former task is visited and the other branches are pruned.

E. Pruning based on Lower Bound

Similar to typical branch-and-bound algorithms, our algorithm keeps a temporarily-optimal schedule and updates it when a better schedule is found. When branching to a child, our algorithm calculates the lower bound of schedule length. If the lower bound is longer than the length of the temporarily-optimal schedule, the branch is pruned.

When our algorithm visits a new node in the branching tree, we use two simple formulas as follows, in order to check the lower bound of the schedule under the node.

$$\sum_j AT_j + \sum_{i \epsilon \varphi} P_i \times T_i \geq N \times TOL \quad (6)$$

$$\sum_j AT_j - \sum_{i \epsilon \omega} P_i \times T_i \geq TIT \quad (7)$$

In the formulas, AT_j denotes the available time of core j. For example, in Figure 3 (a), AT_j is 30 for $0 \leq j \leq 2$, and $AT_3 = 10$. φ is a set of tasks which are not yet scheduled. P_i and T_i denote the degree of data parallelism and execution time of task i, respectively. N is the number of cores, and TOL is the length of the temporarily-optimal schedule. If formula (6) holds, the schedule length under this node cannot be shorter than TOL, and therefore further branches are pruned.

In formula (7), ω denotes a set of tasks which have already been scheduled. TIT represents the total idle time in the temporarily-optimal schedule, and is defined as follows.

$$TIT = N \times TOL - \sum_{i \epsilon all\ tasks} P_i \times T_i \quad (8)$$

Formula (7) checks if the total idle time of the current partial schedule is larger than TIT or not. If yes, further branches under the partial schedule are pruned.

F. Selection of Branch

So far, four rules to prune branches are described. Another important issue in the depth-first branch-and-bound search is how to select a task to go first when multiple child tasks exist.

Out of the children, our algorithm selects the child task which has the earliest start time. In case there exist multiple tasks with the same start time, we select a task based on the PCS strategy which was presented in [4].

IV. EXPERIMENTS

We implemented our proposed scheduling algorithm in C++, and conducted two sets of experiments to test the effectiveness of the proposed algorithm.

In the first experiments, we use 20 sets of 10 tasks, derived from Standard Task Graph (STG) [9]. An integer linear programming (ILP) technique (see Section II) was used as a counterpart to our algorithm. Although the ILP technique is guaranteed to yield optimal schedules, it takes a long time which is often unacceptable. In order to solve the ILP problems, IBM ILOG CPLEX 12.5 was used. The experiments were conducted on dual Xeon processors (E5-2650, 2.00Hz) with 128GB memory.

Table 1 shows scheduling results for 20 task graphs with 10 tasks on 4 cores. ILP and B&B denotes the ILP technique using CPLEX and our branch-and-bound algorithm, respectively. The results in the table show that our algorithm yields the same schedule length as the ILP techniques in any case. Although we have not mathematically proved the correctness of our algorithm yet, our algorithm always found the optimal schedule as long as we tested.

As shown in Table 1, in any cases of 10 tasks, our branch-and-bound algorithm found optimal schedules within a second. On the other hand, the runtime of CPLEX significantly varied depending on the task graph. In the worst case, it took more than 60 fours for CPLEX to find the optimal schedule for 10 tasks.

In the next set of experiments, we compared our branch-and-bound algorithms with two existing heuristic ones. One is the PCS algorithm [4] and the other is the dual-mode algorithm [5]. We used 20 sets of 50 tasks from [9]. With the three algorithms, the task sets are scheduled on 4 cores. The results are shown in Table 2.

The results show that our algorithm always found the best schedule among the three algorithms. The runtimes of the PCS and dual-mode algorithms were always less than 1 second. On the other hand, the runtime of our branch-and-bound algorithm significantly varied depending on the task graph. In the worst case, it took more than 24 fours for our algorithm to find the optimal schedule for 50 tasks. However, it should be noted that, for 16 task graphs out of 20, the runtime of our algorithm is less than 1 minute.

V. CONCLUSIONS

In this paper, we proposed a branch-and-bound algorithm for a task scheduling problem which takes into account both task-parallelism and data-parallelism. We presented four rules to prune non-optimal branches. The experiments show that our algorithm could find best schedules in a practical time.

Table 1. Results for task graphs with 10 tasks on 4 cores

Task graph ID	Schedule Length		Runtime (sec)	
	ILP	B&B	ILP	B&B
rand0000	32	32	6,823	<1
rand0001	43	43	21,788	<1
rand0002	26	26	60,012	<1
rand0003	30	30	71,678	<1
rand0004	36	36	2,588	<1
rand0005	75	75	40,054	<1
rand0006	70	70	46,245	<1
rand0007	94	94	50,019	<1
rand0008	121	121	6,115	<1
rand0009	79	79	58,830	<1
rand0010	23	23	55,539	<1
rand0011	33	33	55,068	<1
rand0012	33	33	15,171	<1
rand0013	31	31	42,571	<1
rand0014	53	53	44,250	<1
rand0015	81	81	<1	<1
rand0016	77	77	<1	<1
rand0017	100	100	41,675	<1
rand0018	72	72	<1	<1
rand0019	70	70	220,650	<1

In future, we plan to formally prove the correctness of our algorithm. Also, we will conduct more extensive experiments to demonstrate the effectiveness of our algorithm more solidly.

ACKNOWLEDGMENT

This work is in part supported by KAKENHI 15H02680.

REFERENCES

[1] S. B. Hassen, H. E. Bal, and C. J. H. Jacobs, "A task and data-parallel programming language based on shared objects," *ACM Trans. on Programming Languages and Systems.*, vol. 20, no. 6, 1998.

[2] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," *IEEE Trans. on Parallel and Distributed Systems.*, vol. 8, no. 11, 1997.

Table 2. Results for task graphs with 50 tasks on 4 cores

Task graph ID	Schedule Length			Runtime (sec)		
	PCS	Dual-mode	B&B	PCS	Dual-mode	B&B
rand0000	168	167	155	<1	<1	8
rand0001	220	211	202	<1	<1	<1
rand0002	173	170	162	<1	<1	<1
rand0003	194	194	181	<1	<1	114
rand0004	167	167	166	<1	<1	<1
rand0005	439	426	397	<1	<1	<1
rand0006	275	270	258	<1	<1	6
rand0007	357	354	339	<1	<1	88,100
rand0008	409	407	387	<1	<1	<1
rand0009	327	356	314	<1	<1	3
rand0010	131	131	128	<1	<1	50
rand0011	181	176	170	<1	<1	<1
rand0012	197	192	179	<1	<1	2
rand0013	186	192	178	<1	<1	7
rand0014	171	167	159	<1	<1	462
rand0015	376	373	345	<1	<1	<1
rand0016	318	319	292	<1	<1	<1
rand0017	377	378	359	<1	<1	6,800
rand0018	403	396	363	<1	<1	<1
rand0019	342	330	323	<1	<1	<1

[3] E. G. Coffman, *Computer and Job-shop Scheduling Theory*, Wiley, 1976.

[4] Y. Liu, L.Meng, I. Taniguchi, and H. Tomiyama, "Novel list scheduling strategies for task graphs with data parallelism," *International Journal on Networking and Computing*, vol. 4, no. 2, 2014.

[5] Y. Liu, L. Meng, I. Taniguchi and H. Tomiyama, "A dual-mode scheduling algorithm for task graphs with data parallelism," *Asia Pacific Conference on Circuits and Systems*, 2014.

[6] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. on Computers*, vol. C-33, no. 11, 1984.

[7] S. Fujita, "A branch-and-bound algorithm for solving the multiprocessor scheduling problem with improved lower bounding techniques," *IEEE Trans. on Computers*, vol. 60, no. 7, 2011.

[8] O. Sinnen, A. V. Kozlov, and A. Z. S. Shahul, "Optimal scheduling of task graphs on parallel systems," *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2008

[9] http://www.kasahara.elec.waseda.ac.jp/schedule/
(Last accessed: June 2016)