ILP-based Scheduling for Malleable Fork-Join Tasks

Kana Shimada
Ritsumeikan University
Japan
kana.shimada@tomiyama-lab.org

Ittetsu Taniguchi
Osaka University
Japan
i-tanigu@ist.osaka-u.ac.jp

Hiroyuki Tomiyama
Ritsumeikan University
Japan
ht@fc.ritsumei.ac.jp

ABSTRACT

This paper studies scheduling of malleable fork-join tasks. In our scheduling problem, each task can be partitioned into multiple sub-tasks, and the sub-tasks are scheduled independently. The optimal number of sub-tasks is determined during scheduling simultaneously. This paper formulates the scheduling problem as an integer linear programming problem.

KEYWORDS

Task scheduling, integer linear programming, multicore

1 INTRODUCTION

Multicore computing attracts an increasing attention because of its better power/performance efficiency than single-core computing. In multicore computing, task scheduling, which assigns tasks to cores and decides the execution order of the tasks on each core, has a significant impact on the system performance. Therefore, multicore task scheduling has been extensively studied in several decades.

A classic task scheduling problem assumes that each task is executed on a single core. Prior algorithms for the problem try to execute as many tasks as possible in parallel on multiple cores, in order to minimize the overall schedule length (a.k.a. makespan). In general, task scheduling is an NP-hard problem, and it is difficult to find an exact solution in a practical time [1]-[3]. Therefore, many heuristic and meta-heuristic approaches to task scheduling have been proposed, for example, in [4]-[8]. Some researchers have extended the scheduling problem in such a way that a task may use multiple cores [9]. Recent works on this direction include [10]-[15]. Liu et al. proposed heuristic algorithms to minimize the scheduling length of a given task-graph [10]. They allow individual tasks to use multiple cores, but the number of cores assigned to each task is assumed to be fixed. Unlike Liu's work, the work in [11] proposes a computational model called malleable, which means that the number of cores assigned to each task is determined at the same time as scheduling. They develop polynomial time algorithms for multiprocessor scheduling problem of malleable tasks. Scheduling for malleable tasks is also studied in [12], [13] and [14]. Yang and Ha's work in [12] allows tasks to run on multiple cores. Their work in [12] also assumes that tasks are malleable. They propose a multi-task mapping/scheduling technique for scalable MPSoC. The solution is based on integer linear programming (ILP). They aim at minimization of hardware cost, while satisfying the deadline constraints of the tasks. In [13], Chen and Chu develop an approximation algorithm for malleable task scheduling. In [14], a malleable task scheduling technique based on ILP was proposed. Each task is executed on multiple cores and many tasks are executed on multiple cores in parallel. They aim at minimization of schedule length. The previous works on malleable task scheduling in [12]-[14] allow a task to run on multiple cores at a synchronous manner, where the task starts and finishes the execution on the multiple cores at the same time. In [15], Kim et al. develop a real-time scheduling algorithm for malleable fork-join tasks. In the scheduling of malleable fork-join tasks, a task is partitioned into multiple sub-tasks (or threads), and the sub-tasks are scheduled independently. The number of sub-tasks is determined during the scheduling process simultaneously. They assume that the tasks are independent, and try to meet deadline constraints of the tasks. This paper studies scheduling of malleable fork-join tasks. Unlike the work in [15], this work assumes a set of dependent tasks, and tries to minimize a makespan. To the best of our knowledge, this is the first paper which studies scheduling of dependent, malleable fork-join tasks.

In this paper, we extend the malleable task model proposed in [14] towards a malleable fork-join task model. Given a set of dependent malleable tasks and a set of homogeneous cores, this work splits the tasks into sub-tasks, assigns the cores to the sub-tasks, and schedules the sub-tasks in such a way that the overall schedule length is minimized. Our task scheduling technique is based on ILP.

This paper is organized as follows. Section 2 reviews a related work on which this work is built. Section 3 presents an ILP formulation of our task scheduling problem, and Section 4 evaluates our scheduling technique. Finally, Section 5 concludes this paper.

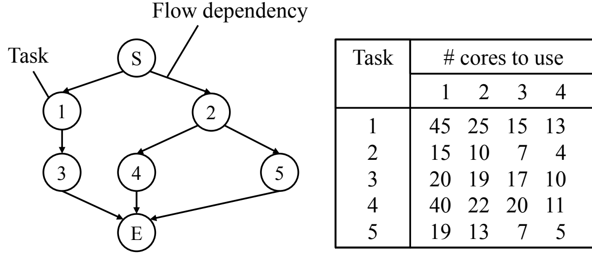
2 SCHEDULING FOR MALLEABLE SYNCHRONOUS TASKS

This work is based on the work in [14] where a Malleable Synchronous (MS) task scheduling technique is proposed. This section reviews the MS scheduling presented in [14].

2.1 Scheduling Example

An example for MS scheduling is shown in Figures 1 and 2. A set of tasks is represented as a directed acyclic graph (DAG) as shown in Figure 1. Each task has the execution times, which depends on the number of cores assigned to the task. For example in Figure 1, the execution time of task 1 is 45 when it is assigned a single core. If two cores are assigned to task 1, its execution

EWiLi'17, 2018, Seoul, South Korea,
Copyright held by Owners/Authors.



Task	# cores to use			
	1	2	3	4
1	45	25	15	13
2	15	10	7	4
3	20	19	17	10
4	40	22	20	11
5	19	13	7	5

Figure 1: Task graph

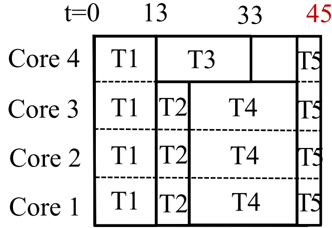


Figure 2: MS scheduling

time is 25. MS scheduling decides the number of cores for each task during scheduling. Figure 2 shows an optimal MS scheduling for the task graph in Figure 1, and the overall schedule length is 45.

2.2 Problem Formulation

The MS scheduling technique presented in [14] is based on Integer Linear Programming (ILP), and scheduling solutions are obtained with a commercial ILP solver.

Let $cores_{i,k}$ be a 0-1 variable, which becomes 1 if task i is assigned k cores, and otherwise 0. The number of cores assigned to task i is expressed as follows.

$$\forall i, \qquad \sum_k cores_{i,k} = 1 \qquad (1)$$

Let $map_{i,j}$ denote a 0-1 decision variable. $map_{i,j}$ becomes 1 if task i is mapped to core j, and otherwise 0. $map_{i,j}$ represents the mapping of task i. Note that $cores_{i,k}$ depends on $map_{i,j}$ and is determined as follows.

$$\forall i, \qquad \sum_j map_{i,j} = \sum_k cores_{i,k} \cdot k \qquad (2)$$

Let $Time_{i,k}$ denote the execution time of task i on k cores. We assume that $Time_{i,k}$ is given. In other words, $Time_{i,k}$ needs to be obtained dynamic profiling or static analysis techniques before task scheduling. Then, $time_i$, the execution time of task i, is given by the following equation.

$$\forall i, \qquad time_i = \sum_k \left(cores_{i,k} \cdot Time_{i,k}\right) \qquad (3)$$

Let $start_i$ and $finish_i$ denote the start time and finish time of task i, respectively. Note that $start_i$ is a decision variable and

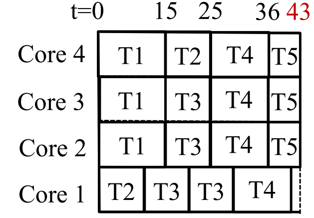


Figure 3: MFJ scheduling

$finish_i$ is a dependent variable defined by the following equation.

$$\forall i, \qquad finish_i = start_i + time_i \qquad (4)$$

If two tasks, $i1$ and $i2$, are mapped to the same core, the execution of the two tasks cannot be overlapped in time. This resource constraint is formulated by the following formula.

$\forall i1, i2, j,$

$$map_{i1,j} + map_{i2,j} < 2 \\ \lor\ finish_{i1} \leq start_{i2} \\ \lor\ finish_{i2} \leq start_{i1} \qquad (5)$$

This work assumes a set of dependent tasks, and the tasks may have a precedence dependency. Let $Flow_{i1,i2}$ denote a precedence dependency between task $i1$ and $i2$. $Flow_{i1,i2}$ is 1 when task $i1$ must be finished before task $i2$ starts. Otherwise, $Flow_{i1,i2}$ is 0. We assume that $Flow_{i1,i2}$ is given. Then, the precedence constraint is expressed as follows.

$$\forall i1, i2, \qquad Flow_{i1,i2} \rightarrow finish_{i1} \leq start_{i2} \qquad (6)$$

This work aims at minimization of the overall schedule length. Therefore, the objective function of our scheduling problem to be minimized is given as follows.

$$\max_i \{finish_i\} \qquad (7)$$

The scheduling problem for malleable synchronous tasks is now formally defined: Given a set of tasks, a set of cores, $Time_{i,k}$ and $Flow_{i1,i2}$, find $map_{i,j}$ and $start_i$ which minimize the objective function (7) subject to the six constraints (1)-(6).

3 SCHEDULING FOR MALLEABLE FORK-JOIN TASKS

In this section, we propose a scheduling technique for Malleable Fork-Join (MFJ) tasks. In MFJ scheduling, tasks are split into sub-tasks and they are scheduled independently. It is possible to schedule them on different cores in parallel or on same cores sequentially. Given a set of tasks and a set of homogeneous cores, this work decides (a) the number of sub-tasks for each task, (b) mapping of the sub-tasks onto the cores, and (c) the start time of each sub-task.

3.1 Scheduling Example

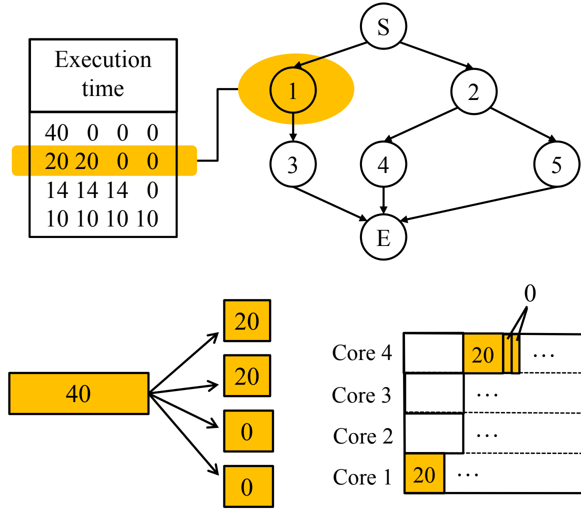


Figure 4: Splitting a task into sub-tasks

Figure 3 shows an example for MFJ scheduling for the task-graph in Figure 1. Similar to MS scheduling in Figure 2, MFJ scheduling determines the number of cores for each task during scheduling. However, unlike MS scheduling where a task is synchronously executed on multiple cores, MFJ scheduling splits a task into sub-tasks and allow them to start asynchronously at different times. For example, in Figure 2, task 2 is split into two sub-tasks. One of the sub-task is executed on core 0 at time 0, while another sub-task is executed on core 3 at time 15. Then, the overall length of the MFJ schedule is 43 time units, which is shorter than the MS scheduling result in Figure 2.

It should be noted that MFJ scheduling and MS scheduling are targeted at different task models. It does not mean that MFJ scheduling outperforms MS scheduling. However, it is not a good idea to apply MS scheduling algorithms to MFJ tasks.

3.2 Problem Formulation

Our MFJ scheduling technique is based on ILP.

Let us assume that a set of tasks and a set of cores are given. This work decides (a) the number of sub-tasks for each task, (b) mapping of the sub-tasks onto the cores, and (c) the start time of each sub-task.

Let $split_{i,k}$ denote a 0-1 decision variable. $split_{i,k}$ becomes 1 if task i is split into k sub-tasks (in other words, if task i is executed on k cores), otherwise 0. The following constraint must hold, where k is within the range of 1 to the number of cores.

$$\forall i, \qquad \sum_k split_{i,k} = 1 \tag{8}$$

Let $Time_{i,k,j}$ denote the execution time of j-th sub-task of task i when task i is assigned k cores. $Time_{i,k,j}$ is 0 for $j > k$. We assume that $Time_{i,k,j}$ is given, and how to obtain $Time_{i,k,j}$ values is out of scope of this paper.

Figure 4 shows an example of MFJ scheduling on four cores. The table on the left top of Figure 4 shows $Time_{1,k,j}$ values for task 1. In this example, task 1 is split into two sub-tasks. In other words, task 1 is assigned two cores. In this case, $split_{1,2}$ becomes 1. Although there are two sub-tasks for task 1, we assume that there exist two virtual sub-tasks whose execution time is zero, as shown in Figure 4. This trick simplifies our ILP formulation.

Using $split_{i,k}$ and $Time_{i,k,j}$, the execution time of j-th sub-task in task i is defined as follows.

$$\forall i, j, \qquad time_{ij} = \sum_k \left(Time_{i,k,j} \cdot split_{i,k}\right) \tag{9}$$

Unlike the previous work in [14], sub-tasks are scheduled independently. Let $start_{i,j}$ and $finish_{i,j}$ denote the start time and the finish time of j-th sub-task in task i, respectively. Note that $start_{i,j}$ is a decision variable and $finish_{i,j}$ is a dependent variable defined by the following equation.

$$\forall i, j, \qquad finish_{i,j} = start_{i,j} + time_{i,j} \tag{10}$$

Next, let $core_{i,j}$ be the identification number of the core which is assigned j-th sub-task in task i. If two sub-tasks, sub-task $j1$ in task $i1$ and sub-task $j2$ in task $i2$, are mapped to the same core, the execution of the two sub-tasks cannot be overlapped in time. This resource constraint is formulated by the following formula.

$$\forall i1, i2, j1, j2\ (i1 \neq i2 \lor j1 \neq j2),$$
$$\begin{aligned} & core_{i1,j1} \neq core_{i2,j2} \\ & \lor\ finish_{i1,j1} \leq start_{i2,j2} \\ & \lor\ finish_{i2,j2} \leq start_{i1,j1} \end{aligned} \tag{11}$$

This work assumes a set of dependent tasks, where the tasks may have a flow dependency among them. Let $start_min_i$ and $finish_max_i$ denote the start time and the finish time of task i, respectively. They are defined as follows.

$$\forall i, \qquad start_min_i = min_j\{start_{i,j}\} \tag{12}$$

$$\forall i, \qquad finish_max_i = max_j\{finish_{i,j}\} \tag{13}$$

Let $Flow_{i1,i2}$ denote a flow dependency from task $i1$ to task $i2$. $Flow_{i1,i2}$ is 1 when task $i1$ must be finished before task $i2$ starts. Otherwise, $Flow_{i1,i2}$ is 0. We assume that $Flow_{i1,i2}$ is given. Then, the precedence constraint is expressed as follows.

$$\forall i1, i2, \qquad Flow_{i1,i2} \rightarrow finish_max_{i1} \leq start_min_{i2} \tag{14}$$

This work aims at minimization of the overall schedule length. Therefore, the objective function of our scheduling problem to be minimized is given as follows.

$$\max_i\{finish_max_i\} \tag{15}$$

Our scheduling problem for malleable fork-join tasks is now formally defined: Given a set of tasks, a set of cores, $Time_{i,k,j}$ and $Flow_{i1,i2}$, find $split_{i,k}$, $core_{i,j}$ and $start_{i,j}$ which minimize the objective function (15) subject to the seven constraints (8)-(14).

Although formulas (11)-(15) are not in a linear form, they can be easily linearized by simple transformation techniques. Actual-

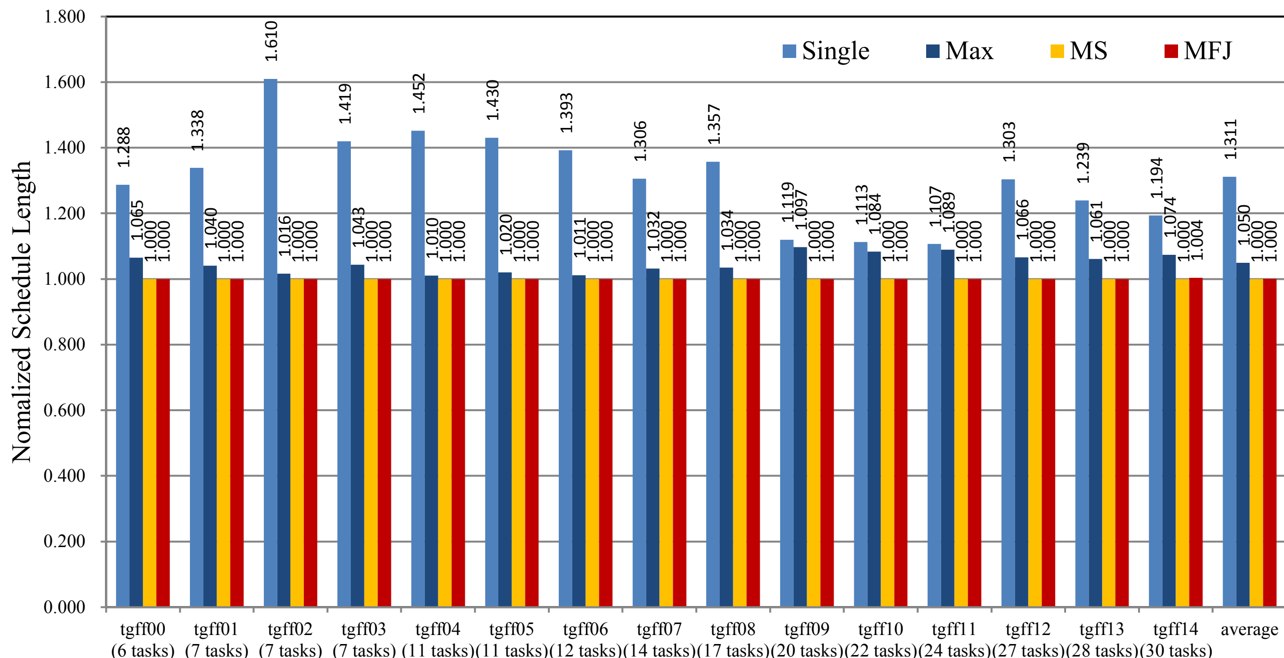


Figure 5: Scheduling results on 2 cores

ly, many ILP solvers are able to translate such non-linear formulas into linear ones automatically.

4 EXPERIMENTS

4.1 Experimental Setup

In order to test the effectiveness of this work, we conducted a set of experiments. We generated fifteen task-graphs using *Task Graph For Free (TGFF)* [16]. Each task-graph contains up to 30 tasks. TGFF assumes that each task is executed on a single core, and $Time_{i,k,j}$ is given only for $k = 1$. Therefore, we set $Time_{i,k,j} = \lceil Time_{i,1,j} \times (0.1 + 0.9/k) \rceil$ for $k > 1$ in our experiments, where each sub-task is assumed to have 10% overhead which cannot be parallelized. For each task-graph, we varied the number of cores from two to eight.

We have compared four scheduling methods including the one presented in this paper as follows.

- *Single*: ILP-based scheduling when each task is assigned a single core.
- *Max*: Each task is assigned all cores, and tasks are scheduled sequentially. The schedule length is $\sum_i \sum_j Time_{i,N,j}$ where N is the number of cores in the target system.
- *MS*: ILP-based malleable synchronous task scheduling presented in [14].
- *MFJ*: ILP-based malleable fork-join task scheduling presented in this paper.

Among the four scheduling methods, Single, MS and MFJ are based on ILP. In order to solve the ILP problems for the three methods, we used IBM ILOG CPLEX 12.7 [17] on dual Intel Xeon E5-2650 processors with 128GB memory. Since ILP is very time-consuming, optimal solutions cannot be found in a practical time for large task-graphs. In our experiments, CPU runtime of CPLEX is limited up to 60 hours, and the best solutions found at that time were used for evaluation. Since we ran CPLEX on 32-threading host computer (dual processors, 8 cores / 16 threads per processor), CPU time of 60 hours is approximately 2 hours in real time.

4.2 Experimental Results

The scheduling results for fifteen task-graphs and their average are shown in Figures 5, 6 and 7. In each figure, the four scheduling methods are compared, and the schedule lengths are normalized to the MS method [14]. Figures 5, 6 and 7 show scheduling results on two, four and eight cores, respectively.

The ILP solver for the Single method found optimal solutions for forty four task graphs out of forty five (fifteen graphs in each of the three figures) within the time limit. The ILP solver for the MS and MFJ methods found optimal solutions for the thirty six task graphs and seven task graphs, respectively.

The schedule length of the Single method was the longest among the four methods in any case. This is because the Single method assigns a single core to each task even when no other task is executable due to flow dependencies. This significantly degrades the CPU utilization.

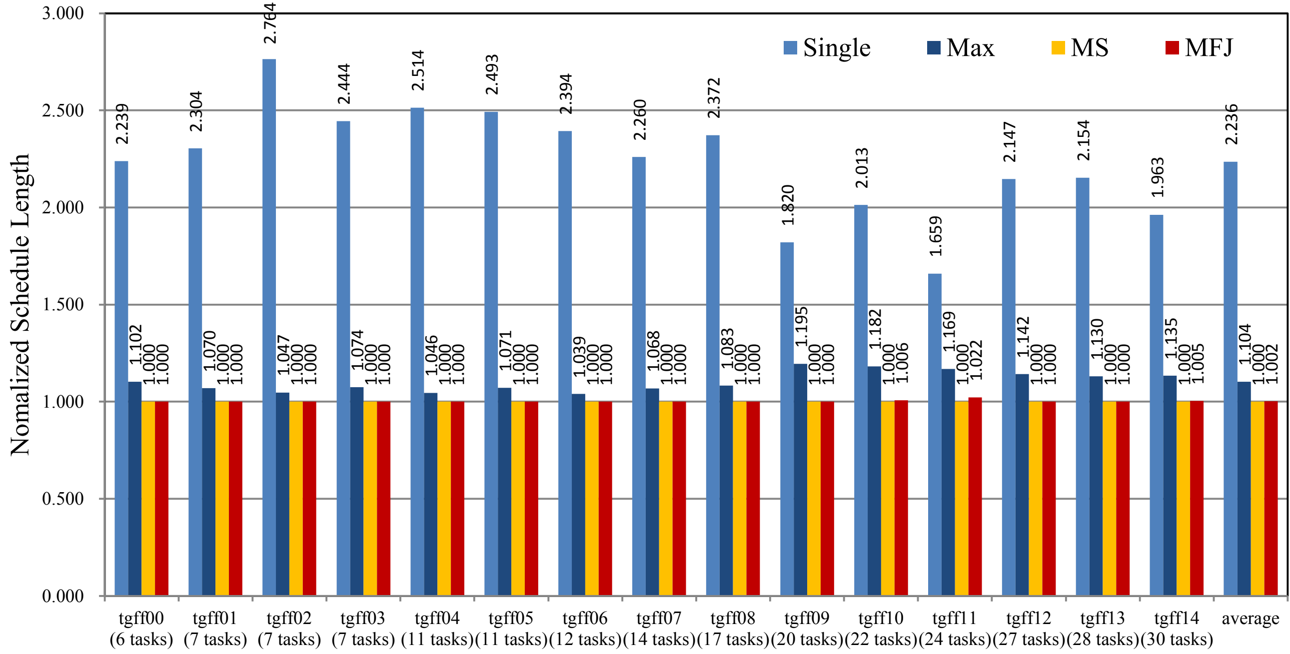


Figure 6: Scheduling results on 4 cores

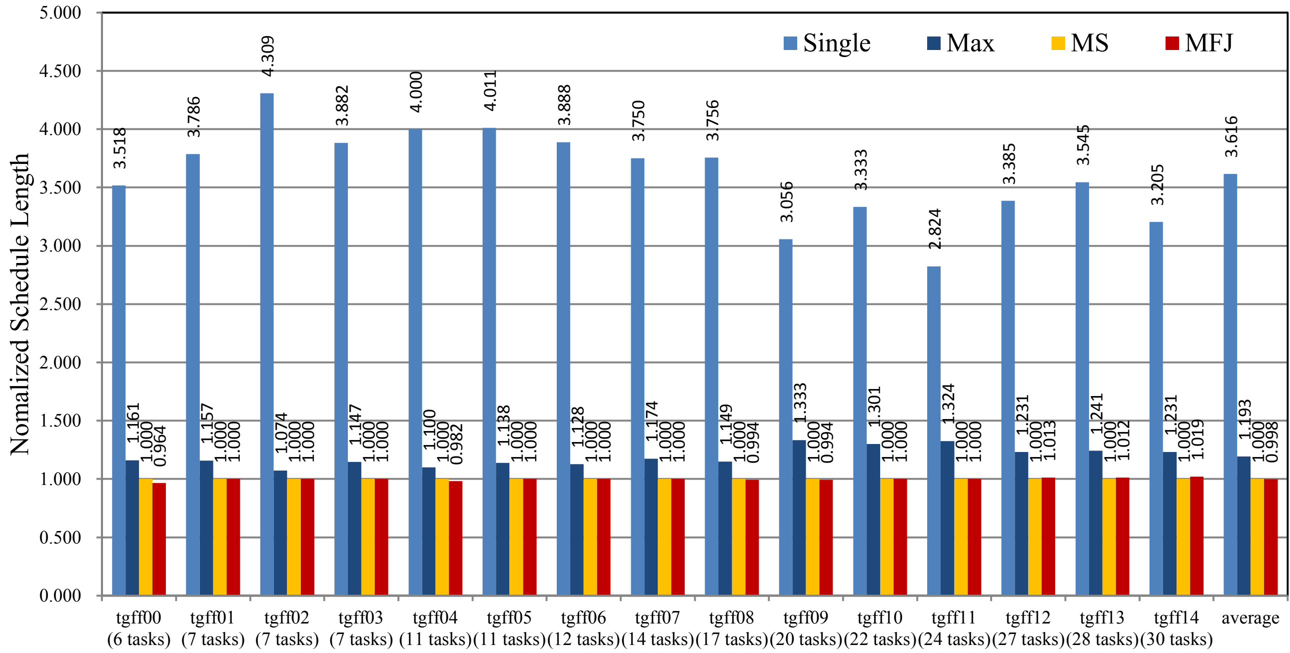


Figure 7: Scheduling results on 8 cores

The CPU utilization of the Max method is 100% since tasks are assigned all cores in the target multicore system and are executed sequentially. The Max method outperforms the Simple method, but is not as good as the MS and MFJ methods. This is because of the performance overhead of parallelization. We assume that each sub-task has 10% overhead. As the number of sub-tasks increases, the overall performance overhead also increases. If a task graph has a rich amount of inter-task parallelism, tasks should be scheduled onto multiple cores without splitting into sub-tasks in order to minimize the overhead. Since the Max method always splits tasks into the maximum number of sub-tasks, it suffers from the overhead of parallelization.

The Single method takes advantage of *inter*-task parallelism, but ignores *intra*-task parallelism. On the other hand, the Max method takes advantage of intra-task parallelism, but ignores inter-task parallelism. The MS method and MFJ method take advantage of both inter-task parallelism and intra-task parallelism in order to minimize the schedule length.

Let us compare the MS and MFJ methods. In Figure 5 where tasks are scheduled on two cores, the MS method and the MFJ method yield the same quality of results for all task graphs except tgff14. Theoretically, the solution space of MFJ completely covers that of MS. Thus, the optimal solution of MFJ should be better than (or at least equal to) the optimal solution of MS. However, due to the wider solution space, the ILP solver was not able to find the good solutions for the MFJ method within a limited time. On the other hand, the ILP solver quickly found good solutions for the MS method. The scheduling results on four cores (Figure 6) are similar to results on two cores (Figure 5). The MS method outperforms the MFJ method for three task graphs out of fifteen, and the two methods are comparable for the other task graphs. The results in Figure 7 are somewhat different from the ones in Figures 5 and 6. For four task graphs (i.e., tgff00, tgff04, tgff08 and tgff09), the MFJ method yields better schedules than the MS method. These task graphs are relatively small, and the ILP solver successfully found good schedules within a limited time. However, for the three larger task graphs (i.e., tgff12, tgff13 and tgff14), the MS method outperformed MFJ.

Our experimental results demonstrate both strength and weakness of our MFJ task scheduling method. On a strong side, the MFJ scheduling is effective when the target system has a large number of cores. On a weak side, the ILP-based MFJ scheduling is not scalable and is hardly applied to large task graphs. Fast heuristic algorithms for MFJ scheduling are necessary.

5 CONCLUSIONS

This paper addressed a scheduling problem for malleable fork-join tasks on multiple cores. We presented a solution technique for the scheduling problem based on integer linear programming formulation. Our scheduling technique decides the number of sub-tasks, allocation of the sub-tasks onto cores and the execution order of the sub-tasks. The experimental results show both strength and weakness of our technique, compared with prior techniques. Our technique found better schedule results than prior techniques when the system has a large number of cores, while our technique failed to find good results for large task graphs in a practical time. In future, we plan to develop fast heuristic algorithms for the scheduling problem.

ACKNOWLEDGMENTS

This work is in part supported by JSPS KAKENHI 15H02680.

REFERENCES

[1] H. Kasahara, S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, vol. *C-33*, pp. 1023-1029, 1984.

[2] S. Fujita, "A branch-and-bound algorithm for solving the multiprocessor scheduling problem with improved lower bounding techniques," *IEEE Transaction on Computers*, vol. 60, pp. 1006-1016, 2010.

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

[4] Y.K. Kwok, I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, pp. 406-471, 1999.

[5] J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee, "Scheduling precedence graph in systems with interprocessor communication times," *SIAM Journal of Computing*, vol. 18, pp. 244-257, 1989.

[6] G.C. Sih, E.A. Lee, "Compile time scheduling heuristic for interconnection constrained heterogeneous processor architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 175-187, 1993.

[7] T. Hagras, J. Janecek, "A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environment," *International Conference on Parallel Processing Workshops*, 2003.

[8] A. Palmer and O. Sinnen, "Scheduling algorithm based on force directed clustering," *International Conference Parallel and Distributed Computing, Applications and Technologies*, pp. 311-318, 2008.

[9] M. Drozdowski, "Scheduling multiprocessor tasks: An overview," *European Journal of Operational Research*, vol. 94, 1996.

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

[11] J. Turek, J.L. Wolf, and P.S. Yu, "Approximate algorithms scheduling parallelizable tasks," *Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[12] H. Yang and S. Ha, "ILP based data parallel multi-task mapping/scheduling technique for MPSoC," *International SoC Design Conference*, 2008.

[13] C. Chen and C. Chu, "A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints", *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, 2013.

[14] K. Shimada, S. Kitano, I. Taniguchi, and H. Tomiyama, "ILP-based scheduling for parallelizable tasks," *IEICE Transactions on Fundamentals*, vol. E100-A, no. 7, 2017.

[15] J. Kim, H. Kim, K. Lakshmanan, R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," *International Conference on Cyber-Physical System*, 2013.

[16] R.P. Dick, D.L. Rhodes, W.H. Wolf, "TGFF: Task graph for free," *International Workshop on Hardwere/Software Codesign*, 1998.

[17] IBM CPLEX Optimize,
http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
(Last accessed : July 16, 2017)