International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 2, pages 131–146, July 2019

A Constraint Programming Approach to Scheduling of Malleable Tasks

Hiroki Nishikawa
Graduate School of Science and Engineering
Ritsumeikan University
Kusatsu, Shiga, 525-0058, Japan

Kana Shimada
Graduate School of Science and Engineering
Ritsumeikan University
Kusatsu, Shiga, 525-0058, Japan

Ittetsu Taniguchi
Graduate School of Information Science and Technology
Osaka University
Suita, Osaka, 565-0871, Japan

Hiroyuki Tomiyama
Graduate School of Science and Engineering
Ritsumeikan University
Kusatsu, Shiga, 525-0058, Japan

Received: February 14, 2019
Accepted: April 11, 2019
Communicated by Shinya Takamaeda-Yamazaki

Abstract

With the increasing demand for high-performance computing, multicore architectures became appealing in various application domains. In order to exploit the parallelism of the multicore architectures, task scheduling has become more important than ever. Classical multicore task scheduling assumes that each task is executed on one of the cores. However, many tasks in modern applications have inherent parallelism and can be multi-threaded. A task is partitioned into threads which can be executed on multiple cores in a fork-join fashion. A multi-threaded task is called *malleable* if the number of threads is flexible and is determined at the same time as task scheduling.

This paper proposes multicore scheduling methods for malleable tasks. Given a set of dependent tasks in the form of directed acyclic graph and homogeneous multiple cores, the proposed methods decide the number of threads for each task and schedule the threads on the multicores simultaneously, with the goal of minimizing the overall schedule length. The proposed scheduling methods are based on constraint programming. Experimental results show that the proposed methods outperform state-of-the-art work which is based on integer linear programming.

Keywords: Task scheduling, multicore, constraint programming

1 Introduction

Multicore task scheduling, which decides the execution order of tasks on multiple cores, has become more important than ever due to increasing number of cores not only in general-purpose computer systems but also in embedded systems. In classical task scheduling problems for multicore architectures, tasks are mapped to the available cores so that the tasks are executed in parallel on different cores, on the assumption that each task is executed on one of the multiple cores [1, 2]. Many tasks in modern applications such as multimedia ones, however, can be partitioned into multiple threads in a data-parallel fork-join manner, and the threads can be executed independently on multiple cores [3]. Many researchers have studied scheduling problems which consider not only task parallelism (i.e. inter-task parallelism) but also data parallelism (i.e. intra-task parallelism) [5]-[15]. As proposed in [5]-[7], their works assumed that the number of threads in each task is fixed. On the other hand, the works in [11] and [15] assumes that tasks are *malleable*. A malleable task has an unfixed number of threads. The number of threads for each task is determined at the same time as scheduling.

Task scheduling problems are known to be in the NP-hard class of computationally intractable problems [4]. The state-of-the-art methods for malleable task scheduling in [11] and [15] are based on integer linear programming (ILP). The methods can obtain optimal schedules for small task graphs, but cannot find optimal schedules, or even one of feasible schedules, for large task graphs in a practical time.

Constraint programming (CP) is one of proven approaches to combinational optimization problems including classical task scheduling problems [16, 20, 21]. This paper, for the first time, proposes a CP-based scheduling method for malleable fork-join tasks. Given a set of dependent malleable tasks in the form of a directed acyclic graph (DAG) and a set of homogeneous multiple cores, the proposed method decides the number of threads for each task and schedule the threads on the multicores simultaneously, with the goal of minimizing the overall schedule length (a.k.a. makespan). This paper also proposes a CP-based scheduling method for malleable synchronous tasks. Similar to malleable fork-join tasks, a malleable synchronous task can be partitioned into an unfixed number of threads. However, the threads cannot be executed independently, and need to be scheduled synchronously with each other [1]. In other words, the threads of a synchronous task need to be started at the same time on different cores. On the other hands, the threads of a malleable fork-join task can be started at different times on different cores, or they can be executed sequentially on the same core.

The contributions of this work are threefold as follows.

- This work proposes a CP-based scheduling method for malleable fork-join tasks.
- This work also proposes a CP-based scheduling method for malleable synchronous tasks.
- This work conducts a set of experiments and shows that the proposed CP-based methods outperform state-of-the-art methods based on ILP.

The remainder of this paper is organized as follows. Section 2 describes related work on task scheduling. Section 3 provides the ILP-based mathematical formulation and our CP-based approach to MS scheduling. This section also presents the experiments and the evaluation. In Section 4, we present a CP-based scheduling of MFJ tasks after the introduction of an ILP-based formulation with the experiments. Finally, Section 5 concludes this paper with future plans.

2 Related Work

In [1], classic techniques on task scheduling for multicore architectures are extensively surveyed. Multiple tasks, which are independent of each other, are executed in parallel on different cores. The authors of [2] develop a ILP-based approach to scheduling of tasks on multiple processors with communication delay. However, both of them assume that each task is not considered with data

[1] If the threads have to synchronize with each other frequently during their execution on a non-preemptive environment, the task is considered as a synchronous one.

parallelism and is executed on a single core. Many real world tasks have not only task parallelism but also have data parallelism. Then task scheduling, which considers both parallelisms, has been studied in [3]- [15]. In [5], Liu et al. proposed list-based scheduling algorithms for data-parallel tasks. Their work assumes that a set of dependent tasks is given in the form of a task-graph, where each task is assigned to a fixed number of cores. Then, they try to minimize the overall schedule length. Yang and Ha's work in [6] also focuses on scheduling of data parallel tasks. Unlike the work in [5], their work in [6] assumes that tasks are malleable, which means that the number of cores for each task is not given and is determined at the same time as scheduling. Their goal is to minimize hardware cost with meeting deadline constraints. In [7], the authors take advantage of data-parallelism and proposed a technique for pipelined task scheduling and mapping on heterogeneous MPSoCs. Chen and Chu in [8] designed a polynomial time approximation algorithm for malleable tasks to find a minimum makespan. Scheduling of MFJ tasks for real-time systems is studied in [12]. In the scheduling of MFJ tasks, a task is partitioned into multiple threads, and each thread is mapped to one of multiple cores, independently. The work aims at evaluation of the tractable and intractable fork-join real-time task model. Lakshmanan et al. in [13] developed an algorithm for MFJ tasks in OpenMP. Saifullah et al. in [14] proposed a real-time task scheduling model which assumes that a task holds the various numbers of threads. Shimada et al. in [11] studies MFJ task scheduling by a mathematical approach based on integer linear programming. They also study in [15] scheduling of malleable tasks based on integer linear programming. In this work, a malleable task is assumed that synchronization among all threads of each task is necessary for start time and end time. Their works are based on an ILP approach, and therefore can hardly be explored due to the excessive time for finding a solution.

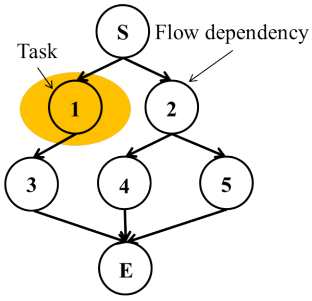
In order to find a feasible solution in a short time, heuristic-based approaches have been developed such as MILP (i.e. Mix Integer Linear Programming), SMT (i.e. Satisfiability Modulo Theory), and CP. In [17], the work studies a MILP approach to scheduling problem of parallel tasks, but it is focus on rigid tasks, where each task is assumed to be executed on a statically fixed number of processors. Liu et al. [18] analyze how efficiently a SAT solver is able to eliminate the solution space for task scheduling problems with mapping. The SAT-based framework proves a significant improvement in terms of the optimality and the scalability. Malik et. al in [19] evaluate the performance of scheduling problem based on satisfiability modulo theory (SMT). Furthermore, CP-based approaches are also proposed in area of scheduling. In [16], the principle of CP methods for combinational optimization problems is surveyed. Derived from the work, Baptiste et al. [20] develop a scheduling problem with a CP approach. Kuchicinski in [21] also proposes a scheduling problem that take communication cost into account, using CP. The authors both in [22] and [23] present scheduling problems, assumed that tasks are executed on multiple processors. However, the works applied with such a solver are not focused on scheduling for malleable tasks. This paper studies a CP-based approach to scheduling of malleable tasks. We try to find better solutions and much faster than the state-of-the-art methods for data parallel task scheduling. To the best of our knowledge, this is the first paper that addresses this topic.

3 Scheduling for Malleable Fork-join Tasks

This section presents an example of MFJ task scheduling, and describes the formulation based on an integer linear programming (ILP) which was presented in [11]. A MFJ task scheduling assumes that each of tasks can be split into threads, and each thread is scheduled, independently. A thread is executed on a core unless the core is overlapped. The number of threads to be split is determined at the same time as scheduling. Given a set of dependent tasks and a set of homogeneous cores, the objective of scheduling for MFJ tasks is to minimize the overall schedule length.

3.1 Problem Description

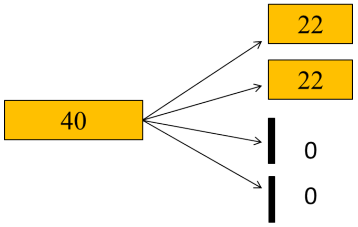
In Figure 1 (a), (b), (c) and (d), an example of MFJ task scheduling on four cores is represented. The table on the top of Figure 1 (a) is a task graph in form of a directed acyclic graph (DAG), and (b) shows $Time_{1,k,j}$ values for task 1. In this example, task 1 is split into two threads. In other



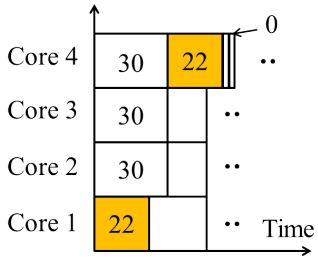
(a) Task graph

Sub-task	Execution time of task 1			
1	40	0	0	0
2	22	22	0	0
3	16	16	16	0
4	14	14	14	14

(b) Execution time of task 1



(c) Split into threads



(d) An optimal schedule

Figure 1: A scheduling example for MFJ tasks

words, task 1 is assigned to two cores. Although there are two threads for task 1, we assume that there exist two virtual threads whose execution time is 0, as shown in Figure 1 (c). Then, one of the examples for scheduling of the threads is represented in Figure 1 (d). This is a very feature of ILP formulation. The execution time of $j-th$ thread in task i is shown.

3.2 ILP Formulation

Scheduling for MFJ tasks presented in [11] is based on ILP, and the scheduling solution is obtained by a commercial solver. The rest of this section briefly describes the ILP formulation presented in [11].

Let $split_{i,k}$ denote a 0-1 decision variable. $split_{i,k}$ becomes 1 if task i is split into k threads. As follows, the constraint must hold, where k is within the range of 1 to the number of cores.

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

Let $Time_{i,k,j}$ denote the execution time of $j-th$ thread of task i when it is assigned to k cores. $Time_{i,k,j}$ is 0 for $j > k$. $Time_{i,k,j}$ is assumed to be given, and how to obtain $Time_{i,k,j}$ values is out of scope in this paper.

$$\forall i,j, \qquad time_{i,j} = \sum_{k} (split_{i,k} \times Time_{i,k,j}) \tag{2}$$

The threads are scheduled independently unless the cores are overlapped. Therefore, let $start_{i,j}$ and $finish_{i,j}$ denote the start time and the finish time of the threads of 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.

$$\forall i,j, \qquad finish_{i,j} = start_{i,j} + time_{i,j} \tag{3}$$

Next, let $core_{i,j}$ be the identified number of the cores which is assigned to $j-th$ thread in task i. If two threads, thread $j1$ in task $i1$ and the thread $j2$ in $i2$, are mapped to the same core, the

execution of the two threads cannot be overlapped in time. This resource constraint is formulated by the following.

$$\begin{aligned} \forall i1,i2,j1,j2, \quad & core_{i1,j1} \neq core_{i2,j2} \\ \vee \quad & finish_{i1,j1} \leq start_{i2,j2} \\ \vee \quad & finish_{i2,j2} \leq start_{i1,j1} \end{aligned} \tag{4}$$

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.

$$\forall i, \quad start\_min_i = min_j\{start_{i,j}\} \tag{5}$$

$$\forall i, \quad finish\_max_i = max_j\{finish_{i,j}\} \tag{6}$$

Let $Flow_{i1,i2}$ denote a flow dependency from task $i1$ to $i2$. $Flow_{i1,i2}$ 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.

$$\forall i1,i2, \quad Flow_{i1,i2} \rightarrow finish\_max_{i1} \leq start\_min_{i2} \tag{7}$$

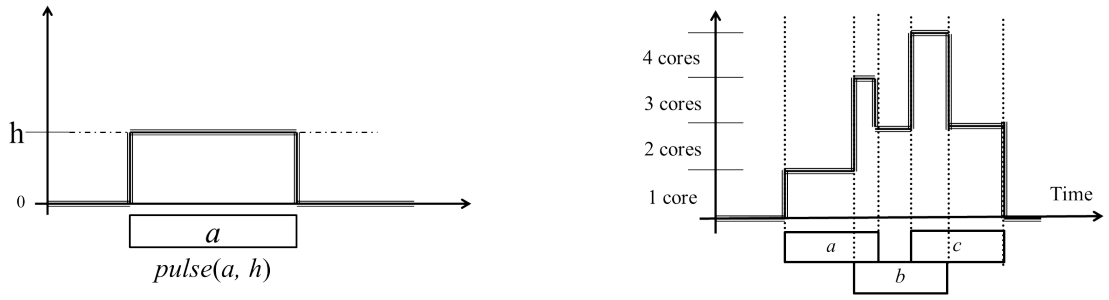
This work aims at minimization of the overall schedule length. Therefore, the objective function of our scheduling problem to be minimized is given as follows.

$$\text{Minimize :} \quad max_i\{finish_i\} \tag{8}$$

Our scheduling problem for MFJ tasks is now formally defined: Given a set of dependent 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 (8) subject to the constraints (1)-(7).

3.3 Constraint Programming Approach

This section proposes a formulation for scheduling of malleable fork-join tasks based on CP. In order to solve an optimization problem with CP, IBM ILOG CP Optimizer is used as a CP solver in this work, whose concepts for expressions and functions of scheduling problems are described in [24] and [25]. ILOG CP Optimizer offers a number of built-in functions. According to IBM ILOG CP Optimizer official publication in [26] and some studies such as the works in [27] and [28], the CP Optimizer has been developed for the internal strong search strategies and techniques. As a default, the CP Optimizer basically has search strategies consists of diverse those of being dynamically changed for the time when the search adapting to the problem at hand. The different search techniques basically include mainly DFS (Depth First Search), LNS (Large Neighborhood Search), GA (Genetic Algorithm), but they are not restricted with such the algorithms. Furthermore, constraint propagation that is one of the internal techniques by ILOG CP Optimizer performs in the search for solutions to CP problems. Constraint propagation removes values from domains that will not take part in any solution. Then, the CP Optimizer starts finding a solution under constraints with back-tracking algorithm. If the CP Optimizer succeeds in obtaining a feasible solution, the solution is added into the constraints as an upper or a bottom bound. Moreover, constraint propagation removes the values of decision variables from domains that will not take part in any other solutions considered with the solution. Again, the CP Optimizer restarts to find a solution. Such strong strategies for solving an optimization problem are employed by the CP Optimizer. A motivation in this paper is obtained due to these features of the CP. By utilizing the built-in functions, optimal schedules can be found efficiently. Even in case optimal schedules are not found in a practical time, the CP optimizer can find better schedules than the ILP-based technique [11]. In the following, we describe the formulation for this scheduling with using CP provided by IBM ILOG CP Optimizer.



(a) Pulse function (b) Cumulative function

Figure 2: A concept of resource constraint

3.3.1 Interval Variable

One of the most important concepts in scheduling with ILOG CP Optimizer is interval variables. An interval variable represents an interval of time while an activity is carried out, and it is described as a start time, an end time and a size (length) of an execution time. One of the features of interval variables is that they are optional even if they are defined in the CP program. Interval variables can be either present or absent. If an interval variable is marked as absent, the interval variable is ignored during the scheduling process.

Let $task_{i,j}$ denote an interval decision variable for $j-th$ thread in task i. It is present in the scheduling if $j-th$ thread in task i is considered, otherwise absent. In our scheduling problem, we are given the execution time of each thread in tasks, thus, we attempt to determine the number of threads for each task, what cores to be mapped on and their start time of execution.

3.3.2 Precedence Constraints

This work assumes a set of dependent tasks, where tasks may have a given precedence dependency. The function, called $endBeforeStart(a, b)$ built in ILOG CP Optimizer, represents the precedence constraint between interval variables a and b are considered to be true provided that both of the two interval variables are present. Therefore, the precedence dependency constraint that $task_{i1,j1}$ must be finished before $task_{i2,j2}$ starts is expressed as follows.

$$\forall j1,j2, \qquad endBeforeStart(task_{i1,j1}, task_{i2,j2}) \tag{9}$$

3.3.3 Alternative Constraints

Next, we set a set of $decision_{i,k,j}$. $decision_{i,k,j}$ denotes the execution time of $j-th$ thread in task i when it is assigned to k cores for the task. In order to guarantee that one of $decision_{i,k,j}$ is present for each $task_{i,j}$ where task i is determined to be split into k threads, we use the alternative function.

$$\forall i,j \qquad alternative(task_i, \cup_k \{decision_{i,k,j}\}) \tag{10}$$

In Formula (10), it should be recalled that task i is always present. Therefore, one in a set of $decision_{i,k,j}$ must be present. Then, all of threads to be split in task i must be present. The constraint is expressed in Formula (11).

$$\forall i,k,j1,j2, \qquad presenceOf(decision_{i,k,j1}) \rightarrow presenceOf(decision_{i,k,j2}) \tag{11}$$

$presenceOf$ is a built-in function of ILOG CP Optimizer, which returns values 0-1 for the existence of the given interval variable.

3.3.4 Resource Constraints

This work tries to minimize the schedule length under a resource constraint on a number of cores. At any moment in time, the total number of cores assigned to active tasks cannot exceed the number of cores in the target system. This resource constraint is expressed with the *pulse* function and the $cumulfunction$ in ILOG CP Optimizer as shown in Figure 2. Figure 2 (a) shows the concept of the pulse function. Let a be an interval variable and h be a scalar value. The value of *pulse(a, h)* indicates h during the interval a. When a is absent, the pulse value is 0. According to [24] and [25], the cumulative function accumulates the values from pulses over time. In shown in Figure 2 (b), here is one of examples with the interval variables a, b and c. This case assumes that the pulse function is utilized with a, b, and c, respectively. Now, the cumulative function is accumulated with $pulse(a, 1) + pulse(b, 2) + pulse(c, 2)$. Applying the concept to our resource constraint, a constraint on a number of cores is expressed as follows.

$$Ncores \geq CumulFunction\{\sum_{i}\sum_{k}\sum_{j} pulse(decision_{i,k,j},\ 1)\} \tag{12}$$

In the formula, $Ncores$ denotes the total number of cores in a system. It should be recalled that $decision_{i,j,k}$ is an interval variable and is present if $j-th$ thread of task i split into k threads is active. Formula (12) means the value of $pulse(decision_{i,j,k}, 1)$ is 1 while $j-th$ thread in task i being active. The value 1 is represented when a core is assigned to a thread in a task, that is, the right side of the inequality shows that sum of the total number of cores assigned to the active threads in tasks.

3.3.5 Objective Function

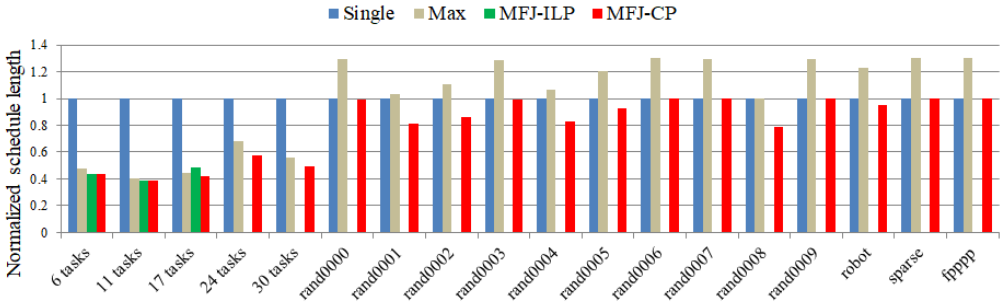
Our objective function is to minimize the overall schedule length, and is defined as follows.

$$\text{Minimize}: \qquad max_{i,j}\{endOf(task_{i,j})\} \tag{13}$$

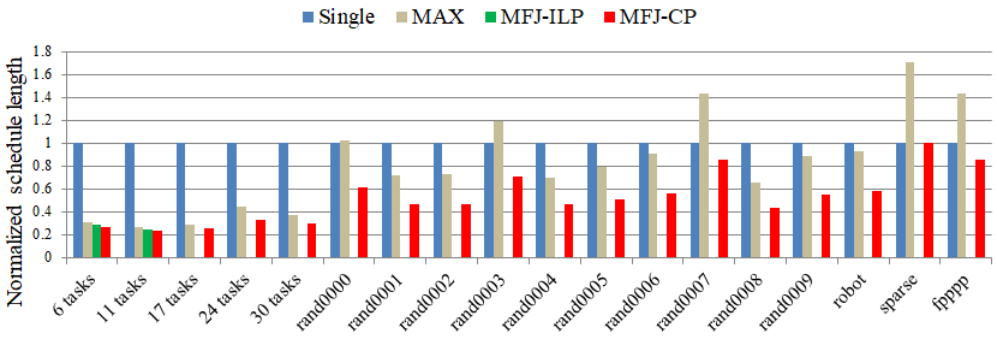
$endOf$ is a built-in function in ILOG CP Optimizer, which returns the end time of the given interval variable. In general, constraint programs do not have any objective functions to be minimized or maximized. Constraint programs search for values for variables which satisfy all of the specified constraints. However, many state-of-the-art CP solvers including ILOG CP Optimizer are capable of finding the best values for variables to optimize a specified objective function. In this work, we take advantage of the optimization capability in ILOG CP Optimizer.

3.4 Experiments

In order to evaluate and compare the performance of the proposed approach, we have conducted a set of experiments. Scheduling for MFJ tasks becomes much more complex than the classical scheduling method in terms of computational costs for parallelism. In order to evaluate the performance of every method in this paper, we have a set of taskgprahs for a small number of tasks composed of 6 to 30 tasks that are generated randomly to DAGs [29]. The rest of taskgraphs is derived from the $Standard\ Task\ Graph(STG)$ developed at Waseda Univerity [30]. Ten out of thirteen taskgraphs, called $rand0000$ to $rand0009$ are randomly generated with exact 50 tasks, and the others, called $robot$, $sparse$, and $fpppp$, are modeled into DAG as benchmarks of actual application. Again, each application is in the following: robot control application has 88 tasks and 131 edges, sparse matrix solver application has 96 tasks and 67 edges, and SPEC fpppp application has 334 tasks and 1145 edges. As mentioned in STG package [30], the task-graphs in these applications are considered without communication costs. Each of application programs are assumed with precedence dependency among tasks. The original taskgraphs in STG assume that each task is executed on single-core only, so that the execution time is provided for assuming single-core execution. In additional, the execution time may be less than 1 when a task is assumed to be executed on 32 cores, so that we multiply them by one hundred. Therefore, in our experiments, the execution times of MFJ tasks on multiple cores are assumed to be $Time_{i,k,j} = 100 \times Time_{i,1,j} \times (0.1 + 0.9/k)$, for



(a) Scheduling results on 4 cores



(b) Scheduling results on 8 cores

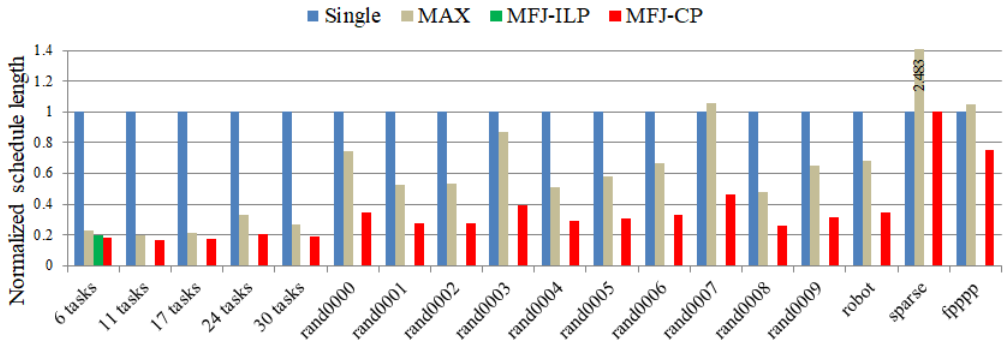
Figure 3: Scheduling results for MFJ tasks on 4 and 8 cores

$k \geq 1$, where $Time_{i,k,j}$ denotes the execution time of $j-th$ thread in task i assigned to k cores, where each thread is assumed to have 10 % overhead for parallelization of a task. The number of cores in the target systems is varied from 4 to 32.

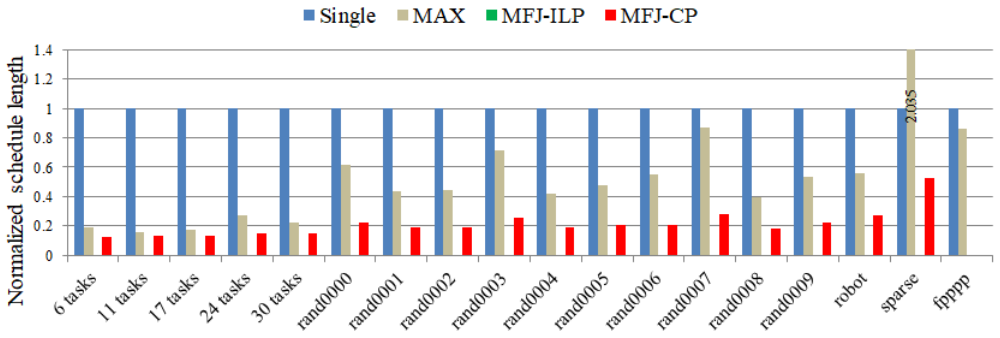
- *Single (optimal)*: Optimal scheduling on the assumption that a task is assigned to a single core. The optimal solutions are provided in the STG package [30].
- *Max*: Every task is executed on all cores, and the tasks are sequentially executed.
- *MFJ-ILP*: ILP-based MFJ task scheduling presented in [11].
- *MFJ-CP*: CP-based MFJ task scheduling presented in this paper.

Figure 3 (a) and (b) and Figure 4 (a) and (b) show scheduling results on 4, 8, 16 and 32 cores, respectively. All of the results are normalized to the Single method. In case of 4-core systems in Figure 3 (a), the MFJ-CP method can find schedules in time, but they are slightly different from the results by the Single method due to poor parallelism against the number of tasks. As less number of tasks, we could find much shorter schedules even on 4 cores in the systems. On the other hand, the results by the MFJ-ILP method are almost missed in the graphs, which means that the method is failed to find any of schedules in time. In case of a small number of tasks, the MFJ-ILP may be possible to find the better schedules than the Single method and the Max method. However, the schedule lengths are the same or longer by 7.1% than the MFJ-CP method.

In the 8-core systems as shown in Figure 3 (b), a much significant improvement is observed. The MFJ-CP method obtains shorter schedules over the other methods. On the other hand, the MFJ-ILP method is failed to find a schedule in almost all the cases. As for the sparse benchmark, the best schedule could be obtained by the execution that each task is assigned to a single core. Compared with the Single method, the MFJ-CP method can shorten the schedule lengths by up to 56.6 %.



(a) Scheduling results on 16 cores



(b) Scheduling results on 32 cores

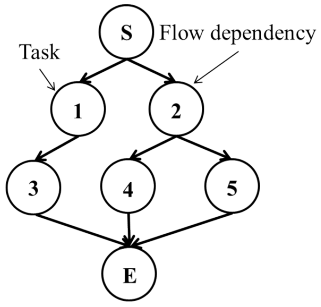
Figure 4: Scheduling results for MFJ tasks on 16 and 32 cores

As increasing number of cores, the MFJ-CP method is still possible to obtain much shorter schedules 65 % on average, but the MFJ-ILP method is again no longer able to find solutions. As for the Max method, it may basically find shorter schedules than the Single method. However, the Max method finds always worse schedules than the MFJ-CP method. On the other hands, the method has an advantage of an ability to find an instant schedule. In the sparse, the tasks include comparative high parallelism against the number of cores in precedence dependency among tasks. Moreover, the overhead by parallelism has influence on the schedule length. Therefore, single-core execution is the better way for this benchmark in poor parallelism of cores against the parallelism of precedence dependency.

In the 32-core systems in Figure 4 (b), the MFJ-ILP method is impossible to find any of the solutions due to high complexity of scheduling on multicore. The same is true for the case of fpppp tried by the MFJ-CP method, but it is successfully able to improve the effectiveness by up to 81.9% with regard to minimization of the schedule lengths in the overall cases.

4 Scheduling for Malleable Synchronous Tasks

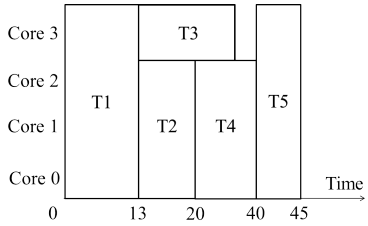
This section presents malleable synchronous (MS) task scheduling. Similar to the scheduling presented in the previous section, MS task can be partitioned into an unfixed number of threads. However, the threads cannot be executed independently, and need to be scheduled synchronously with each other. That is, the threads of a MS task runs at different times on different cores. In this section, we describe a CP-based scheduling method for MS tasks with built-in functions by ILOG CP Optimizer. Scheduling of MS tasks decides the execution order of the tasks and the number of threads for each task at the same time as scheduling. Our scheduling aims to minimize the overall schedule length.



(a) Task graph

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

(b) Execution time of task 1



(c) An optimal schedule

Figure 5: A scheduling example for malleable synchronous tasks

4.1 Problem Definition

Figure 5 (a), (b) and (c) show an example of scheduling for MS tasks. In Figure 5 (a), a set of dependent tasks is represented as DAGs, called a task-graph. Each of the tasks is associated with the execution time which is a function of the number of cores to execute the task. A table of execution time for task 1 is shown in the Figure 5 (b). For instance, the execution time of task 1 is 45 time-units when it is executed on a single core. We assume that each task needs synchronization among all threads in the tasks, and all threads are executed on multiple cores at the same time. The execution time of task 1 is 25 time-units when it is executed on dual cores. It is assumed that the functions of the execution time for the tasks are given, and how to obtain these values is out of scope in this paper. The number of cores assigned to the tasks is determined simultaneously with task scheduling. Figure 5 (c) shows one of the optimal schedules for the task-graph in Figure 5 (a). The overall scheduling length of the schedule is seen as 45 time-units.

4.2 ILP Formulation

The scheduling for MS tasks presented in [15] is based on ILP. In order to compare with the proposed method, the solutions are obtained by a commercial solver. The rest of this section briefly describes the ILP formulation presented in [15].

Let $map_{i,j}$ be a 0-1 decision variable. $map_{i,j}$ becomes 1 if task i is mapped to core j, otherwise 0. Let $cores_{i,k}$ be 0-1 variable, which becomes 1 if task i uses k cores; k is varied from one to the maximum number of cores in the target system, and otherwise 0. Note that $cores_{i,k}$ depends on $map_{i,j}$ and is determined as follows.

$$\forall i, \qquad \sum_{k} cores_{i,k} = 1 \tag{14}$$

$$\forall i, \qquad \sum_{j} map_{i,j} \;=\; \sum_{k} cores_{i,k} \times k \tag{15}$$

Let $Time_{i,k}$ denote the execution time of task i on k cores. $Time_{i,k}$ is assumed to be given as mentioned earlier. The execution time of task i is given by the following equation.

$$\forall i, \qquad time_i = \sum_{k} \{cores_{i,k} \times Time_{i,k}\} \tag{16}$$

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

$$\forall i, \qquad finish_i = start_i + time_i \tag{17}$$

The two tasks, $i1$ and $i2$, cannot be mapped to a core at the same time. That is, the execution of the two tasks cannot be overlapped in time. This resource constraint of cores is formulated by the following inequality.

$$\begin{aligned} \forall i1,i2,j, \quad & map_{i1,j} + map_{i2,j} < 2 \\ \vee \quad & finish_{i1} \leq start_{i2} \\ \vee \quad & finish_{i2} \leq start_{i1} \end{aligned} \tag{18}$$

This work assumes a set of dependent tasks, and some tasks have precedence dependencies. As a precedence dependency between task $i1$ and task $i2$, $Flow_{i1,i2}$ becomes 1 if task $i1$ is ended before task $i2$ starts, and otherwise 0.

$$\forall i1,i2, \qquad Flow_{i1,i2} \rightarrow finish_{i1} \leq start_{i2} \tag{19}$$

The objective is to minimize the overall scheduling length. Therefore, the objective function of the scheduling problem is as follows.

$$\text{Minimize :} \qquad max_i\{finish_i\} \tag{20}$$

Now, the scheduling problem for MS tasks is formally defined: Given a task-graph, a set of cores, $Time_{i,k}$ and $Flowi1,i2$, it decides $map_{i,j}$ and $start_i$ which minimize the objective function (7) subject to the six constraints (1)-(6). Although some of the expressions are not linear, they can be easily transformed into linear forms as presented in[15].

4.3 Constraint Programming Approach

In this section, we propose scheduling of MS tasks based on CP. For the formulation, we use the built-in functions provided by IBM ILOG CP Optimizer as well as the previous section. We describe a CP-based scheduling method for MS tasks in the following.

Let $task_i$ denote an interval decision variable for task i. It must be present in the scheduling problem, and thus the presence status of $task_i$ is true. Next, let $decision_{i,k}$ denote an interval decision variable which decides the number of cores to execute $task_i$. $decision_{i,k}$ is present if k cores are assigned to $task_i$, otherwise absent. This work assumes that the execution time of $decision_{i,k}$ is given, similarly to $Time_{i,k}$ in Section 4.1.

This work assumes a set of dependent tasks, where tasks may have a given precedence dependency. The function, called $endBeforeStart(a, b)$ built in ILOG CP Optimizer, represents the precedence constraint between interval variables a and b are considered to be true provided that both of the two interval variables are present. Therefore, the precedence dependency constraint that $task_{i1}$ must be finished before $task_{i2}$ starts is expressed as follows.

$$\forall i1,i2, \qquad endBeforeStart(task_{i1}\ ,\ task_{i2}) \tag{21}$$

An alternative constraint is one of the functions in ILOG CP Optimizer. Let denote a and b_i interval variables. The function $alternative(a, b_1...b_n)$ represents exactly one of a set of intervals $b_1...b_n$ is present on the condition that interval a is present. The start and the end time of interval a are synchronized with those of b_i which is chosen to be present. If a is absent, every b_i is also absent. Using the alternative function, we can guarantee that one of $decision_{i,k}$ is present for each $task_i$, as shown in Formula (22).

$$\forall i, \qquad alternative(task_i, \cup_k \{decision_{i,k}\}) \tag{22}$$

In Formula (22), it should be recalled that $task_i$ is always present. Therefore, one of $decision_{i,k}$ must be present.

This work tries to minimize the schedule length under a resource constraint on a number of cores. At any moment in time, the total number of cores assigned to active tasks cannot exceed the number

of cores in the target system. This resource constraint is expressed with the pulse function in ILOG CP Optimizer. Figure 2, shown in the previous section, shows the concept of the pulse function. The resource constraint is expressed as follows.

$$Ncores \geq CumulFunction\{\sum_{i}\sum_{k} pulse(decision_{i,k},\, k) \tag{23}$$

In the formula (23), $Ncores$ denotes the number of cores. It should be recalled that $decision_{i,k}$ is an interval variable and is present if k cores are assigned to task i. Formula (23) means the value of $pulse(decision_{i,k}, k)$ is k while task i being active. The value k is the number of cores assigned to a task, that is, the right side of the inequality shows that sum of the number of cores assigned to the active tasks. Therefore, this formulation does not take into account mapping of the tasks onto the cores but simply does the number of cores assigned to them. Formula (23) does not identify what cores to be assigned to a task, thus, the number of combinations of the cores assigned to a task is significantly reduced unlike the formulation addressed in the ILP method.

Our objective function is to minimize the overall schedule length, and is defined as follows.

$$\text{Minimize}: \qquad max_i\{endOf(task_i)\} \tag{24}$$

As earlier mentioned in the previous section, $endOf$ is a built-in function in ILOG CP Optimizer, which returns the end time of the given interval variable. We try to minimize the overall schedule length with the benefit of ILOG CP Optimizer that are capable of finding the best values for variables to optimize a specified objective function.

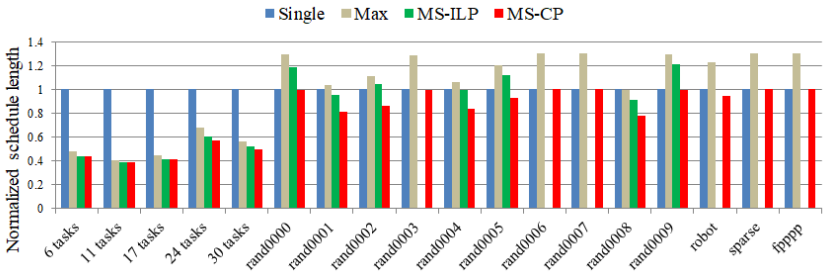
4.4 Expertiments

In order to evaluate and compare the performance of the proposed approach, we have conducted a set of experiments. we have set taskgprahs that are composed of a small number of tasks from 6 to 30 tasks in [29]. The rest of taskgraphs is derived from the *Standard Task Graph*(*STG*) developed at Waseda Univerity [30]. Ten out of thirteen taskgraphs, called *rand*0000 *to rand*0009 are randomly generated with exact 50 tasks, and the others, called *robot*, *sparse*, and *fpppp*, are modeled into DAG as benchmarks of actual application.

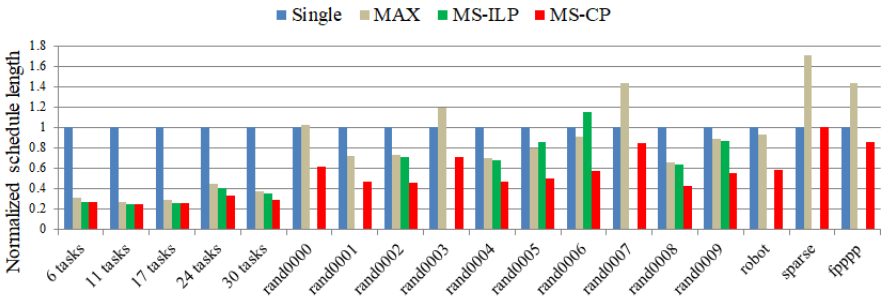
The details of each application are in the following: robot control application has 88 tasks and 131 edges, sparse matrix solver application has 96 tasks and 67 edges, and fpppp application has 334 tasks and 1145 edges. As mentioned in STG [30], the task-graphs in these applications are considered without communication costs. Each of application programs are assumed with precedence dependency among tasks. The original taskgraphs in STG assume that each task is executed on single-core only, so that the execution time is provided for assuming single-core execution. Therefore, in our experiments, the execution times of MS tasks are assumed to be $Time_{i,k} = 100 \times Time_{i,1} \times (0.1 + 0.9/k)$, for $k \geq 1$, where $Time_{i,k}$ denotes the execution time of task i assigned to k cores with 10 % for an overhead by parallelization of tasks. The number of cores in the target systems is varied from 4 to 32.

- *Single (optimal)*: Optimal scheduling on the assumption that a task is assigned to a single core. The optimial solutions are the same as the ones multiplied by 100 in the STG package [30].
- *Max*: Every task is executed on all cores, and the tasks are sequentially executed.
- *MS-ILP*: ILP-based MS task scheduling presented in [15].
- *MS-CP*: CP-based MS task scheduling presented in this paper.

In order to find solutions for the MS-ILP method and the MS-CP method, we use a commercial solver, IBM ILOG CPLEX 12.8 [26]. CPLEX is equipped with two optimization engines, i.e., mathematic programming engine and constraint programming ones. We use the mathematic programming engine and the CP engine for the MS-ILP method and the MS-CP method, respectively. CPLEX



(a) Scheduling results on 4 cores



(b) Scheduling results on 8 cores

Figure 6: Scheduling results for MS tasks on 4 and 8 cores

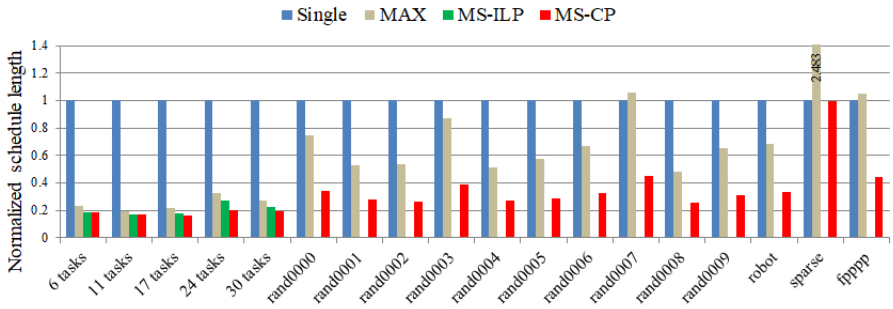
is executed on Intel Core i9 7980XE (2.6GHz) processors with memory of 128GB. The runtime of CPLEX is limited up to 10 hours in CPU time, and then the best schedule found at that moment is used for comparison.

Figure 6 (a) and (b) and Figure 7 (a) and (b) show scheduling results for MS tasks on 4, 8, 16 and 32 cores, respectively. The X-axis of the graphs shows the task-graphs, where *rand*0000 to *rand*0009 include 50 tasks for each task-graph as well as in the previous chapter. The Y-axis shows the schedule length obtained by the four methods. The schedule lengths are normalized to the Single method. In some cases, no solution is found by the MS-ILP method. There is a reason that the MS ILP method is almost failed to obtain even one of feasible solutions in a practical time.

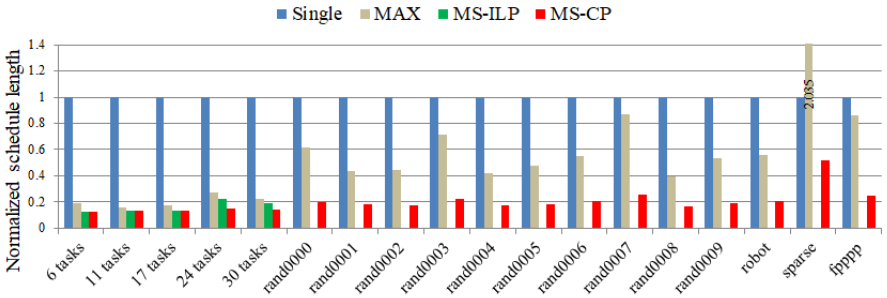
Figure 6 (a) shows the results of the 4-core systems. The results in the MS-CP method surpass the ILP-based method with the improvement by 11% on average, nevertheless there is poor parallelism. As shown in the cases of the applications, the effectiveness of multi-threaded tasks is no more improved than the traditional method. On the other hand, the Max method finds longer schedules in many cases due to the overhead by parallelization.

In the 8-core systems as shown in Figure 6 (b), the effectiveness of the MS-CP method is obviously observed. Although the MS-ILP method fails to obtain shorter schedule in some cases, the MS-CP method is still successfully obtained with better solutions than the other methods. For small task graphs of 6 to 30 tasks, there is slightly difference that a schedule length by the MS-CP method is shorter than the schedule by the MS-ILP method by up to 7.6%. On average, the MS-CP method finds shorter schedules by 18.6% over the results.

In Figure 7 (a) and (b), the results found by the MS-ILP method are almost missed. As increasing number of cores, the number of the threads of each task can be large. On the other hand, the MS-CP method can find feasible schedules, which means that our CP-based approach is obviously able to find good feasible schedules in short time. In the results, we can obtain benefit attributed to a CP-based approach.



(a) Scheduling results on 16 cores



(b) Scheduling results on 32 cores

Figure 7: Scheduling results for MS tasks on 16 and 32 cores

5 Conclusion

This paper studied scheduling methods for MFJ and MS tasks based on constraint programming. Scheduling of MFJ tasks determines the number of threads and the execution order of their threads. Scheduling of MS tasks decides the number of cores to execute each task at the same time as scheduling. The experimental results show the effectiveness of our proposed method. In all of the experiments, we could obtain much better schedules compared with the state-of-the-art methods. Therefore, we have ensured that CP-based scheduling for malleable tasks can quickly find good schedules. In future, we will take into account that more complex problems with communication costs and resource conflicts among tasks

6 Acknowledgment

This work is supported by KAKENHI 15H02680.

References

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

[2] S. Venugopalan and O. Sinnen "Optimal linear programming solutions for multiprocessor scheduling with communication delays," *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 129-138, 2012.

[3] P. F. Dutot, G. Mounie, and D. Trystram, "Scheduling parallel tasks approximation algorithms," *Handbook of scheduling: Algorithms, models and performance analysis*, 2004.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[5] 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.

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

[7] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," *Design Automation and Test in Europe (DATE)*, pp.69-74, 2009.

[8] 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.

[9] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints," *Journal of Computer and System Sciences*, vol.78, no 1, 2012.

[10] D. Ye, DZ. Chen, and G. Zhang, "Online scheduling of moldable parallel tasks," *Journal of Scheduling*, pp. 1-8, 2018.

[11] K. Shimada, I. Taniguchi, H. Tomiyama, "ILP-based scheduling for malleable fork-join tasks," *Accepted for publication in ACM SIGBED Review.*

[12] J. Sun, N. Guan, Y. Wang, Q. Deng, P. Zeng and W. Yi, "Feasibility of fork-join real-time task graph models: Hardness and algorithms." *ACM Transactions on Embedded Computing Systems (TECS)*, vol.15, Issue 1, 2016.

[13] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors." *IEEE Real-Time Systems Symposium*, 2010.

[14] A. Saifullah, K. Agrawal, C. Lu and C. Gill, "Multi-core real-time scheduling for generalized parallel task models." *IEEE Real-Time Systems Symposium*, 2011.

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

[16] F. Rossi, P. V. Beek, and T.Walsh, "*Handbook of constraint programming*," Elsevier, 2006.

[17] A. Goldman and Y. Ngoko, "A MILP approach to schedule parallel independent tasks," *Parallel and Distributed Computing*, pp. 115-122, 2008.

[18] W. Liu, Z. Gu, J. Xu, X. Wu, and Y. Ye, "Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol.22, no. 8, 2011.

[19] A. Malik, C. Walker, M. O'Sullivan, and O. Sinnen, "Satisfiability modulo theory (SMT) formulation for optimal scheduling of task graphs with communication delay," *Computers and Operation Research*, vol. 89, 2018.

[20] P. Baptiste, C. Le Pape, and W. Nuijten, "Constraint-based scheduling: applying constraint programming to scheduling problems," *Springer Science and Business Media*, vol. 39, 2012.

[21] K. Kuchicinski, "Constraints-driven scheduling and resource assignment,"*ACM Transactions on Design Automation of Electronics Systems (TODAES), vol. 8(3), pp.355-383, 2003.*

[22] R. Gedik, D. Kalathia, G. Egilmez, and E. Kirac, "A constraint programming approach for solving unrelated parallel machine scheduling problem," *Computers and Industrial Engineering*, 2018.

[23] K. Xiaohong, R. Li, and Y. Zhang, "Scheduling tasks to multi-processor platform using constraint programming and tabu search," *Natsional'nyi Hirnychyi Universytet. Naukovyi Visnyk*, no. 4, 2016.

[24] I. J. Lustig and J-F. Puget, "Program does not equal program: Constraint programming and its relationship to mathematical programming," *Journal on Interfaces*, vol. 31, no 6, 2001.

[25] P. Laborie, "IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems," *AI and OR Techniques in Constraint Programming for Combinational Optimization*, vol.5547, 2009.

[26] IBM CPLEX Optimizer, https://www-01.ibm.com/ software/commerce/optimization/cplex-optimizer/, (Last accessed: January 3, 2019).

[27] L. Perron, P. Shaw and V. Furnon, "Propagation guided large neighborhood search," *International Conference on Principles and Practice of Constraint Programming*, vol.3258, 2004.

[28] R. Philippe, "Impact-based search strategies for constraint programming," *International Conference on Principles and Practice of Constraint Programming*, vol.3258, 2004.

[29] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graph for free," *International Workshop on Hardware/Software Codesign*, 1998.

[30] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, 2002.